Evolving Neural Networks: Building a more efficient method

Michael Yudanin[1]

[1]Purdue University
yudanin@gmail.com

Abstract

Neuroevolution presents an alternative to traditional approaches for constructing efficient neural networks. Instead of manually selecting hyperparameters or conducting exhaustive grid searches, neuroevolution draws inspiration from biology, allowing the most effective models to "survive" and "reproduce," passing their traits, with some modification, to subsequent generations. Building on prior work and utilizing the Genetic Algorithm approach, this paper proposes a method for evolving neural networks by selecting the most fit models, combining network parameters through crossover, introducing mutations, and iterating this process over multiple generations. Each network undergoes limited backpropagation-based training before fitness evaluation and reproduction. This approach was tested on the MNIST and CIFAR-10 datasets, comparing the evolved networks' performance against well-established architectures. Results show that the evolved networks perform comparably to or even surpass traditional architectures. Directions for future research are also discussed.

Code — https://github.com/yudanin/neuroevo-mnistcifar.git

Introduction

Deep Neural Networks, or DNNs, are learning systems with multiple layers of connected simple units, or neurons, that process inputs. Then, based on the processed results, activation functions pass the status to the next layer of neurons.[1] They are usually trained using backpropagation to adjust the weights and biases that assign varying importance to the neurons as far as their impact on the output is concerned. The DNN architecture, and specifically its hyperparameters - the number of layers, the number of neurons in each layer, the pattern of their interconnection, the method to determine the initial values of the weight and biases, the learning rate and they way to update it with the training of the network, etc. - are determined by the model designer.

The values of these parameters are crucial, as they have a decisive impact on the DNN's performance. They will determine whether the network is likely to fall for local instead of global minima, the amount of computing resources it will need for training, the number of epochs it will require to achieve optimal performance, if at all, and more.

[1]This section is based on the review by (Galván and Mooney 2021)

An alternative to using DNNs with hyperparameters defined by the model designer is applying the principles of evolution to the development of neural networks. This way, the model is provided with a range of options for its parameters - a search space. Then, different instances of the model, or individuals, randomly choose from this repertoire to define their architecture and perform the task assigned to them, e.g., image classification. The models that perform better than others survive and are allowed to procreate - produce slightly modified copies of themselves, new models with some parameters altered. This follows the logic of Darwin's "descent with modification." This way, though the survival of the best-performing, we hope to achieve the goal for which the model is being created.

There are two main strategies of evolutionary optimization: Evolution Strategies (ES) and Genetic Algorithms (GA) (Lange, Tang, and Tian 2023). ES consist in sampling a population of candidates from a parameterized distribution, e.g., a multivariate normal distribution, estimating their fitness, and then updating the distribution to maximize the chances of well-performing candidates to be selected for further consideration. GA keeps a repository of models that perform well and then relies on them for evolving the solution further through modifications mimicking biological evolution: crossing over some characteristics of the parents in the offspring and mutation, slight changes in parents' characteristics. In what follows, we will be considering GA as the main evolutionary method.

The specific copies of them model, or individuals, are evaluated based on a fitness function which assesses their performance on the target task or tasks. This performance determines for each model whether it will propagate its code to future generations. Then, three types of genetic operators are utilized: selection, crossover, and mutation. The selection operator uses the output of the fitness function to determine whether the individual will be chosen for further consideration. The recombination operator, also known as stochastic crossover, exchanges the "genetic material," or certain network components, e.g., filters, layers, or activation function, between selected individuals, usually two parents, in order to determine the characteristic of the next generation, or children. The stochastic mutation operator, similarly to genetic mutation, makes random changes to certain characteristics of children - and thus explores the search

space. The evolutionary process is repeated until the stopping condition is reached, e.g., when the desired performance is achieved or the pre-determined number of evolutionary generations is exhausted. At this point, the population of surviving individuals might include those models that represent the global optimal solutions and thus can be used in production.

Problem Definition

AI researchers interested in the merits of different ways to create better neural networks address the problem of the feasibility of neuroevolution as an alternative to the traditional way to construct neural networks: definition of hyperparameters by the model designer or grid search within the parameters defined by the model designer, and training through backward propagation with gradient descent. Different authors, as we shall see in the next section, focus on evolving better final layer for pre-trained networks, on harnessing the power of language models to evolve better architectures, or on the ways to benchmark different neuroevolutionary methods. The challenge here is to find a way to evolve neural networks that will lead to better performance while saving computational resources, expressed as the effort needed to train or evolve networks that demonstrate good performance.

Related Work

In their paper "Evolving Normalization-Activation Layers" (Liu et al. 2020) Liu, Brock, Simonyan, and Le are trying to evolve normalization layers and activation functions rather than designing them, hoping to achieve better results in comparison with the usual way of defining hyperparameters.

The design of normalization layers and activation functions usually follows common heuristics, e.g., mean subtraction and variance division for normalization layers and choosing scalar-to-scalar activation functions: Sigmoid, ReLU, Softmax, etc. However, these might not be optimal. Firstly, they are arbitrarily chosen and then sometimes decided upon using trial and error, thus being in danger of leading to local rather than global minima in terms of the chosen loss function. Secondly, this approach treats normalization layers and activation functions separately, while it might be beneficial to consider them together.

In contrast, the authors propose to take an evolutionary approach for designing normalization layers and activation functions, where the best performing ones survive and then give rise to modified versions of themselves.

The authors formulate the normalization and activation layer as a tensor-to-tensor computational graph. This graph is a search space consisting of mathematical functions, or basic operations: addition, multiplication, negation, square root, etc. The search space is characterized by a large size and high sparsity. Thus, random search does not yield good results here. Therefore, the authors use evolution as the search method of choice. They evaluate each layer over many different architectures and then use a tournament selection method: a random subset of basic functions are compared, and then the best-performing entity multiplies with modifications in its intermediate node, basic operation, and predecessors. [2]

The performance is judged based on the rejection protocols the authors developed to address the sparsity of the search space. There are two of them:

- Quality: validation accuracy after 100 steps on CIFAR-10
- Stability: filtering out candidate layers if their gradient norm exceeds 10^8 in up to 100 steps.

After applying this method, the authors discovered - or, more precisely, evolved - a number of new layers, EvoNorms, that were different from the commonly used ones. They tested these layers on three architectures, ResNet50, MobileNetV2, and EfficientNet-B0, and found them to perform well.

This paper provides an important window into how neuroevolution works. It has, however, three limitations. Firstly, the authors use Evolution Strategies (ES) rather than Genetic Algorithms (GA). This does not realize fully the potential of neuroevolution to short-cut to better models through propagating useful network features. Secondly, they focus exclusively on the normalization and activation, while there seems to be a considerable potential into opening the number of layers, for example, to evolutionary pressures. Thirdly, they evaluate the performance of the functions they evolve with ready-made well-trained networks. This limits the judgment of effort invested into evolving the models: the resources that went into training the host networks should be taken into consideration as well.

While random choice over a search space and subsequent propagation of the more successful models seems like a promising way to progress, some researchers think that improving search operators would further improve neuroevolution. Chen, Dohan, and So in "EvoPrompting: Language Models for Code-Level Neural Architecture Search" suggest utilizing Language Models as operators for general adaptive mutations and crossover (Chen, Dohan, and So 2023). Specifically, they iteratively evolve in-context prompts and thus prompt-tune the LM that is being utilized. The authors called this technique EvoPrompting.

In-context prompts provide the LLM with an example of the output and then prompt it to generate a response based on it (Raventós et al. 2023). The most common example of in-context learning (ICL) is asking a question and providing a format for the answer, or the seed. This way, the end-user prompting the LM does not need to train the model using computationally-intensive processes and can get to the desired result much quicker.

Chen, Dohan, and So use this method to evolve better deep neural architectures. They start with seeding an LLM, a 62B parameter PALM model that had been pre-trained on 1.3T tokens that included code, with initial code samples, ask the LM to produce modified versions of it, and then test these versions against the task at hand. After computing the fitness of each model, they select the best performing ones, use them as seeds, and repeat the process. This way, the

[2] This method is also known as *elitism*.

LM's vocabulary replaces the search space defined by the researcher in the more traditional neuroevolutionary methods, e.g., (Liu et al. 2020), and at the same time take advantage of what the LLM had learned during its own training about the possible models and their modifications. It is also conceivable that the evolutionary model design produced this way benefits from the the LLMs ability to remember the sequence of prompts from the user. As certain - namely, higher-fitness - models are selected each time for further refinement, the LLM might infer the desired sort of improvements and "play along" by providing model architectures with better fitness prospects next time, similarly to how it gets the context from previous prompts in other tasks, e.g., answers questions within the context provided by the preceding prompts.

The key finding is that EvoPrompting leads LLMs to creating effective neural architectures that outperform more conventional models. The authors first test their hypothesis on a lower-compute problem, MNIST-1D (Greydanus and Kobak 2024), where it produces convolutional architectures that perform better than published manually designed models (4.1). Then they apply the method to designing neural networks to solve problems from the more challenging CLRS Algorithmic Reasoning Benchmanrk (Veličković et al. 2022) where it achieves impressive results as well: the generated novel architectures outperform published models on 21 out of 30 tasks.

While this approach might look promising in terms of generating high-performing models, it relies on having pretrained LLMs. This way, the burden of computation falls on the LLM, not the evolutionary method itself. Therefore, what the method shows is the quality of the LLM mostly, not of neuroevolution. To be more precise, it reflects the quality to the LLM's intelligent design or, perhaps, the ability of the LLM to mimic the logic of evolution. While impressive and potentially very useful in the field, it tells us little about the merits of neuroevolution as opposed to other approached, e.g., grid search coupled with traditional model training. For example, if we want to claim that LLM-assisted, to be completely honest, LLM-driven neuroevolution is more efficient, we will need to factor in the computational resources that went into training and running the LLM. If we do that, the overall efficiency of the method appears questionable.

An important area for evolving neural networks rather than using the traditional approach of arbitrarily setting hyperparameters is having proper performance benchmarks. Lange, Tang, and Tian report on the establishment of such a benchmark, NeuroEvoBench, in "NeuroEvoBench: Benchmarking Evolutionary Optimizers for Deep Learning Applications" (Lange, Tang, and Tian 2023).

The authors argue that the existing benchmarks that have been developed from the networks with gradient descent optimization might not be fully adequate for evolutionary optimization, as the neural network fitness landscapes might be very different for neuroevolution. This they trace to the differences in the nature of considered problems and the search space size.

In order to answer the need for benchmarks, Lange, Tang, and Tian constructed eleven black-box optimization tasks and a standard experimentation protocol. Then, they executed the benchmark for ten evolutionary optimization methods to analyze their quality.

The NeuroEvoBench has the following components:

- 11 problems: Black-Box Optimization tasks of function optimization, Control tasks vision tasks, e.g., CIFAR-10 classification, and sequence produciton tasks, e.g., Sequential MNIST classification.
- 10 optimizers: 6 ES and 4 GA
- 4 fitness evaluation methods: raw, z-score, ranks, and [-1,1] norm
- 3 experiment types:
  1. Random Search, that can be executed with a different number of trial, what the authors refer to as budget: 20 trials, 40 trials, etc. This is used for initial tuning.
  2. Multi-Seed Eval: the best configurations from the Random Search are re-evaluated on multiple random seeds. This tests the robustness of the tuning procedure.
  3. Grid search: finally, the hyperparameters settings are evaluated on the grid search.

The authors built an API for NeuroEvoBench. They tested a variety of evolutionary algorithms and found no clear winner across different configurations.

NeuRoEvoBench is a very useful development. However, as the authors acknowledge, the real world can present a much wider range of scenarios.

There seems to be two possible end-goad for neuroevolution:

- Development of a method to evolve networks for any intent or purpose which would be faster and less resource-consuming than gradient descent optimization for networks with arbitrarily chosen or grid-searched parameters.
- Development of a universal neural network of sorts that will have wide-ranging capabilities and would be, perhaps, able to evolve further without explicit prodding from the model designers.[3]

For the first task, benchmarks have limited utility, as the NNs will have to be benchmarked against specific tasks. For the second, much more ambitious and somewhat holy-grailish undertaking, the benchmarks can be very useful as an intermediate stage: their improved performance across different tasks, something Lange, Tang, and Tian failed to find, would be an indicator that the development is proceeding in the right direction.

Methodology

In order to try and advance the understanding of neuroevolution and developing upon the ideas from (Liu et al. 2020) and (Chen, Dohan, and So 2023) yet trying to focus on Genetic Algorithms, I created a framework where neural network architectures would be generated randomly and then

[3]Perhaps, in this case they should be called model evolvers or even demiurges.

evolve through multiple generations, with the survival of the better-fitting and descent with modification. Each network, after it is created and before being evaluated for fitness and evolving further, will be trained using backpropagation for a limited number of epochs. If selected, it will carry with it the weight and biases of the model but will not pass them to its children.

The models to be evolved are convolutional networks, and the task to determine their fitness is image classification. Each model, once generated, goes through a short, e.g., 5-10 epochs, training through backpropagation by gradient descent on the cross-entropy loss function.

The fitness criteria is accuracy: the percentage of images identified correctly.

The code components used to enable the evolutionary process are:

1. Meta-parameters and the architecture space

   The meta-parameters pertain to the characteristics of the neuroevolutionary process:

   - Initial population size: the number of individual neural networks to start with
   - Number of generations: the number of evolutionary rounds where the parent networks give rise to children
   - Mutation rate: the probability that an individual will change the characteristics received from its parents.
   - Number of survivors: the number of the individuals with the highest fitness in each generation that will stay and give rise to "children"
   - Number of training epochs: the number of epochs each generated model will train using backpropagation
   - Input image height and width (based on the image set used)

2. Architectural search space:

   Providing the options and ranges to choose from for defining network hyperparameters:

   - Number of layers, e.g., 2 to 5
   - Number of filters in a convolution layer, e.g., 8, 16, 32, 64, 128, 256
   - Kernel sizes, e.g., 3 and 5
   - Activation function to choose from, e.g., ReLU, Sigmoid, and Leaky ReLU
   - Pooling layers, e.g., None, Max, and Average

3. Functions and classes:

   - Architecture generation (generate_architecture): randomly chooses values from the architectural space components. Figure 1 provides an example of a randomly generated architecture:

Figure 1: Sample architecture

```
Architecture:
  Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  ReLU()
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  LeakyReLU(negative_slope=0.01)
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  ReLU()
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  Flatten(start_dim=1, end_dim=-1)
  Linear(in_features=1152, out_features=128, bias=True)
  ReLU()
  Linear(in_features=128, out_features=10, bias=True)
```

   - Neural Network class (NN): builds a neural network model based on the generated architecture. Each network entity gets its unique GUID.
   - Training and fitness evaluation: calls NN to generate a model, trains the model with Adam optimizer and cross-entropy loss function on the training set, and evaluates it on the test set. Outputs the percentage of correct identifications.
   - Create population: generates models based on architectures
   - Crossover: mixes parts of "parent" models. There are two crossover functions:
     - Simple crossover: randomly chooses the parent from which the network architecture will be copied
     - Layer exchange:
     (a) Randomly chooses a parent to serve as a basis (parent 1)
     (b) Copies parent 1's architecture - basic architecture
     (c) Drops a random number of layers from the basic architecture
     (d) Chooses a random number of layers from parent 2 and inserts them in the basic architecture wherever possible while keeping the dimensional integrity to form the child architecture.
   - Mutate: based on the mutation rate, randomly chooses from the search space of activation functions and filter numbers. The following characteristics are subject for mutation:
     - Number of filters
     - Kernel size
     - Padding (calculated based on kernel size)
     - Activation function
     - Pooling (if does not break dimensions)
   - Evolve function (evolve): starting with the initially generated population, evaluates fitness for each individual, chooses the most fit ones, and performs crossovers and mutations.

     The selection of the best fit models is done using two algorithms:

     (a) Roulette wheel choice following the stochastic algorithm suggested by Lipowski and Lipowska (Lipowski and Lipowska 2012). This algorithm ensures that the probability of each model to survive and procreate is proportional to its fitness, in our case - accuracy.

(b) Elitist choice: selecting a number of best-performing models.

The Elitist choice has a potential issue - the tendency to always choose the best of the originally generated models and ignore child models. Each model that survives past first generation carries on the improved weights and biases from its previous trainings, thus making it unlikely for even very advantageous architectures generated later to out-perform them, as they would have a smaller number of training epochs. For example, if each generation trains for 5 epochs, a model that had the accuracy of 76% in the first generation might get to 78% in the second, with 10 training epochs overall. A model that was created in the second generation and has 77% accuracy will not be selected over the first one if we use the Elitist method. With roulette wheel they will have a similar probability to survive.

4. The process of evolving models consists in:
   (a) Loading a dataset
   (b) Running the evolutionary process for the specified number of generations.

The assets have been implemented in Google Colab using PyTorch.

The code was executed on Google's A100 GPU runtime.

In order to test the effectiveness of neuroevolution and compare it with traditional training through backpropagation, I conducted experiments on MNIST and CIFAR-10. In both cases, they consisted in:

1. Evolving new networks with simple crossover, i.e., without mixing parents' layers into child layers, and with layer exchange while deciding on survival through Roulette and Elitist methods. This gives four options:
   - Simple crossover, Elitist selection
   - Layer crossover, Elitist selection
   - Simple crossover, Roulette selection
   - Layer crossover, Roulette selection
2. Choosing an architecture commonly known as efficient and training it for the number of epochs equal to the evolutionary effort. The evolutionary effort is calculated as (number of generations) x (number of training epochs) x (population)
   This way, each common architecture model will be trained for
   (number of generations) x (number of training epochs).

Experimental Results

1. MNIST

The first set of experiments has been conducted using MNIST dataset.[4].

The search space for MNIST parameters was:

- Number of convolution layers: 2, 3, 4, 5
- Convolution filters: 16, 32, 64, 128
- Kernel sizes: 3, 5
- Activation functions: Rely, Sigmoid, Leaky ReLu
- Pooling options: None, Max, Average

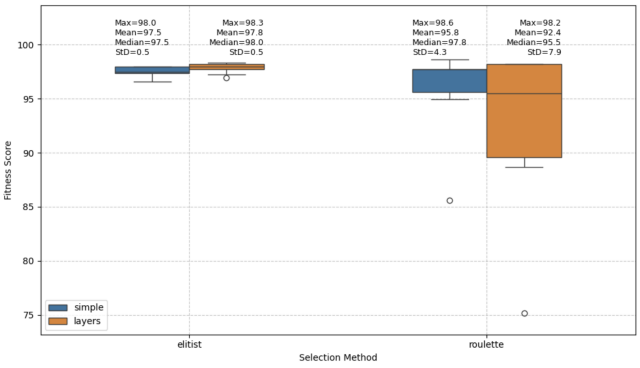
The common parameters for all MNIST experiments were:

- Population size: 15
- Number of generation to evolve: 5
- Mutation rate: 0.1
- Number of survivors in each generation: 8
- Number of training epochs for each model: 3
- Number of output features of the first fully connected layer: 128

Evolutionary experiments Evolving randomly generated networks through:

- Simple crossover, Elitist selection
- Layer crossover, Elitist selection
- Simple crossover, Roulette selection
- Layer crossover, Roulette selection

Figure 2: Fitness Distribution by selection Method and Crossover



In a tabular form, the results are:

Table 1: Statistical summary by method and crossover type

Method	Crossover Type	Mean	Std	Min	Max
Elitist	Layers	97.91	0.45	97.09	98.34
	Simple	97.40	0.46	96.91	98.28
Roulette	Layers	92.79	6.97	76.25	97.78
	Simple	93.47	3.70	87.94	97.84

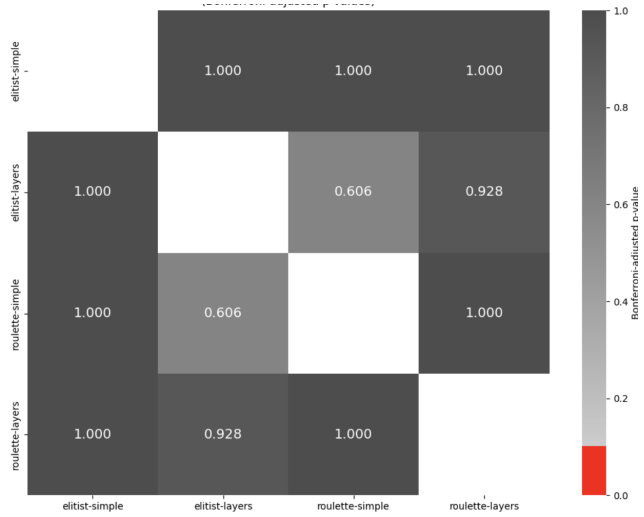
It is interesting to note that the distribution of fitness rankings in the case of Roulette selection is visibly wider than in the case of Elitist selection. The reason is the greater tolerance of Roulette selection to lower-fitness models, which gives higher chances of survival to the child models that are trained less than the surviving parents.

To analyze the differences statistically, I used a non-parametric Pairwise Mann-Whitney U test with Bonferroni

[4]https://pytorch.org/vision/0.19/generated/torchvision.datasets.MNIST.html

correction, since the sample sizes were small. The results are:

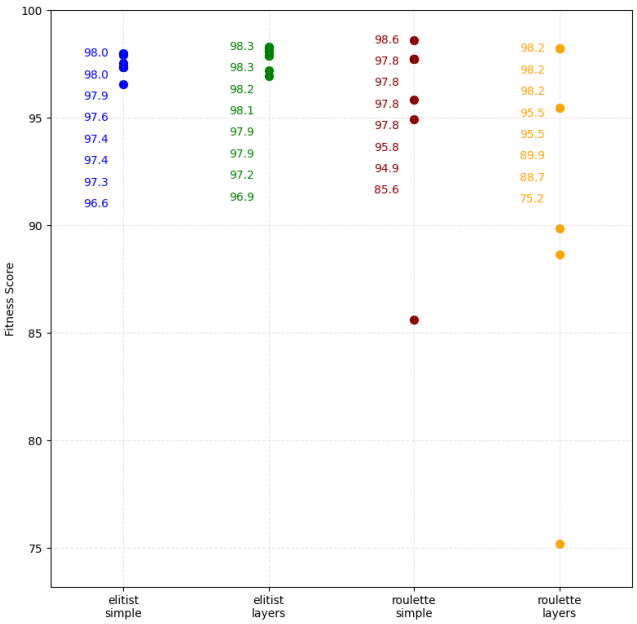
Figure 3: Pairwise Mann-Whitney U test with Bonferroni correction



There are no differences at any reasonable level of significance between the four groups.

We are most interested, however, in the highest fitness models produced as a result of neuroevolution:

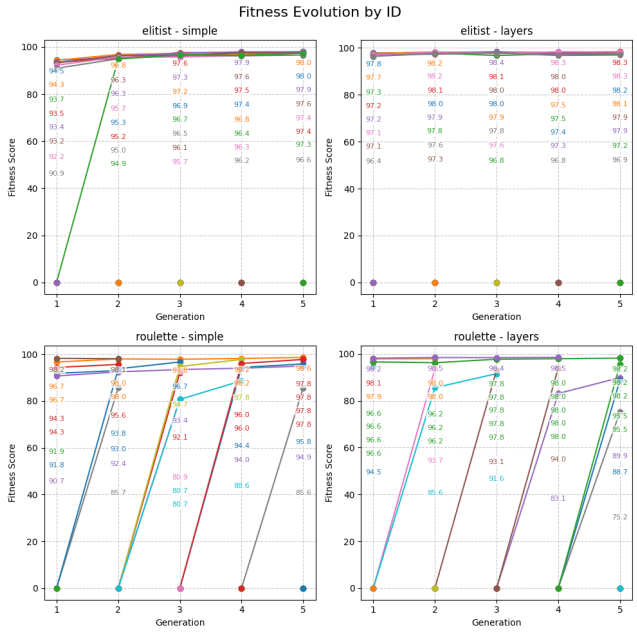
Figure 4: Distribution of Fitness Values



When we look at the distribution of non-zero fitness values, i.e., the surviving models that are able to identify some images correctly after minimal training, we see that there are no significant differences between the different ways to evolve the networks as far as the best evolved models are concerned, and the simple crossover and Roulette selection combination comes on top.

Next, let's examine the evolution of individual models:

Figure 5: Fitness Evolution



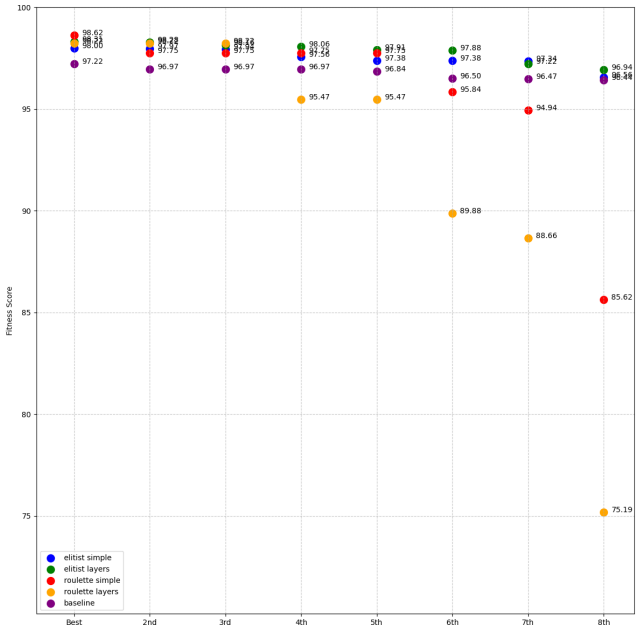
When subjected to the Elitist selection, the process gives little, if any, chances to the evolution of children. In the case of Roulette selection, they have higher chances and indeed evolve into high-performing models.

Comparison with LeNet-5 Architecture Performance In this experiment, a commonly used well-performing LeNet-5 architecture (Lecun et al. 1998) have been trained through backpropagation.

The original Tanh function has been replaced by ReLU, to achieve better results.

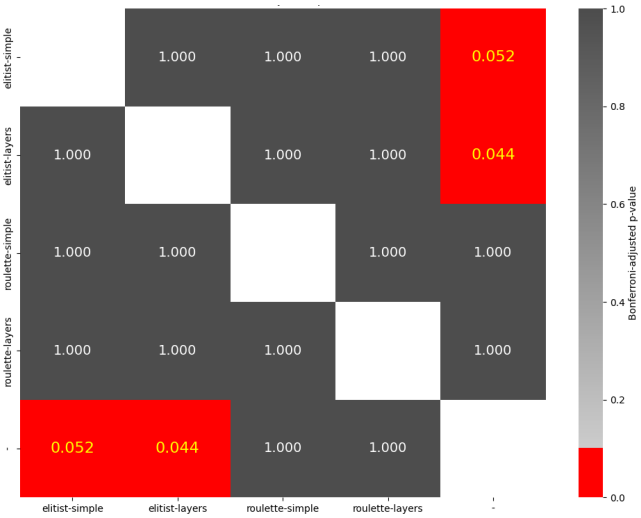
Each of the 15 instances of the LeNet-5 architecture was trained for 15 generations, to maintain the overall effort equal to that of the evolutionary effort.

Figure 6: Comparison: Evolved Models vs LeNet-5 Models



Here again the differences were analyzed using the pairwise Mann-Whitney U tests with a Bonferroni correction:

Figure 7: Comparison: Evolved Models vs LeNet-5 Models



Here the results are quite interesting:

- The Elitist selection method, with and without crossover, performs better than both the Roulette method and the baseline LeNet-5 architecture.
- The difference in performance between different evolutionary methods seems less pronounced for the four top-performing evolved architectures, even though it is not feasible to ascertain that statistically due to the small number of samples.

MNIST is a relatively simple and well-studies dataset. In order to achieve more solid conclusions, we will try the method on CIFAR.

2. CIFAR-10

The second set of experiments has been conducted using CIFAR-10 dataset.[5].

The search space for CIFAR-10 parameters was:

- Number of convolution layers: 4, 5, 6
  a bit higher for CIFAR-10, as it is a more complex dataset with larger images.
- Convolution filters: 32, 64, 128, 256
- Kernel sizes: 3, 5
- Activation functions: Rely, Leaky ReLu
  Sigmoid was not used due to the vanishing gradients problem.
- Pooling options: None, Max, Average

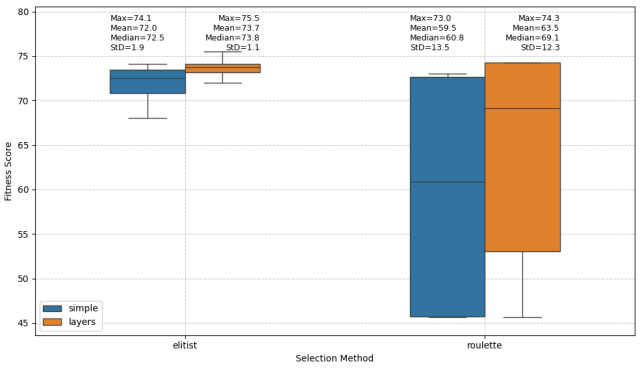
The common parameters for all CIFAR experiments were:

- Population size: 20
- Number of generation to evolve: 8
- Mutation rate: 0.1
- Number of survivors from each generation: 10
- Number of training epochs: 7
- Number of output features of the first fully connected layer: 512

Evolutionary experiments As with MNIST, with CIFAR-10 I evolved randomly generated networks through:

- Simple crossover, Elitist selection
- Layer crossover, Elitist selection
- Simple crossover, Roulette selection
- Layer crossover, Roulette selection

Figure 8: Fitness Distribution by selection Method and Crossover



In a tabular form, the results are:

[5]https://pytorch.org/vision/main/generated/torchvision.datasets.CIFAR10.html

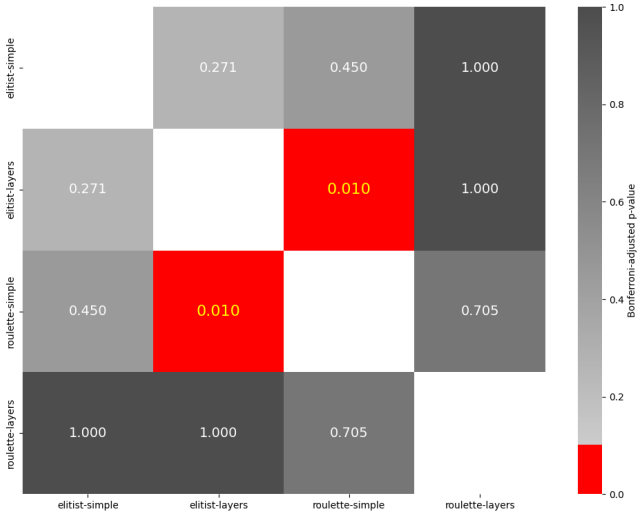
Table 2: Statistical summary by method and crossover type

Method	Crossover Type	Mean	Std	Min	Max
Elitist	Layers	73.69	1.14	72.00	75.53
	Simple	71.98	1.86	68.06	74.09
Roulette	Layers	63.52	12.32	45.66	74.28
	Simple	59.55	13.55	45.62	73.03

As with MNIST, the distribution of fitness rankings in the case of Roulette selection is visibly wider than in the case of Elitist selection.

To analyze the differences statistically, I used a non-parametric Pairwise Mann-Whitney U test with Bonferroni correction, since the sample sizes were small. The results are:

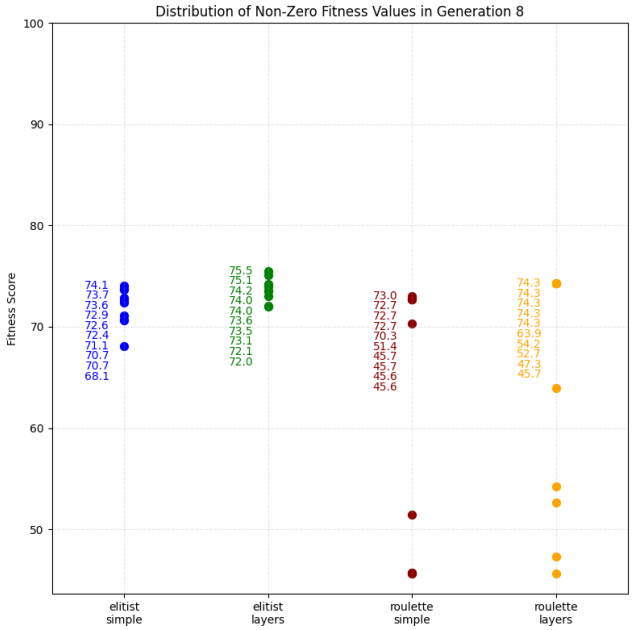
Figure 9: Pairwise Mann-Whitney U test with Bonferroni correction



For CIFAR-10, there is a significant difference (p = 0.01) between the Elitist selection with layers crossover and the Roulette selection with simple crossover, where the former demonstrates higher accuracy.

As we are most interested in the highest fitness:

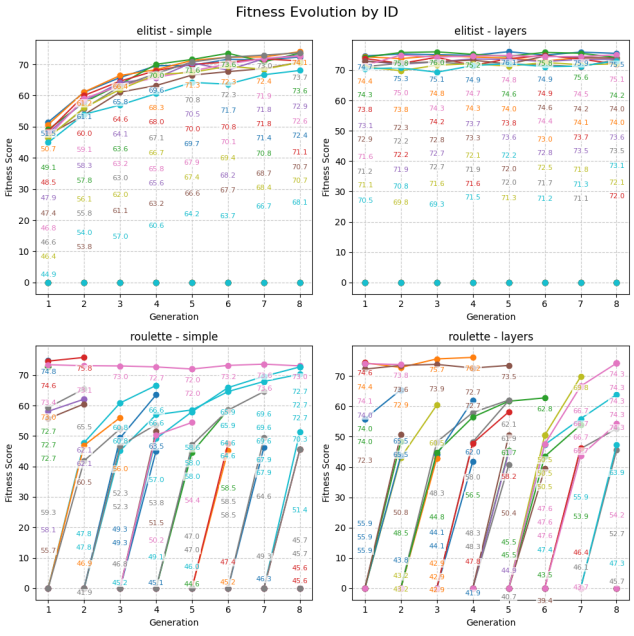
Figure 10: Distribution of Fitness Values



When we look at the distribution of non-zero fitness values for CIFAR-10, we see that even though there is a significant difference between the Elitist layers and Roulette simple methods, the top values are relatively close.

Next, let's examine the evolution of individual models:

Figure 11: Fitness Evolution

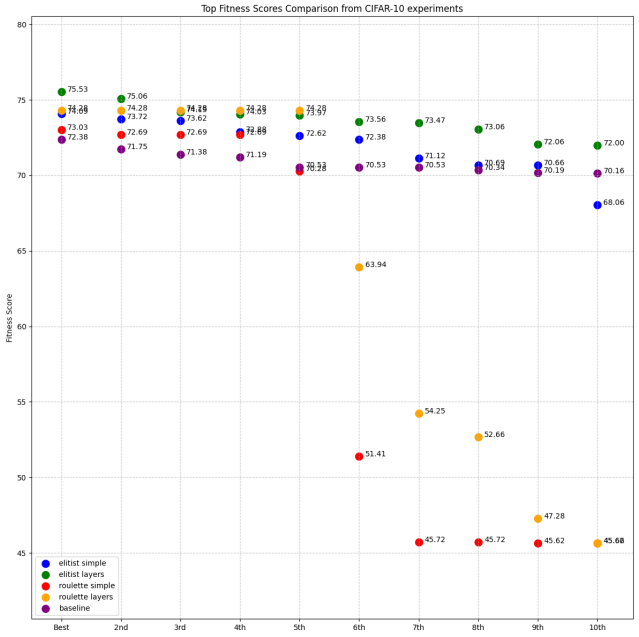


The results here are similar to what we had with MNIST, even though more pronounced in the case of Roulette selection for the models evoled with layer crossover. After 5th

generation, child models outperform parent models.

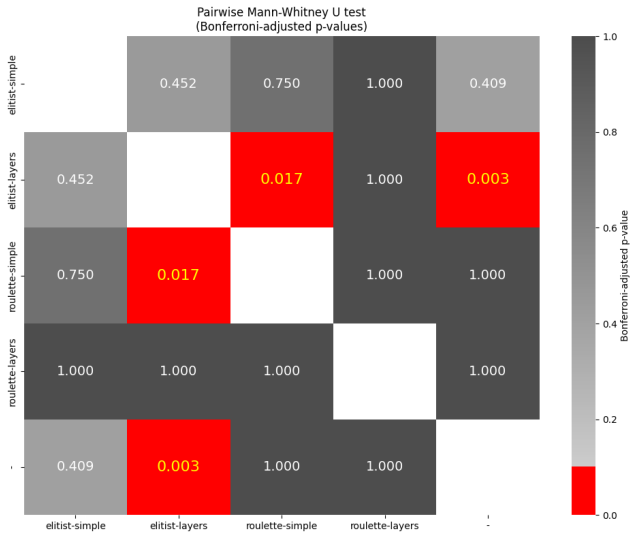
Comparison with EfficientNet-like architecture Architecture Performance In this experiment, a commonly used well-performing EfficientNet-like architecture architecture (Tan and Le 2019) have been trained through backpropagation.

Figure 12: Comparison: Evolved Models vs EfficientNet-like Models



Here again the differences were analyzed using the pairwise Mann-Whitney U tests with a Bonferroni correction:

Figure 13: Comparison: Evolved Models vs EfficientNet-like Models



The results are:

- The Elitist selection method with crossover performs better than both the Roulette method without crossover (simple crossover) and the baseline EfficientNet-like architecture..
- The differnece in performance between different evolutionary methods seems as pronounced for the four top-performing evolved architectures, even though it is not feasible to ascertain that statistically due to the small number of samples.

The most important result is that for CIFAR, the top-performing evolved models are performing better than the top-performing traditionally trained models, even though due to the small sample it is impossible to validate statistically.

Conclusion and Future Directions

The results of the conducted experiments show that for image recognition, neuroevolution can produce neural network architectures that perform at the same level or better than the architectures commonly accepted as standard. Specifically, the approach using an expanded version of crossover - where layers from two neural networks are exchanged to create a new architecture - combined with mutation, proved to be a feasible option.

These findings support neuroevolution as a promising area of research. Since evolving neural networks can be parallelized and may require fewer resources than traditional training methods, this approach offers a possible advantage.

The experiments also demonstrated the advantage of Elitist models over the traditionally trained ones. This seems insignificant for neuroevolution, as these models routinely cut out child models. However, this finding still can be important. The Elitist models train for a smaller number of epochs to achieve results that are better or comparable to those of traditionally trained models. This can be explained by the variety of starting points. Multiple models are generated, and out of them the best ones are selected to continue. This takes a relatively limited amount of resources. In contrast, with the traditional training we have only one architecture that is being trained; and in the case of exhaustive grid search - a great number of architectures whose training consumes a lot of resources before they are compared. Therefore, further research is recommended, as it has potential to propose a way to achieve good performance while saving resources.

The experiments also highlighted the advantage of the models evolved with the elitist selection mode over traditionally trained ones. While this might seem minor for neuroevolution, as elitist models usually discard child models, it is nonetheless significant. The elitist models required fewer training epochs to achieve results that were comparable or superior to those of traditionally trained models, likely due to the diversity of starting points. Multiple models are generated, from which the best are selected to continue - an efficient approach compared to traditional training, where either one architecture is trained or exhaustive grid search is conducted, consuming considerable resources. Further research could identify ways to achieve competitive perfor-

mance with models evolved through elitist selection while optimizing resource use.

Several directions for future research can be suggested.

This study was limited by the resources available, and repeating it with more processing power, a wider search space, larger populations of models, and additional generations could yield more insightful results.

Moreover, this research focused solely on CNNs. Expanding the scope to evaluate the benefits of neuroevolution in other domains, such as text processing and content generation, could be valuable. Specifically, it would be interesting to explore whether evolving models can improve performance on tasks where large language models (LLMs) often struggle, such as solving novel challenges. The Abstraction and Reasoning Challenge (Kaggle n.d.) could be considered as a test bed for this exploration.

Finally, neuroevolution can be useful in the search for Artificial General Intelligence. The only general intelligence we know of, the one demonstrated by humans and many other animals, has been developed through evolution. Combining the methods I explored here with those suggested to mimic neoplasticity in neural network models (Pontes-Filho et al. 2022) can be an interesting direction for research.

Finally, neuroevolution may hold promise in the pursuit of Artificial General Intelligence. The only form of general intelligence we know - exhibited by humans and many other animals - arose through evolution. Combining the methods explored here with approaches that mimic neuroplasticity in neural networks (Pontes-Filho et al. 2022) represents an interesting direction for further research.

References

Chen, A.; Dohan, D.; and So, D. 2023. EvoPrompting: Language Models for Code-Level Neural Architecture Search. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Galván, E.; and Mooney, P. 2021. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6): 476–493.

Greydanus, S.; and Kobak, D. 2024. Scaling Down Deep Learning with MNIST-1D. arXiv:2011.14439.

Kaggle. n.d. Abstraction and Reasoning Challenge. Retrieved November 9, 2024.

Lange, R.; Tang, Y.; and Tian, Y. 2023. NeuroEvoBench: Benchmarking Evolutionary Optimizers for Deep Learning Applications. In *Advances in Neural Information Processing Systems*.

Lecun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324.

Lipowski, A.; and Lipowska, D. 2012. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6): 2193–2196.

Liu, H.; Brock, A.; Simonyan, K.; and Le, Q. 2020. Evolving Normalization-Activation Layers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Pontes-Filho, S.; Olsen, K.; Yazidi, A.; Riegler, M. A.; Halvorsen, P.; and Nichele, S. 2022. Towards the Neuroevolution of Low-level artificial general intelligence. *Frontiers in Robotics and AI*, 9: 1007547.

Raventós, A.; Paul, M.; Chen, F.; and Ganguli, S. 2023. Pretraining task diversity and the emergence of non-Bayesian in-context learning for regression. arXiv:2306.15063.

Tan, M.; and Le, Q. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 6105–6114. PMLR.

Veličković, P.; Badia, A. P.; Budden, D.; Pascanu, R.; Banino, A.; Dashevskiy, M.; Hadsell, R.; and Blundell, C. 2022. The CLRS Algorithmic Reasoning Benchmark. arXiv:2205.15659.