Applied Soft Computing 147 (2023) 110767

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Neuroevolution with box mutation: An adaptive and modular framework for evolving deep neural networks

Frederico J.J.B. Santos [a], Ivo Gonçalves [b], Mauro Castelli [a,*]

[a] NOVA Information Management School (NOVA IMS) Universidade Nova de Lisboa, Campus de Campolide, 1070-312, Lisboa, Portugal
[b] University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, Portugal

ARTICLE INFO

Article history:
Received 26 April 2023
Received in revised form 20 July 2023
Accepted 14 August 2023
Available online 24 August 2023

Keywords:
Neuroevolution
Evolutionary deep learning
Neural architecture search
Supervised learning

ABSTRACT

The pursuit of self-evolving neural networks has driven the emerging field of Evolutionary Deep Learning, which combines the strengths of Deep Learning and Evolutionary Computation. This work presents a novel method for evolving deep neural networks by adapting the principles of Geometric Semantic Genetic Programming, a subfield of Genetic Programming, and Semantic Learning Machine. Our approach integrates evolution seamlessly through natural selection with the optimization power of backpropagation in deep learning, enabling the incremental growth of neural networks' neurons across generations. By evolving neural networks that achieve nearly 89% accuracy on the CIFAR-10 dataset with relatively few parameters, our method demonstrates remarkable efficiency, evolving in GPU minutes compared to the field standard of GPU days.

1. Introduction

Evolutionary Deep Learning (EDL) is a growing field in which various evolutionary computation algorithms demonstrate success when applied to neuroevolution. Neuroevolution encompasses the development of neural networks through evolutionary processes. It primarily focuses on two aspects: Neural Architecture Search (NAS), which seeks the optimal topology for a specific task, and parameter optimization, where evolutionary algorithms optimize parameters of a fixed topology.

NAS has been employed to mutate neural network architectures by initially designing and training them using backpropagation algorithms. However, the performance of NAS algorithms often lags behind traditionally-built neural networks [1,2]. Genetic Programming (GP) [3], an evolutionary technique that mimics natural selection for optimization tasks, has been effectively used in NAS. GP evolves populations of computer programs to solve problems objectively, introducing random variations in offspring structure through genetic operators.

Geometric Semantic Genetic Programming (GSGP) [4] is a specific adaptation of GP. GSGP evolves individuals based on their semantic space (output space) rather than syntactic representation, enabling small syntax structure variations while considering their impact on the results. This paper aims to combine Deep Learning (DL) and GSGP to contribute to bridging the gap between genetically mutated and human-designed neural network architectures. We develop a neuroevolution framework, written in Python and built upon PyTorch, that mutates neural networks whilst using traditional backpropagation techniques for parameter optimization. This technique contrasts with the traditional NAS methodology of randomly evolving numerous topologies and training them over hundreds of iterations. Our method is not only less computationally demanding but also significantly faster. We test the proposed framework considering the ResNet [2] and DenseNet [5] architectures to compare the performance between training complete architectures traditionally and evolving them by progressively adding new layers of neurons until the given architecture is completed. Additionally, we conduct multiple neuroevolution experiments with different hyperparameters to evaluate the framework and its outputs, concentrating on the generalization capability of the generated neural networks. We demonstrate that our approach can evolve and train a neural network on the CIFAR-10 dataset, achieving nearly 89% accuracy in under 30 GPU-minutes, compared to AmoebaNet [6], which attains 96.6% accuracy but requires 3150 GPU-days. The paper is organized as follows: Section 2 recalls relevant concepts exploited in the definition of the proposed neuroevolution method and outlines recent contributions in this area; Section 3 details the neuroevolution method and details the evolutionary model; Section; Section 4 presents the experimental settings, while Section 5 describes the results achieved. Finally, Section 6 concludes the paper and suggests possible future research avenues.

* Corresponding author.
E-mail addresses: m20200604@novaims.unl.pt (F.J.J.B. Santos), icpg@dei.uc.pt (I. Gonçalves), mcastelli@novaims.unl.pt (M. Castelli).

https://doi.org/10.1016/j.asoc.2023.110767

2. Related work

2.1. Deep learning

Deep Learning (DL) is a subfield of machine learning that specializes in developing and applying artificial neural networks (ANNs) to address complex problems across various domains. ANNs are computational models inspired by the human brain's neural structure, consisting of multiple interconnected layers of artificial neurons or nodes. These networks have garnered significant attention due to their capacity to automatically learn hierarchical feature representations from raw data, which results in exceptional performance in a wide range of tasks, including image recognition, natural language processing, and speech recognition.

The foundations of deep learning can be traced back to early works such as LeCun et al.'s Convolutional Neural Networks (CNNs) [7]. Key breakthroughs, such as the development of the Rectified Linear Unit (ReLU) activation function [8] and the introduction of dropout regularization [9], have played a crucial role in deep learning's success across various applications. These pioneering contributions paved the way for rapid advancements in neural network architectures and optimization techniques, which have shaped the current landscape of deep learning.

Recent advancements in neural network architectures, including Residual Networks [2] (ResNets) and Transformer models [10], have further broadened the capabilities and potential applications of deep learning. Given the close relationship between deep learning and neuroevolution, understanding the fundamental concepts and advancements in deep learning is essential for exploring the potential of neuroevolutionary techniques in the context of neural networks and their optimization.

Feedforward neural networks, consist of an input layer, one or more hidden layers, and an output layer. Each layer contains multiple neurons that are fully connected to the neurons in the previous and following layers. The neurons in the input layer receive the raw input data, which is then passed through the network to produce an output. The hidden layers contain non-linear activation functions, such as the sigmoid or ReLU function, that allow the network to model complex relationships in the data.

CNNs are a type of deep learning architecture specifically designed for image and video recognition tasks. The architecture consists of multiple convolutional, pooling, and fully connected layers. The convolutional layers apply a set of filters to the input image, extracting features such as edges, corners, and textures. The pooling layers downsample the feature maps produced by the convolutional layers, reducing the dimensionality of the data and allowing the network to learn features. The fully connected layers at the end of the network use the extracted features to classify the input image. CNNs have been shown to be highly effective for a wide range of image and video recognition tasks, including image classification, object detection, and segmentation.

2.2. Genetic programming

Genetic Programming is a set of techniques used to evolve computer programs, first introduced by Koza in 1992. It belongs to the field of Evolutionary Computation (EC), which includes other methods such as Genetic Algorithms [11], Evolutionary Programming [12], and Evolution Strategies [13]. There are different types of representations in GP, but this work only focuses on the tree-based GP, as originally defined by Koza.

Programs are represented as trees, where nodes represent functions and leaves represent constants. The nodes form the function set, which are the operations that may exist in each program. The leaves form the terminal set, which are the possible values that may appear in the program. These two sets are problem-specific and must be handcrafted for each problem.

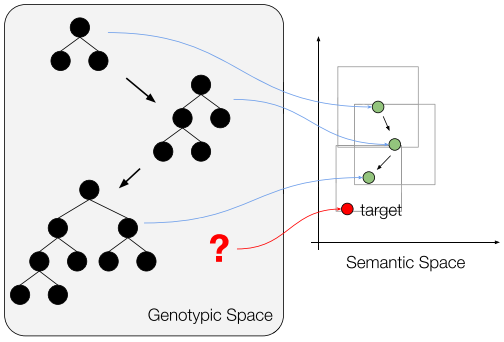

Fig. 1. Execution of multiple GSM steps considering a simple example with only two training samples. We may notice the structure changing in the genotypic space whilst the semantics are bound inside gray boxes, exemplifying the semantic awareness of the GSM operator.

The GP process starts by generating a random initial population of individuals, where one individual represents a program. The process then continues until a given stopping criterion is reached, such as a certain number of generations or a specific amount of time. Each individual's fitness is measured by running the program and evaluating its performance on some problem tasks. A new population is then created by applying one of the following operations to each individual: parent selection, reproduction, crossover, or mutation.

In parent selection, a set of individuals is selected based on their fitness to be reproduced with certain probabilities. Reproduction simply copies some of the selected individuals into the new population without modification. Crossover generates new individuals by combining random parts of two individuals. Mutation generates new individuals by replacing a random part of an individual with a new randomly generated one. The individual with the best fitness is selected as the solution to the problem set.

2.2.1. Geometric semantic Genetic Programming

In Genetic Programming (GP), the semantics of an individual can be represented as a vector of output values on input data in a semantic space S. The target vector can also be represented in the same semantic space S. The distance between an individual and the target vector can be measured to determine the error between the individual's semantics and the ground truth. However, genetic operators in GP only operate on the genotypic (or syntactic) space, searching for a structure that minimizes the error between an individual's semantics and the target vector. This aspect means that the offspring of an individual can be significantly different from its parent in terms of semantics.

To address this limitation, GSGP was introduced by Moraglio [4], which uses Geometric Semantic Operators such as Geometric Semantic Mutation (GSM) and Geometric Semantic Crossover (GSC) that operate on both the genotypic and semantic spaces. This work only considers GSM since the focus is solely on mutation operations. GSM enforces a "box mutation", i.e., it restricts the mutation to a small perturbation of the parent individual, delimited by a mutation step (ms). In other words, the mutation does not introduce drastic changes to the individual's structure, but only slightly modifies it, ensuring the search space is confined around the parent's structure (see Fig. 1). It is worth noting that GSM has only been applied to supervised regression tasks, with recent extensions to binary classification tasks [14,15].

2.3. Review of neuroevolution methods

Despite the competitive results achieved with DL, recent years have witnessed a greater interest in the definition of methods to automatically design neural networks' topologies. This interest is motivated by the fact that the success of deep neural networks on a given task depends significantly on the design of their architectures. Recent works can be categorized into two main groups: evolution-based methods [16] also known as neuroevolution [17], and reinforcement learning (RL) methods [18]. In addition to these categories, other alternative approaches have been proposed, hill-climbing [19], Bayesian optimization [20], Monte Carlo-based simulations [21], and a combination of these approaches [22]. Considering that neuroevolution-based methods are the most investigated ones [23] and taking into account that the method proposed in this study belongs to this area, this section reviews the most recent neuroevolution techniques proposed in the literature, aiming at highlighting relevant differences with respect to the method proposed in this paper. For a review of the different approaches to perform neural architecture search, the interested reader is referred to [24] where swarm and evolutionary computing approaches for deep learning are discussed, and the work of Wistube [25], in which different approaches, including Bayesian methods, are analyzed.

Neuroevolution is a subfield of evolutionary computation that specifically focuses on optimizing ANNs through evolutionary algorithms. This approach is rooted in the principles of natural selection and genetic variation, with the aim of optimizing neural network architectures and their parameters to solve complex tasks. The general approach of neuroevolution methods can be summarized by considering the following steps: (1) the initial step consists of defining the search space and initializing a population of admissible solutions. Each solution encodes a neural network architecture; (2) once the fitness evaluation of the initial population is complete, the entire population initiates the evolutionary process. Throughout the evolutionary process, selection and evolutionary operators are used to create new solutions; (3) once the termination condition is met, the best-performing solution (i.e., neural network) is returned.

Thus, to analyze existing evolutionary approaches for NAS, it is fundamental to analyze three main ingredients: the encoding strategy, the initialization strategy, and the encoding space. Concerning the encoding strategy, the evolutionary process may consider fixed-length encoding strategies (like in genetic algorithms) and variable-length encoding strategies (like in genetic programming). In the first case, the main advantage relies on the possibility of evolving the network topology by using the genetic operators originally designed for individuals presenting the same length, as proposed in the work of Xie and Yuille [26] and Loni and coauthors [27]. However, the main limitation of fixed-length strategies is the experience necessary to properly choose the length of the candidate solutions, a parameter that has a strong impact on the performance of the resulting neural network. This issue can be addressed by relying on variable-length encoding, thanks to the possibility of freely evolving the network's topology without imposing any limit on the length of the encoding. On the other hand, with this encoding, it is often necessary to redefine the genetic operators, as proposed in the work of Sun and coauthors [28]. From the analysis of the existing contributions, it emerges that genetic algorithms represent the most common choice, especially when considering early neuroevolution contributions [29,30]. Genetic programming [31,32] and evolutionary strategies [33,34] are used in a limited number of contributions, especially due to the computational time they usually required. On the other hand, the possibility of relying on a non-fixed length for the encoding of the solutions provides significant advantages in most of the problems and especially when no information is available to decide the length of the encoded solutions (and, as a consequence, the number of layers and neurons of the topology). A small number of neuroevolution methods rely on swarm intelligence. For instance, Byla and Pang [35] proposed DeepSwarm, a NAS method based on Ant Colony Optimization to generate an ant population that uses the pheromone information to collectively search for the best neural architecture. Junior and Yen [36] defined a NAS method based on swarm optimization to design a topology for an image classification task. Finally, some contributions proposed the use of memetic algorithms for the NAS problem [37,38] but the good-quality results are achieved at a cost of an increased computational time, thus making these methods suitable only for simple problems.

Moving to the initialization strategies, and similarly to other applications of evolutionary computation methods, it is possible to identify three main procedures: a simple strategy, a random strategy, and a strategy that considers some existing knowledge. The simple strategy consists of an initialization that only considers a few primitive layers. For instance, Real and coauthors [39] relied on this method for building initial individuals representing a single-layer architecture. Despite its simplicity, Xie and coauthors [26] showed that this simple initialization can result in an evolutionary process in which well-performing architectures are obtained. Concerning the random initialization, the initial individuals are randomly generated using different primitive layers, thus resulting in a population representing different types of architectures and with different shapes [28]. Finally, the last initialization method takes advantage of the existing knowledge concerning state-of-the-art architectures and includes them in the original population. While this initialization method can make it extremely difficult to evolve novel competitive architectures with respect to the ones in the initial population, it is commonly used in all the works aiming at refining the performance of state-of-the-art architectures [40].

To conclude the discussion concerning the fundamental differences in existing neuroevolution methods, the remaining part of this section describes the different encoding space strategies commonly used in the literature.

The population's encoding space encompasses all the legitimate individuals that have been encoded and can be categorized into three distinct types based on the units they utilize: layer-based, block-based, and cell-based encoding spaces [23]. Additionally, certain evolutionary NAS methods prioritize the connections between units rather than the configuration of the basic unit itself. This specific encoding space is referred to as the topology-based encoding space [41].

The layer-based encoding refers to the use of primitive layers, such as convolution layers and fully-connected layers, as the fundamental building blocks. While this approach results in an extensive search space, it requires a significant computational time to search for a promising individual since there are numerous possibilities for constructing a well-performing neural network using these primitive layers. Moreover, relying solely on primitive layers may not yield the desired performance since primitive layers alone cannot adequately represent skip connections, a fundamental component for achieving satisfactory performance with some neural network topologies (i.e., like ResNet [2]). Examples of methods relying on layer-based encoding have been proposed by Tanaka [42] and Zhu [43]. In the former work, the authors automated the design of a recurrent neural network by relying on the covariance matrix adaptation-evolution strategy (CMA-ES) [44] and found a topology that gives improved recognition performance on a speech recognition problem. In the latter work, the authors proposed an evolutionary-based method that by relying on specifically-designed genetic operators achieved competitive results on a computer vision task.

To limit the issues associated with the layer-based encoding approach, block-based encoding allows combining different types of blocks (i.e., like ResBlock [2], DenseBlock [5], InceptionBlock [45], etc.) that serve as the basic unit of the encoding space. More specifically, the blocks within the architecture exhibit distinctive topological relationships, such as the residual connection in ResBlock. These blocks demonstrate promising performance and often require fewer parameters for constructing the architecture. Consequently, it is generally simpler to discover a suitable architecture within the block-based encoding space compared to the layer-based encoding space. Several works relying on the block-based encoding space have been proposed: for instance, Baldeon-Calisto and coauthors [46] proposed the use of multi-objective evolutionary algorithms to design a convolutional neural network (AdaResU-Net) that combines the structure of the state-of-the-art U-Net [47] with a residual learning framework to improve information propagation and promote an efficient training. Experimental results showed that the resulting topology achieves excellent segmentation performance while presenting a significative reduction in terms of the number of trainable parameters. Suganuma and coauthors [48] also relied on the block-based encoding space to evolve the topology of a CNN. In particular, highly functional modules such as a convolutional block and tensor concatenation are encoded as the node functions in cartesian genetic programming. The resulting CNN achieved results comparable with the state-of-the-art on the CIFAR 10 and CIFAR 100 datasets. Hassanzadeh and coauthors [49] proposed a genetic algorithms-based method to evolve a small neural network for a medical image segmentation problem. The block-based encoding space was considered to evolve a U-Net-based deep network topology. The resulting network showed excellent segmentation performance by also using less trainable parameters with respect to existing human-designed networks.

The cell-based encoding space is a particular case of block-based encoding. The cell-based encoding allows combining the layers in the cell more freely while the connections between different cells are determined by the human expertise [50]. Chen and coauthors [51] used cell-based encoding to design the topology of a CNN for a computer vision classification task. Despite the reduction in terms of computational time when compared to existing architectures, the authors needed 12 GPU hours for evolving a topology to classify CIFAR-10 images. The cell-based encoding space is considered also in the work of Fan and coauthors [33], which optimized an encoder–decoder architecture for retinal vessel segmentation. The evolved topology achieves top performance among all compared methods on the three datasets by also using fewer trainable parameters compared to existing competitors. Despite its popularity, Frachon and coauthors [52] pointed out the lack of theoretical basis to guarantee that methods relying on the cell-based encoding space can achieve a good-performing topology.

2.3.1. Open issues identified

From the analysis of recent contributions in the field of neuroevolution, it is possible to draw the following observations:

- Despite the plethora of existing studies, concerns have been raised regarding the sluggish learning capabilities and computational costs associated with evaluating evolutionary algorithms when utilized for designing deep neural network architectures [53].
- Due to the vast search space, achieving outstanding performance through network structure evolution becomes difficult when relying on evolutionary principles that have been defined decades ago and that were not specifically designed for designing deep neural networks. Moreover, the vast majority of the existing works on neuroevolution do not consider the fitness landscape properties when defining a NAS algorithm [30].
- Existing methods, while based on gradient-free optimization, need to evaluate the quality of the evolved network at each iteration through a fitness function. Calculating the fitness function requires the execution of the backpropagation algorithm, thus turning the evolution unbearably slow. In this sense, methods allowing for an incremental evaluation of the evolved topologies are fundamental [54].
- The choice of the encoding may hinder the potential of the evolutionary process. In this sense, it is important to design a mechanism that allows for an efficient and effective exploration of the search space.

To tackle these challenges, we propose a neuroevolution method that:

- allows for the quick evolution of a neural network topology;
- it is based on recently defined genetic operators for genetic programming that allow inducing a unimodal search space and that have been adapted to efficiently explore the search space of the neural networks' topologies;
- allows performing an incremental evaluation of the evolved topology, thus reducing the computational time and making it possible to execute it on a consumer laptop.
- is based on a simple encoding that, by exploiting theoretical properties of the search space, makes it possible to navigate a search space characterized by the absence of local (non-global) optimal solutions.

Before concluding the review of existing works on neuroevolution algorithms, it is important to highlight that Section 2.3 aimed at presenting the most common approaches to deal with the automatic evolution of neural networks' topologies. However, the field of neuroevolution is witnessing an increasing number of contributions in recent years, and it is out of the scope of this section to analyze all the existing works. Thus, the interested reader can refer to recent works presenting surveys on NAS from different perspectives. White and coauthors [55] highlighted that from 2020 to the end of 2022, over 1000 papers on neuroevolution have been published. Thus, their work proposed a taxonomy of the existing methods by considering the search spaces, algorithms, best practices, and open issues in this area. Li and coauthors [54] reviewed evolutionary deep learning approaches from the perspective of automated machine learning (AutoML). In particular, following a new taxonomy, they considered methods from data preparation and model generation to model deployment. Finally, they discussed applications and open issues in the area of neuroevolution. Zhan and coauthors [56] analyzed the existing work of neuroevolution by proposing a two-level taxonomy. The higher level includes four categories based on when evolutionary computation can be adopted in optimizing deep neural networks (i.e., data processing, model search, model training, and model evaluation and utilization). At the lower level, authors reviewed existing works in each category by analyzing how specific evolutionary computation methods have been used. Finally, Li and coauthors [57] reviewed more than 500 contributions in evolutionary machine learning by also focusing on neuroevolution. The authors analyzed the current limitations and discussed future research directions.

2.4. Semantic learning machine

The Semantic Learning Machine (SLM) [58,59] is a neuroevolution algorithm that does not use backpropagation to optimize parameters. It employs GSM, is based on GSGP principles, and suits regression problems. The SLM mutation adds a new random neural network to the parent network with randomly initialized parameters. However, there are constraints in the mutation process. The random network can only receive connections from the

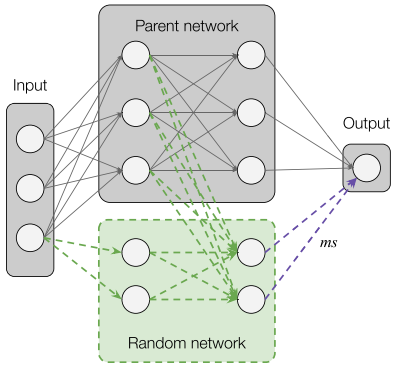

Fig. 2. Diagram of a random mutation. The random network receives the initial input for the first layer and both random and parent networks' first layers' output for the second layer. New connections are dashed green lines, while dashed purple connections show connections affected by the mutation step ms.

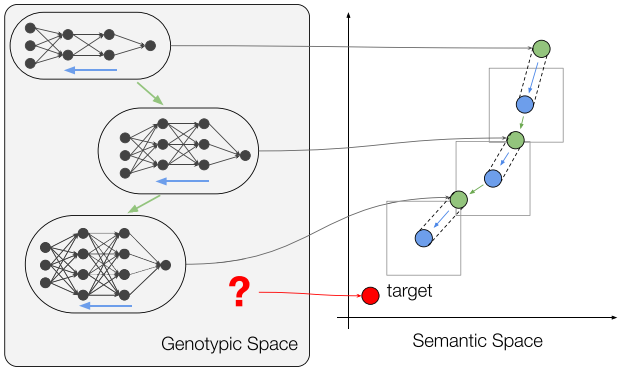

Fig. 3. Representation of model evolution using our method, illustrated in both genotypic and semantic spaces. Green arrows and points signify syntactic mutations and pre-optimization semantics, respectively, while blue arrows and points denote the application of the backpropagation algorithm and post-optimization semantics. Within the semantic space, a box is observed constraining the mutation options, effectively demonstrating the controlled evolution of our approach.

parent network and not feed back into it to avoid altering the parent network semantics. The parent network's output layer takes the random network's last hidden layer as input. The weights connecting the random network to the output layer are restricted by the mutation step ms, enforcing a "box mutation" similar to the GSM (see Fig. 2).

3. Methodology

In this section, we detail the neuroevolution method in Section 3.1, then go into detail about the structure of the evolutionary model in Section 3.2.

3.1. Neuroevolution

Our approach to neuroevolution is inspired by GSGP and focuses on the mutation genetic operator. In this mutation process, the framework progressively adds new parameters (artificial neurons) to the neural network and trains them on the problem dataset. Subsequently, it compares the performance of the mutated individual to that of the parent individual (the non-mutated neural network) in the validation dataset.

Our work adapts the Semantic Learning Machine (SLM) concept to Deep Learning but with key differences. GSGP and SLM were originally designed for regression problems, whereas our method focuses on classification tasks. We experimentally adapt the concept of GSGP for classification tasks, recognizing that the randomly added network cannot optimize the residual error between the parent network and the target vector, as there are no real distances to be measured.

Our method evolves the topology of models and optimizes the parameters directly with the backpropagation algorithm, i.e., it only considers the topology when representing the model in the genotypic space. We contrast this representation to SLM, which takes both topology and parameters into its genotypic space. We benefit from the best of both worlds, as the backpropagation algorithm constantly optimizes the model's parameters towards our target, while the evolution randomly (but in a controlled fashion) mutates the topology of the model to lead its semantics towards the target vector (Fig. 3).

Backpropagation Optimization. With regard to backpropagation optimization, our focus is on fast convergence to minimize the number of epochs needed per mutation. Our approach is guided by Smith's recommendations [60] on optimizers, learning rate, batch size, momentum, and weight decay. Preference is given to Rectified Adam [61] as it is ideal for fast convergence [62] and has fewer hyperparameters to be tuned, making it a more suitable choice than Adam [63] for an environment where fine-tuning is not achievable.

As the model's architecture changes with each mutation and considering that the learning rate (η) is an important hyperparameter to achieve a constant and stable learning process, we decided to implement a Learning Rate Finder [64] (LRF) that tests multiple values within the range $[\eta_{min}, \eta_{max}]$ and selects η^*, which corresponds to the steepest negative gradient.

Evolution Strategy. The trainer is the system responsible for mutating, fitting, and evaluating the individuals (models). In each generation, the trainer decides which mutation to apply and fits the individual to the training data for a fixed number of epochs. The individual is then evaluated and if it performs better than the elite individual (the best individual found so far), the elite individual is replaced. Then, the parent is reloaded, and the process is repeated for p individuals and g generations. The fitness function is either the loss function or accuracy for the validation data multiplied by a threshold. This threshold is a hyperparameter and was introduced to prevent very small improvements from leading to large models with little improvement in performance (see Fig. 4).

The trainer has three main options for mutations: adding a new block, a new layer, or new connections. However, to streamline model evolution and simplify the mutation process, our experiments always mutate by adding a new block. Instead of adding new layers as separate mutations, a random number of layers are added when introducing a new block. This decision was made to take "larger" steps when searching for the optimal neural architecture. The pseudocode in Algorithm 1 summarizes the steps of the entire process. The subsequent section provides details on each component used to construct a topology (i.e., blocks and layers)

3.2. Evolutionary model

The model's structure comprises interconnected blocks (detailed in Section 3.2.1) and a *Merge Block*. A block is defined as a module containing a number of sequentially connected layers whereas the *Merge Block* is the module that combines the outputs of all the blocks. Each new mutation adds a new block to the model, increasing its complexity. Following the concept of GSGP and GSM, the parent network semantics must remain unchanged when a new mutation adds a new block, such that each new block never feeds back into the parent network (Fig. 5). We set a

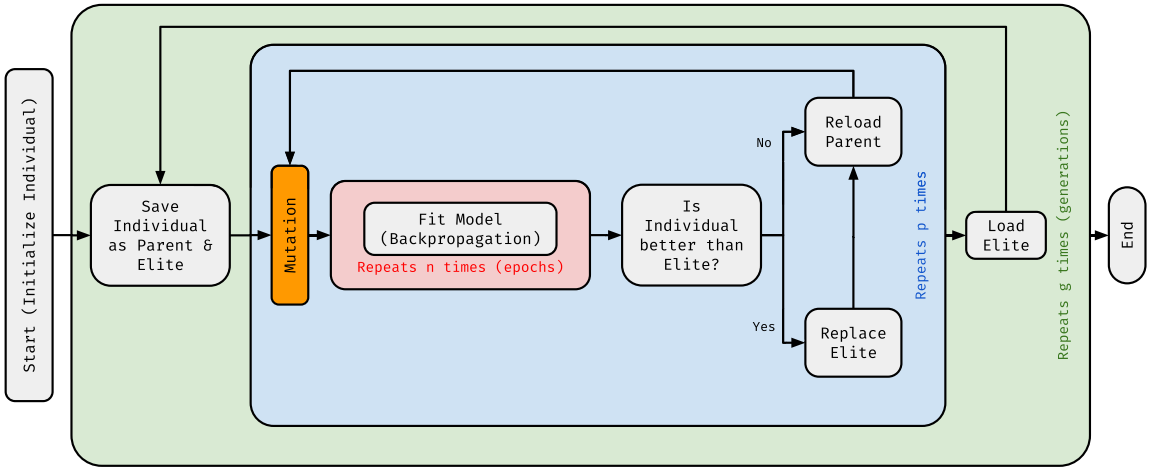

Fig. 4. Neuroevolution flowchart. The red process represents the fitting loop of n epochs. The blue process represents the mutation loop that runs for p times. The green process represents the generation loop, which repeats g times.

Algorithm 1: The proposed neuroevolutionary process. Concerning the time complexity of the algorithm, when we evolve the network's topology without freezing the existing blocks, the complexity, at each epoch, corresponds to the complexity resulting from the execution of the backpropagation algorithm. On the other hand, when we freeze the existing blocks, while the upper bound for the time complexity still corresponds to the execution of the backpropagation algorithm, the time needed to evolve the topology will be smaller due to the fact that we must execute backpropagation only on the added block and not on the entire network.

Require: Number of generations (g), population size (p), connection weights (CW), performance threshold (t), number of training epochs (n), Learning Rate (η), Use Learn Rate Finder (LRF, $bool$)

```
Parent ← Initialize empty model
Elite ← Parent
for i in 1 to g do
  for j in 1 to p do
    Individual ← Parent
    Randomly choose block type (linear or convolutional)
    Randomly choose block connection based on CW
    Randomly choose block configuration and number of
    layers for new block
    Randomly select parameters for each layer
    Mutate Individual – Add new block with chosen
    options
    if LRF = True then
      η ← LearningRateFinder(Individual)
    end if
    Train – Fit Individual with backpropagation for n
    epochs
    Compute the top-1 accuracy of Individual on the
    validation dataset
    if Individual_Acc > Elite_Acc * t then
      Elite ← Individual
    end if
  end for
  Parent ← Elite
end for
return Elite
```

hyperparameter (Freeze Evolved Blocks, *FEB*) to enable or disable the optimization of the parent network when optimizing the parameters of a mutated model, i.e., the model either trains all its parameters or only the parameters of the newly added block. We experimentally find that the decision to freeze (not optimize) or not freeze (optimize) the parameters of the parent network plays a significant role in how well the models evolve.

3.2.1. Blocks

A block is defined as a module containing a number of sequentially connected layers. The last layer of the block is a specific kind of layer which we define as the *Output Layer*, and its output is always a vector with the same shape as the model's output. We go into detail about Layers and the *Output Layer* in Section 3.2.2.

The first layer of a block receives a single input, which can originate either directly from the problem dataset or any other layer (except for an *Output Layer*) within previously added blocks. This input may be user-defined (in cases where the user manually mutates the model) or randomly chosen based on specified probabilities (detailed below). The input is then fed forward sequentially through the layers and into the *Output Layer*, which connects to the *Merge Block*. The output of each layer is stored in the *Block Inputs* to be used as potential input for future blocks. We implement two block types: Linear and Convolutional (see Fig. 6).

Linear Block. This block can receive either singular or multi-dimensional inputs. If the input is multi-dimensional, it is transformed into one dimension (by flattening the tensor). A layer in this block consists of a sparsely connected feedforward module with a user-defined number of nodes, a regularization module, and an activation function. The *Output Layer* is a simple feedforward module that connects the block to the *Merge Block*.

Convolutional Block. This type of block receives multi-dimensional inputs by default. A layer in this block includes a two-dimensional convolutional module, followed by a regularization module and an activation function. The *Output Layer* has a pooling module followed by a flattening module, that converts the input into a one-dimensional vector, followed by a feedforward module that, as with the Linear Block, connects the block to the *Merge Block*.

The convolutional block can have a residual (skip) connection [2], where a downsample layer transforms the input shape to match the output shape of the block's last layer. This downsample layer consists of a convolutional module with a kernel size of (1,1), no padding, a regularization module, and no activation function. The output of the downsample layer is then added to the output of the last layer before applying the activation function.

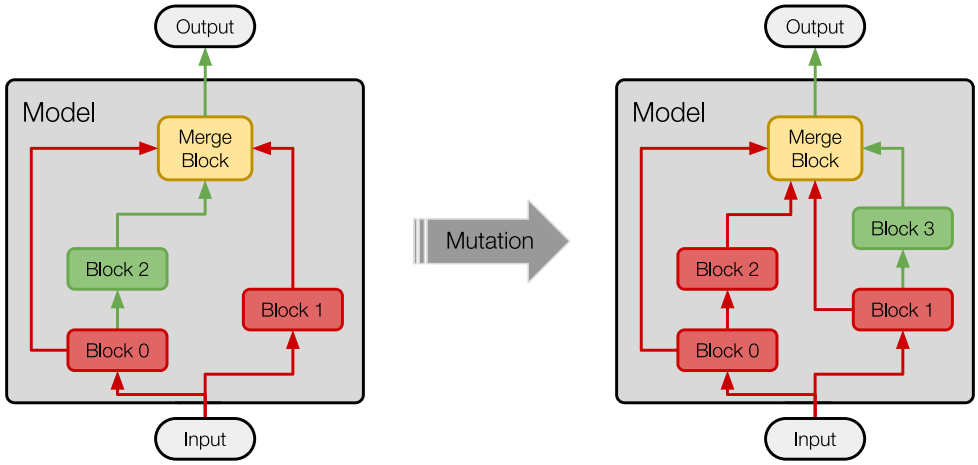

Fig. 5. The inner structure of an evolutionary model composed of blocks and the *Merge Block* (yellow). Red-colored blocks and arrows represent frozen parameters, while green ones represent unfrozen parameters. The model goes through a mutation that adds Block 3, freezing the parameters of Block 2 in the process. When the model is optimized, only the parameters of Block 3 are trainable.

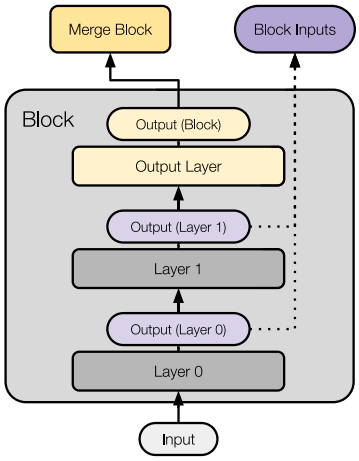

Fig. 6. An example of the inner structure of a block with two layers is illustrated here, showing the flow of storing the output of each layer in the *Block Inputs*. For the sake of simplicity, we omit this storage system in our figures going forward.

Furthermore, an alternative approach inspired by DenseNet [5] can be employed, which involves the concatenation of the input with the output of the last layer before feeding it into the *Output Layer*. This concatenation mechanism enables the network to retain detailed information from previous layers, promoting the integration of both low-level and high-level representations. It is important to note that to ensure compatibility for concatenation, the input and output feature maps must have the same width and height. During the evolutionary process, if a convolutional block does not maintain the same feature dimensionality, the concatenation operation is not applied.

The following hyperparameters are defined when a block is created; every layer created inside the block will share these hyperparameters by default. Bias is enabled for all layers if it is enabled for the block. Weight initialization may be uniformly or normally distributed, either using the Xavier Initialization [65] or Kaiming Initialization [2]; by default, they are initialized with the Kaiming Normal initialization (default PyTorch values). Bias initialization can be performed with zeros [2,65] or with default (PyTorch) values, and by default, biases are initialized with a uniform distribution of $[-1/\sqrt{fan_{in}}, 1/\sqrt{fan_{in}}]$ where fan_{in} is the number of input features. Regularization and activation function modules, detailed further in Section 3.2.2, are defined at the block level so we can measure the impact of each type when evaluating evolved models.

Connecting Blocks. To decide which input the block receives, we consider factors such as the type of block and where it should connect in the model. We use a hyperparameter called connection weights (CW) to assign weights to each block based on its maturity; more recent blocks may be more or less likely to receive a connection, depending on the hyperparameter value.

For instance, a recently added block might be assigned a greater weight than an older block, implying a higher likelihood of forming a new connection. This allocation can also vary depending on the type of block, as we often observe that feed-forward layers usually appear at the end of convolutional neural network architectures. Therefore, in the case of linear blocks, we might want to adhere to traditional neural network architectures and restrict their connection possibilities.

In a simplified scenario, where connection weights are common to all block types, we denote $CW = [w_1, w_2, \ldots, w_m]$, with m being the maximum block 'age' we want to consider. These weights are assigned in a chronologically inverse order, with w_1 as the weight of the most recently added or 'youngest' block, w_2 as the weight of the subsequent block, and so forth. In scenarios where the network continues to grow beyond the size of the given CW array, the weight w_m is assigned to all blocks 'older' than the mth block. This system ensures that the weight allocation remains scalable as the network grows and incorporates new blocks.

These weights are then transformed into weighted probabilities (Fig. 7). Given the connection weights $[w_1, w_2]$ and k existing blocks, the probability of a new block connecting to the youngest existing block is computed as:

$$P_{\text{youngest}} = \frac{w_1}{w_1 + k * w_2}$$

Meanwhile, the probability of said new block connecting to any other block or the model's input is computed as:

$$P_{\text{other}} = \frac{w_2}{w_1 + k * w_2}$$

The probability of choosing a layer inside a block is equally divided between all layers of the block. For example, if a block has 5 layers, the probability of connecting to layer 3 is the same as that of connecting to layer 1 or layer 5. Each block is then given a connection index, two values representing the block and layer that it is connected to.

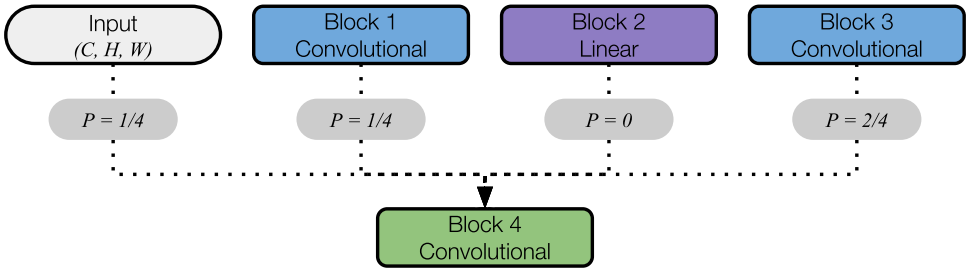

Fig. 7. Probabilities for connecting Block 4 to the rest of the model assuming the connection weights (CW) are initially [2, 1]. When considering Block 4, the connection weights are adjusted to become $CW_{Block4} = [2, 0, 1, 1]$. The weight of Block 2 has been set to 0 because, in our model, convolutional blocks cannot receive connections from linear blocks. Our framework reflects this constraint by zeroing out the corresponding weight.

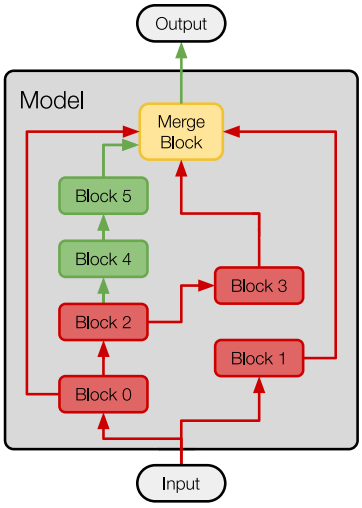

Fig. 8. Model with BB enabled. The most recently added Block 5 connects to Block 4, forming a branch together. Since Block 2 has more than one outgoing connection, it is not part of the branch. Both Block 2 and Block 4 do not connect to the Merge Block, as they both are part of block branches. Green and red represent unfrozen and frozen parameters, respectively.

To allow for more complex structure evolutions, we introduce the concept of **Block Branches** (BB): a group of interconnected blocks where only the last block connects to the model's output. BB enables the creation of a larger block structure that can be considered a single unit. A block is eligible to be part of a branch only if it connects to the most recently added block. The block branch terminates when a block has more than one output connection (Fig. 8).

When BB is enabled, the first outgoing connection of a block may only be established to its last layer because the block is disconnected from the *Merge Block*. Subsequent outgoing connections of said block can connect to any layer. The connection limitation is in place to preserve the concept of box mutation. The restriction ensures that the block that is disconnected from the *Merge Block* still contributes its entire structure to the model's output.

3.2.2. Layers

In this work, we define a layer as a composition of modules that preprocess the input, the neural network layer itself, and post-processing modules. A module can be a traditional fully connected linear neural network layer, activation function, or regularization layer. We use different types of modules, such as pooling, reshaping, main, regularization, and activation function modules. Pooling modules are used for feature extraction and keeping the block's parameters in check, while reshaping modules flatten or reshape input tensors as needed. The main module is a feedforward module with possible sparse connections in linear blocks or a two-dimensional convolutional module in convolutional blocks. Regularization modules like Batch Normalization, Instance Normalization, Layer Normalization, Dropout, or no regularization module are determined at the block level, affecting the model's generalization ability. Activation functions such as Rectified Linear Unit (ReLU) [8], Gaussian Error Linear Unit (GELU) [66], and Leaky ReLU [67] are chosen at the block level.

Output Layer. In a classification model, the last layer typically consists of a fully connected feedforward layer outputting scalars, which are then passed through a *softmax* function to obtain class probabilities. If our framework had a traditional last layer, it would receive inputs from all blocks, and its weights would either be all trainable or non-trainable. However, to preserve box mutation, all the weights of evolved blocks should remain unaltered, including the weights that connect to the output, while connections to the mutation block should undergo backpropagation optimization.

To achieve this *partial* last layer optimization, we break down the last linear layer into *Output Layers* and a *Merge Block* (Fig. 9). Each block has an *Output Layer* as its last layer, acting as the block's classifier and contributing to the model's output. All *Output Layers* are summed together at the *Merge Block*. The model hyperparameter, Freeze Evolved Output Layers (FEOL), allows evolved blocks' output layers to remain trainable (unfrozen) when new blocks are added, effectively functioning like Double Adaptive Mutation [68] and altering how existing blocks contribute to the model's output. This hyperparameter is not applicable if FEB is disabled, as FEB unfreezes all parameters in the model.

4. Experimental settings

In this section, we present the experimental settings and procedures that were employed to evaluate our proposed framework for evolving deep neural networks. The experiments conducted in this study are designed to assess the effectiveness of our method in various scenarios, specifically focusing on two main experiments: (1) evolving models to recreate the ResNet-18, Resnet-34 and DenseNet-121 architectures and measure their performance against the traditional method of training these architectures, and (2) freely letting the models evolve without architectural constraints.

In every experiment, we conducted 30 runs for each sub-experiment with a batch size of 256, calculating the mean, standard deviation, and best run. Experiments were run on GPUs, either using an RTX 3080 Ti or a GTX 1070. Due to differences in computational power between the cards used, we do not report on run time unless explicitly stated. For all figures in this section, the mean is represented by a line, and the standard deviation

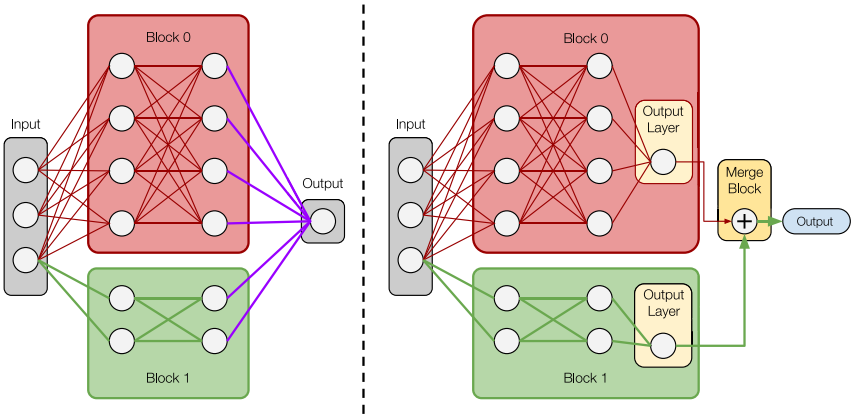

Fig. 9. Green represents unfrozen parameters; red represents frozen parameters. **Left.** Neural network with a traditional last layer to output the result. The purple lines represent the output weights, and they can either be all frozen or unfrozen. **Right.** By breaking down the output into *Output Layers* and *Merge Block*, we are effectively able to optimize only the weights of the *Output Layer* of Block 1, as represented by the green lines going to the *Merge Block*.

Table 1
Hyperparameters used to recreate the three architectures using the framework to establish the baselines, as well as the options for the evolution experiments. We only produce LRF comparisons for the ResNet-18 architecture due to computational constraints.

Model hyperparameters	Baselines	Evolution (ResNet-18)	Evolution (ResNet-34)	Evolution (DenseNet-121)
Block Branches (BB)	True	{True, False}	{True, False}	{True, False}
Epochs (total)	20	34	170	60
Epochs (per mutation)	–	2	10	10
Use Learning Rate Finder (LRF)	False	{True, False}	False	False
Learning Rate (η)	0.05	{True, False (0.05)}	0.05	0.05
Freeze Evolved Blocks (FEB)	False	{True, False}	{True, False}	{True, False}
Freeze Evolved Output Layers (FEOL)	False	{True, False}	{True, False}	{True, False}

is represented by the shaded area, averaged over thirty runs. Vertical semi-transparent dashed lines indicate when a model is mutated.

4.1. Exploring the performance of evolved architectures

We regarded the ResNet and DenseNet designs as an ideal testbed for the theory behind our method for evolving neural networks. If we were to evolve models to have the same topology as these architectural designs, how would they perform compared to the traditional training of the ResNet and DenseNet networks?

As a baseline for this experiment, we recreated the ResNet-18, ResNet-34, and DenseNet-121 architectures using the framework and trained them using standard backpropagation. We compared the baselines against forced mutations that evolve the model into these architectures. For these forced evolution sub-experiments, we tested various combinations of values (Table 1).

4.2. Unrestricted neuroevolution

In this experiment, we evolved models from scratch without any restrictions. After considering the results regarding LRF (Section 5.1.1), we opted not to use the LRF and instead employed a fixed learning rate ($\eta = 0.05$). Furthermore, we introduced the *Reset N Fit* hyperparameter, which reinitializes the model's parameters and retrains them from a blank state. We experimented with different hyperparameter values, as detailed in Table 2. The various hyperparameter setups are designed to yield similar run times across all sub-experiments.

5. Results

5.1. Experiment 1: Exploring the performance of evolved architectures

In this section, we reported the results achieved from the first set of experiments, in which we aim at recreate the ResNet-18, ResNet-34, and DenseNet-121 architectures using the proposed framework and training them using standard backpropagation.

Table 2
Hyperparameters for unrestricted neuroevolution.

Hyperparameter	Value
Epochs (per mutation)	{5, 10}
Population	{5, 10}
Generations	{10, 20}
Reset and Fit Parameters at end of training	{True, False}
Use Learning Rate Finder (LRF)	False
Learning Rate (η)	0.05
Block Branches (BB)	True
Freeze Evolved Blocks (FEB)	False
Freeze Evolved Output Layers (FEOL)	False

5.1.1. ResNet-18

Fig. 10 displays the learning curves achieved when evolving the ResNet-18 architecture. In particular, on the left, the figure shows the value of the loss function on the training set at each epoch while, on the right, it displays the validation accuracy. Fig. 11 shows the validation accuracy achieved by the evolved ResNet-18 architecture with different hyperparameters' values. Based on the observed results, it appears that the choice of hyperparameters' values may affect the performance of the resulting network. Before entering into a more detailed analysis, Table 3 summarizes the results achieved considering the ResNet-18 architecture.

Referring to Table 3, we note that no sub-experiment is able to achieve the same level of accuracy as the baseline ResNet-18 (91.6%).

Impact of Block Branches (BB). It is important to note that BB inherently disables FEB within the block branch. Considering the topology of ResNet-18, since the block branch constitutes the entire model, FEB is effectively disabled for the entire model. The

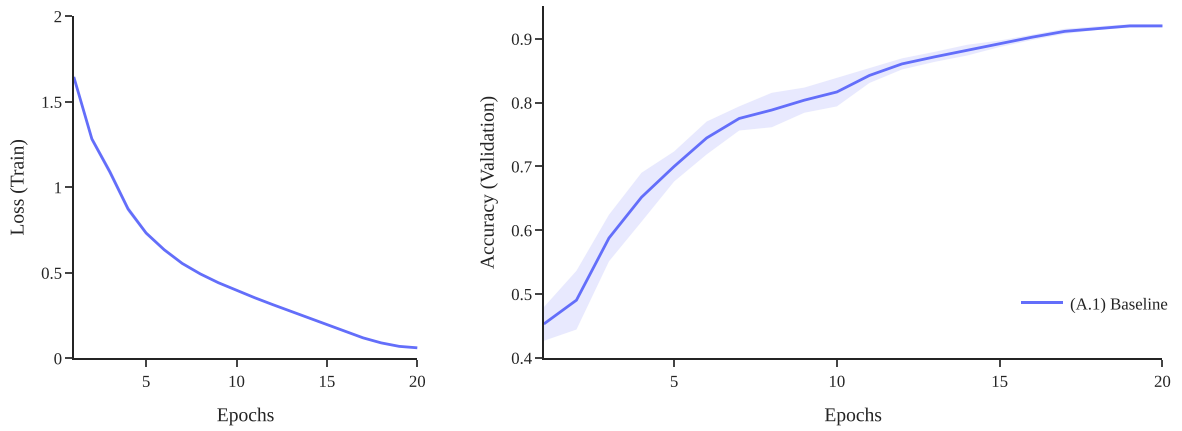

Fig. 10. Training progression for the baseline sub-experiment A.1 consisting of the ResNet-18 architecture. **Left.** Train loss. **Right.** Validation accuracy with final accuracy of 91%.

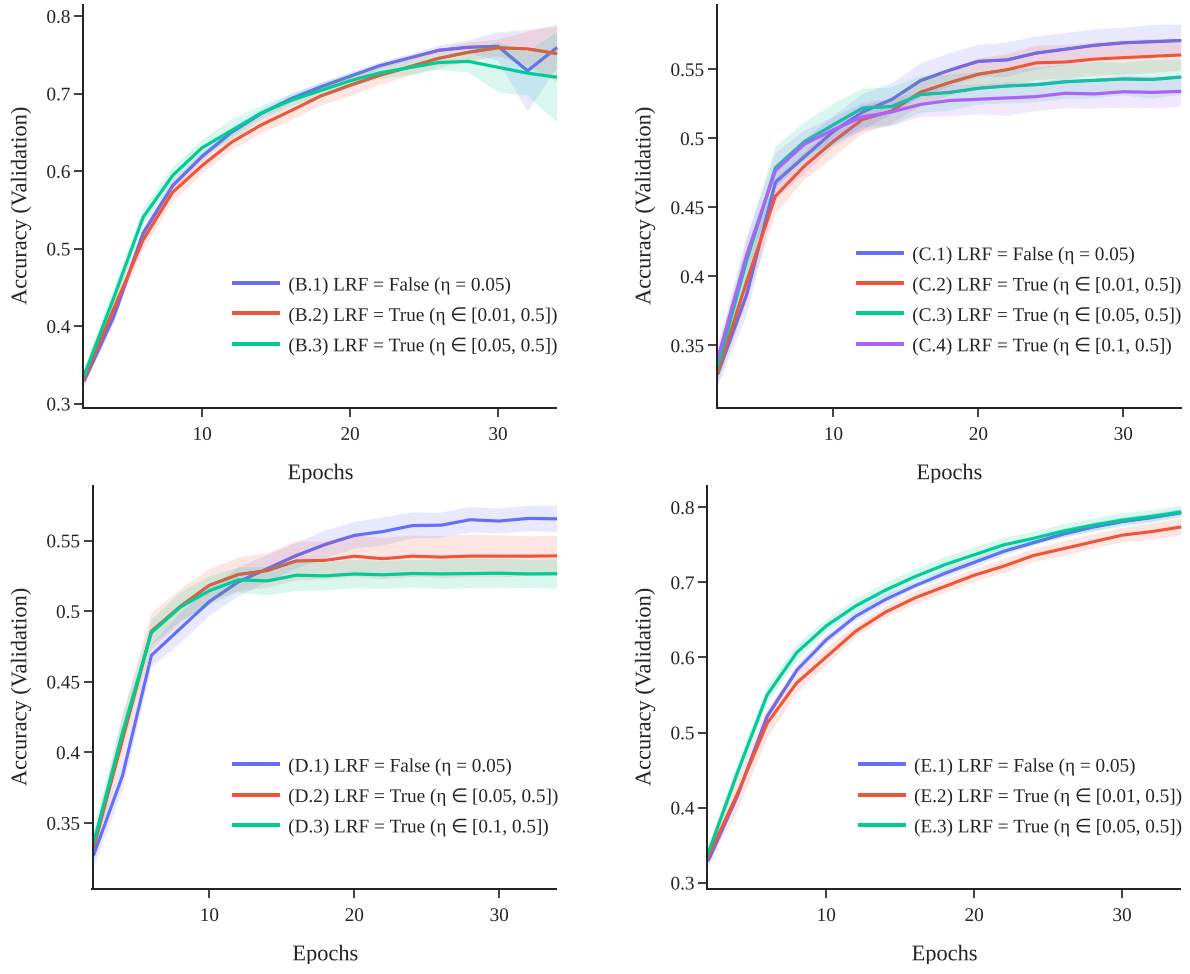

Fig. 11. Progression of validation accuracy for sub-experiments with varying hyperparameters.

key difference between sub-experiments B and E is that the entire model trains throughout the evolution in sub-experiment B, with every block connecting directly to the *Merge Block*. In contrast, in sub-experiment E, only the last block added is connected to the *Merge Block* while the entire model still trains during the whole evolution.

Though the ResNet Experiment may not be the ideal setting to examine the BB hyperparameter, as a block branch represents the entire model throughout the evolution steps, we still observe that sub-experiment E, with BB enabled, attains the best overall performance. Sub-experiment E.3 achieves a score of 79.41%, outperforming the best sub-experiment B.1, where BB is disabled, with a score of 76.33%. Additionally, it is worth mentioning that E.3 (±0.82%) demonstrates more consistent results compared to B.1 (±2.75%).

Impact of Freeze Evolved Blocks (FEB). We empirically prove that, with regards to neural networks, preserving the parent's contribution, as in GSGP, leads to significantly worse performance

Table 3
Comparison of sub-experiments using the ResNet-18 architecture on the CIFAR-10 test dataset, showcasing the impact of various hyperparameter settings on the model's performance.

Sub-experiment	BB	FEB	FEOL	LRF	$\eta_{\max}$	Epochs	Test accuracy [%]	Test accuracy (Best) [%]
Baseline (A.1)	–	–	–	–	0.05	20	**91.6 ± 0.17**	**92.06**
(B.1)			–		0.05	34	76.33 ± 2.75	78.9
(B.2)			–	✓	[0.01, 0.5]	34	75.43 ± 3.75	78.44
(B.3)			–	✓	[0.05, 0.5]	34	72.39 ± 6.01	77.81
(C.1)		✓			0.05	34	56.79 ± 1.17	59.38
(C.2)		✓		✓	[0.01, 0.5]	34	55.85 ± 1.19	58.26
(C.3)		✓		✓	[0.05, 0.5]	34	54.03 ± 1.31	56.08
(C.4)		✓		✓	[0.1, 0.5]	34	53.08 ± 1.0	54.99
(D.1)		✓	✓		0.05	34	56.44 ± 0.94	59.13
(D.2)		✓	✓	✓	[0.05, 0.5]	34	53.65 ± 1.45	56.69
(D.3)		✓	✓	✓	[0.1, 0.5]	34	52.26 ± 1.05	54.43
(E.1)	✓	–	–		0.05	34	79.3 ± 0.48	80.19
(E.2)	✓	–	–	✓	[0.01, 0.5]	34	77.51 ± 1.1	78.88
(E.3)	✓	–	–	✓	[0.05, 0.5]	34	79.41 ± 0.82	80.66

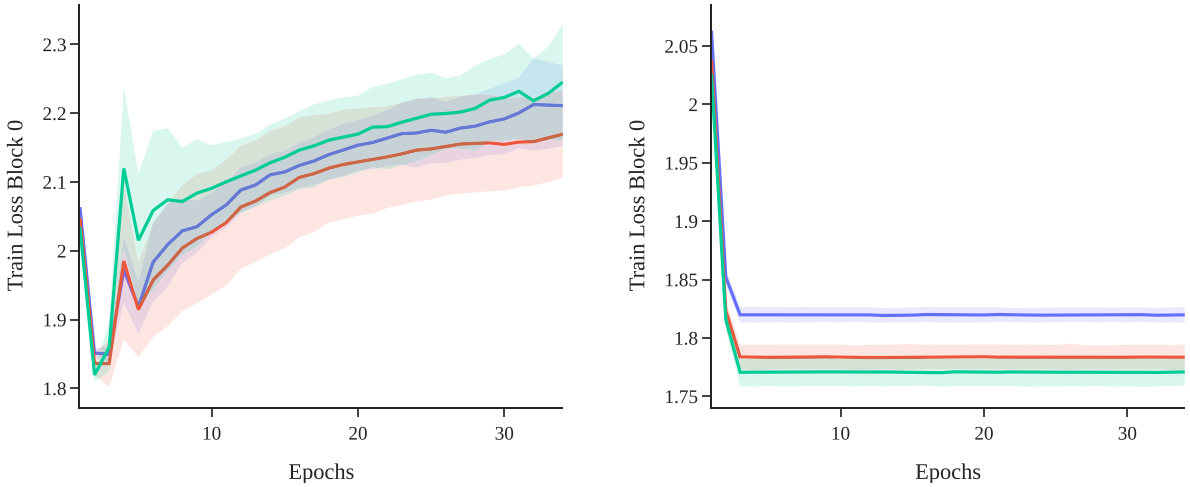

Fig. 12. Loss of Block 0 with Freeze Evolved Blocks (FEB) disabled (**left**, sub-experiment B) and enabled (**right**, sub-experiment D). The loss for models with FEB disabled (**left**) decreases for a few epochs and then increases as the model grows in size. This pattern suggests that Block 0 focuses on learning complex features while it serves as the last block of the model (until epoch = 4). However, as more blocks are added, Block 0 shifts towards learning basic features, leading to a decline in classification performance (i.e., increased loss). This contrasts with the models that have FEB and FOEL enabled (**right**), where Block 0 maintains a steady loss and, intuitively, does not make the "switch" from complex to more basic feature extraction.

than when we allow the parent structure to be changed by either disabling FEB or enabling BB. We observe that enabling FEB leads to overall worse performance. Sub-experiments C and D (FEB, maximum accuracy achieved of 56.79% for sub-experiment C.1) perform substantially worse than sub-experiments B (no FEB, maximum accuracy achieved of 76.33% for sub-experiment B.1).

A possible explanation for this drop in performance may be that convolutional layers are feature extractors, and as the model grows, they need to adapt to new blocks that are introduced. In such cases, the initial layers need to focus on learning basic, lower-level features, while the deeper layers should capture more complex, higher-level features. If the earlier layers are not allowed to adapt due to FEB, they may struggle to effectively extract the necessary basic features, thus hindering the model's overall performance.

As our framework is modular, we are able to measure the loss related to each block. Our observations align with the intuition that in evolutions with FEB, the new block consistently attempts to learn more complex and unique features of each class rather than basic feature maps (Fig. 12).

Impact of Freeze Evolved Output Layers (FEOL). We may only compare sub-experiments C and D when considering FEOL. Regarding the test accuracy, there is no significant difference between enabling (D.1; 56.44% ± 0.94%) or disabling (C.1; 56.79% ± 1.17%) FEOL.

Impact of Learning Rate Finder (LRF). We find that for sub-experiments B, C, and D, enabling LRF leads to overall worse performance (w.r.t. the accuracy). However, results were mixed for sub-experiment E. Using LRF proved marginally better for E.3 (79.96%; $\eta_{max}[0.05, 0.5]$) vs. 79.32% (E.1; $\eta_{max} = 0.05$) for no LRF. We conclude that using LRF is not ideal, as it increases evolution runtime without necessarily increasing model performance. We suggest a different approach for choosing the learning rate in Section 6 for future experiments.

Training an Evolved Model from Scratch. We can reverse-engineer the idea of this experiment: if evolving the ResNet-18 model leads to worse performance when compared to training it when it is already constructed, we can then apply the same concept to unrestrained (i.e., randomly) evolved models. We introduce *Reset N Fit*, a hyperparameter that resets and unfreezes the model's parameters, allowing it to be fully trainable, and retrains the model.

5.1.2. ResNet-34

Concerning the results achieved when considering the ResNet-34 architecture, Fig. 13 displays, on the left, the value of the loss function on the training set at each epoch while, on the right, it displays the validation accuracy. Table 4 summarizes the results achieved considering the ResNet-18 architecture.

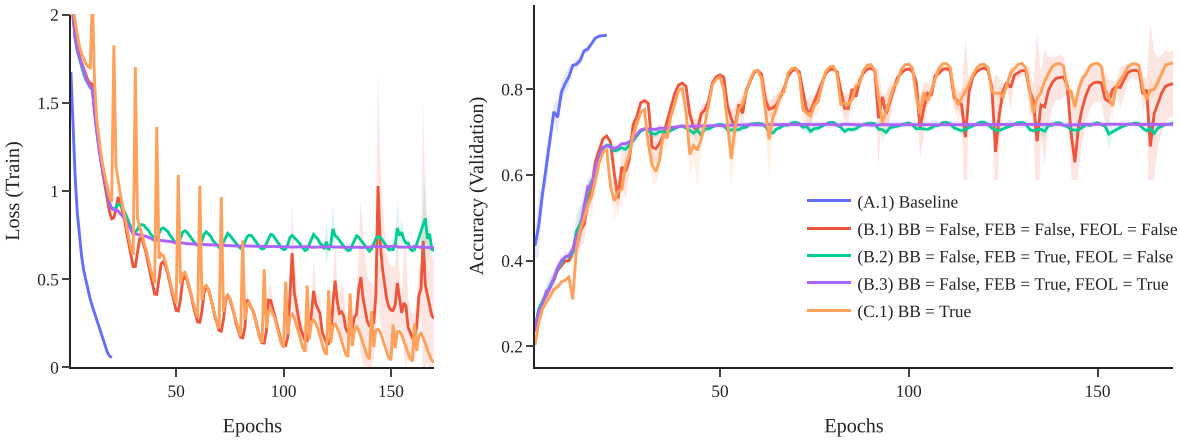

Fig. 13. Training progression for ResNet-34 on the CIFAR-10 dataset. **Left.** Train loss. **Right.** Validation accuracy.

Table 4
Comparison of sub-experiments using the ResNet-34 architecture on the CIFAR-10 test dataset.

Sub-experiment	BB	FEB	FEOL	η_{max}	Epochs	Test accuracy [%]	Test accuracy (Best) [%]
Baseline (A.1)	–	–	–	0.05	20	**92.16 ± 0.29**	**92.43**
(B.1)			–	0.05	170	81.54 ± 7.29	85.29
(B.2)		✓		0.05	170	72.5 ± 0.29	72.81
(B.3)		✓	✓	0.05	170	72.27 ± 0.45	72.88
(C.1)	✓	–	–	0.05	170	85.66 ± 0.24	86.01

Table 5
Comparison of sub-experiments using the DenseNet-121 architecture on the CIFAR-10 test dataset.

Sub-experiment	BB	FEB	FEOL	η_{max}	Epochs	Test accuracy [%]	Test accuracy (Best) [%]
Baseline (A.1)	–	–	–	0.05	20	**92.97 ± 0.15**	**93.19**
(B.1)			–	0.05	60	88.24 ± 0.22	88.49
(B.2)		✓		0.05	60	87.68 ± 1.83	88.96
(B.3)		✓	✓	0.05	60	88.38 ± 0.38	88.76
(C.1)	✓	–	–	0.05	60	88.54 ± 0.8	89.33

Similar to the results of the ResNet-18 experiment (see Section 5.1.1), no sub-experiment is able to achieve the same level of accuracy as the baseline ResNet-34 (92.16%) as shown in Table 4.

Impact of Block Branches (BB). Considering the ResNet-34 architecture, enabling BB results in an improvement in model performance. Sub-experiment C.1, with BB enabled, achieves a test accuracy of 85.66%, a significant improvement over the best sub-experiment without BB (B.1), which produces an accuracy of 81.54

Impact of Freeze Evolved Blocks (FEB). Similar to the observations in the ResNet-18 experiment, enabling FEB reduces the model's performance. As evident in sub-experiments B.2 and B.3, where FEB is enabled, the test accuracy is significantly lower than in sub-experiment B.1, where FEB is disabled. This pattern further supports the hypothesis that convolutional layers need the ability to adapt as the model evolves and new blocks are introduced.

Impact of Freeze Evolved Output Layers (FEOL). Just as in the ResNet-18 experiment, the effect of FEOL is not significant. Sub-experiment B.3, with FEOL enabled, achieves a test accuracy very similar to sub-experiment B.2, which has FEOL disabled.

5.1.3. DenseNet-121

Focusing on the results achieved when considering the DenseNet-121 architecture, Fig. 14 displays, on the left, the value of the loss function on the training set at each epoch while, on the right, it displays the validation accuracy. Table 5 summarizes the results achieved considering the DenseNet-121 architecture.

In line with the previous experiments, none of the DenseNet-121 sub-experiments, even the best-performing one (Sub-experiment C.1 with an accuracy of 88.54%), could match the accuracy of the baseline DenseNet-121, which achieved a test accuracy of 92.97% (Table 5).

The DenseNet architecture has a unique design, where each block concatenates features from preceding blocks. This design feature likely influenced the closely ranged performance of all sub-experiments, which presented test accuracies varying from 87.68% to 88.54%, very different results from previous experiments on the ResNet architecture. Regardless of the variations in the hyperparameters (BB, FEB, and FEOL), since a new block always has access to the feature maps obtained from previous blocks and can modify these feature maps, it effectively nullifies the concept of FEB and FEOL, bringing the performance of sub-experiments B to par with the performance of sub-experiment C.1. These results are promising, as we mention in Section 6 for future work, since sub-experiments with FEB enabled perform just as well as one with FEB disabled and, as previously explained, FEB allows for much faster training times.

5.2. Experiment 2: Unrestricted neuroevolution

In this section, we reported the results achieved by evolving models from scratch without any restrictions. These experiments are fundamental to assessing the suitability of the proposed framework to build neural networks that show satisfactory performance (see Table 6).

Based on the experimental evidence, we can state that our framework successfully evolves complex models. Although sub-experiment A.3 performs the best on average (84.97% ± 2.44%), the best model of A.2 achieves nearly 89% accuracy, and we believe the most relevant result is sub-experiment A.1. The best

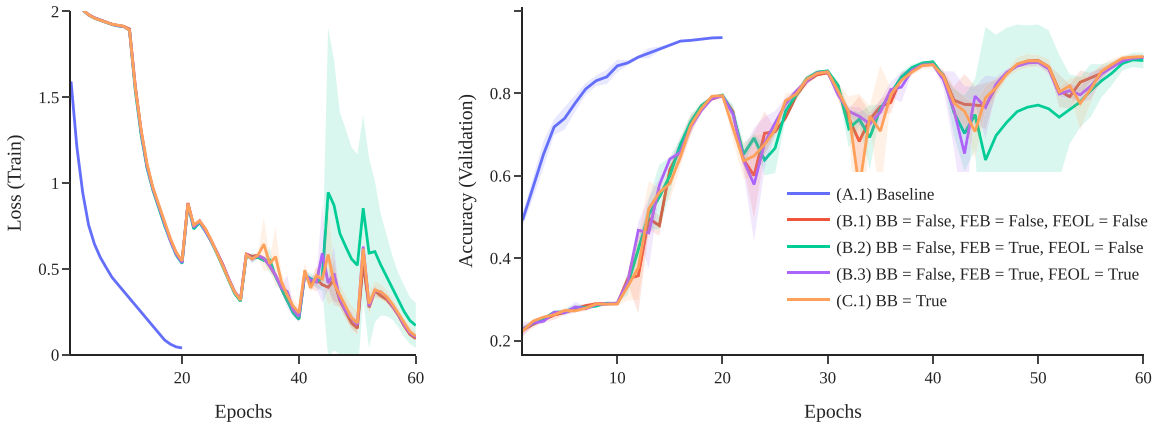

Fig. 14. Training progression for DenseNet-121 on the CIFAR-10 dataset. **Left.** Train loss. **Right.** Validation accuracy.

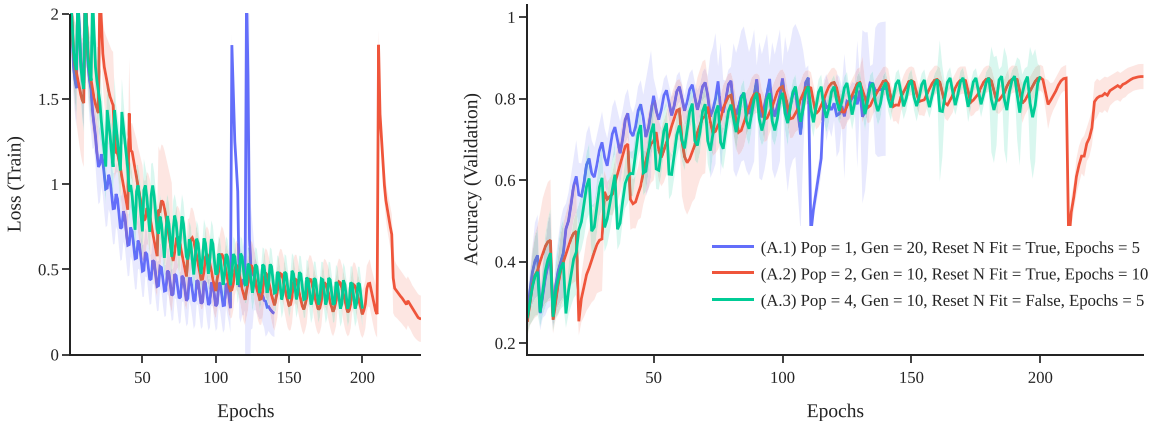

Fig. 15. Results for the evolution of models. **Left.** Train loss. **Right.** Validation accuracy.

Table 6
Comparison between experiments in the evolution experiment on the test dataset (CIFAR-10). We report the mean and standard deviation of the accuracies, averaged over thirty runs, and the best run of each experiment.

Experiment	Pop.	Gen.	Reset & Fit	Epochs (per mutation)	Epochs (total)	Accuracy [%]	Accuracy (Best) [%]	Parameters (Best) [Millions]
(A.1)	1	20	✓	5	100	84.46 ± 3.16	88.69	3.2
(A.2)	2	10	✓	10	200	84.58 ± 3.04	88.93	15.76
(A.3)	4	10		5	200	84.97 ± 2.44	88.4	11.23

model of sub-experiment A.1 attains 88.69% accuracy on the test set while having only 3.2M parameters. For comparison, the ResNet-18 model architecture from Section 5.1.1 achieves 91.6% (±0.17%) accuracy and has 11M parameters.

It appears that resetting the model's parameters and retraining it does not affect its validation accuracy meaningfully, as seen in Fig. 15. Sub-experiment A.1 is visibly better on the validation set before resetting its parameters, and A.2 maintains similar accuracy before and after resetting and retraining. This disproves our hypothesis from Section 5.1.1, where we suggested retraining the model from scratch after evolving it. Although more research should be conducted, our initial findings indicate that using our method to evolve models leads to different topologies more attuned to incremental evolutions.

In terms of discovery, we observe that these models took 16 min, on average, to evolve on a single GPU (Fig. 16). As previously mentioned, models found through NAS usually take multiple days on multiple GPUs.

Considering that the results of all sub-experiments A were very close, our intuition is that having a big population for each generation might not be compelling as we had to trade off the number of epochs per mutation to bring down the running time: A.2 with a population of 2 had 10 epochs per mutation vs. A.3 with a population of 4 had to be decreased to 5 epochs. Interestingly, the number of parameters is overall higher when the population is 1, but the best model of sub-experiment A.1 had a relatively low amount of parameters (3.2M). Overall, these are excellent results for such a new framework and method of evolving neural networks.

6. Conclusions

Despite being limited by computational resources, we successfully evolved models with high accuracy on our problem dataset by experimentally exploring numerous theoretical possibilities for the neuroevolution process. One of our best-evolved models achieved nearly 89% accuracy on the test set of the CIFAR-10 dataset problem with only 3.2M parameters.

In addition to the evolved model surpassing the manual evolution of the model mimicking the ResNet-18 architecture (84.97% ± 2.44% for our model versus 79.41% ± 0.82% for ResNet-18), the evolved models also demonstrated competitive performance

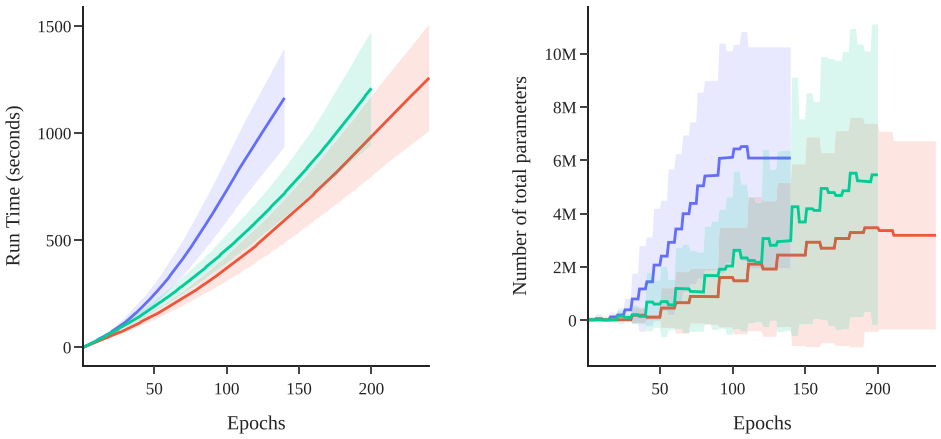

Fig. 16. Left. Run time of evolution in seconds. All evolutions ran on a 3080 Ti. **Right.** Number of parameters throughout mutations.

when compared to the other architectures tested. For instance, the best performing evolved models achieved accuracies of 86.01% (Table 4) and 89.33% (Table 5), versus ours' accuracy of 88.93%.

For future work, there are several ways to improve the framework:

- Implement convolutional blocks with groups and dilation to enhance the convolutional kernel's ability to analyze larger image areas without increasing the number of parameters.
- As corroborated by our findings presented in Section 5.1.3, enabling each block to receive multiple inputs and then concatenate them via pooling or other inexpensive methods can lead to significant acceleration in the evolution process.
- Enable mutations similar to the Bottleneck design in ResNet architectures by allowing layers (except the last one) to change their dimensions when the convolutional block is set to keep input dimensions.
- Explore optimizing the last block before unfreezing the entire branch for block branches. Similarly, when a new layer is added to a block, optimize the parameters of the layer and output layer for 1 or 2 epochs without altering the previous layers in the block.
- Increase block diversity by adding custom blocks, such as Residual Block and Bottleneck from ResNet or Transformer Blocks.
- Develop the framework to work with multiple GPUs, as this method of evolving neural networks is an excellent candidate for utilizing multiple training devices, with each device representing an individual in the population at every generation.

By addressing these areas for improvement, we believe the framework can be further refined to produce even better neural network architectures.

Declaration of competing interest

The authors declare no competing interests.

Data availability

No data was used for the research described in the article

Acknowledgments

This work is funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the projects CISUC - UID/CEC/00326/2020, UIDB/04152/2020 - Centro de Investigação em Gestão de Informação (MagIC)/NOVA IMS, and by European Social Fund, through the Regional Operational Program Centro 2020.

References

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., An image is worth 16x16 words: Transformers for image recognition at scale, 2020, arXiv preprint arXiv:2010.11929.
[2] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[3] J.R. Koza, Genetic programming as a means for programming computers by natural selection, Stat. Comput. 4 (1994) 87–112.
[4] A. Moraglio, K. Krawiec, C.G. Johnson, Geometric semantic genetic programming, in: International Conference on Parallel Problem Solving from Nature, Springer, 2012, pp. 283–293.
[5] G. Huang, Z. Liu, L. Van Der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 4700–4708.
[6] E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Regularized evolution for image classifier architecture search, in: Proceedings of the Aaai Conference on Artificial Intelligence, vol. 33, 2019, pp. 4780–4789.
[7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE (1998).
[8] V. Nair, G.E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 807–814.
[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958.
[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).
[11] J.H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications To Biology, Control, and Artificial Intelligence, MIT Press, 1992.
[12] L.J. Fogel, A.J. Owens, M.J. Walsh, Artificial Intelligence Through Simulated Evolution, John Wiley & Sons, 1966.
[13] H.-G. Beyer, H.-P. Schwefel, Evolution strategies–a comprehensive introduction, Nat. Comput. 1 (2002) 3–52.
[14] I. Bakurov, M. Castelli, F. Fontanella, L. Vanneschi, A regression-like classification system for geometric semantic genetic programming, in: Proceedings of the 11th International Joint Conference on Computational Intelligence (IJCCI 2019), vol. 1, SciTePress-Science and Technology Publications, 2019, pp. 40–48.
[15] I. Bakurov, M. Castelli, F. Fontanella, A.S. di Freca, L. Vanneschi, A novel binary classification approach based on geometric semantic genetic programming, Swarm Evol. Comput. 69 (2022) 101028.
[16] T. Bäck, D.B. Fogel, Z. Michalewicz, Handbook of evolutionary computation, Release 97 (1) (1997) B1.
[17] K.O. Stanley, J. Clune, J. Lehman, R. Miikkulainen, Designing neural networks through neuroevolution, Nat. Mach. Intell. 1 (1) (2019) 24–35.
[18] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, 2016, arXiv preprint arXiv:1611.02167.

[19] T. Elsken, J.-H. Metzen, F. Hutter, Simple and efficient architecture search for convolutional neural networks, 2017, arXiv preprint arXiv:1711.04528.
[20] R. Chandra, A. Tiwari, Distributed Bayesian optimisation framework for deep neuroevolution, Neurocomputing 470 (2022) 51–65.
[21] T. Deng, J. Wu, Efficient graph neural architecture search using Monte Carlo tree search and prediction network, Expert Syst. Appl. 213 (2023) 118916.
[22] A. Kapoor, E. Nukala, R. Chandra, Bayesian neuroevolution using distributed swarm optimization and tempered MCMC, Appl. Soft Comput. 129 (2022) 109528.
[23] Y. Liu, Y. Sun, B. Xue, M. Zhang, G.G. Yen, K.C. Tan, A survey on evolutionary neural architecture search, IEEE Trans. Neural Netw. Learn. Syst. (2021).
[24] A. Darwish, A.E. Hassanien, S. Das, A survey of swarm and evolutionary computing approaches for deep learning, Artif. Intell. Rev. 53 (2020) 1767–1812.
[25] M. Wistuba, A. Rawat, T. Pedapati, A survey on neural architecture search, 2019, arXiv preprint arXiv:1905.01392.
[26] L. Xie, A. Yuille, Genetic cnn, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 1379–1388.
[27] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, M. Sjödin, DeepMaker: A multi-objective optimization framework for deep neural networks in embedded systems, Microprocess. Microsyst. 73 (2020) 102989.
[28] Y. Sun, B. Xue, M. Zhang, G.G. Yen, Evolving deep convolutional neural networks for image classification, IEEE Trans. Evol. Comput. 24 (2) (2019) 394–407.
[29] K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evol. Comput. 10 (2) (2002) 99–127.
[30] E. Papavasileiou, J. Cornelis, B. Jansen, A systematic literature review of the successors of "neuroevolution of augmenting topologies", Evol. Comput. 29 (1) (2021) 1–73.
[31] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2017, pp. 497–504.
[32] S. Bianco, M. Buzzelli, G. Ciocca, R. Schettini, Neural architecture search for image saliency fusion, Inf. Fusion 57 (2020) 89–101.
[33] Z. Fan, J. Wei, G. Zhu, J. Mo, W. Li, Evolutionary neural architecture search for retinal vessel segmentation, 2020, arXiv preprint arXiv:2001.06678.
[34] M. Neshat, M.M. Nezhad, E. Abbasnejad, L.B. Tjernberg, D.A. Garcia, B. Alexander, M. Wagner, An evolutionary deep learning method for short-term wind speed prediction: A case study of the lillgrund offshore wind farm, 2020, arXiv preprint arXiv:2002.09106.
[35] E. Byla, W. Pang, Deepswarm: Optimising convolutional neural networks using swarm intelligence, in: Advances in Computational Intelligence Systems: Contributions Presented At the 19th UK Workshop on Computational Intelligence, September 4-6, 2019, Portsmouth, UK 19, Springer, 2020, pp. 119–130.
[36] F.E.F. Junior, G.G. Yen, Particle swarm optimization of deep neural networks architectures for image classification, Swarm Evol. Comput. 49 (2019) 62–74.
[37] P.R. Lorenzo, J. Nalepa, Memetic evolution of deep neural networks, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 505–512.
[38] B.P. Evans, H. Al-Sahaf, B. Xue, M. Zhang, Genetic programming and gradient descent: A memetic approach to binary image classification, 2019, arXiv preprint arXiv:1909.13030.
[39] E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q.V. Le, A. Kurakin, Large-scale evolution of image classifiers, in: International Conference on Machine Learning, PMLR, 2017, pp. 2902–2911.
[40] Y. Hu, S. Sun, J. Li, X. Wang, Q. Gu, A novel channel pruning method for deep neural network compression, 2018, arXiv preprint arXiv:1805.11394.
[41] T. Wu, J. Shi, D. Zhou, Y. Lei, M. Gong, A multi-objective particle swarm optimization for neural networks pruning, in: 2019 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2019, pp. 570–577.
[42] T. Tanaka, T. Moriya, T. Shinozaki, S. Watanabe, T. Hori, K. Duh, Automated structure discovery and parameter tuning of neural network language model based on evolution strategy, in: 2016 Ieee Spoken Language Technology Workshop (Slt), IEEE, 2016, pp. 665–671.
[43] H. Zhu, Z. An, C. Yang, K. Xu, E. Zhao, Y. Xu, EENA: efficient evolution of neural architecture, in: Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops, 2019.
[44] S. Kern, S.D. Müller, N. Hansen, D. Büche, J. Ocenasek, P. Koumoutsakos, Learning probability distributions in continuous evolutionary algorithms–a comparative review, Nat. Comput. 3 (2004) 77–112.
[45] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[46] M. Baldeon-Calisto, S.K. Lai-Yuen, AdaResU-net: Multiobjective adaptive convolutional neural network for medical image segmentation, Neurocomputing 392 (2020) 325–340.
[47] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, in: Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18, Springer, 2015, pp. 234–241.
[48] M. Suganuma, M. Kobayashi, S. Shirakawa, T. Nagao, Evolution of deep convolutional neural networks using cartesian genetic programming, Evol. Comput. 28 (1) (2020) 141–163.
[49] T. Hassanzadeh, D. Essam, R. Sarker, Evou-net: an evolutionary deep fully convolutional neural network for medical image segmentation, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 181–189.
[50] X. Dong, Y. Yang, Nas-bench-201: Extending the scope of reproducible neural architecture search, 2020, arXiv preprint arXiv:2001.00326.
[51] Y. Chen, T. Pan, C. He, R. Cheng, Efficient evolutionary deep neural architecture search (nas) by noisy network morphism mutation, in: Bio-Inspired Computing: Theories and Applications: 14th International Conference, BIC-TA 2019, Zhengzhou, China, November 22–25, 2019, Revised Selected Papers, Part II 14, Springer, 2020, pp. 497–508.
[52] L. Frachon, W. Pang, G.M. Coghill, Immunecs: Neural committee search by an artificial immune system, 2019, arXiv preprint arXiv:1911.07729.
[53] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, X. Wang, A comprehensive survey of neural architecture search: Challenges and solutions, ACM Comput. Surv. 54 (4) (2021) 1–34.
[54] N. Li, L. Ma, G. Yu, B. Xue, M. Zhang, Y. Jin, Survey on evolutionary deep learning: Principles, algorithms, applications and open issues, ACM Comput. Surv. (2022).
[55] C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, F. Hutter, Neural architecture search: Insights from 1000 papers, 2023, arXiv preprint arXiv:2301.08727.
[56] Z.-H. Zhan, J.-Y. Li, J. Zhang, Evolutionary deep learning: A survey, Neurocomputing 483 (2022) 42–58.
[57] N. Li, L. Ma, T. Xing, G. Yu, C. Wang, Y. Wen, S. Cheng, S. Gao, Automatic design of machine learning via evolutionary computation: A survey, Appl. Soft Comput. (2023) 110412.
[58] I. Gonçalves, S. Silva, C.M. Fonseca, Semantic learning machine: A feed-forward neural network construction algorithm inspired by geometric semantic genetic programming, in: Progress in Artificial Intelligence: 17th Portuguese Conference on Artificial Intelligence, EPIA 2015, Coimbra, Portugal, September 8-11, 2015. Proceedings 17, Springer, 2015, pp. 280–285.
[59] I. Gonçalves, An Exploration of Generalization and Overfitting in Genetic Programming: Standard and Geometric Semantic Approaches (Ph.D. thesis), Department of Informatics Engineering, University of Coimbra, Portugal, 2017.
[60] L.N. Smith, A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay, 2018, arXiv preprint arXiv:1803.09820.
[61] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, J. Han, On the variance of the adaptive learning rate and beyond, 2019, arXiv preprint arXiv:1908.03265.
[62] L.N. Smith, N. Topin, Super-convergence: Very fast training of neural networks using large learning rates, in: Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications, vol. 11006, SPIE, 2019, pp. 369–386.
[63] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.
[64] L.N. Smith, Cyclical learning rates for training neural networks, in: 2017 IEEE Winter Conference on Applications of Computer Vision, WACV, IEEE, 2017, pp. 464–472.
[65] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feed-forward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
[66] D. Hendrycks, K. Gimpel, Gaussian error linear units (gelus), 2016, arXiv preprint arXiv:1606.08415.
[67] A.L. Maas, A.Y. Hannun, A.Y. Ng, et al., Rectifier nonlinearities improve neural network acoustic models, in: Proc. Icml, vol. 30, Atlanta, Georgia, USA, 2013, p. 3.
[68] I. Gonçalves, S. Silva, C.M. Fonseca, On the generalization ability of geometric semantic genetic programming, in: Genetic Programming: 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings 18, Springer, 2015, pp. 41–52.