Thiago Zafalon Miranda

Grammar-based Neuroevolution of Fully Convolutional Networks

Doctoral Thesis presented to the Programa de Pós-Graduação em Ciência da Computação of the Universidade Federal de São Carlos as a partial requirement to obtain the degree of Doctor of Computer Science

Universidade Federal de São Carlos

Programa de Pós-graduação em Ciência da Computação

Supervisor Ricardo Cerri

Co-supervisor: Márcio Porto Basgalupp

São Carlos, São Paulo - Brazil

2025

FUNDAÇÃO UNIVERSIDADE FEDERAL DE SÃO CARLOS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO - PPGCC/CCET
Rod. Washington Luís km 235 - SP-310, s/n - Bairro Monjolinho, São Carlos/SP, CEP 13565-905
Telefone: (16) 33518233 - http://www.ufscar.br

DECLARAÇÃO

Declaro, para os devidos fins, que a banca de defesa de tese de doutorado, conforme abaixo, foi realizada com a presença de todos os membros.
12/08/2025 14h00min
Candidato Thiago Zafalon Miranda
Título Grammar-based Neuroevolution of Fully Convolutional Networks

Presidente Ricardo Cerri UFSCar
Titular Helena de Medeiros Caseli UFSCar
Titular Sandra Eliza Fontes de Avila Unicamp
Titular Gisele Lobo Pappa UFMG
Titular Rodrigo Coelho Barros PUC/RS

O(a) aluno(a) está apto(a) agora a receber o título de "Doutor(a) em Ciência da Computação". A homologação do título e o respectivo processo de expedição e registro de diploma aguardam a finalização da versão final do texto da tese e de seu depósito no Repositório Institucional da UFSCar para se iniciarem.

Ivan Rogério da Silva
Secretaria do PPGCC

Documento assinado eletronicamente por **Ivan Rogerio da Silva**, **Assistente em Administração**, em 13/08/2025, às 08:35, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufscar.br/autenticacao, informando o código verificador **1936192** e o código CRC **9F2CEFBB**.

Referência: Caso responda a este documento, indicar expressamente o Processo nº 23112.024306/2025-56

SEI nº 1936192

Modelo de Documento: Declaração, versão de 02/Agosto/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Thiago Zafalon Miranda, realizada em 12/08/2025.

Comissão Julgadora:

Prof. Dr. Ricardo Cerri (UFSCar)

Profa. Dra. Helena de Medeiros Caseli (UFSCar)

Profa. Dra. Sandra Eliza Fontes de Avila (UNICAMP)

Profa. Dra. Gisele Lobo Pappa (UFMG)

Prof. Dr. Rodrigo Coelho Barros (PUC-RS)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Acknowledgements

I would like to express my deepest gratitude to my family, whose unwavering support and love have been the foundation of this journey. I am especially thankful to my mother, whose emotional support, endless encouragement, and unconditional love have sustained me through every step. I am equally grateful to my wife, whose own academic journey has been a constant source of inspiration. Her love, understanding, and support gave me strength during the most challenging moments.

I am deeply thankful to Diorge Brognara for his valuable insights, thoughtful discussions, and steadfast friendship. His presence and perspective throughout this journey enriched both my academic work and personal experience.

I extend my heartfelt thanks to my supervisor, Professor Ricardo Cerri, for his extraordinary dedication and mentorship. His support went far beyond expectations—always available beyond working hours, continuously seeking grants, hardware, and new opportunities to enhance both this research and my development as a scholar. His commitment has been truly invaluable.

I am also grateful to Victor Padilha, who guided me in understanding and working effectively within a supercomputing environment; Leonardo Martinussi, who assisted me with the Euler computer cluster; Professors Paulo Matias and Hélio Crestana Guardia for their support with the UFSCars computing cluster.

To all of you, my sincerest thanks.

Last, but not least, this study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 - PhD scholarship with process number 88887.682413/2022-00. Research carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0).

Abstract

The design of complex and deep neural networks is often performed by identifying and combining building blocks and progressively selecting the most promising combination. Neuroevolution automates this process by employing evolutionary algorithms to guide the search. Within this field, grammar-based evolutionary algorithms have been demonstrated to be powerful tools to describe and thus encode complex neural architectures effectively. In this context, this research proposes a novel grammar-based multi-objective neuroevolutionary approach for generating fully convolutional networks. The proposed method, named Multi-Objective gRammatical Evolution for FUlly convolutional Networks (MOREFUN), includes a new efficient way to encode skip-connections, facilitating the description of complex search spaces and the injection of domain knowledge in the search procedure, the generation of fully convolutional networks upsampling of lower-resolution inputs in multi-input layers, the usage of multi-objective fitness, and the inclusion of data augmentation and optimizer settings in the grammar. The best networks found by the algorithm outperformed those generated by previous grammar-based evolutionary algorithms, achieving 90% accuracy on CIFAR-10 without using transfer learning, ensembles, or test-time data augmentation, while having a relatively small number of parameters.

Keywords: evolutionary algorithm; neural network; neuroevolution; multi-objective optimization; grammatical evolution; image classification;

Contents

1	**INTRODUCTION**	**8**
1.1	**Context**	**8**
1.2	**Hypothesis**	**9**
1.3	**Summary of results and main contributions**	**9**
1.4	**Fundamentals**	**9**
1.4.1	Multi-objective Optimization	9
1.4.2	Evolutionary Algorithms	11
1.4.3	Formal Languages and their Relation to Evolutionary Algorithms	12
1.4.4	Neural Architecture Search	18
1.4.5	Most Influential Works	19
2	**LITERATURE REVIEW**	**27**
2.1	**Related works found via Snowball**	**27**
2.1.1	Works based on manual or exhaustive search	27
2.1.2	Works based on reinforcement learning	28
2.1.3	Works based on gradient	29
2.1.4	Works based on evolutionary search	31
2.2	**Related works found using systematic review methods**	**33**
3	**PROPOSED ALGORITHM**	**44**
3.1	**Population initialization**	**45**
3.2	**Individual's representation**	**46**
3.3	**Multi-input layers**	**53**
3.4	**Grammar primitives**	**54**
3.5	**Mutation**	**57**
3.6	**Fittest selection**	**58**
3.7	**Principal differences between MOREFUN and DENSER**	**59**
4	**EXPERIMENTS**	**61**
4.1	**Scenarios**	**61**
4.1.1	Default scenario	62
4.1.2	Single objective optimization	66
4.1.3	With validation	67
5	**CONCLUSION AND FUTURE WORKS**	**71**
	REFERENCES	**73**

1 Introduction

1.1 Context

Deep Neural Networks (DNNs) are popular machine learning algorithms used to address classification and regression problems in multiple application domains, such as natural language processing (Otter, Medina and Kalita 2020) and image processing (Alam et al. 2020).

Developing DNNs to efficiently address a specific class of problems can be both challenging and onerous for a human designer (Géron 2019, Ronneberger, Fischer and Brox 2015) as it requires both domain knowledge and a trial-and-error process that resembles a human-powered evolutionary algorithm: a specialist generates an initial collection of models, evaluates their performances, identifies the best models (using one or more metrics to compare them), and recombines the most promising ideas or components of the best solutions to generate a new generation of models. This process is repeated until the stopping criteria are met.

The automation of this process using evolutionary algorithms (EAs), often called neuroevolution, requires the definition of the search space for the problem (Xue et al. 2021), that is, a description of which models are suitable for the problem at hand. Recent works have shown that neuroevolutionary algorithms can generate DNNs that perform comparably to hand-crafted models (Assunção et al. 2019).

Within the field of neuroevolution, some solutions are based on Grammar-based Evolutionary Algorithms (GEAs) (O'Neill and Ryan 2001), which describe the search space using a grammar. The main advantage of GEAs for neuroevolution is that complex neural network structures can be efficiently and elegantly represented through a collection of formal grammar rules (Assunção et al. 2019). Furthermore, grammar-based representations create an interface that cleanly separates the genotype from the phenotype, enabling the modification of the genotype and the phenotype (in a programming sense) independently of one another. For example, the designer can add new rules to a grammar (i.e., new phenotypes) without rewriting the code that manipulates the genotype, which simplifies the injection of domain knowledge into the search algorithm. Recent works, such as (Assunção et al. 2019, Assunção et al. 2019, Assunção et al. 2019), proved the feasibility and competitiveness of this approach in designing neural networks.

1.2 Hypothesis

Given the context previously mentioned, the hypothesis of this research is that grammar-based neuroevolutionary algorithms (GBNA) can be enhanced with multi-objective optimization (MOO) to generate smaller and more efficient models while simultaneously speeding up the search process by avoiding regions of the search space that contain wastefully large and slow-to-train models.

To evaluate the hypothesis, a GBNA was proposed and evaluated in terms of the quality of the models it generated and the GPU days required to run the algorithm.

1.3 Summary of results and main contributions

The results were favorable to the hypothesis, as experimental runs with MOO required substantially less time and produced smaller models when compared to those without it.

The main contributions of the algorithm include: mechanisms to describe skip connections, including from ones that connect smaller layers to larger ones to enable the creation of U-Net-like architectures (Ronneberger, Fischer and Brox 2015); multi-objective optimization; simplified pipeline, relying on the grammar to describe mechanisms that other algorithms, such as Deep Evolutionary Network Structured Representation (DENSER) (Assunção et al. 2019), implemented using an additional Genetic Algorithm (GA); simplified encoding, relying on SGE (Structured Grammatical Evolution) (Lourenço, Pereira and Costa 2015) instead of DSGE (Dynamic Structured Grammatical Evolution) (Lourenço et al. 2018).

1.4 Fundamentals

This research is built on top of multiple concepts, which will be discussed in the following sections: multi-objective optimization in Section 1.4.1, evolutionary algorithms in Section 1.4.2, grammar-based evolutionary algorithms in Section 1.4.3, neural architecture search and neuroevolution in Section 1.4.4, and, finally, most influential works in Section 1.4.5.

1.4.1 Multi-objective Optimization

Real-world problems often require the simultaneous optimization of multiple incommensurable objectives, that is, objectives that cannot be directly compared because they are not measured with the same units, e.g., age, measured in years, and weight, measured in kilograms. When trying to solve such problems, which are called Multi-Objective Opti-

mization problems (MOO), if one optimizes a single objective, then the final solution may be unacceptable for the other objectives. An example of one such problem in computer science is “find a machine learning model for a given task, such as classification, optimizing its runtime and its accuracy”: if we optimize just for runtime, we could find a “maximally fast” model that simply outputs the same class without looking at the input data.

Three of the most common approaches to handling MOO problems are (Freitas 2004): weighted objective combination, lexicographic optimization, and the Pareto approach. The weighted combination of objectives consists of transforming the MOO problem into a single-objective optimization problem and then ranking the solutions using this new objective. For example, if we go back to the problem of optimizing the runtime and accuracy of classification models, then a combined objective could be “maximize $(accuracy * 100) - (runtime/60)$”. Although this approach appears attractive at first glance, it intermixes quantities measured in different unities and requires finding weights for the objectives.

The lexicographic approach handles those issues by assigning priorities to each objective and comparing the solutions using only the objective with the highest priority. If and only if they are equivalent, they are compared using the objective with the subsequent highest priority. This process is repeated until one solution is found to be superior or the objectives are exhausted, in which case the solutions are considered equivalent. An inherent problem of this approach is that it will sacrifice arbitrarily large improvements in all objectives for possibly minuscule benefits in the current highest-priority objective; for example, if accuracy has the highest priority, then a model with 0.00001% greater accuracy that is orders of magnitude slower would be preferable to another that is faster but with slightly lower accuracy. This is particularly problematic for objectives whose measurements are “noisy”.

Finally, the Pareto-based approach is built on the concept of Pareto domination. A solution is said to dominate another solution if it is as good as the other solution for all objectives and better in at least one objective. Table 1 contains examples of solutions for an optimization problem with two objectives that must be maximized. In the Pareto approach, the most interesting solutions are the non-dominated ones, as they offer unique compromises for the objectives.

Evolutionary algorithms are frequently used in multi-objective optimization for several reasons. Primarily, they have minimal requirements, needing only three functions: one to generate solutions, one to modify solutions, and one to evaluate the solutions. Second, they are relatively easy to implement and parallelize. Additionally, when combined with appropriate fitness evaluation functions and selection criteria, they can produce a diverse range of solutions (Zitzler 1999).

Table 1 – Examples of Pareto Domination

Solution ID	O_1	O_2	Is dominated by	Dominates
A	2	3	{C, D, F}	
B	3	2	{C, D, F}	
C	4	6		{A, B}
D	6	5		{A, B, F}
E	8	0		
F	6	4	{D}	{A, B}

1.4.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are inspired by Darwin's Theory of Evolution (Eiben, Smith et al. 2003), which describes how a population of individuals living in an environment with finite resources changes over time. The premise is that the limited resources force the individuals to compete, which causes individuals more well-adapted for such competition to have a higher probability of outliving and reproducing than the less well-adapted individuals.

The EAs implement those ideas by viewing a collection of solutions to a problem, such as machine learning models created for an image classification problem, as competing individuals. The evolutionary pressure is simulated by evaluating the fitness of those solutions, that is, measuring how well they solve the problem. In the context of classification models, the fitness of a model could be its accuracy, runtime requirements, or even a combination of both. The best, or most fit, individuals are then used to generate other solutions, while the least fit ones are discarded.

Before we proceed, it is important to describe a few of the least-intuitive expressions that are often employed in the EA literature:

- Genotype: EAs quite often encode the individuals using some data structure that can be easily and efficiently manipulated. A neural network, for example, might be encoded as an adjacency matrix, which, in turn, could be flattened and ultimately be represented as a sequence of integers. The term genotype is often used to refer to individuals in this encoded form. A genotype is usually only useful during the execution of the evolutionary algorithm;
- Codon: a small part of a genotype. In the adjacency matrix example, a cell of the matrix could be considered a codon;
- Phenotype: the result of decoding a genotype, usually a solution. Most often, such a solution is also useful outside the evolutionary algorithm, e.g., a neural network.

There are multiple ways to generate new individuals from a collection of preexisting

individuals, most notably: a single individual can be modified slightly to generate a new solution, or pairs of individuals can be combined to generate a new solution that contains parts of each "parent" solution. The former process is often called mutation, while the latter is often called breeding or reproduction.

The EA repeats the process of evaluating existing individuals, generating new ones, and discarding the least promising ones until predefined criteria are met, such as executing a certain number of cycles or finding at least one individual with a given fitness level. The evolutionary loop of a generic algorithm is illustrated in Figure 1.

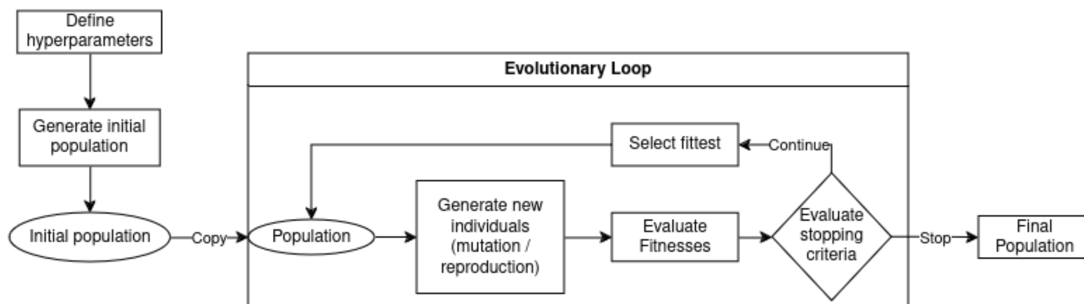


Figure 1 – Flowchart of a Generic Evolutionary Algorithm - The order of the steps "Generate new individuals" and "Evaluate Fitness" is an implementation detail. For example, if the algorithm uses the fitness of the individuals to determine which ones will participate in the generation of new individuals, then the order may be inverted.

EAs can be used to perform multi-objective optimization by using an appropriate fitness evaluation strategy. For instance, if the algorithm is being used to evolve Decision Trees (DT) (Jankowski and Jackowski 2015), and the fitness of a given DT is a tuple $(number_of_nodes, accuracy)$, then the EA could rank the DTs using the lexicographic approach; the first ones would be the fittest and the last ones would be the least fit.

Finally, it is important to emphasize that EA is an expression used to describe a class of algorithms, which can themselves be organized into many categories, such as Genetic Algorithms (Freitas 2013, Eiben, Smith et al. 2003), Learning Classifier Systems (Urbanowicz and Moore 2009), and many others[1]. A sub-type of particular interest for this research is Grammar-based Evolutionary Algorithms (GEAs), which will be discussed in Section 1.4.3.

1.4.3 Formal Languages and their Relation to Evolutionary Algorithms

Researchers have investigated and improved multiple components of evolutionary algorithms, such as mutation (Slowik and Kwasnicka 2020), breeding (Hassanat et al. 2019), fitness evaluation and fittest selection (Deb et al. 2002), and individual representation (O'Neill et al. 2004). Grammar-based Evolutionary Algorithms are particularly con-

[1] <https://github.com/fcampelo/EC-Bestiary>

cerned with the description of the search spaces and with the representation of individuals. Such aspects are important because a sub-optimal individual representation may negatively affect other aspects of the algorithm to the point of reducing the effectiveness of the evolutionary process to a random search (Castle and Johnson 2010); similarly, a poorly chosen search space may be wastefully large or constrained to an uninteresting region of the solution space.

The main idea of GEAs is to use grammars to describe the search space, and each sequence of tokens, or phrases, recognized by this grammar represents an individual. Before describing how this is done and how it is beneficial, formal languages and grammars will be briefly reviewed.

A grammar can be described by a 4-tuple $G = (N, T, P, S)$, where N denotes a set of non-terminal symbols (or variables); T denotes a set of terminal symbols; P denotes a set of production rules; and $S \in N$ denotes the initial symbol of the grammar.

A production rule is a function that maps sets of symbols to other sets of symbols, usually mapping a single non-terminal to a sequence of terminals and non-terminals. The set of sequences of non-terminals that can be generated using the production rules is called the "language recognized (or described) by the grammar". Figure 2 illustrates how such sequences can be generated; we start with the initial symbol S, then repeatedly use the production rules to replace the non-terminals until our sequence of symbols (or phrase) contains only non-terminals.

Although the grammar shown in Figure 2 is relatively simple, this thesis contains larger and more complex grammars; thus, to simplify their representations, the following conventions will be adopted:

- As it is common in the GEA literature, the definition of N, T, and S will be omitted, as they can all be inferred from the production rules: S is the symbol on the left-hand side of the first production rule; all symbols that appear on the left-hand side of a rule are non-terminals; any symbol that is not a non-terminal is a terminal;
- The syntax `?`, such as in the rule `a: b? c`, indicates that the preceding expression may appear one or zero times. This means that the rule `a: b? c` actually represents two rules: `a: c` and `a: b c`;
- The syntax `~3..7`, such as in the rule `a: b~3..7 c`, indicates that the expression may appear from 3 to 7 times. This means that `a: b? c` is equivalent to `a: b~0..1 c`;
- Finally, the syntax `...` indicates that parts of the grammar that are irrelevant to the given context have been omitted.

```
Grammar definition:
N = {X, Y, Z}
T = {c, o, _}
S = X
P = {(0) X: cYcY
     (1) X: cYcYZA
     (2) Y: o
     (3) Z: _
}

Tokenstream generation
- starting with S=X, then replacing X using the production rule 1
    before = X
    after  = cYcYZX
- replace Y using production rule 2
    before = cYcYZX
    after  = cocYZX
- replace Y using production rule 2
    before = cYcYZX
    after  = cocoZX
- replace X using production rule 0
    before = cocoZX
    after  = cocoZcYcY
- replace Y using production rule 2
    before = cocoZcYcY
    after  = cocoZcocY
- replace Y using production rule 2
    before = cocoZcocY
    after  = cocoZcoco
- replace Z using production rule 3
    before = cocoZcoco
    after  = coco_coco

Process finishes, as the phrase contains only terminal symbols
```

Figure 2 – Example of phrase generation.

The first convention is interesting because the mechanism used to identify non-terminals suggests that all grammars will be Context-Free. There are multiple types of grammars, each being able to represent more or less complex languages depending on restrictions imposed on the production rules, such as the maximum number of symbols appearing on the left side of the rule. Context-free Grammars, for example, require that the left side of the rules contain exactly one symbol (a non-terminal), while Recursively Enumerable Grammars have no restrictions. For an in-depth discussion of grammars, see (Hopcroft and Ullman 1969); but for the purposes of this work, that will not be necessary.

As previously mentioned, for GEAs, the token streams (or phrases) generated by a grammar represent an individual, more specifically, the phenotype of an individual. In most GEAs, it is implicitly assumed that there is a trivial method of converting such phrases into actual solutions. For example, if the GEA generates a token stream "neural network - dense layer - 10 neurons - sigmoid function - dense layer - 10 neurons - softmax function", then it is assumed that there is a way to convert those literal characters into a proper neural network containing two dense layers, the first using a sigmoid activation function and the second one a softmax function. This is usually not discussed because it is an uninteresting implementation detail; the GEA could generate a valid `.json` string, which could then be parsed by a specific neural network framework, such as Keras[2]. Usually, however, the grammar is designed using symbols that make it easier for humans to reason about the generated solutions.

In this context, the genotype is usually a data structure that can be processed with a grammar to generate a sequence of tokens (the phenotype). This process is called genotype-to-phenotype mapping (or translation). This, combined with the previously mentioned assumption about the existence of a trivial function that converts the sequence of tokens into an actual solution, means that the expression "phenotype" can be used to refer to the immediate output of the mapping process or to the actual solution produced by invoking the implicit conversion function.

One of the simplest genotypes and associated genotype-to-phenotype translation mechanisms is the one employed by the original Grammatical Evolution (O'Neill and Ryan 2001). The genotypes were simple sequences of bits, and the translation could be summarized as: create an initial sequence of tokens that just contains the starting symbol of the grammar, then repeatedly substitute its leftmost non-terminal using the production rules until it contains only terminal symbols. Whenever the set of rules r that could be used to transform the leftmost symbol contained more than one rule, it consumed 8 bits from the genotype and interpreted them as an unsigned integer (i.e., a integer between 0 and 255), then selected the n-th rule, with $n = c \bmod |r|$. Figure 3 illustrates this process.

The Grammatical Evolution algorithm, and works inspired by it, are of particular importance to this research and, as such, will be discussed in-depth in Section 1.4.5. All of them share some features, such as the mapping (or translation) process that creates an interface that cleanly separates the phenotypes from genotypes. Having decoupled representations for genotypes allows algorithm designers to choose data structures to represent the genotypes without having to consider how it would affect the phenotype. Similarly, it enables the description of intricate phenotypes that describe complex solutions, such as neural networks, without necessarily incurring an unwieldy increase in the complexity of

[2] <https://www.tensorflow.org/versions/r2.15/api_docs/python/tf/keras/models/model_from_json>

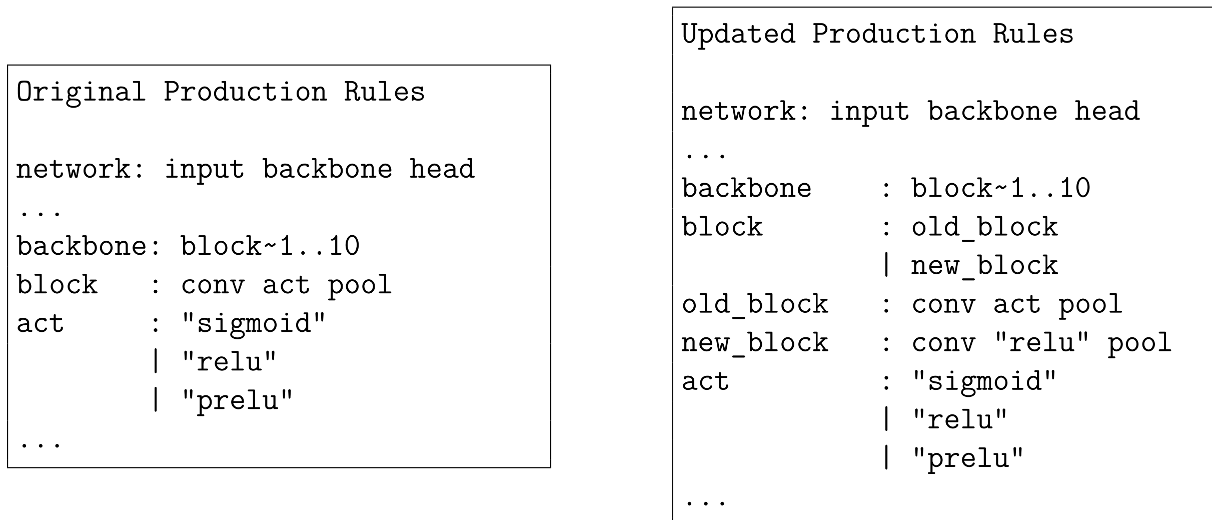
```
Grammar definition:
N = {X, Y}
T = {a, b}
S = X
P = {(0) X -> Xa
     (1) X -> a
     (2) X -> XY
     (3) Y -> b
}

Genotype (already interpreted as a sequence of integers):
genotype=51234549

Mapping process:
- first step
    initial token stream = X
    initial genotype     = 51234549
    r = {0, 1, 2}
    n = 5 mod |{0, 1, 2}|  = 2
    transform X using rule (2)
    resulting token stream = XY
    resulting genotype     = 1234549

- second step
    initial token stream = XY
    initial genotype     = 1234549
    r = {0, 1, 2}
    n = 1 mod |{0, 1, 2}|  = 1
    transform X using rule (2)
    resulting token stream = aY
    resulting genotype     = 234549

- final step
    initial token stream = aY
    initial genotype     = 234549
    expand using the only available rule (no need to consume genotype)
    resulting token stream = ab
    resulting genotype     = 234549

Process finishes, as the token stream contains only terminal symbols
```

Figure 3 – Example of phrase generation using Grammatical Evolution (2001) mechanisms.

the genotype manipulation.

Another major benefit of using grammars to describe the search space is that it makes it easier to inject domain knowledge into the search process by customizing the grammar. A simple example is shown in Figure 4, where the probability of certain features

appearing in our population is increased by just adding an explicit expansion of the rule.



```
Original Production Rules

network: input backbone head
...
backbone: block~1..10
block   : conv act pool
act     : "sigmoid"
        | "relu"
        | "prelu"
...
```

```
Updated Production Rules

network: input backbone head
...
backbone     : block~1..10
block        : old_block
             | new_block
old_block    : conv act pool
new_block    : conv "relu" pool
act          : "sigmoid"
             | "relu"
             | "prelu"
...
```

Figure 4 – Example of preference injection. On the right, ‘block‘ has an additional expansion that explicitly instantiates the activation layer

Two important observations related to grammars and GEAs. First, the grammars used in GEAs are not required to be proper grammars; the production rules, for instance, may contain duplicates, which would technically make them lists, not sets. This duplication may be desirable, as illustrated in the previous example, where the user increased the likelihood of certain expansions. Second, the grammar is often just another hyperparameter of the algorithm; it is defined before the execution starts and is accessed as a constant whenever necessary; it is not part of any individual or population, even though the individual becomes meaningless without the grammar. This means that the grammar is not subject to mutation, breeding, or any other individual-related operation. In summary, GEAs differ from other EAs mostly in how they describe the search space, genotypes, and phenotypes, as illustrated in Figure 5.

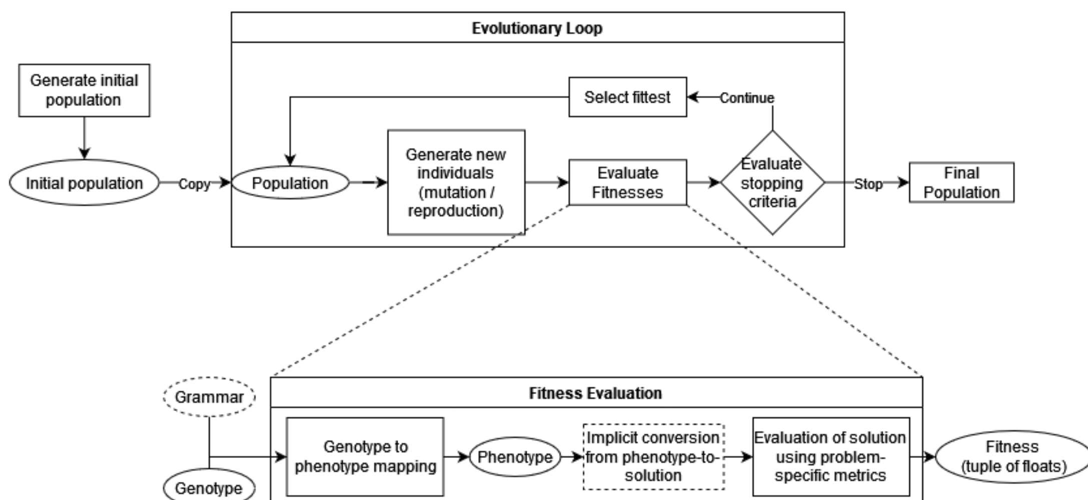


Figure 5 – Example of Generic Grammar-based Evolutionary Algorithm.

1.4.4 Neural Architecture Search

Designing and training a neural network for a specific task is not trivial. Researchers rarely discuss how many models and training sessions were required to achieve the results they ultimately publish; one of the few publications that actually mention numbers, (Ronneberger, Fischer and Brox 2015), comments that authors have submitted 78 attempts to achieve certain results. Nevertheless, it is well known from experiments that this process is expensive and tiresome, as it requires substantial computational power, domain knowledge, trial-and-error, and overall machine learning expertise.

The automated search for neural networks, often called Neural Architecture Search (NAS), is experiencing a sudden growth due to increased demand for neural network solutions, decreased compute costs, and a streamlined development process with widely used and free libraries. The field is diverse and there are multiple algorithms and approaches, but the three main components of a NAS solution are:

- search space: defines which solutions the algorithm can explore. For example, which type of layers, connections, and activation functions can be used in the model, which data augmentation mechanism can be used during training, which learning rate scheduler should be used, etc;
- search strategy: defines the method used to navigate the search space and find promising architectures. For example, the search strategy can be based on random search (Li and Talwalkar 2020), exhaustive search (Ying et al. 2019), reinforcement learning (Zoph and Le 2016), evolutionary algorithms (Sun et al. 2018), gradient-based optimization (Liu, Simonyan and Yang 2018), and so on;
- performance estimation: defines how to evaluate the quality of a network architecture without necessarily fully training and testing it. For example, the performance estimation strategy could consist of training and evaluating the model on the actual data, using a proxy task, or relying on another machine learning system designed to estimate the predictive power of the models. The performance estimation strategy can reduce the computational cost and time of NAS.

Arguably, the most important component is the search space definition, as it ultimately limits the quality of the solutions that can be found. A good search space description may synergize with the search strategy, enabling the researchers to inject domain knowledge into the NAS process by directing the search efforts to the most promising regions of the search space, and, as previously exemplified, that is one of the benefits of grammar-based algorithms.

Using evolutionary algorithms to generate or optimize neural networks is not a recent idea (Stanley et al. 2019). In 1989, for example, evolutionary algorithms were

used to optimize the weights of the connections of the neurons of neural networks; in 1990, to generate network topologies, whose weights were adjusted with a modified version of the backpropagation algorithm (Harp, Samad and Guha 1990); and, in 1992, to generate the network topologies and the weights of the connections between the neurons (Dasgupta and McGregor 1992). However, only recently, due to the previously mentioned factors, has it become feasible to employ such algorithms to generate large models for real-world applications. The use of evolutionary algorithms to perform NAS is often called neuroevolution.

1.4.5 Most Influential Works

One of the most influential works for this research is the original Grammatical Evolution (O'Neill and Ryan 2001). A deeper analysis of the genotypical representation and mapping process reveals many properties that were discussed or improved in subsequent works. For example, during the genotype-to-phenotype mapping, it is possible for a genotype to be exhausted before all non-terminals have been replaced. In such scenarios, the genotype would be effectively duplicated in a process called "genotype wrapping", allowing the mapping to continue.

An immediate consequence of this temporary[3] duplication is that the algorithm could enter infinite loops for certain combinations of genotypes and recursive production rules. Such infinite loops can be handled with ease by limiting the number of wrappings that a genotype could undergo; if this number is reached, the algorithm halts the mapping process and marks the genotype as ill-formed. Although this approach solves the infinite loop issues, it is rather limited since it occurs during mapping time, thus wasting computational resources.

The opposite scenario is also possible: the mapping process may only require a few codons from a very long genotype, rendering most of it useless. The authors (O'Neill and Ryan 2001) argue that this is desirable since it allows mutations to be neutral.

Another property of this algorithm, arguably one of the most relevant and undesirable, is that the *meaning* of a codon depends on the values of previous codons. A mutation that changes the value of the *n-th* codon could affect not only it but also the value of all subsequent codons. This is illustrated in Figure 6, where the value of the first codon changed from 0 to 1 and rendered all subsequent codons meaningless. Although not illustrated in this figure, it is possible to imagine scenarios where a single value change results in a completely different chain of expansions instead of "just disabling" the rest of the genotype.

[3] Temporary because it only lasts until the mapping process is finished.

Last but not least, the algorithm suffers from low locality (Rothlauf and Oetzel 2006). In the context of EAs, locality refers to the size of phenotypical changes relative to genotypical changes; if a small change in the genotype causes a small change in the phenotype, then the algorithm is said to have a high locality. Conversely, if a small change in the genotype causes a large change in the phenotype, the algorithm is said to have a low locality. The lower the locality, the more similar the entire search process becomes to a random search (O'Neill et al. 2004). For the Grammatical Evolution, this is exemplified in Figure 6, where a small change in the genotype led to a large change in the phenotype.

```
Production rules = [
S : AB
A : a
B : BC  (expansion 0)
  | b   (expansion 1)
C : c   (expansion 0)
  | Cc  (expansion 1)
  | ccc (expansion 2)
]

Original genotype = [0, 1, 5]
Mapped phenotype =
S
(_) -> A B
(_) -> a B
(0 mod 2 = 0) -> a B C
(1 mod 2 = 1) -> a b C
(5 mod 3 = 2) -> a b c c c

Genotype after single mutation = [1, 1, 5]
Mapped phenotype =
S
(_) -> A B
(_) -> a B
(1) -> a b
```

Figure 6 – Example of positional dependency of the original Grammatical Evolution.

Researchers proposed multiple algorithms based on, or inspired by, Grammatical Evolution. π-Grammatical Evolution (O'Neill et al. 2004) is a position-independent variation on Grammatical Evolution, where the codons no longer necessarily translate the leftmost non-terminal of the token stream. The codons, instead of encoding a single integer, now encode a pair of integers $(nont, rule)$; the *rule* serves the same purpose that the integer of Grammatical Evolution did, to choose which rule is going to be selected to transform the non-terminal; while *nont* is used to decide which non-terminal of the string

is going to be expanded. Effectively, this gives the algorithm the possibility of choosing when to expand which terminal. Based on the results of multiple experiments, the authors concluded that this was beneficial.

Later on, Structured Grammatical Evolution (SGE) (Lourenço, Pereira and Costa 2015) was proposed with two substantial modifications. First, the grammar is no longer allowed to have recursive rules, which forces the search space to be finite and, perhaps more importantly, guarantees that the algorithm will not enter infinite loops during the mapping of genotypes to phenotypes. Second, each gene is now associated with a specific non-terminal symbol, which requires a new mapping mechanism but improves mutations by reducing unintended changes to derivations of other non-terminals.

Before discussing SGE's mapping mechanism, we will first analyze its genotype, which is particularly important for this research as the genotype of the proposed algorithm, discussed in Chapter 3, is based on it. Each SGE genotype contains a fixed number of genes, exactly one for each non-terminal symbol in the grammar. Each gene is a sequence of integers, with a length equal to the maximum number of times that its non-terminal can be expanded (or mapped). Each integer present in a gene is a value between zero and the number of different expansions that the non-terminal the gene is associated with can undergo. Figure 7 demonstrates what SGE genotypes look like for a sample grammar.

- The gene associated with the non-terminal S is a list with size 1 because S can only be expanded once. The only value that can appear in the list is 0 because there is only one way to expand S;
- The gene associated with B is also a list with size 1 for the same reason: B can only be expanded once. But, since B contains three possible expansions, the values allowed in the list are $\{0, 1, 2\}$;
- The gene associated with A is the most interesting because it can be expanded twice, so its list of integers has size 2. Since A has 3 different expansions, then the values of the list are $\{0, 1\}$.

The final shape of the genotype (also called the genotype template or the genotype skeleton), thus, depends on the grammar. As the algorithm must identify how many times each symbol can be expanded, the grammar must be pre-processed before the initialization of the first population. Section 3.1 of (Lourenço, Pereira and Costa 2015) includes a detailed explanation of the pre-processing algorithm, including pseudo-codes, but the core idea is to exhaustively expand the initial symbol of the grammar using a depth-first search and counting for each symbol, what is the maximum number of times it

was expanded. This is guaranteed to terminate because the absence of recursive production rules means that there are no infinitely expandable symbols.

```
Production rules = [
S : AB
A : aa
  | a
B : Ab
  | bb
  | b
]

Genotype template
[
Gene S: [0 <= v <= 0]
Gene A: [0 <= v <= 1, 0 <= v <= 1]
Gene B: [0 <= v <= 2]
]

Sample Genotype 1
[
[0],
[1, 1],
[0]
]

Sample Genotype 2
[
[0],
[0, 1],
[2]
]
```

Figure 7 – Example of SGE Genotypes.

The SGE's mapping process is, by virtue of its more robust genotype, rather simple. As with the original Grammatical Evolution algorithm, we start with the initial symbol of the grammar and repeatedly replace the leftmost non-terminal using the production rules, but now we always consume an integer from the genotype, even when there is only a single possible expansion for a non-terminal. Figure 8 illustrates this process.

The authors of SGE also proposed a modified version of SGE called Dynamic Structured Grammatical Evolution (DSGE) (Lourenço et al. 2018) that uses a more compact genotype and handles recursive grammars by adding a hyperparameter to limit the recursion depth during runtime. All the algorithms from the DENSER family (Assunção et al. 2019, Assunção et al. 2019, Assunção et al. 2019), which are very

```
Production rules = [
S : AB
A : aa
  | a
B : Ab
  | bb
  | b
]

Sample Genotype
[S: [0], A: [1, 1], B: [0]]

First step
Starting genotype  = [S: [0], A: [1, 1], B: [0]]
Starting phenotype = [S]
Final phenotype    = [AB] (consume "0" from gene associated with S)
Final genotype     = [S: [], A: [1, 1], B: [0]]

Second step
Starting genotype  = [S: [], A: [1, 1], B: [0]]
Starting phenotype = [AB]
Final phenotype    = [aB] (consume "1" from gene associated with A)
Final genotype     = [S: [], A: [1], B: [0]]

Third step
Starting genotype  = [S: [], A: [1], B: [0]]
Starting phenotype = [aB]
Final phenotype    = [aAb] (consume "0" from gene associated with B)
Final genotype     = [S: [], A: [1], B: []]

Third step
Starting genotype  = [S: [], A: [1], B: []]
Starting phenotype = [aAb]
Final phenotype    = [aab] (consume "1" from gene associated with a)
Final genotype     = [S: [], A: [], B: []]
```

Figure 8 – Example of SGE Mapping Process.

influential to this research, employ DSGE. Its genotype and translation mechanisms are substantially more complex than SGE's, and it is arguable if its potential benefits outweigh such increased complexity, especially in the context of neuroevolutionary algorithms. For example, the memory saved by having a more compact genotype is orders of magnitude smaller than any decently sized phenotype (a neural network); handling recursion during runtime, instead of during a pre-processing phase may allow experiments to write more elegant grammars but at the cost of making it harder to reason about since the number of expansions is no longer self-evident. Finally, there is also an increased implementation

cost; the simplicity of SGE makes it easier to implement, test, and modify experiments compared to DSGE. For those reasons, SGE's genotype, instead of DSGE, was chosen as the genotypical base of the algorithm presented in this thesis, described in Chapter 3.

It is important to emphasize that DSGE, SGE, and GE still rely on the same evolutionary loop and principles: a grammar is used to describe the search space, and phenotypes are generated by repeatedly transforming the leftmost non-terminal of a token stream using genotypes to determine which production rules should be used. This changes with Deep Evolutionary Network Structured Representation (DENSER) (Assunção et al. 2019), which combines DSGE with a Genetic Algorithm (GA) and was proposed specifically as a neuroevolutionary algorithm.

DENSER's genotype has two levels, the outer one is manipulated by the GA component of the algorithm and determines the high-level structures of the model; it can be viewed as a list of 3-tuples, each containing a non-terminal and two integers, such as $[(features, 1, 10), (classification, 1, 2), (softmax, 1, 1)]$. The symbol of a tuple indicates a starting symbol for the grammar used by DSGE, while the integers indicate the minimum and the maximum number of times they can be used.

The inner level of the DENSER's genotype is manipulated by the DSGE component of the algorithm and is used to determine low-level information of the model, such as the number of kernels in the convolution layer and the size of such kernels. The DSGE used in DENSER is a slightly modified version that handles continuous values; instead of having the grammar designer enumerate all possible values that a continuous variable can assume, it allows the designer to just type the minimum and maximum values.

It is unclear what the benefits of having such a two-level approach are and if they outweigh the added complexity. The authors claim that "The novelty of DENSER relies on the combination of these two levels. Without the GA level, it would not be possible to encapsulate the genetic material, which facilitates the application of the genetic operators, and thus eases evolution.", but it appears that the same structure could be perfectly represented using DSGE, or SGE, by adding an appropriate initial rule to the grammar, such as `S: features~1..10 classification~1..2 softmax`.

The authors conducted experiments using multiple strategies, such as using different learning rate policies, modifying the number of epochs, and including test-time data augmentation. The results were extremely promising and demonstrated that grammar-based methods could generate competitive models.

Fast DENSER (F-DENSER) (Assunção et al. 2019) is a modification of DENSER that employs a "$(1 + \lambda)$ Evolutionary Strategy" to reduce the number of individuals that have their fitnesses evaluated each generation. More interestingly, F-DENSER also modifies the DENSER genotype to enable the creation of skip connections, but only between the

elements of the genotype's outer level.

F-DENSER handles mismatches between tensor shapes (when it merges the output of multiple layers) by downsampling all tensors to the smallest tensor size, as seen in its source code[4]. This limited merging strategy disallows the generation of architectures like the U-Net (Ronneberger, Fischer and Brox 2015). Lastly, the only strategy implemented to merge multiple layers after they have been adjusted to have the same size is the depth-wise concatenation.

The population initialization has also been modified based on observations of DENSER experiments: the networks generated during initialization had random sizes, but after a few generations, the average model size tended to decrease; so the authors of F-DENSER added a constraint to the initialization phase to limit the maximum network sizes to try and skip this "model shrinking phase" of the evolution. This idea is very interesting and has been incorporated into MOREFUN, the algorithm presented in this thesis and discussed in Chapter 3.

Experimentally, the authors observed a substantial reduction in run-time, likely related to the reduced number of individuals trained, competitive results, and, on average, smaller networks, likely related to the addition of skip connections.

Fast Denser++ (Assunção et al. 2019) modifies F-DENSER in a very interesting way: the authors added a value to the genotype to determine for how long the model should be trained; such is usually a hyperparameter, but in this context, it becomes susceptible to mutations, which means that some models may be trained for longer than others. A notable consequence of this increase in train time is that the final output of the algorithm is a collection of fully trained models. Experimental results suggest that the individuals generated by F-Denser++, without further training, are able to outperform those generated by F-Denser.

Finally, the last work that is particularly influential to this research is (Deb et al. 2002), which presents the Nondominated Sorting Genetic Algorithm II (NSGA-II). This is a relatively old but still very relevant multi-objective evolutionary algorithm that uses the Pareto approach. During evolution, the "working set" of solutions increases due to mutation and reproduction, so the algorithm must prune it to the original set. This pruning is often called fittest selection, as NSGA-II does this by sorting the candidate solutions into Pareto ranks, that is, sets of solutions that are non-dominated. After sorting the solutions, each rank is added into the pruned set P, which is initially empty, until the point where adding another rank would make it larger than the target size t (which usually is the size of the population during the beginning of the cycle). Usually, the situation is such that adding the entire rank would make the set have more than t

[4] The implementation of the algorithm is publicly available at: <https://github.com/fillassuncao/fast-denser>

elements, but not adding it would make it have fewer than t, so a subset of the rank must be selected. NSGA-II selects this subset of solutions by sorting them using a density-based metric: individuals in less-dense regions of the objective space are preferred over those in more-dense regions. The idea is that such individuals are located in a less-explored region of the search space, and preserving them would lead to a potentially more diverse population.

In this chapter, the fundamental concepts that support the research hypothesis were presented: that grammar-based neuroevolutionary algorithms can be enhanced with multi-objective optimization to generate smaller, more efficient neural network models while simultaneously accelerating the search process. These concepts included multi-objective optimization approaches, evolutionary algorithms, formal languages and grammar-based representations, and neural architecture search methodologies. The experimental results demonstrated that runs incorporating multi-objective optimization required substantially less computational time while producing smaller but equally effective models, which provided evidence supporting the central hypothesis. The following chapters will cover the literature review, the proposed MOREFUN algorithm, the experiments performed for its publication and this thesis, and the final conclusion.

2 Literature Review

This Chapter is dedicated to related works that, although not as fundamental as those discussed in Chapter 1, are still relevant to this research. The first section of this chapter contains works that were identified by the Snowball method (Wohlin 2014), while the second section contains the results of a search that employs the mechanism of a systematic review.

2.1 Related works found via Snowball

The neural architecture search works can be grouped by the search strategy employed by the authors: manual search and exhaustive search, reinforcement learning (RL), gradient-based, or evolutionary (EA). Each sub-section is dedicated to one such group.

2.1.1 Works based on manual or exhaustive search

The works discussed in this section employ the simplest search strategies, such as exhaustive search and manual search. It is important to emphasize that search-related information for the manually designed models is scarce, as researchers rarely disclose information about unsuccessful models or the search process itself. Nevertheless, it is common knowledge that such a process is tedious, resource-intensive, and error-prone (Ronneberger, Fischer and Brox 2015).

One of the most well-known manually designed models is the VGG (Simonyan and Zisserman 2014). The architecture is rather simple, consisting of a stack of convolutional blocks, each being a sequence of two or three convolution layers, all with kernel size=3 and ReLU activation, followed by a max-pooling layer. The head of the model consisted of three fully connected layers. When released, the model won first place in ILSVRC, set a new state-of-the-art for the ImageNet dataset, and demonstrated that deeper representations are beneficial for classification accuracy. The authors included in the manuscript a link to a reference implementation of the algorithm[1].

In (He et al. 2016), the authors advanced the trend of increasing network depth through the introduction of ResNet. Deeper networks were substantially harder to train due to the vanishing gradient problem, which the authors addressed using residual blocks. Such blocks contained "residual connections" (also referred to as skip connections or shortcut connections), which perform identity mapping to propagate untransformed data from

[1] <https://www.robots.ox.ac.uk/~vgg/research/very_deep/>

earlier layers to later layers. ResNets won first place in ILSVRC and set a new state-of-the-art for the ImageNet dataset. The authors did not include a reference implementation of the algorithm in the manuscript.

In (Huang et al. 2017), the authors presented DenseNets by taking the idea of ResNets to the extreme and adding skip connections from each layer to every other layer that had the same feature-map size. Upon publication, the model set the state-of-the-art for multiple datasets and required less computation. The authors included in the manuscript a link to a reference implementation of the algorithm[2].

In (Sandler et al. 2018), the authors presented MobileNetV2. As the name suggests, the work focuses on reduced computational cost to enable model deployment to mobile devices. The authors attribute the excellent performance of the model to two key contributions: linear bottlenecks, which involve removing the non-linear activation function from the end of bottleneck blocks; and inverted residual connections, which involve rewriting residual connections to add skip connections between the bottlenecks instead of between expansions. The authors included in the manuscript a link to a reference implementation of the algorithm[3].

One of the most famous neural architecture search works is (Ying et al. 2019), where the authors introduce a dataset of neural network architectures called NAS-Bench-101. The dataset was created by an exhaustive search of a carefully designed search space comprising 423,624 architectures. Each model was fully trained and evaluated, with properties and metrics such as topology, test accuracy, and training time recorded. The authors made the dataset publicly available to enable other researchers to experiment with NAS without having to train the models. The authors report that the generation of the dataset required over 100 TPU years.

Although NAS-Bench-101 appears to be a valuable resource, its search space has limited overlap with that of MOREFUN (the algorithm proposed by this research), rendering it ineffective for query-based acceleration. This mismatch is caused by two key features of MOREFUN: the incorporation of upscaling layers and the frequent use of skip connections.

2.1.2 Works based on reinforcement learning

Researchers have also explored neuroevolutionary algorithms via reinforcement learning. One of the earliest and most computationally expensive of such works is (Zoph and Le 2016), where the authors presented a named Neural Architecture Search (NAS). The authors used a Recurrent Neural Network (RNN) to design networks, which are then trained and evaluated, with the validation accuracy of the networks used

[2] <https://github.com/liuzhuang13/DenseNet>
[3] <https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet>

as reward signals for the RNN controller. The proposed algorithm generated models that achieved a 3.65% test error rate on CIFAR-10 after running for 28 days on 800 GPUs (Pham et al. 2018). The authors made part of their implementation publicly available, adding their RNN controller to the TensorFlow library[4], but I could not find the reference implementation of the entire algorithm.

In (Pham et al. 2018), the authors proposed ENAS, an algorithm three orders of magnitude less computationally expensive than NAS (Zoph and Le 2016). The main idea of ENAS is to construct a large computational graph to represent the search space, where each subgraph represents a neural network. This representation allows the sharing of model weights, drastically reducing the time required to train the models. ENAS required just 0.5 GPU days and generated models that achieved a 2.89% error rate on CIFAR-10. The authors did not include a reference implementation of the algorithm in their manuscript, but at the time of writing of this thesis, it is possible to find their reference implementation[5] via Google Search; it is unclear if the implementation was made available immediately after the publication or only years later.

In (Ding et al. 2021), the authors proposed BNAS, which is based on ENAS. The new algorithm replaces the architecture used in ENAS with a *Broad Scalable Architecture* (BCNN), which reduces the runtime from 0.5 GPU days to 0.1. The generated models have a 2.67% test error rate on CIFAR-10. The authors did not include in the manuscript a link to a reference implementation of the algorithm.

In (Zoph et al. 2018), the authors proposed NASNet, which is built on top of NAS (Zoph and Le 2016). The authors reason that it is more efficient to search a good "cell" and repeatedly stack it to create an architecture than to search for complete models, and that such cells could also be transferred to different computer vision datasets. Their experiments suggest that this is true, as the models they generated set the new state-of-the-art for CIFAR-10 with a 2.4% test error rate after running for 4 days on 500 GPUs, which is less than the original Neural Architecture Search but still orders of magnitude more than ENAS. The authors did not include in the manuscript a link to a reference implementation of the algorithm.

2.1.3 Works based on gradient

Another popular type of search strategy is gradient-based, where the algorithms navigate the search space by trying to minimize some metrics using gradient descent. One of the earliest and most influential of such works is (Liu, Simonyan and Yang 2018), where the authors presented DARTS. Using continuous relaxation, DARTS treats the search space not as a set of discrete architectures but as a weighted mixture of candidate

[4] <https://github.com/tensorflow/models>
[5] <https://github.com/google-research/google-research/tree/master/enas_lm>

operations, and this allows it to search for the optimal architecture via gradient descent. The algorithm achieved results comparable with the state-of-the-art and required from 1.5 to 4 GPU days, depending on the settings. The authors included in the manuscript a link to their reference implementation[6].

In (Chen et al. 2019), the authors proposed P-DARTS as a potential solution to a recurring issue with DARTS-based algorithms called "depth gap". This problem arises from a mismatch between the depth of architectures during the search phase and the evaluation phase; cells that performed well in shallow architectures may underperform when used in deeper architectures. P-DARTS addresses this by dividing the search phase into multiple stages, progressively increasing the network depth at the end of each stage. Additionally, it incorporates search space regularization, a technique that gradually increases the "weight" of skip-connections during the search to discourage their overuse in the early stages, as they tended to be chosen over other operations. P-DARTS generated models that set the new state-of-the-art on CIFAR-10 with a 2.5% test error rate after running for 0.3 days on a single GPU. The authors included in the manuscript a link to a reference implementation of the algorithm[7].

There are many other DARTS-based algorithms, such as Fair-DARTS (Chu et al. 2020), SDARTS-ADV (Chen and Hsieh 2020), U-DARTS (Huang et al. 2023), β-DARTS (Ye et al. 2022), and Shapley-NAS (Xiao et al. 2022)[8]. Each identifies a potential improvement point for DARTS and explores a promising solution, but the results are not substantially different (see Table 4 in Chapter 4): they all require less than a GPU day to run (except U-Darts, which requires 4 GPU days) and they generate models with approximately 3.3 million parameters that achieve approximately a 2.5% test error rate on CIFAR-10.

In (Dong and Yang 2019), the authors proposed GDAS, which views the search space as a directed acyclic graph containing billions of sub-graphs, each representing a neural network. The algorithm then, similarly to DARTS, uses a differentiable sampler to find models via gradient descent. Results were competitive with the state-of-the-art, and the algorithm could be executed in four hours on a single GPU. The authors included in the manuscript a link to their reference implementation[9].

In (Zhou, Xie and Kung 2021), the authors proposed EoiNAS, which is built on top of GDAS. The authors highlight that architecture weight (used during search) does not necessarily reflect the importance of each operation and propose a new indicator named Exploiting Operation Importance for effective Neural Architecture Search (EoiNAS), and

[6] <https://github.com/quark0/darts>
[7] <https://github.com/chenxin061/pdarts>
[8] Important observation: all of the darts-derived works have included links to reference implementations in the manuscript.
[9] <https://github.com/D-X-Y/GDAS>

a new pruning strategy based on said indicator to guide the search. The algorithm required 0.6 GPU days and generated a model that achieved a 2.50% test error rate on CIFAR-10. The authors did not include in the manuscript a link to a reference implementation of the algorithm.

In (Xie et al. 2019), the authors presented SNAS and introduced a novel search gradient that optimizes the same objective as reinforcement-learning-based models but does so more efficiently by assigning credit to structural decisions. Models generated by SNAS achieved a 2.85% test error rate on the CIFAR-10 dataset after training for 1.5 days on a single GPU. The authors included a link to their reference implementation in the manuscript[10].

2.1.4 Works based on evolutionary search

The last search strategy is evolutionary. As mentioned in Chapter 1, the idea of using evolutionary algorithms to generate neural networks is not new, but only recently has it become technologically viable to use them to generate large networks. One of the earliest such examples is (Xie and Yuille 2017), where the authors describe Genetic CNN. The algorithm uses a fixed-size binary string to encode the network structures, which is inflexible and difficult to work with, as noted in the “limitations” section of the paper. Additionally, the algorithm relies mainly on hard-coded parameters rather than encoding them in the genotype; e.g., all convolutions have the same number of filters, all kernels have the same size, and tensors in a “network blocks” have the same number of channels. After running for 17 GPU days, the algorithm evolved models that achieved a 7.10% test error rate on the CIFAR-10. The authors did not include a reference implementation of the algorithm in the manuscript.

In (Real et al. 2019), the authors present AmoebaNet, an evolutionary algorithm that operates on the same search space as NASNet (Zoph et al. 2018). One of the key features of the algorithm is the concept of aging evolution, which favors younger genotypes during the tournament selection to prevent premature convergence. The algorithm generated models that achieved a 2.55% test error rate on CIFAR-10, but its runtime was 3.75 years longer than NASNet, that is, it required 3150 GPU days to reduce the test error rate by 0.1%. The authors included in the manuscript a link to a reference implementation of the algorithm[11].

In (Liu et al. 2018), the authors presented Hierarchical Evolution, which describes models as a hierarchy of components, with the top-most element representing the entire architecture and the bottom-most representing fundamental operations, such as convolutions and pooling. A particularly interesting contribution of the paper is that the

[10] <https://github.com/SNAS-Series/SNAS-Series>
[11] <https://tfhub.dev/google/imagenet/amoebanet_a_n18_f448/classification/1>

authors also conducted experiments using random search and achieved similar results, and concluded that a trivial search strategy coupled with a carefully designed search space can yield strong results. The algorithm required 300 GPU days to generate models that achieved a 3.75% test error rate on CIFAR-10. The authors did not include a reference implementation of the algorithm in the manuscript.

In (Yang et al. 2020), the authors presented CARS. The authors observed that the high computational cost traditionally associated with evolutionary algorithms was caused by the frequent training of models and proposed weight sharing via SuperNet as a potential solution. The algorithm, via a modified NSGA-III, also performs multi-objective optimization to evolve models with smaller sizes. After running for 0.4 GPU days, the algorithm generated models that achieved a 2.81% test error rate on CIFAR-10. The authors included in the manuscript a link to a reference implementation of the algorithm[12].

In (Sinha and Chen 2021), the authors proposed EvNAS, which uses a weight-sharing strategy similar to CARS, although the older algorithm is not referenced in the manuscript. After running for 3.83 GPU days, the algorithm generated models that achieved a 2.52% test error rate on CIFAR-10. The authors highlight an interesting benefit of evolutionary algorithms when compared to gradient-based: EAs do not have to be explicitly engineered to mitigate "the skip connection issue" that affects many gradient-based algorithms, as the skip connections do not affect the model fitness as much as they affect the gradients. The authors did not include a reference implementation of the algorithm in the manuscript.

In (Xue et al. 2021), the authors described MOGIG-Net, a multi-objective neuroevolutionary algorithm that the authors claim to have a reduced runtime due to weight sharing, implemented as "ad-hoc crossover and mutation operators". The authors do not report the actual runtime of the algorithm (just that "The results in this study have been detected after two weeks of calculation.") nor basic information required for reproducibility, such as which optimizer was used to train the networks or the learning rate regime. They also did not include a reference implementation of the algorithm in the manuscript. Despite these shortcomings, the paper is included in the experimental comparisons of Chapter 4 because it has been compared with many publications.

In (Xue, Chen and Słowik 2023), the authors described MOEA-PS, another multi-objective neuroevolutionary algorithm with reduced runtime. The main contributions are the modification of the NSGA-II with the introduction of a queue to store good solutions (similar to elitist strategies of a traditional evolutionary algorithm) and the usage of "probability selectors" that indicate which known blocks should be used to create a new model. The algorithm required 2.6 GPU days and generated models that achieved a 2.76% test error rate on CIFAR-10. The authors did not include a reference implementation in

[12] <https://github.com/zhaohui-yang/CARS>

the manuscript.

In (Dong et al. 2022), the authors described MSNAS, a genetic algorithm that partitions the search space into convolution space and pooling space, with encoding and crossover operations adapted to this new space. The algorithm required 0.23 GPU days and generated models that achieved a 2.68% test error rate on CIFAR-10, but it is interesting to observe that the authors halved the image sizes and used only 50% of the training dataset for training, which makes the runtime less impressive, but the test error rate substantially more impressive. The authors did not include a reference implementation in the manuscript.

2.2 Related works found using systematic review methods

Figure 9 shows the search string used to identify papers describing novel neural architecture search algorithms.

```
"neural architecture search"
OR "nas"
OR "neuro*evolution*"
OR (("automl" OR "auto ml" OR "auto-ml")
    AND ("neural network*" OR "convolutional network*")
   )
OR (("grammatical evolution*"
        OR "evolutionary computing"
        OR "genetic algorithm*"
        OR "genetic programming")
     AND ("neural network*" OR "convolutional network*")
   )
```

Figure 9 – Search string used in the literature review.

On 2024/01/14, the search string was entered on Scopus, which yielded 14654 results, and on ScienceDirect, which rejected it and similar alternatives due to the number of boolean connectors (max 8), as shown in Figure 10. Given the problems with ScienceDirect, Scopus served as the sole data source for the results presented here.

I filtered the results by subject area “Computer Science”; language “English”; publication date “between 2016 and 2023, inclusive”; number of citations > 10. Finally, I de-duplicated DOIs, which resulted in a set of 809 papers.

The boundary dates were chosen to include Zoph’s seminal work (Zoph and Le 2016) and to exclude papers that are too recent to have been thoroughly discussed by the

Title, abstract or author-specified keywords

```
"neural architecture search"
OR "nas"
OR (("automl" OR "auto ml" OR "auto-ml")
   AND ("neural network" OR "convolutional network")
  )
OR (("grammatical evolution" OR "evolutionary computing" OR "genetic
algorithm" OR "genetic programming")
   AND ("neural network" OR "convolutional network")
  )
```

Use fewer boolean connectors (max 8 per field)

Figure 10 – ScienceDirect search error.

community. The minimum number of citations was used as a proxy for the value assigned to that work by the community, and the numeric value 10 was chosen arbitrarily.

After filtering the initial results, the 809 abstracts were downloaded, read, and filtered again to keep all the papers that "describe a novel neural architecture search" and exclude all papers that "simply apply a known algorithm to a new dataset". To minimize the risk of excluding relevant papers, whenever the abstract alone was insufficient to determine if the paper described a novel NAS algorithm, it was kept. This yielded a reduced list of 554 papers.

Finally, all the manuscripts were downloaded and classified as "reproducible" and "not reproducible" using the following criteria, ordered by priority:

- if all experiments were conducted in private datasets, the paper was considered "not reproducible";
- if the paper included a link to their reference implementation, then it was considered "reproducible";
- everything else was considered "not reproducible";

This resulted in a list of 49 papers, which are discussed in the following sections.

In (Kordmahalleh et al. 2017), the authors presented an algorithm that evolves hierarchical recurrent neural networks using a genetic algorithm, specifically for the identification of time-delayed gene regulatory networks. The evolutionary component of the algorithm is used to evolve the architecture of the model and the weights.

In (Rapaport, Shriki and Puzis 2019), the authors presented EEGNAS, a GA that evolves convolutional neural networks (CNNs) with up to 10 layers. The algorithm was designed specifically for the domain of electroencephalography, but it has interesting characteristics. First, the search space description was not particularly precise, as it included multiple types of invalid architectures or invalid components, such as convolution

layers with kernel sizes larger than their inputs, which the algorithm handled by discarding the architecture and generating a replacement. The algorithm also did not prevent the generation of duplicate architectures; instead, it used a dynamic mutation rate that increased when less than 70% of the population was architecturally unique. The algorithm recognized multiple objectives but optimized only one per run. Finally, one of the most interesting contributions of the paper, from a neural architecture search perspective, was the set of experiments they conducted with weight sharing: the authors described experiments with three weight-sharing strategies (one being the null strategy, i.e., no weight sharing) and demonstrated that for their domain and search space, weight sharing could reduce the average train time of a network from 30 seconds to just 10 seconds. The experiments were conducted on a single GPU and required at most a day to run.

In (Véniat, Schwander and Denoyer 2019), the authors presented SANAS, an algorithm that focuses on runtime model selection instead of the traditional train-time model selection. The algorithm was developed for keyword spotting in real-time audio streams, where latency is particularly important. The main idea is to use a smaller and faster model when the stream is easier to process (e.g., with less noise) and a larger, slower model when the stream is harder to process. The algorithm trains a single HyperNet with a sampling strategy to choose which nodes and paths (i.e., subnetwork) should be used, and this sampling mechanism is what allows the algorithm to choose different model sizes and architectures during runtime. The authors describe the train time as approximately 1 GPU day.

In (Weng et al. 2019), the authors presented NAS-Unet, an algorithm that implements a search strategy similar to DARTS but optimized for the generation of U-Net-like (Ronneberger, Fischer and Brox 2015) models, which are known to perform well in semantic segmentation tasks. The algorithm describes three types of cells: downsampling, upsampling, and normal, and the algorithm searches the space of networks that have an equal amount of upsampling and downsampling cells, which guarantees the formation of U-shaped models. The authors conducted experiments on multiple semantic segmentation datasets, each requiring a different amount of GPU time but no more than 1.5 GPU days.

In (Weng et al. 2019), some of the authors of NAS-Unet proposed a similar algorithm named CNAS, which evolves convolutional neural networks for image classification using a DARTS-based search mechanism instead of U-Nets for semantic segmentation. The experiments required less than 3 GPU days and generated models that achieved a 4.23% test error rate on CIFAR-10.

In (Jin, Song and Hu 2019), the authors presented Auto-Keras, a framework to perform AutoML, similar to WEKA and Auto-Sklearn, but focused on deep neural networks. The goal of the authors was to create a free, easy-to-use, and local alternative to cloud-based AutoML services. They demonstrated the efficacy of the framework in multiple

datasets, most notably on CIFAR-10, where they were able to generate models with a 3.60% error rate after 0.5 GPU days.

In (Balaprakash et al. 2019), the authors proposed a NAS algorithm designed specifically for non-image non-text cancer data. The search strategy was based on reinforcement learning, and the search space was described using a graph. The work was focused on the scalability of the proposed solution and describes runs with multiple supercomputer configurations. The generated models were substantially smaller and faster to train than the state-of-the-art manually designed models, with similar predictive power.

In (Tian et al. 2020), the authors presented E^2GAN, an algorithm that searches architectures of Generative Adversarial Networks (GANs) by combining a Markov Decision Process (MDP) view of the optimization problem with an off-policy reinforcement learning search mechanism. The experiment ran for 0.3 GPU hours and generated highly competitive models with the state-of-the-art.

In (Ding et al. 2020), the authors proposed AutoSpeech, a DARTS-based algorithm that searches convolutional neural networks for speaker recognition tasks. The experiment required 5 GPU days and generated models that outperformed VGG and ResNet; models that the authors describe as being developed for image classification tasks.

In (Chu et al. 2020), the authors proposed FairDARTS, which was previously described in Section 2.1.3 as one of the multiple DARTS-based algorithms that search architectures for computer vision tasks and achieve similar results to other DARTS-based algorithms.

In (Lu et al. 2020), the authors presented the algorithm MSuNAS and the models it evolved, which were named NSGANetV2. The algorithm searched both the architecture of the models and their weights, using one surrogate for each [13]. The architecture surrogate was trained online, and the weights surrogate (and SuperNet) offline. The algorithm also performed multi-objective optimization using NSGA-II[14]. The authors claim that the algorithm generated models that were competitive with the state-of-the-art and could be run 20x faster, but they did not include the time required to train the SuperNet in the comparisons.

In (Byla and Pang 2020), the authors presented DeepSwarm, a NAS algorithm based on ant colony optimization (Dorigo and Gambardella 1997), a type of evolutionary algorithm inspired by how ants explore the ambient and map their exploration using pheromones. The paper is well written, and the authors made their code publicly available, but they do not discuss many important aspects of their NAS implementation, such as

[13] Surrogates, in the context of NAS, refer to mechanisms that can be used to estimate the performance of the models being evolved or to compare them; for example, one could train a Decision Tree to estimate the test error rate of a neural network based on its architecture and use the output of this Decision Tree as the fitness of the network instead of training it.

[14] Interestingly, one of the authors of NSGA-II is also an author of this work.

the search space being explored, besides mentioning that it evolves convolutional neural networks for image classification and employs weight re-usability to decrease costs. After running for 1.25 GPU days, it generated models that achieved an 11.31% test error rate on CIFAR-10.

In (Stamoulis et al. 2019), the authors presented Single-Path NAS, an efficient gradient-based NAS algorithm. The authors claim that it runs up to 5000 times faster than comparable previous algorithms. The key innovation lies in the design of the search space. Traditionally, a large supernet containing multiple paths was used, but the proposed algorithm views it as a single-path “nested” supernet. For instance, instead of having separate nodes for convolutional layers with kernel sizes 3, 5, and 7, the proposed approach uses a single convolutional layer with a kernel size of 7, where the 5 and 3 versions are “cropped” from the larger kernel. They did not experiment on CIFAR-10, but their results on ImageNet show that the algorithm achieves state-of-the-art results, for mobile models (i.e., reduced size), after running for just 0.15 GPU days.

In (Guo et al. 2020), the authors presented HNAS, an algorithm designed to evolve hierarchical networks for supersampling, that is, reconstructing high-resolution images from lower-resolution inputs. The search space is relatively simple, consisting of networks composed of a stack of upsampling and “normal” blocks. The authors used a recurrent neural network to design the models and a joint reward that combined the quality of the models and their computational cost, effectively transforming a multi-objective problem into a single-objective one. The authors did not disclose how much time was required to run the algorithm, but the generated models achieved results comparable to the state-of-the-art.

In (Gonzalez and Miikkulainen 2020), the authors presented GLO, an algorithm that does not perform the traditional neural architecture search; instead, it focuses on the evolution of loss functions. The authors conducted experiments comparing the generated loss functions with the traditional cross-entropy and demonstrated that it is superior in terms of accuracy, training speed, and data requirements.

In (Yao et al. 2020), the authors presented an algorithm that performs a very targeted neural architecture search, evolving models that can be used to replace “interaction functions” that are used, for example, in recommender systems. The algorithm evolves both the model architecture and the weights, but the search space and search strategy are specialized for the task of evolving alternative “interaction functions”.

In (Yan et al. 2020), the authors presented a DARTS-based algorithm that searches for neural network cells for the task of magnetic resonance imaging reconstruction. The algorithm yielded promising results, but from a NAS perspective, it is highly specialized and not particularly novel.

In (Liu et al. 2020), the authors presented AutoFIS, a DARTS-inspired algorithm

that searches for the most useful "feature representations" for factorization models used in recommender systems. The algorithm operates on a fixed neural architecture that contains "gates" that allow or inhibit the propagation of certain features, and the search process is focused on learning which gates must be open or closed. Since the number of gates is large, the authors employ the relaxation mechanism of DARTS to view the search space as continuous instead of discrete. The authors conducted experiments with public and private datasets and ran the algorithm live on the Huawei App Store, where it improved the model being used by approximately 20%.

In (Chu, Zhang and Xu 2020), the authors described MoGA. During a deeper examination of the code provided by the authors, it became clear that it can only be used to evaluate the models found by the algorithm, not to run the algorithm itself. As such, this paper does not meet the reproducibility criteria previously described.

In (Sun et al. 2019), the authors presented EvoCNN, a simple genetic algorithm that evolves sequential convolutional networks similar to VGG-16. This is a complex case for the "reproducible or not" question, as the code provided by the authors is broken, calling functions that do not exist and passing invalid arguments to functions that do exist. Ultimately, it was kept because the code, although broken, can at least be used by researchers to study implementation details.

In (Marchisio et al. 2020), the authors presented NASCaps, a multi-objective neuroevolutionary algorithm designed to search for Capsule Networks (Sabour, Frosst and Hinton 2017) for IoT devices. The algorithm is a modified version of NSGA-II, and it optimizes the accuracy, energy consumption, memory consumption, and latency of the models. The hardware-related metrics are modeled by the algorithm, while the accuracy is estimated from a training session with a reduced number of epochs. The results were promising, but an inspection[15] of the source code suggests that the algorithm used the test accuracy, not the train or validation, during evolution, resulting in data leakage[16].

In (Wang et al. 2020), the authors presented AutoREC, an AutoML open-source framework designed to simplify the generation of recommender systems. It is built on top of AutoKERAS. The 3-page short paper describes the system, illustrates how to use the framework, and presents results of experiments on subsets of benchmark datasets.

In (Fang et al. 2020), the authors presented FNA++, a NAS algorithm that searches for semantic segmentation and object detection using modified backbones that were previously developed for image classification. The algorithm modifies the backbones via parameter remapping, a technique that allows a trained network component, such as a convolutional layer, to be recreated as a larger version of itself while keeping the trained

[15] See lines 1044, 581, 652, and 825 at <https://github.com/ehw-fit/nascaps/blob/master/nsga/main.py>
[16] <https://www.ibm.com/think/topics/data-leakage-machine-learning>

weights, achieving an effect similar to traditional weight sharing. Experiments demonstrated that the algorithm is orders of magnitude faster than other semantic segmentation NAS approaches.

In (Zhang et al. 2020), the authors tackled the problem of catastrophic forgetting that affects supernet-based NAS algorithms in certain conditions. The key contribution of the paper is NSAS, a loss function devised to mitigate the problem. They conducted experiments using RandomNAS (Li and Talwalkar 2020) and GDAS (Dong and Yang 2019) with the proposed loss function, which demonstrates that the proposed loss function improves results but does not solve the problem entirely, suggesting that this is a promising approach for future research.

In (Zhang et al. 2020), the authors presented DSO-NAS, another gradient-based algorithm that, similarly to DARTS, uses relaxation to treat the discrete search space as continuous. After running for 0.8 GPU days, the algorithm produced networks that achieved a 2.74% test error rate on CIFAR-10. The overall algorithm and results are very similar to other DARTS-based or DARTS-inspired algorithms.

In (Zheng et al. 2021), the authors described a framework for neural architecture search and a novel optimization strategy named MIGO. Similarly to (Chu, Zhang and Xu 2020), upon further inspection, it became clear that the paper does not actually meet the reproducibility criteria. The authors claimed the code is publicly available and included a URL to the reference implementation in their manuscript, but important parts of the algorithm were not shared, and the authors do not respond to requests for sharing such parts: <https://github.com/MAC-AutoML/XNAS/issues/17>.

In (Balaha, Balaha and Ali 2021), the authors presented a complex 9-phase system for the classification of COVID-19 images. The overall pipeline consists of identifying regions of interest, cropping them, feeding them to an ensemble of models, and using the majority class of the ensemble as the system prediction. The relevant phase, from an AutoML perspective, is the optimization of hyperparameters of the networks using a GA, but there is no novelty. The algorithm performed similarly to other algorithms it was compared with, achieving from 98% to 100% accuracy.

In (Ding et al. 2021), the authors presented DiffMG, a NAS algorithm designed for heterogeneous graph neural networks, that is, networks used to model problems that can be described by graphs that contain different types of nodes. Applying DARTS to this type of network is particularly expensive because it requires simulating the message passing between each edge in the graph. The key contribution of the algorithm is avoiding the expensive computations associated with the message passing. The proposed algorithm achieved the best results for node classification and recommendation tasks.

In (Termritthikun et al. 2021), the authors presented EEEA-Net, a multi-objective

neuroevolutionary algorithm that leverages a mechanism named "early exit", which effectively simply discards networks that have more trainable parameters than a predefined threshold, to generate models that achieved a 2.46% test error rate on CIFAR-10, after running for 0.52 days. The author's experimental results show the algorithm dominating (in the Pareto sense) most of the state-of-the-art algorithms. A cursory inspection of the source code [17] suggests that the experiments may suffer from data leakage, as the authors used the test accuracy during evolution.

In (Zimmer, Lindauer and Hutter 2021), the authors presented Auto-Pytorch, which, similarly to AutoKeras, is an open-source framework for NAS but built on top of Pytorch instead of TensorFlow. The framework achieved state-of-the-art performance on several tabular benchmarks, but it can also be used for computer vision tasks, where it achieved results comparable to the state-of-the-art.

In (Lu et al. 2021), the authors described NAT. Upon a deeper analysis of the repository linked in the manuscript, it became clear that the work does not satisfy the reproducibility criteria, as the repository does not contain the reference implementation of the described algorithm, only the code required to evaluate the fully-trained models [18].

In (Liu et al. 2021), the authors presented AdaLSN, a neural architecture search designed to evolve models for skeleton detection. The search mechanism is rather simple, relying on a genetic algorithm, but the search space and genetic encoding are highly specific. The model achieved state-of-the-art results for skeleton detection and competitive results in other image-to-mask tasks, such as edge detection and road extraction.

In (Yu et al. 2021), the authors presented a new type of 3D convolution, named 3D-CDC, to efficiently learn spatio-temporal features and a NAS method to search for multi-rate and multi-modal networks. The NAS component is a two-phase algorithm, both based on DARTS; the first phase searches for cells to construct the backbone of the network, and the second phase searches for the multi-rate and multi-modal components based on the backbones found. The proposed algorithm achieved state-of-the-art results on the three datasets the authors used for the experiments and required 48 days.

In (Wang et al. 2021), the authors presented BiX-NAS, a NAS algorithm designed to generate models for semantic segmentation, with architectures that resemble the U-Net (Ronneberger, Fischer and Brox 2015). The algorithm has two phases: the first phase is gradient-based and searches for good architectural components, while the second phase is evolutionary-based and tries to optimize the architectures by exploring different skip connection arrangements. The proposed algorithm achieved the best results for all datasets

[17] Lines 84 and 93: <https://github.com/chakkritte/EEEA-Net/blob/58ba0720b7e481e5421929d70e5345a3dbd3d4ef/EEEA/cifar/eeea/data/datasets.py>

[18] The authors ignore requests for the release of the reference implementation: <https://github.com/human-analysis/neural-architecture-transfer/issues/1>

and algorithms the authors experimented with.

In (McNally et al. 2021), the authors presented a novel strategy for weight transfer and conducted NAS experiments with 2D human pose estimation datasets. The search algorithm is a traditional Genetic Algorithm combined with multi-objective fitness (minimizing loss and number of trainable parameters), while the search space is tailored for the task. The best models generated, named EvoPose2d-S and EvoPose2D-L, set the new state-of-the-art for 2D human pose estimation while having a smaller memory footprint and inference time.

In (Liberis, Dudziak and Lane 2021), the authors presented μNAS, a multi-objective neuroevolutionary designed to evolve networks for extremely resource-scarce environments, in order of 64 KB of memory. The search strategy is conducted using Amoeba (Real et al. 2019), while the multi-objective aspect is handled via Random Scalarization (Paria, Kandasamy and Póczos 2020).

In (Pan et al. 2021), the authors presented AutoSTG, a NAS algorithm designed to evolve neural networks for spatiotemporal graph prediction tasks. The search space and algorithm are tailored for the domain, but the optimization mechanism used during the search leverages the fact that all computations in AutoSTG are differentiable to implement a gradient-based strategy similar to DARTS.

In (Lakhmiri, Digabel and Tribes 2021), the authors presented HyperNOMAD, an extension of NOMAD, to optimize the architecture and learning parameters of neural networks. NOMAD is a software that implements the MADS algorithm, which is a Derivative-Free Optimization algorithm. The search space of HyperNOMAD is rather simple, consisting of VGG-like models. The algorithm generates models that achieve a 22.4% test error rate on CIFAR-10 if initialized from default values and a 7.46% test error rate if initialized from VGG.

In (Huan, Quanming and Weiwei 2021), the authors presented SANE, an adaptation of DARTS to evolve graph neural networks. The authors craft a search space focused on node aggregators and layer aggregators to achieve state-of-the-art results on multiple datasets.

In (Perez et al. 2021), the authors presented ATTIC, an extremely simple algorithm to classify bug tickets using TF-IDF and basic classifiers from Scikit-learn (Pedregosa et al. 2011). The authors employ a genetic algorithm to tune the hyperparameters of the Multilayer perceptron, such as the number of neurons, technically performing NAS; but without a particularly novel search space or encoding, the algorithm is not very interesting from a NAS perspective.

In (Jiang et al. 2021), the authors presented NASLung, a neural architecture search designed for the generation of 3D convolutional neural networks to classify pulmonary

nodules. The search mechanism is built on top of Partial Order Pruning (Li et al. 2019), and assumes that narrower networks are always more efficient and less accurate than wider ones. This assumption allows the authors to evaluate a network and, based on its efficiency and accuracy, partition the search space. The search space was tailored for their task and includes Convolutional Attention Modules to allow users to understand what the model was "paying attention" to when making a certain prediction. The proposed algorithm achieved results comparable to the state-of-the-art but with models that had 1/40 of the parameters.

In (Wang et al. 2021), the authors presented LaNAS, a Monte Carlo Tree Search algorithm that recursively partitions the search space into good and bad regions, with one of the most important contributions being a mechanism that allows the search algorithm to learn how to best partition the search space. The authors claim the algorithm generated a network that achieved a 99% test accuracy on CIFAR-10; however, there are multiple issues on their repository mentioning they are unable to reproduce their results[19,20]. Notably, one of the answers provided by the authors is "*After a couple years later, I don't encourage you to chase 99% accuracy on CIFAR10. It is not a good metric for either judging a good search algorithm or something quite useful. As for your question, sorry I completely forget the details now*". The discussion thread was closed, and the repository is no longer accepting comments.

In (Shen et al. 2022), the authors presented JSNET, a DARTS-based algorithm that generates networks for image restoration tasks. The most interesting aspect of the paper, from a NAS perspective, is the joint search space that, according to the authors, differs from similar methods by including both operation modules (convolutions) and attention modules. The algorithm achieved results comparable to the state-of-the-art and required up to 14 times less GPU time to run.

In (Guo et al. 2022), the authors presented DNAL, a neural architecture search algorithm that, starting from a given architecture, produces smaller networks while minimizing the loss of accuracy. The algorithm was able to reduce the number of parameters in some models by 18x while reducing the accuracy from 93.77% to 89.27%. The search mechanism is similar to DARTS but uses a "scaled sigmoid" instead of the traditional sigmoid.

In (Zhou et al. 2021), the authors presented OSNet, a convolutional neural network for person re-identification. NAS is not the focus of the paper; the authors use a gradient-based search strategy just to identify the ideal placement of a new type of layer, named instance normalization, into the proposed OSNet. The proposed model achieved state-of-the-art results on multiple datasets.

[19] <https://github.com/facebookresearch/LaMCTS/issues/10>
[20] <https://github.com/facebookresearch/LaMCTS/issues/30>

In (Wei, Zhao and He 2022), the authors presented F^2GNN, a DARTS-based algorithm that evolves graph neural networks. The authors propose a framework to unify existing topology designs with feature selection and fusion into a single framework and search space.

In (Shi et al. 2022), the authors presented Genetic-GNN, a genetic algorithm that evolves graph neural networks, handling both the architecture of the network and associated hyperparameters. From an NAS perspective, the search mechanism, which consists of traditional genetic operators and the evolutionary cycle, is not particularly novel.

In (Wang et al. 2022), the authors presented LightHuBERT, a transformer compression framework that, similarly to DNAL, starts from a model and produces smaller, more efficient models. Notably, the proposed algorithm relies on distillation, which requires not only a seed architecture but also the weights of a fully trained model.

The appendix of this thesis contains links to the repositories of the reproducible works described in the Chapter. In the next Chapter, the proposed algorithm of this research is described in detail and compared with similar algorithms.

3 Proposed Algorithm

This chapter is dedicated to the presentation of the algorithm Multi-Objective gRammatical Evolution for FUlly convolutional Networks (MOREFUN), which was published in (Miranda et al. 2023). The reference implementation of the algorithm is available at: <https://github.com/Mirandatz/morefun/tree/develop/morefun>.

To simplify descriptions, the following conventions will be adopted for the remainder of the chapter:

- *italics* will be used to identify named data structures, e.g., *connection markers*;
- the term "layer" will be used with the same meaning of a TensorFlow/Keras layer[1], that is, it will be used to describe any potentially stateful "output-generating" component of neural network. Examples: a convolutional layer with its respective weights, a batch-normalization layer with its respective weights, an image-rotation component used for data augmentation during training, or even an activation function, such as sigmoid. As such, layer effectively refers to any building block of a neural network.

The most interesting properties of MOREFUN, which will be detailed in the following sections, are:

- it is a grammar-based evolutionary algorithm, which means that a grammar describes the search space and that there is a clear interface that separates genotypes and phenotypes, reducing the required effort to improve the algorithm in the future and simplifying the introduction of domain knowledge into the search process;
- the genotypes, and the grammar used to describe the search space, contains novel structures to represent the skip connections, enabling the creation of models similar to the U-Net (Ronneberger, Fischer and Brox 2015), which are suitable for tasks more complex than classification, such as semantic segmentation;
- its fittest selection mechanism employs multi-objective optimization.

It is important to observe that although MOREFUN is capable of generating U-Net-like models, the experiments discussed in Chapter 4 focus on image classification tasks. The generation of U-Net-like models, which is of particular interest for semantic segmentation tasks, had to be left for future work due to computational resource constraints.

[1] <https://keras.io/api/layers/>

A high-level description of MOREFUN is: an evolutionary loop that consists of generating mutants, adding them to the population, evaluating the fitness values of the population, and selecting the fittest individuals for the next generations. The next section describes the initial population generation.

3.1 Population initialization

The initial population is randomly generated but respects two constraints. The first constraint, inspired by F-DENSER (Assunção et al. 2019), refers to network size: models with too many learnable parameters or layers are discarded and regenerated. This is done for the same reason that the authors of F-DENSER limited their initialization, that is, to try to skip the "shrinking phase" that was observed with DENSER. This "shrinking phase" was a trend observed in the initial generations of the algorithm, where the average network sizes tended to decrease.

The second constraint is related to phenotypical novelty: each individual must have a unique phenotype. This requirement has two benefits: first, it improves the diversity of the population; second, it reduces the computational cost by preventing multiple evaluations of the same phenotype, which are particularly time-consuming for deep neural networks. A pseudo-code of the initialization procedure is shown in Algorithm 1.

Algorithm 1 Population Initialization (Miranda et al. 2023).

Require:
Population size: pop_size
Max failures: max_fails

```
 1: K ← {}                      ▷ Known phenotypes, used to speed phenotype novelty checks
 2: P ← {}
 3: failures ← 0
 4: repeat
 5:     g ← create_random_genotype()
 6:     if phenotype(g) ∈ K then
 7:         failures ← failures + 1
 8:     else
 9:         P ← P ∪ {g}
10:         K ← K ∪ {phenotype(g)}
11:     end if
12:     if failures = max_fails then
13:         abort("Unable to generate initial population")
14:     end if
15: until |P| = pop_size
16: return P, K
```

The initial population is then processed by the evolutionary framework depicted in Algorithm 2. In each cycle of the evolutionary loop, the individuals produce descendants

via mutation (detailed in Section 3.5), that is, from a population P, a mutated population M is generated. The novelty requirement employed during the initial population generation is also used during mutation for the same reasons.

The sizes of the populations P and M are hyperparameters of the algorithm: the initial population size and the number of mutations per generation, respectively. If the mutation is unable to generate the required number of novel solutions, the evolutionary process halts, and the results of the previous generation are returned. This rare state may be reached if the search space has been exhausted or because a maximum number of non-novel individuals have been generated in a single generation.

After the mutation phase, the solutions are evaluated, and the best individuals of $P \cup M$ are selected for the next cycle. The fitness evaluation and fittest selection mechanisms are detailed in Section 3.6.

Algorithm 2 Evolutionary Loop (Miranda et al. 2023).

Require:
Initial population: P
Nr. of generation to run: max_gens
Nr. of mutants to create per generation: nr_muts
Max failed mutation attempts per generation: max_fails
Known phenotypes: K

```
 1: pop_size ← |P|
 2: for i ∈ {1, 2, . . . , max_gens} do
 3:     M ← gen_mutants(P, K, nr_muts, max_fails)
 4:     if M = {} then
 5:         return P                         ▷ Unable to create enough novel mutants
 6:     end if
 7:     K ← K ∪ {phenotype(g) | g ∈ M}
 8:     P ← select_fittest(P ∪ M, pop_size)
 9: end for
10: return P, K
```

3.2 Individual's representation

MOREFUN encodes the networks using a genotype representation named *composite genotype*, which consists of two parts. The first, called *layers and optimizer segment*, is a standard (SGE) genotype (Lourenço, Pereira and Costa 2015)[2], which encodes information about the learning algorithm (optimizer), the layers of the network, and the data augmentation mechanisms used during training. The second part, called *topology segment*, is a data structure that describes how network layers are connected. Figure 11 provides a high-level graphical representation of this *composite genotype*.

[2] For details, see Section 1.4.5.

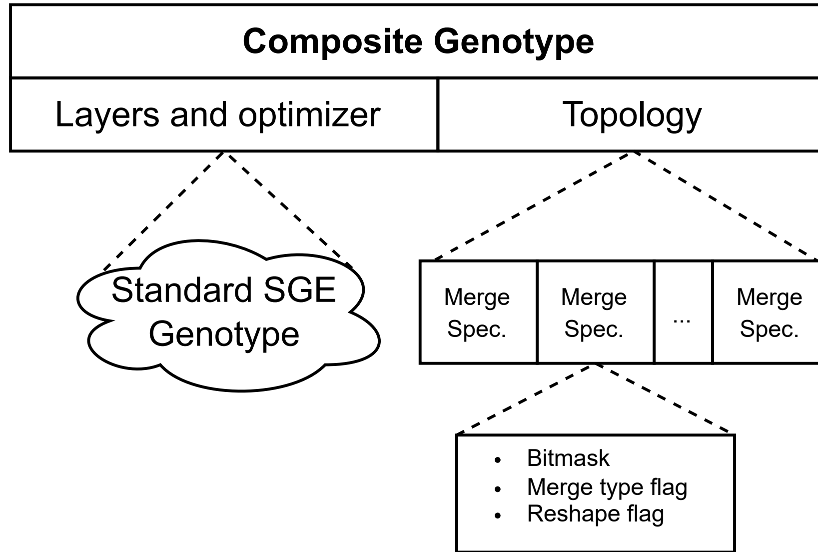


Figure 11 – Graphical representation of the composite genotype.

In terms of implementation, the *layers and optimizer segment* is simply a jagged array of integers, as illustrated in Figure 7. The shape of such depends on the grammar used to describe the search space; what differs from one individual to another, and thus what may be modified during mutation, are the values contained in the arrays.

The integers inside the *layers and optimizer segment* are used with a grammar to generate a sequence of tokens (or strings) that represent neural network components. An exhaustive enumeration of the components that can be encoded by such integers, along with their parameters, is shown in Table 2.

One of MOREFUN's most distinctive features is related to the grammar, which may contain special tokens named *connection makers*. When present on an individual, such tokens appear in the string generated by the genotype-to-phenotype mapping; more specifically, they appear between tokens that describe network layers. Such special tokens can have two values: *fork* and *merge*. The *fork marker* indicates that the output of the last generated layer could be forwarded to additional layers besides the one generated immediately after it. The *merge layer* indicates that the next generated layer could receive as input not only the output of the last generated layer, but also from any previously generated layer marked as a fork. Thus, fork and merge actions can be viewed as dual (split vs. merge) and can be used to implement skip connections.

The components shown in Table 2 represent the components used in the experiments discussed in Chapter 4, which is a subset of the implemented collection of components. For instance, the algorithm supports "dropout layers", but as they have not been used in the experiments[3], they were omitted from the table. The full list of supported elements is an

[3] Preliminary experiments showed batch normalization outperformed dropout. Given limited computational resources, removing a component that increased the search space without yielding improvements was a natural choice.

Table 2 – Elements of Layers and Optimizer Segment (Miranda et al. 2023).

Layer	Numeric Params.	Non-numeric Params.
Flip	-	mode ∈ {horizontal, vertical, both}
Rotate	max degrees	-
Translate	max shift	-
Conv2D	nr. filters, kernel size, strides	-
Pool2D	size, strides	pooltype ∈ {average, max}
Normalization	-	batchnorm ∈{yes, no}
Activation	-	function ∈{relu, prelu}
Optimizer	learning rate, momentum, beta1, beta2, epsilon, sync period ams_grad	algorithm ∈ {adam, ranger}

implementation detail that does not affect the overall algorithm[4]; it has evolved with the reference implementation without changing any particular mechanism. The most recent version of the supported components can be found at <https://github.com/Mirandatz/morefun/blob/develop/morefun/neural_networks/layers.py>.

To aid visualization, Figure 12 was synthesized to demonstrate what a string, generated by translating the *layers and optimizer segment* with the standard SGE translation mechanism (Lourenço, Pereira and Costa 2015), looks like for a fictional individual. This string describes a network with five layers: the first is a convolutional layer composed of 128 filters, kernels with a size of three, and a stride of one. The second layer is an activation layer with ReLU (Fukushima 1975) function that potentially "splits" into multiple further connections due to the "fork marker". The third layer is a convolution layer with 128 filters, kernels of size three, and a stride of one. The fourth layer is an activation layer with a ReLU function. Finally, the fifth layer is an activation layer with PReLU (He et al. 2015) that receives as input the values generated by the last convolution layer, and, depending on the values of the associated *topology segment*, the values generated by the first layer.

The second part of a MOREFUN genotype, called *topology segment*, is a list of *merge specifications*, as illustrated in Figure 11. The length of this list is equal to the number of merge tokens generated by the *layers and optimizer segment*. A *merge specification* is a data structure that contains detailed instructions about the connectivity of a layer marked with a merge token. Each merge specification is composed of the following elements:

1. a bitmask with a length equal to the number of fork layers created before the current merge; a value 1 in the n-th position indicates that the n-th fork point should be

[4] This is an excellent real-world example of the benefits of using a grammar-based algorithm: implementing new phenotypical elements, such as dropout layers, did not require modifying the genotype machinery, as the genotype-to-phenotype mapping mechanism transparently handled the new elements.

```
conv filter_count 128 kernel_size 3 stride 1
relu
fork
conv filter_count 128 kernel_size 3 stride 1
relu
fork
merge
prelu
```

Figure 12 – Illustrative example of a string generated from a *layers and optimizer segment*, organized in multiple lines to improve readability (Miranda et al. 2023).

used as input (to the current merge), while a 0 indicates that it should not;

2. a flag that indicates what type of multi-input layer should be created, such as Add or Concatenate;

3. a flag that indicates how mismatches between input shapes should be resolved, namely downsampling all inputs to the smallest size or upsampling all inputs to the largest size.

The connection markers, combined with the *topology segment*, allow MOREFUN to create networks with an arbitrary number and arrangement of skip connections, with the only restriction being that the final network forms a directed acyclic graph.

The mapping process of MOREFUN, which ultimately generates a neural network, consists of translating the *layers and optimizers segment* to produce a string and parsing it, creating one layer at a time and connecting them in sequence.

During the parsing phase, whenever a "fork" is encountered, the last created layer is marked as a fork layer. Whenever a "merge" is encountered, a multi-input layer (Add or Concatenate) is generated and connected to the previously created layer and some of the fork layers.

MOREFUN determines what type of multi-input layer should be generated, which fork layers should be used as input, and how shape mismatches should be handled by consuming a *merge specification* from the *topology segment*.

Components that describe the optimizer and data augmentation are also generated and associated with the final network, but are used only during training. A pseudo-code description of the genotype-to-phenotype mapping is presented in Algorithm 3. The algorithm's inputs are the composite genotype of an individual, split into *layers and optimizer* and *topology segments*, and the grammar that describes the search space. Although the grammar is a hyperparameter and, therefore, can be viewed as a constant

value throughout the execution of the algorithm, it was included as input to this procedure to improve clarity.

The function *sge_mapping* (Line 1) uses the SGE mapping algorithm (Lourenço, Pereira and Costa 2015) to translate the "layers and optimizer segment", which is a jagged array of integers, into a token stream (or string). As shown in Figure 12, the token stream is a simple sequence of characters.

The function *parse_object* (Line 6) consumes tokens to generate an actual object (such as a convolution layer). It consumes only as many tokens as necessary to generate a single object, returning a tuple containing the remaining (unparsed) tokens and the parsed object.

The function *pop_element* (Line 11) fetches the next element of the topology segment, which is a *merge specification*.

The function *get_forks_mask* (Line 12) retrieves the forks mask from a given merge specification. It is a "getter method" from a programming perspective.

The function *select* (Line 13) receives two lists of the same length, the first containing layers and the second containing boolean values. It selects elements from the first list if the element at the same position in the second list is "true". Effectively, this function identifies the fork layers that will be used to create the input of a merge layer.

The function *get_reshape_flag* (Line 15) is another getter, similar to the function *get_forks_mask*. In this case, it retrieves the reshape flag from a merge specification.

The function *reshape* (Line 16) receives a list of tensors (i.e., outputs from layers) and a reshape flag and outputs another list of tensors. The generated tensors all have the same shape. For example, if the value of the reshape flag is "downsample", then the tensors are downsampled to the size of the smallest one; conversely, if its value is "upsample", the smallest ones are upsampled to the size of the largest one.

The function *get_merge_type* (Line 17) is another getter. It retrieves the merge type flag from the merge specification.

The functions *make_multi_input_layer* and *connect* (Lines 18 and 19, respectively) create a multi-input layer, such as Add or Concatenate, depending on the value of the merge type flag, and connect it to the reshaped layers generated by *reshape*.

Since the processing consumes the tokens (as in formal grammar style), the condition $tokens =$ (Line 27) means that the entire string was processed.

After this point, the function *walk_dag_backwards*, used on Line 28, navigates the network from the last layer to the first and assembles the network. The function *generate_head*, used on Line 29, generates a classification head and adds the optimizer to the model.

The generated head depends on the dataset being processed, but it consists of a convolution layer with as many filters as classes in the dataset, followed by a global max-pooling layer and a softmax activation. This classifier head substitutes the traditional fully connected layer with a softmax output, generating a tensor with the same shape and equivalent values. We chose this mechanism because it enables the generation of fully convolutional networks (Long, Shelhamer and Darrell 2015), that is, models that can natively handle inputs with different shapes.

Algorithm 3 Genotype to Phenotype Mapping (Miranda et al. 2023).

```
Require:
    layers and optimizer segment: los
    topology segment: ts
    Grammar: grammar

 1: tokens ← sge_mapping(los, grammar)
 2: previous_layer ← new Input layer
 3: fork_layers ← Empty list
 4: optimizer ← None
 5: repeat
 6:     obj, remaining_tokens ← parse_object(tokens)
 7:     tokens ← remaining_tokens
 8:     if is_fork(obj) then
 9:         fork_layers ← append(previous_layer, fork_layers)
10:     else if is_merge(obj) then
11:         merge_spec ← pop_element(ts)
12:         bitmask = get_forks_mask(merge_spec)
13:         selected_forks ← select(fork_layers, bitmask)
14:         inputs ← selected_forks ∪ {previous_layer}
15:         rf ←get_reshape_flag(ts)                    ▷ Downsample or upsample
16:         reshaped_inputs ← reshape(inputs, rf)
17:         mf ← get_merge_type(ts)                     ▷ Add or concatenate
18:         layer ← make_multi_input_layer(mf)
19:         connect(layer, reshaped_inputs)
20:         previous_layer ← layer
21:     else if is_optimizer(obj) then
22:         optimizer ← obj
23:     else                                ▷ obj is just a normal neural network layer
24:         connect(obj, previous_layer)
25:         previous_layer ← obj
26:     end if
27: until tokens = ""
28: network ← walk_dag_backwards(previous_layer)
29: model ← generate_head(network, optimizer)
30: return model
```

As the algorithm implies, the *topology segment* (and its *merge specifications*) is tailored for a string with a specific quantity and arrangement of *connection markers*, and as such, when MOREFUN creates new individuals, it must create the *layers and*

optimizer segment first. More specifically, to create the *composite genotype*, it creates the *layers and optimizer segment*; processes it to generate a string; scans the string to identify the arrangement of *connection markers*; creates the *topology segment* using the scanned information; and combines both to create a *composite genotype*.

Careful analysis of the algorithm reveals that adding connection markers to the grammar (and consequently to genotypes) increases the size of the search space exponentially. To demonstrate this, consider the grammar illustrated in Figure 13 and the string generated by translating an individual's layers and optimizer segment from this search space. The grammar describes a search space composed of networks with the same seven convolution layers, differing only in how they are connected.

The last "merge" may receive as input any combination of the previous "forks", plus the output of the preceding layer, resulting in $2^4 + 1$ possible inputs; the penultimate "merge" has $2^3 + 1$ possibilities; and so on. Thus, the search space of this grammar contains approximately $\prod_{n=1}^{n=4} 2^n$ neural networks. Therefore, if we add a "merge" to the rule "last", changing it to `last: "merge" conv`, we increase the number of networks to $\prod_{n=1}^{n=5} 2^n$.

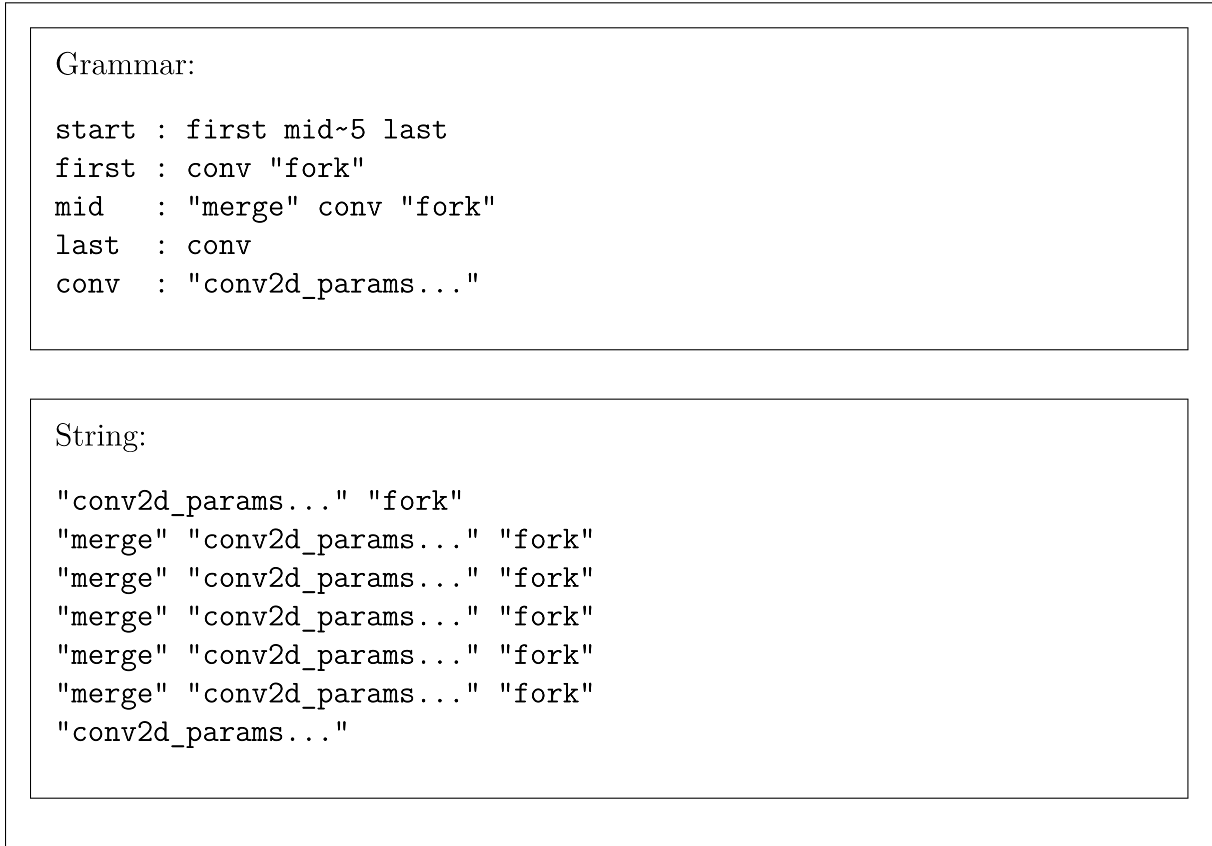


Figure 13 – Example of simple grammar and a token stream belonging to it. The grammar describes a search space of networks that contain between three and six convolution layers, with the convolutions' parameters omitted to reduce the token stream's size. The token stream is the result of translating the layers and optimizer segment of an individual belonging to the search space described by the grammar (Miranda et al. 2023).

3.3 Multi-input layers

MOREFUN, during the time of the experiments, supported two types of multi-input layers that it generates for the merge tokens: Add, which adds the values of the input tensors and outputs a tensor with the same shape, and Concatenate, which preserves the values of the input tensors by stacking them depth-wise. Both require the input tensors to have the same height and width (Add also requires them to have the same depth), and as such, MOREFUN reshapes inputs as necessary.

The reshaping procedure depends on the value *reshape flag* of the associated *merge specification*. If the value indicates that inputs should be downsampled, MOREFUN selects the smallest input shapes (measured as $width * height$) as the "target shape". Then, for each other tensor, check if its shape matches the target shape; if it does, it's used as is; if it does not, MOREFUN creates a convolution layer to reshape it and uses the output of this convolution as input.

The convolution layer used for downsampling has a kernel with size one (as in (He et al. 2016)), a number of filters equal to the depth of the target shape, and strides equal to the scale factor between the target shape and the source shape.

For upsampling, the target shape is the largest of the sources. When it is necessary to reshape a tensor, we utilize a convolution layer with kernel size one, stride one, and a number of filters equal to the depth of the target shape, followed by an upsampling layer with a factor equal to the scale between the source shape and the target shape.

3.4 Grammar primitives

As in other grammar-based evolutionary algorithms, the grammar is not a "domain-specific language" that describes a neural network; it describes a search space for the evolutionary algorithm, that is, a set of possible neural networks. Effectively, the grammar is a hyperparameter of the algorithm, meaning that users may run MOREFUN with different grammars.

The "building blocks" of the grammars can be divided into three categories: data augmentation, learning, and optimizer settings. The data augmentation blocks include layers that rotate, translate, and flip the inputs. The learning blocks include common neural network layers, such as convolutions and pooling. Finally, the optimizer blocks contain the identification and configuration of the optimizer, such as ADAM or SGD, learning rate, momentum, etc.

Formally, the set of all grammars recognized by MOREFUN can be described using another grammar, named meta-grammar (or upper grammar), whose formal description using LARK's[5] syntax is shown in Figure 14 and Figure 15. The definitions of the terminals have been omitted, as they are of no importance; but as a curiosity, they are usually defined to be equal to the name of the terminal, but enquoted, e.g., `BATCHNORM: /"batchnorm"/`. The symbol `_NL` is reserved by LARK and matches new lines; it is used in this context to improve the readability of the grammars.

[5] <https://lark-parser.readthedocs.io/en/stable/>

```
start       : _NL* rule+
rule        : NONTERMINAL ":" (layer_and_maybe_emptylines
            | block | optimizer_and_maybe_emptylines)
NONTERMINAL : NAME

block        : block_option ("|" block_option)*
block_option : maybe_merge symbol_range+ maybe_fork _NL*
maybe_merge  : [MERGE]
maybe_fork   : [FORK]

symbol_range : NONTERMINAL ["~" RANGE_BOUND [".." RANGE_BOUND]]
RANGE_BOUND  : INT

layer_and_maybe_emptylines : layer _NL*
layer : random_flip  | random_rotation | random_translation
      | resizing | random_crop | conv_layer | max_pooling_layer
      | avg_pooling_layer | batchnorm_layer | activation_layer

random_flip : RANDOM_FLIP _flip_args

random_rotation : RANDOM_ROTATION _float_args

random_translation : RANDOM_TRANSLATION _float_args

resizing : RESIZING height width
height  : HEIGHT _int_args
width   : WIDTH _int_args

random_crop : RANDOM_CROP height width

conv_layer   : CONV filter_count kernel_size strides
filter_count : FILTER_COUNT _int_args
kernel_size  : KERNEL_SIZE _int_args
strides      : STRIDE _int_args

batchnorm_layer : BATCHNORM
```

Figure 14 – MOREFUN’s upper grammar, part 1.

```
max_pooling_layer : MAXPOOL pool_sizes strides
avg_pooling_layer : AVGPOOL pool_sizes strides
pool_sizes        : POOL_SIZE _int_args

activation_layer : RELU | GELU | SWISH | PRELU

optimizer_and_maybe_emptylines : optimizer _NL*
optimizer : sgd | adam | ranger

sgd           : SGD learning_rate momentum nesterov
learning_rate : LEARNING_RATE _float_args
momentum      : MOMENTUM _float_args
nesterov      : NESTEROV _bool_args

adam : ADAM learning_rate beta1 beta2 epsilon amsgrad
beta1: BETA1 _float_args
beta2: BETA2 _float_args
epsilon: EPSILON _float_args
amsgrad: AMSGRAD _bool_args

ranger         : RANGER learning_rate beta1 beta2 epsilon amsgrad
                 sync_period slow_step_size
sync_period    : SYNC_PERIOD _int_args
slow_step_size : SLOW_STEP_SIZE _float_args

_int_args : INT_ARG
          | "(" INT_ARG ("|" INT_ARG)* ")"
INT_ARG   : QUOTED_INT

_float_args : FLOAT_ARG
            | "(" FLOAT_ARG ("|" FLOAT_ARG)* ")"
FLOAT_ARG : QUOTED_FLOAT

_bool_args : BOOL_ARG
           | "(" BOOL_ARG ("|" BOOL_ARG)* ")"
BOOL_ARG   : BOOL

_flip_args  : FLIP_MODE
            | "(" FLIP_MODE ("|" FLIP_MODE)* ")"
```

Figure 15 – MOREFUN’s upper grammar, part 2.

3.5 Mutation

MOREFUN's mutation phase, illustrated in Algorithm 4, consists in trying to generate *mutants_per_gen* new individuals using the procedure described below (where *mutants_per_gen* is a hyperparameter).

To generate a single mutant, MOREFUN randomly selects an individual from the population, creates a copy of this individual, and mutates the copy instead of the original. This non-destructive approach to mutations simplifies the implementation and enables the generation of new individuals without losing the ones already created, which means they can be used as mutation candidates multiple times.

Mutation candidates are chosen from the pool of individuals available at the beginning of the mutation phase, which means that mutants generated in the current mutation phase are not eligible as mutation candidates.

To mutate an individual, MOREFUN can either mutate its *topology segment* or its *layers and optimizer segment*. To mutate the *topology segment*, it randomly selects one of its *merge specification* and mutates it by: (a) toggling a random bit of the bitmask, which adds or removes a connection between a fork layer and a merge layer; (b) changing the flag that indicates how mismatches between input shapes should be resolved, alternating between upsampling and downsampling; or (c) changing the flag that indicates the type of multi-input layer that should be generated, alternating between Add or Concatenate layers.

To mutate the *layers and optimizer segment*, MOREFUN uses the standard SGE mutation (Lourenço, Pereira and Costa 2015). As a brief reminder, an SGE genotype contains multiple lists of integers; mutating consists of changing one of the integers. Modifying the *layers and optimizer segment* may alter the quantity and arrangement of *connection markers*, which may invalidate the *topology segment*. Thus, it is necessary to regenerate the *topology segment* whenever this mutation occurs.

Similarly to the initial population generation procedure, the mutation procedure ensures phenotypical novelty. Whenever it generates an individual whose phenotype was already seen (at any generation), this individual is discarded, a failure counter is incremented, and MOREFUN attempts to generate a new one. This failure counter was implemented to prevent potential infinite loops that could happen, for example, if the search space was exhausted. If the failure counter reaches a specific value, defined by a hyperparameter of the algorithm, the evolutionary cycle halts, and the last generation is returned. If the mutation phase successfully generates the required number of mutants, MOREFUN resets the failure counter and advances to the next stage.

Algorithm 4 Mutation Process (Miranda et al. 2023)

Require:
Initial population: P
Number of mutants to generate: num_muts
Known phenotypes: K
Max failures: max_fails

```
1: muts ← {}
2: failures ← 0
3: repeat
4:     candidate ← select_random(P)
5:     mutant ← mutate(candidate)
6:     if phenotype(mutant) ∈ K then
7:         failures ← failures + 1
8:     else
9:         muts ← muts ∪ {mutant}
10:        K ←K ∪ {phenotype(mutant)}
11:    end if
12:    if failures = max_fails then
13:        return {}
14:    end if
15: until |muts| = num_muts
16: return muts
```

3.6 Fittest selection

To evaluate an individual's fitness, MOREFUN creates the layers described by its *layers and optimizer segment* and connects them according to its *topology segment*. It then synthesizes the optimizer, described by the *layers and optimizer segment*, and uses it to train the model until its loss on the training partition does not improve for a few epochs, defined by a hyperparameter. MOREFUN utilizes two objective functions; the first function, f_1, is the model's final train loss, while the second objective function, f_2, is the model's size, measured as the number of trainable parameters.

To select which individuals will participate in the next generation, MOREFUN combines the current population with the newly generated mutants into a single collection; evaluates the fitness of all unevaluated individuals; and, using NSGA-II (Deb et al. 2002), selects the p fittest individuals, with p being the size of the initial population. More precisely, the mechanisms of NSGA-II that MOREFUN leverages are the ranking of solutions into Pareto fronts and the selection of the last individuals using objective-space density.

The non-selected individuals are removed from the population, but their phenotypes are maintained in a separate data structure for the novelty constraint of subsequent mutation phases.

The multi-objective approach is attractive for two reasons. First, it allows for the

generation of networks with similar performances that differ significantly in size. Since MOREFUN ultimately generates a collection of solutions (the final population), it is beneficial to have solutions that offer different compromises between predictive power and size. Second, if one assumes that the selective pressure towards smaller models may reduce the overall runtime, as less time is spent exploring regions of the search space with inefficiently large models, then it may decrease the overall runtime of the algorithm.

3.7 Principal differences between MOREFUN and DENSER

MOREFUN was inspired by the DENSER family of algorithms, but it differs from them in multiple aspects. The first notable difference is that MOREFUN has no GA elements, relying instead on the grammar to represent structural repetition.

As a reminder, DENSER employs a hierarchical genotype with the outer level being manipulated by a GA and the inner level by DSGE. The outer level can be viewed as a list of 3-tuples, each containing a non-terminal and two integers, such as: $[(features, 1, 10), (classification, 1, 2), (softmax, 1, 1)]$. The symbol of a tuple indicates a starting symbol for the grammar used by DSGE, while the integers indicate the minimum and the maximum number of times they can be used. As mentioned in Chapter 1, this structure is redundant and its functionality could be perfectly implemented with grammar rules, which is what MOREFUN does.

Another key distinction between MOREFUN and the DENSER family of algorithms is in how they handle skip connections, which happens implicitly in the DENSER family and explicitly in MOREFUN.

The final search-space difference between the algorithms is that MOREFUN is capable of generating architectures similar to U-Net (Ronneberger, Fischer and Brox 2015), because it can create different types of merging layers and upscale their outputs. From a theoretical standpoint, this is beneficial, as the U-Net is known to perform well on semantic segmentation tasks, which require detailed per-pixel information. It is interesting to observe that, although this feature did appear in the smallest models generated by MOREFUN, including the one illustrated in Figure 17, it is unclear how beneficial it is for classification tasks.

Regarding the search mechanism, MOREFUN and the DENSER algorithms differ substantially in how the training costs are minimized. F-DENSER relies on a "$(1 + \lambda)$ Evolutionary Strategy" to reduce the number of evaluations, and F-DENSER++ includes the training time of the model in the genotype. MOREFUN, on the other hand, relies on the multi-objective pressure to explore more size-efficient models.

A less important difference is related to the GE ancestry of the algorithms. The

DENSER family is based on DSGE, while MOREFUN is based on SGE. As mentioned in Chapter 1, SGE was chosen because the potential benefits of DSGE, especially in the context of neural architecture search, were not considered worth the disproportional increase in complexity, as the potential time and memory saved by DSGE are orders of magnitude smaller than the time and memory required to evaluate the fitness of a single model.

The final and least important distinction of the algorithms is related to the treatment of the grammar. For GE algorithms, the grammar is just another hyperparameter; changing it should be similar to changing the number of generations. Checking if a grammar is well-formed, however, is not as simple as verifying if a number is greater than zero. The problem with invalid grammar is that it may be partially valid, that is, some genotypes may expand only the valid parts of the grammar; if the malformed parts of the grammar are rarely described by a genotype, possibly due to early-generations constraints on model size, then the software may crash in unexpected and hard-to-reproduce ways after running for dozens of hours. To avoid this, MOREFUN employs a meta-grammar[6], which ensures that user-provided grammars are well-formed. This meta-grammar is a hard-coded grammar that recognizes the shape and content of all valid grammars. It can also be used to warn users about potentially undesirable expansions, such as expansions that lead to sequences of non-linearities.

[6] In the reference implementation, they are named upper grammars: <https://github.com/Mirandatz/morefun/blob/develop/morefun/grammars/upper_grammars.py>

4 Experiments

This chapter describes the experiments conducted to evaluate the proposed algorithm and the research hypothesis. Some of the results discussed in the section were published in (Miranda et al. 2023). All MOREFUN experiments used the grammar shown in Figure 16 and the CIFAR-10 dataset (Krizhevsky, Hinton et al. 2009).

The CIFAR-10 dataset was selected for two reasons: its relatively small size optimizes experiment runtime (e.g., by fitting multiple models on the GPU), and its frequent use in computer vision research. This widespread use enables comparison of results without re-implementing algorithms, a process prone to errors, especially since authors often do not provide public reference implementations. Consequently, all metrics from other algorithms are reported as presented by their authors.

4.1 Scenarios

The original experiments had MOREFUN optimizing both model size and model efficacy, with the former being measured as the number of learnable parameters and the latter as the loss on the train partition of the dataset. This scenario, called "default", is the focus of the publication (Miranda et al. 2023), where MOREFUN was compared with other algorithms from the literature.

A second scenario, called "single objective", was devised to establish the benefits of the multi-objective optimization. This scenario consisted of running MOREFUN with a single optimization objective: the model efficacy.

The last scenario, called "with-validation", is similar to "default", but with the train partition split into two disjoint sets named "train-without-validation" and "validation". The models were trained on the "train-without-validation" and their efficacies were measured as the loss on the "validation" partition.

All scenarios consist of five runs with different seeds using the following hyperparameters: initial population size = 20; max generations = 50; mutants per generation = 5; max mutation failures per generation = 500; batch size = 256 (for model training); early stop, using six consecutive epochs without improvement in the train loss as stop criterion. Other settings, such as optimizer, learning rate, and data augmentation, are part of the search space and are enumerated in the grammar, illustrated in Figure 16.

```
start       : data_aug learning optimizer
data_aug    : flip rotate translate
flip        : "random_flip" "horizontal"
rotate      : "random_rotation" ("15" | "30" | "45")
translate   : "random_translation" ("0.1" | "0.15" | "0.20")
learning    : intro_a mid | intro_b mid
intro_a     : conv_128_3 act norm conv_128_3 act norm "fork"
intro_b     : conv_128_5 act norm conv_128_5 act norm "fork"
mid         : pool block_1~1..2 pool block_2~1..2 pool block_3~1..2
    pool block_3~1..2
block_1     : "merge" conv_128_3 act norm "fork"
            | "merge" conv_128_5 act norm "fork"
block_2     : "merge" conv_256_3 act norm "fork"
            | "merge" conv_256_5 act norm "fork"
block_3     : "merge" conv_512_3 act norm "fork"
            | "merge" conv_512_5 act norm "fork"
conv_128_3 : "conv" "filter_count" "128" "kernel_size" "3" "stride" "1"
conv_128_5 : "conv" "filter_count" "128" "kernel_size" "5" "stride" "1"
conv_256_3 : "conv" "filter_count" "256" "kernel_size" "3" "stride" "1"
conv_256_5 : "conv" "filter_count" "256" "kernel_size" "5" "stride" "1"
conv_512_3 : "conv" "filter_count" "512" "kernel_size" "3" "stride" "1"
conv_512_5 : "conv" "filter_count" "512" "kernel_size" "5" "stride" "1"
pool        : maxpool | avgpool
maxpool     : "maxpool" "pool_size" "2" "stride" "2"
avgpool     : "avgpool" "pool_size" "2" "stride" "2"
norm        : "batchnorm"
act         : relu | prelu
relu        : "relu"
prelu       : "prelu"
optimizer   : adam | ranger
adam        : "adam" "learning_rate" "0.001" "beta1" "0.9" "beta2"
    "0.999" "epsilon" "1e-07" "amsgrad" "false"
ranger      : "ranger" "learning_rate" "0.001" "beta1" "0.9" "beta2"
    "0.999" "epsilon" "1e-07" "amsgrad" "false" "sync_period" "6"
    "slow_step_size" "0.5"
```

Figure 16 – The grammar used in the experiments. The redundancy and prolixity of the production rules are not strictly necessary; they are an artifact of the simplicity of the currently implemented parsing mechanism. I opted for a larger grammar and a simpler parser to reduce implementation overheads.

4.1.1 Default scenario

In each of the five runs, the model with the smallest train loss from the last generation was selected, and its accuracy on the test partition was measured. Table 3 presents the characteristics of these networks. The results from smaller models, such

as those from the last run (Seed 4), whose architecture is shown in Figure 17, are particularly notable. The fitnesses of all models in the last generation across all five runs are illustrated in Figure 18.

Table 3 – Best models, default scenario (Miranda et al. 2023)

Seed	Train acc.	Test acc.	Params (millions)
0	97.86	89.37	8.58
1	98.92	89.33	13.05
2	98.84	87.34	16.68
3	99.14	90.57	20.43
4	98.19	90.00	8.23
mean/std	98.59 ±0.54	89.32 ±1.21	13.39 ±5.25

Table 4 compares MOREFUN with other algorithms from the literature on the CIFAR-10 dataset, grouped by search strategy. The GE algorithms were inferior to almost all non-GE algorithms in every aspect.

One of the reasons for the elevated runtime costs that is shared between all GE algorithms is the naive model training strategy, which (Yang et al. 2020) identified as a recurring factor in the most expensive NAS algorithms. F-Denser improved on DENSER by trying to reduce such costs with a $(1 + \lambda)$ Evolutionary Strategy (ES), which reduced the number of trained models, and thus the overall algorithm runtime, by a factor of twenty. MOREFUN, similarly to DENSER, trained all models, but tried to reduce the costs by leveraging side-effects of multi-objective optimization, directing the search towards smaller, and thus faster-to-train, models. Although both were successful when compared to DENSER, they still required days of GPU time, while the state-of-the-art algorithms required just hours.

A second reason for the elevated costs is related to *what* is being searched; all GEs, as demonstrated by the grammars they used in the experiments, searched for complete architectures, but (Zoph et al. 2018) suggests that it is more efficient to search for "good cells" and simply stack them to create a complete architecture. This is not an intrinsic limitation of GE algorithms; rather it is a consequence of the grammars that were used in the experiments; indeed, one could trivially design a grammar that contains a single rule for the final architecture consisting of a repeated number of cells, while dedicating the remainder of the grammar to different cell designs. For MOREFUN, specifically, the grammar was not designed with cell optimization in mind, instead it was created to try and incorporate domain knowledge into the search space via the skip connection markers.

The design of the grammar also had consequences for the second metric of the experiments, the test error rate of the models. Although the search space is an important factor, it is not simple to identify which particular aspect, if any, contributed the most

Figure 17 – Architecture of the best model found on our fifth run (Seed 4). The "Shortcut Conv" blocks are 1x1 convolutions used handle shape mismatches between the tensors with different depths (He et al. 2016).

to the underwhelming results because most works used different search spaces; training regimes, from optimizers to learning rate schedulers; evaluation regimes, using test-time data augmentation or ensembles; different training data augmentation methods; etc.

The results point to interesting possibilities for future works: modifying the GE algorithms to allow weight sharing between components of different models; replacing the fitness evaluation strategy to use less computationally expensive proxies for the test accuracy, such as using a single minibatch for training (Abdelfattah et al. 2021), training the model on a downsampled subset of the train partition (Dong et al. 2022), using surrogates (Lu et al. 2020); and conducting experiments with grammars designed for cell optimization, instead of the whole architecture.

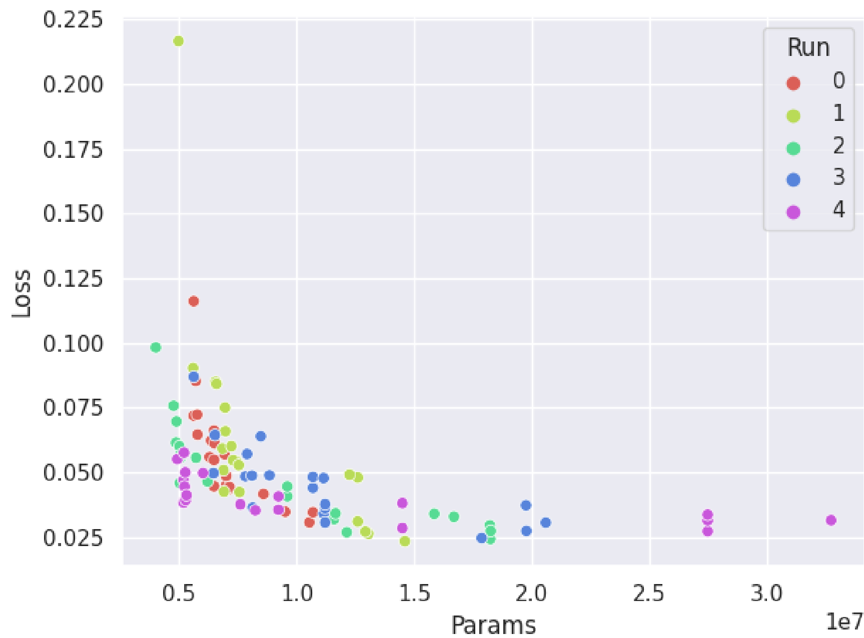


Figure 18 – Fitnesses of the individuals present in the last generation, default scenario (Miranda et al. 2023)

Table 4 – Comparison on CIFAR-10 (Miranda et al. 2023)

Architecture	Params (M)	GPU days	Test Error (%)	Search Strategy
ResNet(depth=110) (He et al. 2016)	1.7	-	6.61±0.06	Manual
ResNet(depth=1202)(He et al. 2016)	10.2	-	7.93	Manual
DenseNet-BC (Huang et al. 2017)	25.6	-	3.46	Manual
VGG (Simonyan and Zisserman 2014)	20.1	-	6.66	Manual
MobileNetV2 (Sandler et al. 2018)	2.2	-	4.26	Manual
NAS (Zoph and Le 2016)	37.4	22400	3.65	RL
ENAS (Pham et al. 2018)	4.6	0.5	2.89	RL
BNAS (Ding et al. 2021)	3.3	0.1	2.67	RL
NASNet-A (Zoph et al. 2018)	3.3	1800	2.65	RL
DARTS (first order) (Liu, Simonyan and Yang 2018)	3.3	1.5	3.00	gradient
DARTS (second order) (Liu, Simonyan and Yang 2018)	3.3	4.0	2.76	gradient
GDAS (Dong and Yang 2019)	3.4	0.2	2.93	gradient
SNAS (Xie et al. 2019)	2.8	1.5	2.85	gradient
P-DARTS (Chen et al. 2019)	3.4	0.3	2.50	gradient
FairDARTS (Chu et al. 2020)	2.8	0.4	2.54	gradient
SDARTS-ADV (Chen and Hsieh 2020)	3.3	1.3	2.61	gradient
EoiNAS (Zhou, Xie and Kung 2021)	3.4	0.6	2.50	gradient
U-DARTS (Huang et al. 2023)	3.3	4	2.59	gradient
β-DARTS (Ye et al. 2022)	3.78	0.4	2.51±0.07	gradient
Shapley-NAS (Xiao et al. 2022)	3.4	0.3	2.47	gradient
DSO-NAS (Zhang et al. 2020)	3.0	0.9	2.74	gradient
Genetic CNN (Xie and Yuille 2017)	-	17	7.10	EA
AmoebaNet-B (Real et al. 2019)	2.8	3150	2.55	EA
Hireachical Evolution (Liu et al. 2018)	15.7	300	3.75	EA
CARS (Yang et al. 2020)	3.0±0.33	0.4	2.81±0.12	EA
EvNAS (Sinha and Chen 2021)	3.7±0.08	3.83	2.52±0.04	EA
MOGIG-Net (Xue et al. 2021)	3.0	14	3.13	EA
MOEA-PS(Xue, Chen and Słowik 2023)	3.0	2.6	2.77	EA
MSNAS(Dong et al. 2022)	3.25	0.23	2.68±0.08	EA
DeepSwarm(Byla and Pang 2020)	-	1.25	11.31	EA
DENSER (Assunção et al. 2019)	-	45.12	11.81	GE
F-DENSER (Assunção et al. 2019)	-	2.3	12.24	GE
F-DENSER++ (Assunção et al. 2019)	-	7.06	11.27	GE
MOREFUN	13.39±5.25	3.1	10.6±1.21	GE

4.1.2 Single objective optimization

This scenario was created to investigate the benefits of the multi-objective optimization and to investigate the validity of the hypothesis. The only difference between this scenario and the default one is that the fitness of this is no longer a tuple $(model_size, model_efficacy)$, but instead it is a single scalar that represents the model efficacy.

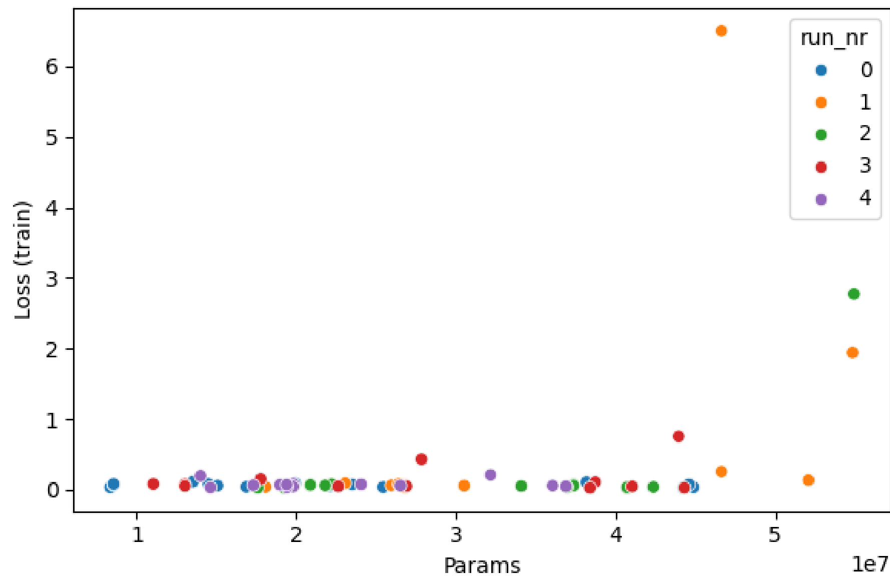


Figure 19 – Fitnesses (train loss and model size) of the individuals present in the last generation - *Single Objective Scenario*

Figure 19 illustrates the fitness of individuals from the last generation across all runs. Note that this figure is less precise than similar figures from other scenarios for two reasons: first, individuals mapping to models too large to train are not shown, as their fitness could not be computed; second, not all runs in this scenario reached the target of 50 generations. Thus, this figure under-represents the model sizes produced during evolution.

This approach, as hypothesized, produces inefficiently large models, which substantially increases the time required to train the models. After 20 generations, it is common for models to require more than 40 gigabytes of VRAM to be trained, and after around 30 generations, most of the mutants are too large to be trained. Eventually, this culminates in a generation where the number of viable mutants produced within the maximum number of attempts is insufficient to create a new generation, halting the evolutionary loop. Out of five runs, only one reached the target number of generations, 50. The best models of this run are shown in Table 5.

Table 5 – Best Models - Single Objective Scenario

Train Loss	Train acc.	Test loss	Test acc.	Params (millions)
0.03	0.99	0.6	0.87	44.25
0.05	0.98	0.67	0.88	22.12
0.04	0.99	0.84	0.85	44.86
0.11	0.97	0.97	0.85	13.51
0.09	0.97	0.67	0.87	19.93

The runtime difference between scenarios could not be precisely measured due to the shared server used for experiments. As a reference, the average multi-objective run required approximately 3.1 days to evolve 50 generations, while the single-objective required approximately 22 days to evolve 30 generations.

4.1.3 With validation

The last scenario, called with validation, is built on top of the default scenario. First, the train partition of the dataset, which contained 50000 instances, was divided into two disjoint sets: “validation”, with 12500 instances; and “train-without-validation”, with the remaining 37500 instances. Then, the algorithm was modified to train the models on the larger partition and to compute the “model efficacy” component of the multi-objective fitness as the loss on the smaller partition, that is, train until loss on the validation partition plateaus. Everything else remained the same.

The fitnesses of the models from the last generation, across all runs, are illustrated in Figure 20, while the characteristics of the best models of each run is shown in Table 6. Figure 21 illustrates the train loss.

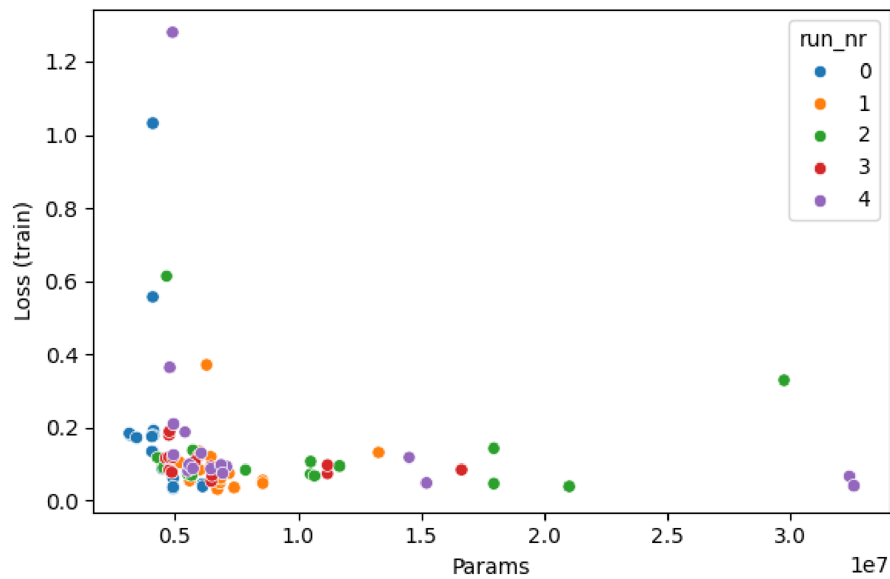


Figure 20 – Fitnesses (train loss and model size) of the individuals present in the last generation - *With Validation Scenario*

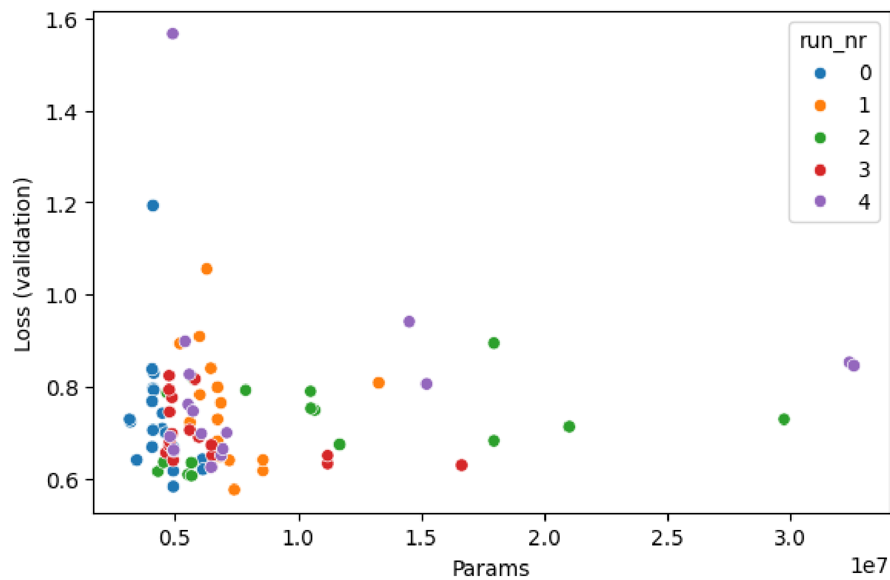


Figure 21 – Train losses and model sizes of the individuals present in the last generation - *With Validation Scenario*

Table 6 – Best Models - With Validation Scenario

Seed	Train acc.	Val acc.	Test acc.	Params (millions)
0	98.50	87.44	87.25	4.93
1	98.14	88.48	87.94	7.41
2	96.64	87.65	86.66	4.67
3	98.20	87.65	86.95	6.48
4	97.45	86.17	85.99	5.73
mean/std	97.79±0.67	87.48±0.75	86.96 ±0.64	5.84±1.01

The target number of generations, 50, was chosen arbitrarily, but an analysis of the hypervolume of objective space covered by the solutions suggests that the value is not inadequate. Figure 22 shows the evolution of the hypervolume across different runs; progress often plateaus by the 30th generation.

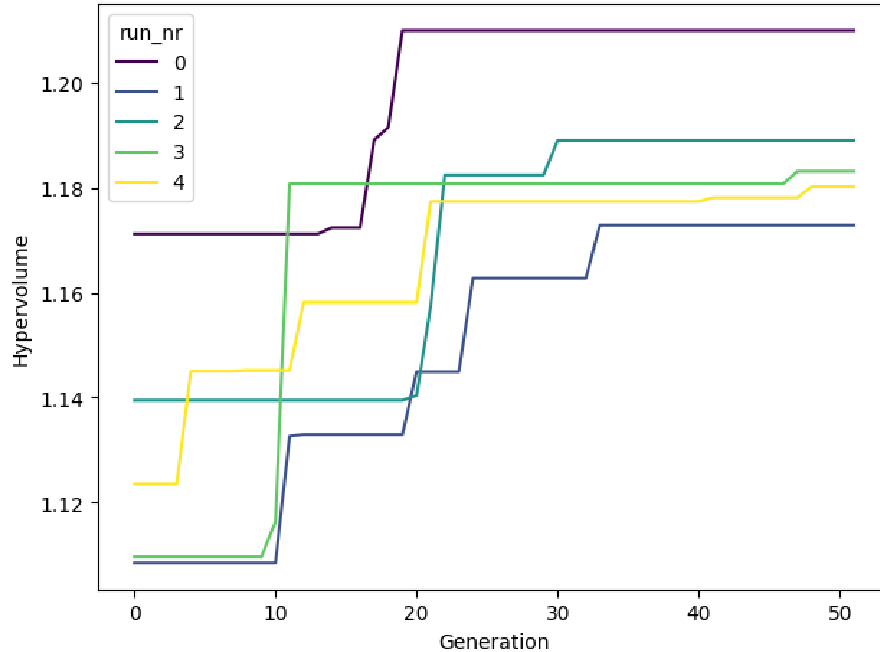


Figure 22 – Evolution of hypervolume of objective space over 50 generations across 5 runs

This chapter presented the experimental evaluation of the MOREFUN algorithm and the proposed research hypothesis. Three distinct scenarios were investigated: a "default" multi-objective optimization, a "single objective" optimization focused solely on efficacy, and a "with validation" scenario similar to the default but incorporating a validation partition.

The "default" scenario demonstrated MOREFUN's ability to find models that balance both model size and efficacy. The best models from five runs showed competitive test accuracies, averaging 89.32%, with a mean of 13.39 million parameters. Notably, the smallest model achieved a 90.00% test accuracy with only 8.23 million parameters, highlighting the effectiveness of multi-objective optimization in discovering efficient architectures. While MOREFUN successfully reduced runtime compared to DENSER family of algorithms by favoring smaller models, it still required days of GPU time, a common challenge for GE algorithms that train full architectures, in contrast with the state-of-the-art algorithms that often require only hours.

The "single objective" scenario provided clear evidence for the benefits of multi-objective optimization. As hypothesized, optimizing for efficacy alone led to the evolution of unnecessarily large models, often exceeding 40 gigabytes of VRAM and frequently halting the evolutionary process due to memory constraints.

The “with-validation” scenario was introduced as a crucial control experiment, aligning the methodology more closely with standard machine learning practices, to proactively address concerns that the “default” scenario, which optimizes the train loss instead of the validation loss.

Overall, the experiments confirmed the efficacy of multi-objective optimization in navigating the trade-off between model size and performance and, perhaps more interestingly, had the desired side effect of reducing the search time. The results also highlighted limitations inherent in current GE approaches for neural architecture search, primarily concerning runtime costs due to the training of entire architectures. Future works could explore strategies to address these limitations, such as incorporating weight sharing, utilizing cheaper proxy fitness evaluations, and designing grammars specifically for cell optimization.

5 Conclusion and Future Works

In the earlier chapters, I highlighted the recent increase in computing power and the advances of neural network libraries, such as PyTorch and TensorFlow, as favorable factors for the development and adoption of NAS algorithms. More recently, however, we have also witnessed a significant increase in model sizes and the computational resources required to train them, particularly for Large Language Models (LLMs).

As we move into late 2024 and early 2025, LLMs have become the most discussed and, arguably, the most researched models, but their size and data requirements are such that it is more cost-effective to hire a team of researchers to manually design and optimize their architectures than to conduct a single additional round of training. In this context, NAS algorithms are not an economically viable approach for the currently most well-funded types of models.

For other types of models and domains, such as convolutional neural networks used for computer vision tasks, NAS may be viable, especially if the available compute continues to increase, and the community continues to improve the search mechanism. Perhaps in the future, using NAS will be as common as using grid search is common for traditional models today.

As for Grammar-Based Neuroevolutionary algorithms (GBNA), the benefits of having a descriptive and easy-to-manipulate search space do not outweight the computational costs and underwhelming results, at least not with the current naive approach to fitness evaluation.

This research demonstrated that the multi-objective approach can be combined with grammar-based neuroevolutionary algorithms to ameliorate the runtime costs, but not to the point of being competitive with the state-of-the-art. Such algorithms, however, are well suited to modifications due to their modularity and could potentially be improved in multiple ways.

Future research could explore multiple promising approaches. To provide a practical roadmap for researchers with varying computational resources, these are ordered from the least to the most computationally expensive[1]. The initial strategies, in particular, are designed to significantly reduce runtime costs and enable the exploration of larger search spaces.

[1] Although it is not possible to precisely ascertain which approaches are going to be more computationally expensive, it is reasonable to assume that under ideal conditions, some of them should not require substantially more computational resources than the current implementation. The estimated computational costs are, therefore, educated guesses.

- Fitness evaluation with proxies or surrogates: the most direct way to decrease runtime would be to replace the fitness evaluation strategy. This approach would involve running the algorithm for a few generations to create a dataset of architectures and their corresponding performances. This dataset would then be used to train a surrogate model (e.g., a gradient boosting tree or a small neural network) to estimate an architecture's performance directly from its genotype. This proxy would replace the costly full training and evaluation cycle for subsequent generations.

- Block-based weight sharing: implementing a weight-sharing mechanism could also dramatically decrease runtime. A interesting possibility would be a mechanism that identifies recurring "blocks" (i.e., specific expansions of a given non-terminal of the grammar) and caches a set of pre-trained weights for them. When a new architecture is generated that uses a known block, it would sample weights from this cache instead of initializing them randomly, significantly accelerating its convergence.

- Cell-based architecture search: another possibility would be to shift from a global architecture search to a cell-based search. This would require designing a grammar that describes only the structure of a computational cell and then modifying the algorithm to build a full network by stacking these evolved cells. While this approach could reduce the runtime per generation, it requires a careful and potentially iterative design process for the search space itself, which could increase the overall computational costs of the research.

- Application to semantic segmentation: finally, a promising but likely more computationally intensive path would be to apply the proposed algorithm to semantic segmentation tasks to leverage its capability of generating U-Net-like structures. Since semantic segmentation is a task substantially more complex than image classification, the costs would be higher, but one could also explore different types of tasks.

Finally, it is worth noting that multiple approaches could be combined at once and the contribution of each evaluated via ablation studies. For instance, one could implement both the cell-based architecture search and the block-based weight sharing, which would likely yield interesting results.

References

Abdelfattah et al. 2021 Abdelfattah, M. S.; Mehrotra, A.; Dudziak, Ł.; Lane, N. D. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021.

Alam et al. 2020 Alam, M.; Samad, M.; Vidyaratne, L.; Glandon, A.; Iftekharuddin, K. Survey on deep neural networks in speech and vision systems. *Neurocomputing*, v. 417, p. 302–321, 2020. ISSN 0925-2312.

Assunção et al. 2019 Assunção, F.; Lourenço, N.; Machado, P.; Ribeiro, B. Denser: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines*, Springer, v. 20, n. 1, p. 5–35, 2019.

Assunção et al. 2019 Assunção, F.; Lourenço, N.; Machado, P.; Ribeiro, B. Fast denser: Efficient deep neuroevolution. In: Springer. *European Conference on Genetic Programming*. 2019. p. 197–212.

Assunção et al. 2019 Assunção, F.; Lourenço, N.; Machado, P.; Ribeiro, B. Fast-denser++: Evolving fully-trained deep artificial neural networks. *arXiv preprint arXiv:1905.02969*, 2019.

Balaha, Balaha and Ali 2021 Balaha, H. M.; Balaha, M. H.; Ali, H. A. Hybrid covid-19 segmentation and recognition framework (hmb-hcf) using deep learning and genetic algorithms. *Artificial Intelligence in Medicine*, Elsevier, v. 119, p. 102156, 2021.

Balaprakash et al. 2019 Balaprakash, P. et al. Scalable reinforcement-learning-based neural architecture search for cancer deep learning research. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2019. p. 1–33.

Byla and Pang 2020 Byla, E.; Pang, W. Deepswarm: Optimising convolutional neural networks using swarm intelligence. In: Springer. *Advances in Computational Intelligence Systems: Contributions Presented at the 19th UK Workshop on Computational Intelligence, September 4-6, 2019, Portsmouth, UK 19*. 2020. p. 119–130.

Castle and Johnson 2010 Castle, T.; Johnson, C. G. Positional effect of crossover and mutation in grammatical evolution. In: Springer. *European Conference on Genetic Programming*. 2010. p. 26–37.

Chen and Hsieh 2020 Chen, X.; Hsieh, C.-J. Stabilizing differentiable architecture search via perturbation-based regularization. In: *International Conference on Machine Learning*. 2020. p. 1554–1565.

Chen et al. 2019 Chen, X.; Xie, L.; Wu, J.; Tian, Q. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019. p. 1294–1303.

Chu, Zhang and Xu 2020 Chu, X.; Zhang, B.; Xu, R. Moga: Searching beyond mobilenetv3. In: IEEE. *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020. p. 4042–4046.

Chu et al. 2020 Chu, X.; Zhou, T.; Zhang, B.; Li, J. Fair darts: Eliminating unfair advantages in differentiable architecture search. In: *European Conference on Computer Vision*. 2020. p. 465–480.

Dasgupta and McGregor 1992 Dasgupta, D.; McGregor, D. R. Designing application-specific neural networks using the structured genetic algorithm. In: IEEE. *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. 1992. p. 87–96.

Deb et al. 2002 Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, IEEE, v. 6, n. 2, p. 182–197, 2002.

Ding et al. 2020 Ding, S.; Chen, T.; Gong, X.; Zha, W.; Wang, Z. Autospeech: Neural architecture search for speaker recognition. *arXiv preprint arXiv:2005.03215*, 2020.

Ding et al. 2021 Ding, Y.; Yao, Q.; Zhao, H.; Zhang, T. Diffmg: Differentiable meta graph search for heterogeneous graph neural networks. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2021. p. 279–288.

Ding et al. 2021 Ding, Z. et al. Bnas: Efficient neural architecture search using broad scalable architecture. *IEEE Transactions on Neural Networks and Learning Systems*, IEEE, 2021.

Dong et al. 2022 Dong, J. et al. A cell-based fast memetic algorithm for automated convolutional neural architecture design. *IEEE Transactions on Neural Networks and Learning Systems*, 2022. To appear.

Dong and Yang 2019 Dong, X.; Yang, Y. Searching for a robust neural architecture in four gpu hours. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019. p. 1761–1770.

Dorigo and Gambardella 1997 Dorigo, M.; Gambardella, L. M. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, IEEE, v. 1, n. 1, p. 53–66, 1997.

Eiben, Smith et al. 2003 Eiben, A. E.; Smith, J. E. et al. *Introduction to evolutionary computing*. : Springer, 2003. v. 53.

Fang et al. 2020 Fang, J. et al. Fna++: Fast network adaptation via parameter remapping and architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 43, n. 9, p. 2990–3004, 2020.

Freitas 2004 Freitas, A. A. A critical review of multi-objective optimization in data mining: a position paper. *ACM SIGKDD Explorations Newsletter*, ACM, v. 6, n. 2, p. 77–86, 2004.

Freitas 2013 Freitas, A. A. *Data mining and knowledge discovery with evolutionary algorithms*. : Springer Science & Business Media, 2013.

Fukushima 1975 Fukushima, K. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, Springer, v. 20, n. 3-4, p. 121–136, 1975.

Géron 2019 Géron, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* : O'Reilly Media, 2019.

Gonzalez and Miikkulainen 2020 Gonzalez, S.; Miikkulainen, R. Improved training speed, accuracy, and data utilization through loss function optimization. In: IEEE. *2020 IEEE congress on evolutionary computation (CEC).* 2020. p. 1–8.

Guo et al. 2022 Guo, Q.; Wu, X.-J.; Kittler, J.; Feng, Z. Differentiable neural architecture learning for efficient neural networks. *Pattern Recognition*, Elsevier, v. 126, p. 108448, 2022.

Guo et al. 2020 Guo, Y.; Luo, Y.; He, Z.; Huang, J.; Chen, J. Hierarchical neural architecture search for single image super-resolution. *IEEE Signal Processing Letters*, IEEE, v. 27, p. 1255–1259, 2020.

Harp, Samad and Guha 1990 Harp, S. A.; Samad, T.; Guha, A. Designing application-specific neural networks using the genetic algorithm. In: *Advances in neural information processing systems.* 1990. p. 447–454.

Hassanat et al. 2019 Hassanat, A. et al. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, MDPI, v. 10, n. 12, p. 390, 2019.

He et al. 2015 He, K.; Zhang, X.; Ren, S.; Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: *Proceedings of the IEEE international conference on computer vision.* 2015. p. 1026–1034.

He et al. 2016 He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016. p. 770–778.

Hopcroft and Ullman 1969 Hopcroft, J. E.; Ullman, J. D. *Formal languages and their relation to automata.* : Addison-Wesley Longman Publishing Co., Inc., 1969.

Huan, Quanming and Weiwei 2021 Huan, Z.; Quanming, Y.; Weiwei, T. Search to aggregate neighborhood for graph neural network. In: IEEE. *2021 IEEE 37th International Conference on Data Engineering (ICDE).* 2021. p. 552–563.

Huang et al. 2017 Huang, G.; Liu, Z.; Maaten, L. V. D.; Weinberger, K. Q. Densely connected convolutional networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2017. p. 4700–4708.

Huang et al. 2023 Huang, L. et al. U-darts: Uniform-space differentiable architecture search. *Information Sciences*, Elsevier, v. 628, p. 339–349, 2023.

Jankowski and Jackowski 2015 Jankowski, D.; Jackowski, K. Evolutionary algorithm for decision tree induction. In: Springer. *IFIP International Conference on Computer Information Systems and Industrial Management.* 2015. p. 23–32.

Jiang et al. 2021 Jiang, H.; Shen, F.; Gao, F.; Han, W. Learning efficient, explainable and discriminative representations for pulmonary nodules classification. *Pattern Recognition*, Elsevier, v. 113, p. 107825, 2021.

Jin, Song and Hu 2019 Jin, H.; Song, Q.; Hu, X. Auto-keras: An efficient neural architecture search system. In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019. p. 1946–1956.

Kordmahalleh et al. 2017 Kordmahalleh, M. M.; Sefidmazgi, M. G.; Harrison, S. H.; Homaifar, A. Identifying time-delayed gene regulatory networks via an evolvable hierarchical recurrent neural network. *BioData mining*, Springer, v. 10, p. 1–25, 2017.

Krizhevsky, Hinton et al. 2009 Krizhevsky, A.; Hinton, G. et al. Learning multiple layers of features from tiny images. Citeseer, 2009.

Lakhmiri, Digabel and Tribes 2021 Lakhmiri, D.; Digabel, S. L.; Tribes, C. Hypernomad: Hyperparameter optimization of deep neural networks using mesh adaptive direct search. *ACM Transactions on Mathematical Software (TOMS)*, ACM New York, NY, USA, v. 47, n. 3, p. 1–27, 2021.

Li and Talwalkar 2020 Li, L.; Talwalkar, A. Random search and reproducibility for neural architecture search. In: PMLR. *Uncertainty in artificial intelligence*. 2020. p. 367–377.

Li et al. 2019 Li, X.; Zhou, Y.; Pan, Z.; Feng, J. Partial order pruning: for best speed/accuracy trade-off in neural architecture search. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019. p. 9145–9153.

Liberis, Dudziak and Lane 2021 Liberis, E.; Dudziak, Ł.; Lane, N. D. μnas: Constrained neural architecture search for microcontrollers. In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. 2021. p. 70–79.

Liu et al. 2020 Liu, B. et al. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. In: *proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020. p. 2636–2645.

Liu et al. 2021 Liu, C.; Tian, Y.; Chen, Z.; Jiao, J.; Ye, Q. Adaptive linear span network for object skeleton detection. *IEEE transactions on image processing*, IEEE, v. 30, p. 5096–5108, 2021.

Liu et al. 2018 Liu, H.; Simonyan, K.; Vinyals, O.; Fernando, C.; Kavukcuoglu, K. Hierarchical representations for efficient architecture search. In: *International Conference on Learning Representations*. 2018.

Liu, Simonyan and Yang 2018 Liu, H.; Simonyan, K.; Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Long, Shelhamer and Darrell 2015 Long, J.; Shelhamer, E.; Darrell, T. Fully convolutional networks for semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015. p. 3431–3440.

Lourenço et al. 2018 Lourenço, N.; Assunção, F.; Pereira, F. B.; Costa, E.; Machado, P. Structured grammatical evolution: a dynamic approach. In: *Handbook of Grammatical Evolution*. : Springer, 2018. p. 137–161.

Lourenço, Pereira and Costa 2015 Lourenço, N.; Pereira, F. B.; Costa, E. Sge: a structured representation for grammatical evolution. In: Springer. *International Conference on Artificial Evolution (Evolution Artificielle)*. 2015. p. 136–148.

Lu et al. 2020 Lu, Z.; Deb, K.; Goodman, E.; Banzhaf, W.; Boddeti, V. N. Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search. In: Springer. *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*. 2020. p. 35–51.

Lu et al. 2021 Lu, Z. et al. Neural architecture transfer. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 43, n. 9, p. 2971–2989, 2021.

Marchisio et al. 2020 Marchisio, A. et al. Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020. p. 1–9.

McNally et al. 2021 McNally, W.; Vats, K.; Wong, A.; McPhee, J. Evopose2d: Pushing the boundaries of 2d human pose estimation using accelerated neuroevolution with weight transfer. *IEEE Access*, IEEE, v. 9, p. 139403–139414, 2021.

Miranda et al. 2023 Miranda, T. Z.; Sardinha, D. B.; Neri, F.; Basgalupp, M. P.; Cerri, R. A grammar-based multi-objective neuroevolutionary algorithm to generate fully convolutional networks with novel topologies. *Applied Soft Computing*, Elsevier, v. 149, p. 110967, 2023.

O'Neill and Ryan 2001 O'Neill, M.; Ryan, C. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 5, n. 4, p. 349–358, 2001.

Otter, Medina and Kalita 2020 Otter, D. W.; Medina, J. R.; Kalita, J. K. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, IEEE, v. 32, n. 2, p. 604–624, 2020.

O'Neill et al. 2004 O'Neill, M.; Brabazon, A.; Nicolau, M.; Garraghy, S. M.; Keenan, P. π grammatical evolution. In: Springer. *Genetic and Evolutionary Computation–GECCO 2004: Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004. Proceedings, Part II*. 2004. p. 617–629.

Pan et al. 2021 Pan, Z. et al. Autostg: Neural architecture search for predictions of spatio-temporal graph. In: *Proceedings of the Web Conference 2021*. 2021. p. 1846–1855.

Paria, Kandasamy and Póczos 2020 Paria, B.; Kandasamy, K.; Póczos, B. A flexible framework for multi-objective bayesian optimization using random scalarizations. In: PMLR. *Uncertainty in Artificial Intelligence*. 2020. p. 766–776.

Pedregosa et al. 2011 Pedregosa, F. et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

Perez et al. 2021 Perez, Q.; Jean, P.-A.; Urtado, C.; Vauttier, S. Bug or not bug? that is the question. In: IEEE. *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 2021. p. 47–58.

Pham et al. 2018 Pham, H.; Guan, M.; Zoph, B.; Le, Q.; Dean, J. Efficient neural architecture search via parameters sharing. In: *International Conference on Machine Learning*. 2018. p. 4095–4104.

Rapaport, Shriki and Puzis 2019 Rapaport, E.; Shriki, O.; Puzis, R. Eegnas: Neural architecture search for electroencephalography data analysis and decoding. In: Springer. *Human Brain and Artificial Intelligence: First International Workshop, HBAI 2019, Held in Conjunction with IJCAI 2019, Macao, China, August 12, 2019, Revised Selected Papers 1*. 2019. p. 3–20.

Real et al. 2019 Real, E.; Aggarwal, A.; Huang, Y.; Le, Q. V. Regularized evolution for image classifier architecture search. In: *Proceedings of the aaai conference on artificial intelligence*. 2019. v. 33, n. 01, p. 4780–4789.

Ronneberger, Fischer and Brox 2015 Ronneberger, O.; Fischer, P.; Brox, T. U-net: Convolutional networks for biomedical image segmentation. In: Springer. *International Conference on Medical image computing and computer-assisted intervention*. 2015. p. 234–241.

Rothlauf and Oetzel 2006 Rothlauf, F.; Oetzel, M. On the locality of grammatical evolution. In: Springer. *European conference on genetic programming*. 2006. p. 320–330.

Sabour, Frosst and Hinton 2017 Sabour, S.; Frosst, N.; Hinton, G. E. Dynamic routing between capsules. *Advances in neural information processing systems*, v. 30, 2017.

Sandler et al. 2018 Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018. p. 4510–4520.

Shen et al. 2022 Shen, H.; Zhao, Z.-Q.; Liao, W.; Tian, W.; Huang, D.-S. Joint operation and attention block search for lightweight image restoration. *Pattern Recognition*, Elsevier, v. 132, p. 108909, 2022.

Shi et al. 2022 Shi, M. et al. Genetic-gnn: Evolutionary architecture search for graph neural networks. *Knowledge-based systems*, Elsevier, v. 247, p. 108752, 2022.

Simonyan and Zisserman 2014 Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sinha and Chen 2021 Sinha, N.; Chen, K.-W. Evolving neural architecture using one shot model. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2021. p. 910–918.

Slowik and Kwasnicka 2020 Slowik, A.; Kwasnicka, H. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, Springer, v. 32, p. 12363–12379, 2020.

Stamoulis et al. 2019 Stamoulis, D. et al. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In: Springer. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 2019. p. 481–497.

Stanley et al. 2019 Stanley, K. O.; Clune, J.; Lehman, J.; Miikkulainen, R. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, Nature Publishing Group, v. 1, n. 1, p. 24–35, 2019.

Sun et al. 2018 Sun, Y.; Xue, B.; Zhang, M.; Yen, G. G. A particle swarm optimization-based flexible convolutional autoencoder for image classification. *IEEE transactions on neural networks and learning systems*, IEEE, v. 30, n. 8, p. 2295–2309, 2018.

Sun et al. 2019 Sun, Y.; Xue, B.; Zhang, M.; Yen, G. G. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 24, n. 2, p. 394–407, 2019.

Termritthikun et al. 2021 Termritthikun, C.; Jamtsho, Y.; Ieamsaard, J.; Muneesawang, P.; Lee, I. Eeea-net: An early exit evolutionary neural architecture search. *Engineering Applications of Artificial Intelligence*, Elsevier, v. 104, p. 104397, 2021.

Tian et al. 2020 Tian, Y. et al. Off-policy reinforcement learning for efficient and effective gan architecture search. In: Springer. *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VII 16*. 2020. p. 175–192.

Urbanowicz and Moore 2009 Urbanowicz, R. J.; Moore, J. H. Learning classifier systems: a complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, Wiley Online Library, v. 2009, n. 1, p. 736398, 2009.

Véniat, Schwander and Denoyer 2019 Véniat, T.; Schwander, O.; Denoyer, L. Stochastic adaptive neural architecture search for keyword spotting. In: IEEE. *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019. p. 2842–2846.

Wang et al. 2021 Wang, L.; Xie, S.; Li, T.; Fonseca, R.; Tian, Y. Sample-efficient neural architecture search by learning actions for monte carlo tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 44, n. 9, p. 5503–5515, 2021.

Wang et al. 2022 Wang, R. et al. Lighthubert: Lightweight and configurable speech representation learning with once-for-all hidden-unit bert. *arXiv preprint arXiv:2203.15610*, 2022.

Wang et al. 2020 Wang, T.-H. et al. Autorec: An automated recommender system. In: *Proceedings of the 14th ACM Conference on Recommender Systems*. 2020. p. 582–584.

Wang et al. 2021 Wang, X. et al. Bix-nas: Searching efficient bi-directional architecture for medical image segmentation. In: Springer. *Medical Image Computing and Computer Assisted Intervention–MICCAI 2021: 24th International Conference, Strasbourg, France, September 27–October 1, 2021, Proceedings, Part I 24*. 2021. p. 229–238.

Wei, Zhao and He 2022 Wei, L.; Zhao, H.; He, Z. Designing the topology of graph neural networks: A novel feature fusion perspective. In: *Proceedings of the ACM Web Conference 2022*. 2022. p. 1381–1391.

Weng et al. 2019 Weng, Y.; Zhou, T.; Li, Y.; Qiu, X. Nas-unet: Neural architecture search for medical image segmentation. *IEEE access*, IEEE, v. 7, p. 44247–44257, 2019.

Weng et al. 2019 Weng, Y.; Zhou, T.; Liu, L.; Xia, C. Automatic convolutional neural architecture search for image classification under different scenes. *IEEE access*, IEEE, v. 7, p. 38495–38506, 2019.

Whitley et al. 1989 Whitley, L. D. et al. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In: Fairfax, VA. *Icga*. 1989. v. 89, p. 116–123.

Wohlin 2014 Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 2014. p. 1–10.

Xiao et al. 2022 Xiao, H.; Wang, Z.; Zhu, Z.; Zhou, J.; Lu, J. Shapley-nas: Discovering operation contribution for neural architecture search. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022. p. 11892–11901.

Xie and Yuille 2017 Xie, L.; Yuille, A. Genetic cnn. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017. p. 1379–1388.

Xie et al. 2019 Xie, S.; Zheng, H.; Liu, C.; Lin, L. SNAS: stochastic neural architecture search. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. : OpenReview.net, 2019.

Xue, Chen and Słowik 2023 Xue, Y.; Chen, C.; Słowik, A. Neural architecture search based on a multi-objective evolutionary algorithm with probability stack. *IEEE Transactions on Evolutionary Computation*, IEEE, 2023.

Xue et al. 2021 Xue, Y.; Jiang, P.; Neri, F.; Liang, J. A multi-objective evolutionary approach based on graph-in-graph for neural architecture search of convolutional neural networks. *Int. J. Neural Syst.*, v. 31, n. 9, p. 2150035:1–2150035:17, 2021.

Xue et al. 2021 Xue, Y.; Jiang, P.; Neri, F.; Liang, J. A multi-objective evolutionary approach based on graph-in-graph for neural architecture search of convolutional neural networks. *International Journal of Neural Systems*, v. 31, n. 09, p. 2150035, 2021.

Yan et al. 2020 Yan, J.; Chen, S.; Zhang, Y.; Li, X. Neural architecture search for compressed sensing magnetic resonance image reconstruction. *Computerized Medical Imaging and Graphics*, Elsevier, v. 85, p. 101784, 2020.

Yang et al. 2020 Yang, Z. et al. Cars: Continuous evolution for efficient neural architecture search. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020. p. 1829–1838.

Yao et al. 2020 Yao, Q.; Chen, X.; Kwok, J. T.; Li, Y.; Hsieh, C.-J. Efficient neural interaction function search for collaborative filtering. In: *Proceedings of The web conference 2020*. 2020. p. 1660–1670.

Ye et al. 2022 Ye, P. et al. b-darts: Beta-decay regularization for differentiable architecture search. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022. p. 10874–10883.

Ying et al. 2019 Ying, C. et al. Nas-bench-101: Towards reproducible neural architecture search. In: PMLR. *International conference on machine learning*. 2019. p. 7105–7114.

Yu et al. 2021 Yu, Z. et al. Searching multi-rate and multi-modal temporal enhanced networks for gesture recognition. *IEEE Transactions on Image Processing*, IEEE, v. 30, p. 5626–5640, 2021.

Zhang et al. 2020 Zhang, M. et al. One-shot neural architecture search: Maximising diversity to overcome catastrophic forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 43, n. 9, p. 2921–2935, 2020.

Zhang et al. 2020 Zhang, X.; Huang, Z.; Wang, N.; Xiang, S.; Pan, C. You only search once: Single shot neural architecture search via direct sparse optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 43, n. 9, p. 2891–2904, 2020.

Zheng et al. 2021 Zheng, X. et al. Migo-nas: Towards fast and generalizable neural architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 43, n. 9, p. 2936–2952, 2021.

Zhou et al. 2021 Zhou, K.; Yang, Y.; Cavallaro, A.; Xiang, T. Learning generalisable omni-scale representations for person re-identification. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 44, n. 9, p. 5056–5069, 2021.

Zhou, Xie and Kung 2021 Zhou, Y.; Xie, X.; Kung, S.-Y. Exploiting operation importance for differentiable neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, IEEE, 2021.

Zimmer, Lindauer and Hutter 2021 Zimmer, L.; Lindauer, M.; Hutter, F. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 43, n. 9, p. 3079–3090, 2021.

Zitzler 1999 Zitzler, E. *Evolutionary algorithms for multiobjective optimization: Methods and applications.* : Citeseer, 1999. v. 63.

Zoph and Le 2016 Zoph, B.; Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Zoph et al. 2018 Zoph, B.; Vasudevan, V.; Shlens, J.; Le, Q. V. Learning transferable architectures for scalable image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018. p. 8697–8710.

Appendix

Links to repositories of reproducible works

Table 7 – Links to repositories of reproducible works

Reference	URL
(Kordmahalleh et al. 2017)	<https://github.com/mmoradik/GRN>
(Rapaport, Shriki and Puzis 2019)	<https://bitbucket.org/eladrapaport/eegnas>
(Véniat, Schwander and Denoyer 2019)	<http://github.com/TomVeniat/SANAS>
(Weng et al. 2019)	<https://github.com/tianbaochou/NasUnet>
(Weng et al. 2019)	<https://github.com/tianbaochou/CNAS>
(Jin, Song and Hu 2019)	<https://autokeras.com/>
(Balaprakash et al. 2019)	<https://github.com/scrlnas2019/nas4candle>
(Tian et al. 2020)	<https://github.com/Yuantian013/E2GAN>
(Ding et al. 2020)	<https://github.com/VITA-Group/AutoSpeech>
(Chu et al. 2020)	<https://github.com/xiaomi-automl/FairDARTS>
(Lu et al. 2020)	<https://github.com/mikelzc1990/nsganetv2>
(Byla and Pang 2020)	<https://github.com/Pattio/DeepSwarm>
(Stamoulis et al. 2019)	<https://github.com/enyac-group/single-path-nas>
(Guo et al. 2020)	<https://github.com/guoyongcs/HNAS-SR>
(Gonzalez and Miikkulainen 2020)	<https://github.com/sgonzalez/SwiftCMA>
(Yao et al. 2020)	<https://github.com/quanmingyao/SIF>
(Yan et al. 2020)	<https://github.com/yjump/NAS-for-CSMRI>
(Liu et al. 2020)	<https://github.com/zhuchenxv/AutoFIS>
(Chu, Zhang and Xu 2020)	<https://github.com/xiaomi-automl/MoGA>
(Sun et al. 2019)	<https://github.com/sunkevin1214/codes>
(Marchisio et al. 2020)	<https://github.com/ehw-fit/nascaps>
(Wang et al. 2020)	<https://github.com/datamllab/AutoRec>
(Fang et al. 2020)	<https://github.com/JaminFong/FNA>
(Zhang et al. 2020)	URL is too long, mouse over this
(Zhang et al. 2020)	<https://github.com/XinbangZhang/DSO-NAS>
(Balaha, Balaha and Ali 2021)	URL is too long, mouse over this
(Ding et al. 2021)	<https://github.com/LARS-research/DiffMG>
(Termritthikun et al. 2021)	<https://github.com/chakkritte/EEEA-Net>
(Zimmer, Lindauer and Hutter 2021)	<https://github.com/automl/Auto-PyTorch>
(Lu et al. 2021)	URL is too long, mouse over this
(Liu et al. 2021)	<https://github.com/sunsmarterjie/SDL-Skeleton>
(Yu et al. 2021)	<https://github.com/ZitongYu/3DCDC-NAS>
(Wang et al. 2021)	<https://github.com/tiangexiang/BiX-NAS>
(McNally et al. 2021)	<https://github.com/wmcnally/evopose2d>
(Liberis, Dudziak and Lane 2021)	<https://github.com/eliberis/uNAS>
(Pan et al. 2021)	<https://github.com/panzheyi/AutoSTG>
(Lakhmiri, Digabel and Tribes 2021)	<https://github.com/bbopt/HyperNOMAD>
(Huan, Quanming and Weiwei 2021)	<https://github.com/AutoML-4Paradigm/SANE>
(Perez et al. 2021)	<https://github.com/qperez/ATTIC>
(Jiang et al. 2021)	<https://github.com/fei-aiart/NAS-Lung>
(Wang et al. 2021)	<https://github.com/facebookresearch/LaMCTS>
(Shen et al. 2022)	<https://github.com/it-hao/JSNet>
(Guo et al. 2022)	<https://github.com/QingbeiGuo/DNAL>
(Zhou et al. 2021)	<https://github.com/KaiyangZhou/deep-person-reid>
(Wei, Zhao and He 2022)	<https://github.com/LARS-research/F2GNN>
(Shi et al. 2022)	<https://github.com/codeshareabc/Genetic-GNN>
(Wang et al. 2022)	<https://github.com/mechanicalsea/lighthubert>