Evolutionary AutoML

Automatic CNN Generation with BNF Grammars

HANI Moad HOANG Thi Huyen Trang HOFFMANN Yann

ABSTRACT

In recent years, deep learning with Convolutional Neural Network (CNN) as its most prominent technique, has been successfully applied in many domains such as image classification and natural language processing. However, designing CNNs is not easy to master, it involves a large number of choices from many possible hyperparameters, and does not guarantee obtaining the best results. To solve this problem, recent studies have focused on the automatic generation of CNNs and have achieved positive results. In this study, we apply Genetic Evolution (GE) to design and optimize CNN architecture. We propose BNF grammars that define and construct the structure of CNN models. To evaluate our approach, we use the CIFAR-10 dataset to validate the evolution of the generated models, using the metric of accuracy. The obtained results show that the proposed grammars achieve competitive results when compared with similar state-of-the-art approaches.

Keywords

grammatical evolution, convolutional neural networks, automatic design

1. INTRODUCTION

Evolution being a randomized generate-and-test process presents some similarities to the monkey-with-typewriter process. However, natural selection can produce unlikely results. Could a monkey accidentally type the Hamlet line "methinks it is like a weasel"? The chances are virtually zero. So, how does an evolutionary algorithm _do?

In fact, the odds of a monkey exactly typing a complete work such as Shakespeare's Hamlet is so tiny that the chance of it occurring during a period of time of the order of the age of the universe is minuscule, but not zero. For this kind of problems and according to Richard Dawkin's Weasel, the search space is a set of strings of characters of fixed length, the Number of errors in the string is the fitness that should be minimized and of course we need an algorithm that will randomly generate an initial string of characters and repeat this process until a new generation (target string) has been found; it's The Stochastic Hill-climber Algorithm.

Figure 1: Monkey typing

Theory can help understanding and improving evolutionary algorithms (EA) since when working with EA, it is easy to conjecture some general rule from observations, but without theory it is hard to distinguish between "we often observe" and "it is true that.

- Reason: it is often hard to falsify a conjecture experimentally (the conjecture might be true "often enough", but not in general).
- Danger: misconceptions prevail in the EA community and mislead the future development of the field.

By analyzing rigorously simplified situations, theory can suggest

- which algorithm to use
- which representation to use
- which operators to use
- how to choose parameters
- As with all particular research results, the question is how representative such a result is for the general usage of GA.

2. STATE-OF-THE-ART

Evolutionary algorithms form a subset of evolutionary computation in that they generally only involve techniques implementing mechanisms inspired by biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within which the solutions "live" (see also fitness function). Evolution of the population then takes place after the repeated application of the above operators.

In this process, there are two main forces that form the basis of evolutionary systems: Recombination and mutation create the necessary diversity and thereby facilitate novelty, while selection acts as a force increasing quality.

Many aspects of such an evolutionary process are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen. On the other hand, selection operators can be either deterministic, or stochastic. In the latter case, individuals with a higher fitness have a higher chance to be selected than individuals with a lower fitness, but typically even the weak individuals have a chance to become a parent or to survive.

Evolutionary computation practitioners - Incomplete list: Kalyanmoy Deb - David E. Goldberg - John Henry Holland - John Koza - Peter Nordin -Ingo Rechenberg - Hans-Paul Schwefel - Peter J. Fleming - Carlos M. Fonseca - Lee Graham...

The use of Darwinian principles for automated problem solving originated in the fifties. It was not until the sixties that three distinct interpretations of this idea started to be developed in three different places.

Evolutionary programming was introduced by Lawrence J. Fogel in the US, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies.

These areas developed separately for about 15 years. From the early nineties on they are unified as different representatives ("dialects") of one technology, called evolutionary computing. Also in the early nineties, a fourth stream following the general ideas had emerged – genetic programming. Since the 1990s, evolutionary computation has largely become swarm-based computation, and nature-inspired algorithms are becoming an increasingly significant part.

These terminologies denote the field of evolutionary computing and consider evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas.

Genetic Algorithms have the ability to deliver a 'good-enough' solution 'fast-enough'. This makes genetic algorithms attractive for use in solving optimization problems. The reasons why GAs are needed are as follows:

- Solving Difficult Problems

In computer science, there is a large set of problems, which are NP-Hard. What this essentially means is that, even the most powerful computing systems take a very long time (even years!) to solve that problem. In such a scenario, GAs prove to be an efficient tool to provide usable near-optimal solutions in a short amount of time.

- Failure of Gradient Based Methods

Traditional calculus based methods work by starting at a random point and by moving in the direction of the gradient, till we reach the top of the hill. This technique is efficient and works very well for single-peaked objective functions like the cost function in linear regression. But, in most real-world situations, we have a very complex problem called as landscapes, which are made of many peaks and many valleys, which causes such methods to fail, as they suffer from an inherent tendency of getting stuck at the local optima.

That being said, GAs are very general in nature, and just applying them to any optimization problem wouldn't give good results. In fact, In genetic algorithms, there is no "one size fits all" or a magic formula which works for all problems. Even after the initial GA is ready, it takes a lot of time and effort to play around with the parameters like population size, mutation and crossover probability etc. to find the ones which suit the particular problem.

Grammatical Evolution was firstly introduced by Ryan et al.[9][11], who explored a unique way of using grammars to evolve programs in the aspect of automatic programming. To describe the working scheme in Grammatical Evolution briefly, GE uses an evolutionary algorithm to evolve variable-length binary strings, which are considered as the genome of individuals and used to represent corresponding integer string codons. At the same time, these integer codons can determine which derivation rule is going to be used to produce mathematical expression, string, or even program segment needed. Moreover, all these codons work together to form a valid solution. The details of this working mechanism of GE will be discussed later. GE is set up so that the EA component is an independent module out of the outputted program by taking the virtue of the genotype-phenotype mapping mechanism. And the BNF, like the search algorithm, is a plugin part of the system that in charge of the outputted language and syntax. Based on these characters, GE theoretically has the ability to evolve programs in any computer language.

GE Mechanism Similar to other Evolutionary Algorithms, Grammatical Evolution got inspiration from the biological process in nature. It is simulating the process of production of protein from the genetic material of an organism, and protein is the fundamental material for an organism to maintain basic live operation and expression of heritable traits [5].

Discussion: Grammatical Evolution is such a method to theoretically solve almost any kinds of problem in the way of optimization consistently if the definition of the problem is precise and adequate. But as a matter of fact, still, many hindrances are placed on the way towards that possibility. The limit of computation power causes some of the hindrances, and some of other hindrances are the result of the structure of GE itself. In this section, the main problem GE is currently facing will be discussed, and some immature personal ideas are declared as well.

The first hindrance, which is also the least important one for GE is the limitation of computation power. As it has been declared in the previous part of this chapter, Grammatical Evolution still uses EA as its core to evolve its population. Due to the design of Evolutionary Algorithm, it always needs to maintain a relatively significant population to keep those 'potential' gene fragments for the global optimum or even local optimum, which may be dispersed in many different individuals throughout the entire evolution process. This design demands more computation re-sources for sure, if we compare this to those strongly oriented searching methods. However, it is also the essences of EA as well as nature, that the composed of several simple parts can sometimes produce surprising results.

The Grammar file, which is used to indicate how genotype is mapped into phenotype in grammatical evolution, also has a significant influence on the performance of GE. Different from our intuition, the grammar file is not merely an external file for grammatical evolution system. It plays one of the essential roles in the whole process of evolution. The below illustrates the mechanism of grammatical evolution from another perspective. Mapping, search mechanism, and evaluating mechanism include almost all manipulation we have to solve our problem. Among these, the gram-mar file defines every rule of mapping process wheres the design of GE algorithm controls the mapping mechanism. Any tiny modification in the grammar file can cause a considerable difference in the GE process. However, in the field of GE, such vital files have to be purely written by people. This lead to the fact that the great performance of Grammatical Evolution has great reliance on an expert-written and well-designed grammar file for most problems. This problem is not severe in some widely-used test problems since many different grammars have been tested in the community for millions of times and many researchers have done much work for these, and these grammar files are acceptable for these problems. But in the more general case, especially for those applications or user-customized problems from non-expert users, the grammar file from them may become the road-block toward the better performance of GE. Dirk Schweim et al. [12] studied the structure of grammar for GE and advised the average branching factor, which is the expected number of non-terminals chosen in mapping one genotype codon to a phenotype tree code, should be as close to 1 as possible to help with the efficiency of GE. However, it is still uneasy about writing a proper and efficient grammar for a specific problem, since the average branching factor is the only evaluation of an existing grammar.

Meanwhile, the design of GE itself also brought some problems. In canonical grammatical evolution, the selection of non-terminal is revealed by doing a mod calculation over the codons. Since the number of available derivation rules is usually small, it is easy to get the same result in this calculation, even if the codons are different. This mapping mechanism of grammatical evolution implies an N to 1 relationship between the genotype and phenotypes. That is, every phenotype has a large number of corresponding geno-

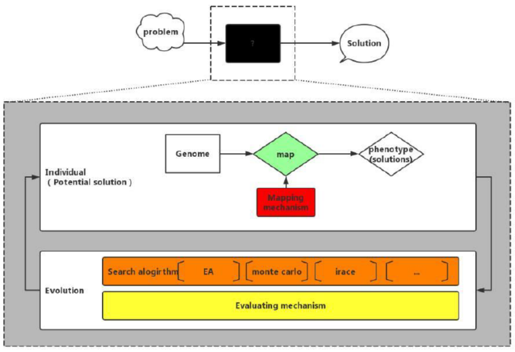


Figure 2: Another perspective of Grammatical Evolution

types. In theory, a phenotype in solution space is accessible to be found in the case that it has at least one correspondent point in the search space. The N to 1 relationship between genotypes and phenotypes in GE is highly redundant, as it can have a number of points in search space, it actually needs to locate the global optimum or local optimum we are searching for. This character may sometimes increase the possibility for GE to get the optimum point. However, it also decreases the efficiency of the search process, since many candidates genotypes tested in the search process may finally point to one same phenotype, and the evaluation of phenotype may be computation costly in some problems.

On the other side, this mapping mechanism also has the problem of low locality. The derivation of genotype to phenotype is a repeated nesting loop since the selection of one derivation rules can influence the later derivation. This derivation way may causes a phenomenon different from the expression of the gene in the natural world. Because in GE, a neighborhood genotype may have a phenotype with no similarity. This character diverses from our intuition that neighboring genotypes should usually correspond to neighboring phenotypes.

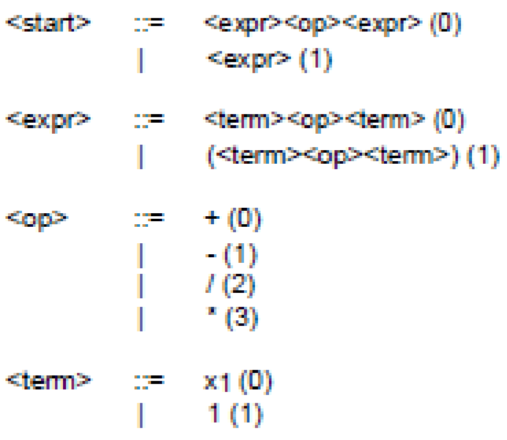
In other words, if we visualize the landscape of fitness of all possible points in search space, what we get is a rugged space full of ravines and spikes. The term of low locality is used to represent this character in the community of GE. It can cause the search process much harder since the direction of evolution is hard to find for either local or global optimums.

In fact, many variants of GE have tried to solve the problem of high redundancy and low locality. For example, the SGE system that made great efforts in this regard. However, in theory, a GE variant algorithm with lower redundancy and higher locality does not mean the change of search space. These two characters are only helpful for the GE system to improve its efficiency in the searching process. This leads to the fact that an 'advanced' GE variant system cannot make sure that it always performs better than a canonical GE system since they are usually dancing on the same stage.

Grammatical Evolution and Structured Grammatical Evolution (SGE)

In Grammatical Evolution, it is not necessary that all codons have to be used. In this example, the last codons are not used ,and the last two codons are remained and have totally no influence for the phenotype. On the other side, if the length of the genome is not long enough to derive all non-terminal, this individual will be regarded to be invalid. A technique called 'Wrapping' can be used to relieve the influence of problem. While wrapping is used and the genome of an individual is not long enough to derive a phenotype, the codons are used in a circle, just like the genome are wrapped up. After the last codon is used, the first codons of the genome are going to be orderly used from the first codon again. In addition, it is likely some slight differences may exist between different GE variants.

Structured Grammatical Evolution is a recent variant of canonical grammatical evolution, which was firstly published in the work by Lourenço et al. [8] from the University of Coimbra. One point to note is that, since they have also named their system as Structured Grammatical Evolu-tion (SGE), the word SGE can represent the algorithm as well as the corresponding system. In this section, the main difference between SGE and canonical GE will be introduced, and meanwhile, some important information about the corresponded SGE system will be delivered. As it has been introduced before, GE uses a context-free grammar to realize the target of mapping genotype into the phenotype. Due to the mechanism of it works, one of the problems it comes with is the problem of high redundancy and low locality, which could be potentially harmful to the efficiency of GE [10, 7]. SGE is proposed to relieve the issues of locality and redundancy of canonical grammatical evolution by replacing the context-free grammar to a structured mapping method. Different from the situation in GE, a one-to-one mapping mechanism between the genotype and the non-terminals are used in the SGE. To archive this target, a pre-processing procedure is required. By such a procedure, the standard context-free grammar can be translated into SGE-used grammar. In SGE, every genotype is represented by several sets of integers, rather than a long integer string in canonical grammatical evolution. Here, one example is used to demonstrate this difference of representing method between SGE and canonical grammatical evolution. Considering we have following context-free grammar:



```
<start>   ::=   <expr><op><expr> (0)
          |     <expr> (1)

<expr>    ::=   <term><op><term> (0)
          |     (<term><op><term>) (1)

<op>      ::=   + (0)
          |     - (1)
          |     / (2)
          |     * (3)

<term>    ::=   x1 (0)
          |     1 (1)
```

Figure 3: Example of context-free Grammar

One individual with the genotype of [27; 7; 55; 22; 3; 4; 30; 16; 203; 24] can be finally be derived into the phenotype (1=1) + x1 x1. And it is obvious that this phenotype can have many potential genotype because of the working mechianism of GE, such like [7; 7; 55; 22; 3; 4; 30; 16; 203; 24]. However, in SGE, one phenotype can only have one genotype. For this case, the genotype in SGE is written in [[0][1,0][2,0,3][1,1,0,0]]. Each bracket here is representing one non-terminal in order. In the first bracket we have only a 0, it means that for the first non-terminal, rule (0) of first non-terminal (<start>) will be used for derivation from the <start> to <expr><op><expr>. And for the second bracket, we have two value 1 and 0, which means rule (0) and rule (1) of second non-terminal (<expr>) will be used for derivation respectively to (<term><op><term>)<op> <term><op><term>. This process continues until the translation for all four non-terminals end. Figure 4 demonstrate this process in a more intuitive way. Due to the reason that codons controlling different kinds of non-terminals are separated, even they are still mapping in a depth-first way, there is no different to map all codons belongs to the same non-terminals into terminals according to the order of non-terminals at once, just as a layer structure does.

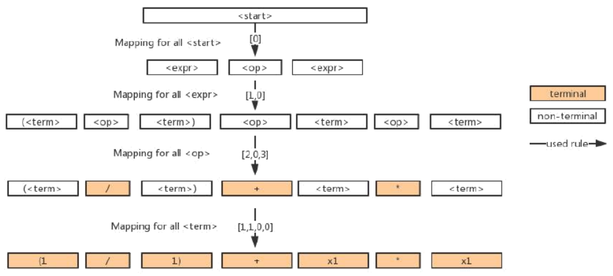
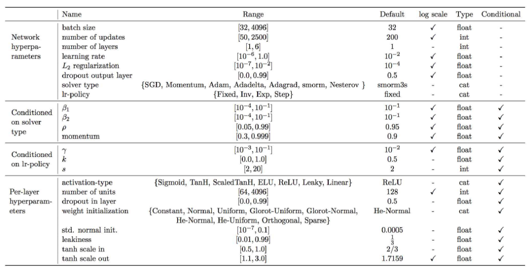


Figure 4: Example of mapping process in SG2E

This new mapping mechanism is the main difference between GE and SGE, which has also brought several characteristics as results of that, which are different from canonical GE: Because of the way SGE deal with grammar, no recursion in Grammar is permitted for the grammar file for SGE. Pre-processing is mandatory to translate standard context-free gram-mar to SGE-used grammar. The maximum recursion level must be pre-defined for transferring a context-free grammar to a grammar without any recursion, to limit the genotype size. All integers in genotype are bounded by the number of possible options of the corresponding non-terminals, as a result of every integer is representing. A derivation rule to use. However, in Grammatical Evolution, integers in genotype could theoretically be any natural number, since it uses a 'mod' calculation to choose which derivation rule to use. The SGE's structure ensures that one variation on one codon would not affect the derivation of other non-terminals, and this characteristic lead to the high locality in theory. In SGE, the relation between genotype and phenotype is always one to one, since every codon is directly referring to a derivation rules, whereas in GE, the relation between genotype and phenotype is usually N to 1. This design reduces the redundancy of canonical GE and does the search for optimum more efficient in theory. Since the shape of genotype in SGE is restricted to a set of list with the sizes of occurrence number of corresponding non-terminals, the variation operation in the evolution process has less diversity than canonical GE. For example, the crossover in SGE can only be performed on candidates with the same structure of genotype. Some advanced operation technique for GE (e.g., derivation-tree based crossover technique) is not permitted in SGE.

Hyper-parameter Tuning

Even though most of the Grammatical Evolution systems have the ability to evolve executable computer program fragments or mathematical expression automatically, some parameters are still necessary to control the process of the evolution process. As these parameters are on a high-level aspect, they are usually called hyper-parameters to distinguish from parameters in low-level aspects, such as those parameters in evolved programs. These hyper-parameters usually has a great influence on the performance of the Grammatical Evolution system. For example, when PonyGE2 system is being tested, several different hyper-parameter sets as follows are tested on the same problem:



	Name	Range	Default	log scale	Type	Conditional
Network hyperparameters	batch size	$[32, 4096]$	32	✓	float	-
	number of updates	$[50, 2500]$	200	✓	int	-
	number of layers	$[1, 6]$	1	-	int	-
	learning rate	$[10^{-6}, 1.0]$	10^{-2}	✓	float	-
	L_2 regularization	$[10^{-7}, 10^{-2}]$	10^{-4}	✓	float	-
	dropout output layer	$[0.0, 0.99]$	0.5	✓	float	-
	solver type	{SGD, Momentum, Adam, Adadelta, Adagrad, smorm, Nesterov }	smorm3s	-	cat	-
	lr-policy	{Fixed, Inv, Exp, Step}	fixed	-	cat	-
Conditioned on solver type	β_1	$[10^{-4}, 10^{-1}]$	10^{-1}	✓	float	✓
	β_2	$[10^{-4}, 10^{-1}]$	10^{-1}	✓	float	✓
	ρ	$[0.05, 0.99]$	0.95	✓	float	✓
	momentum	$[0.3, 0.999]$	0.9	✓	float	✓
Conditioned on lr-policy	γ	$[10^{-3}, 10^{-1}]$	10^{-2}	✓	float	✓
	k	$[0.0, 1.0]$	0.5	-	float	✓
	s	$[2, 20]$	2	-	int	✓
Per-layer hyperparameters	activation-type	{Sigmoid, TanH, ScaledTanH, ELU, ReLU, Leaky, Linear}	ReLU	-	cat	✓
	number of units	$[64, 4096]$	128	✓	int	✓
	dropout in layer	$[0.0, 0.99]$	0.5	-	float	✓
	weight initialization	{Constant, Normal, Uniform, Glorot-Uniform, Glorot-Normal, He-Normal, He-Uniform, Orthogonal, Sparse}	He-Normal	-	cat	✓
	std. normal init.	$[10^{-7}, 0.1]$	0.0005	-	float	✓
	leakiness	$[0.01, 0.99]$	$\frac{1}{3}$	-	float	✓
	tanh scale in	$[0.5, 1.0]$	2/3	-	float	✓
	tanh scale out	$[1.1, 3.0]$	1.7159	✓	float	✓

Figure 5: Hyperparameters

As explained during the final presentation, Hyperparameters can be very sensitive in fact a change sometimes - seen as 'small' - will always have great impact on the whole process of learning.

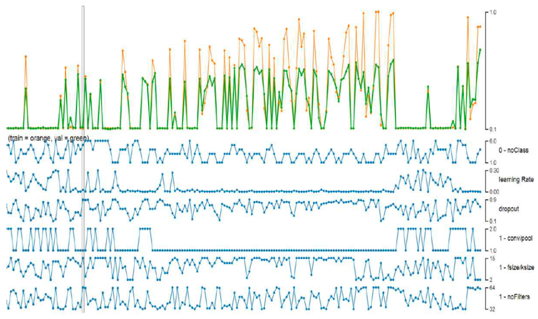


Figure 6: Sensitivity in Hyper-parameters tuning

3. PROBLEM PRESENTATION

The problematic is How can we create a universal language (ie. meaning based on a benchmark to get an optimized solution : a grammar to evolve CNN's).

Here we're about to answer some important questions in order to understand why should we use GE for this case study?

But before that let's understand why we do theory?

As previously mentioned many results can only be obtained by theory (ie: if we make a statement on a very large or even infinite set [all TSP instances on n vertices] or [all possible algorithms for a problem]), thus theory is crucial for results. Furthermore, we do theory because of the approach, in other words we want to understand how things work and this is exactly the working principle of evolution computation and secondly for self correctness and self guiding effect of proving in order to trigger for new ideas.

So, how can one explain that Genetic Algorithms is the best fit to solve this AutoML problemtic - Two reasons 1. First one related to the results that should be :

- Flexible: applicable to different problems
- Robust: can deal with noise
- Adaptive: deal with dynamic environments
- Autonomous: without human interactions
- Decentralized: without central authority

2. Secondly, the approach, in fact evolutionary algorithms approaches are suitable for not well behaved functions:

- Non-linear,
- Non-convex,
- Non-differentiable,
- Discontinuous.

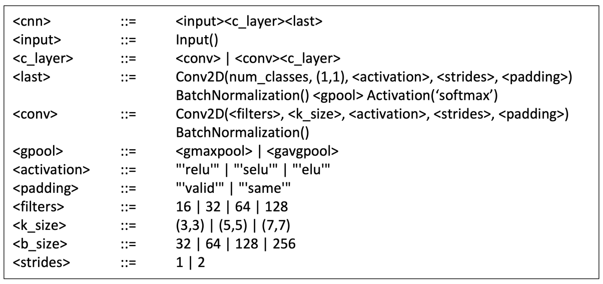
This being said, we're going to see in the next section, the adopted approach to solve the defined problematic.

4. APPROACH

In the quest for ever better performing neural networks, we take a reductionist approach on the problem. The two objects under study are grammars and their resulting CNN. The former can be viewed as a factory that sets the rigid rules of production of the latter. The act of squeezing the sheer complexity of CNN topology into basic rules invariably creates an informational bottleneck that sheds light onto CNNs themselves. We score the grammar by the accuracy of the best model they can produce within some pre-defined constraints. They are then ordered by their respective score.

We have divided our grammars into two subcategories.
The first (here named Evol Grammar), as in Figure 7, provides incremental changes to the IEEE grammar while significantly improving on the final score.

The second group due to its more experimental nature ends up being less fruitful. It however informs on potential future



```
<cnn>          ::=     <input><c_layer><last>
<input>        ::=     Input()
<c_layer>      ::=     <conv> | <conv><c_layer>
<last>         ::=     Conv2D(num_classes, (1,1), <activation>, <strides>, <padding>)
                       BatchNormalization() <gpool> Activation('softmax')
<conv>         ::=     Conv2D(<filters>, <k_size>, <activation>, <strides>, <padding>)
                       BatchNormalization()
<gpool>        ::=     <gmaxpool> | <gavgpool>
<activation>   ::=     "'relu'" | "'selu'" | "'elu'"
<padding>      ::=     "'valid'" | "'same'"
<filters>      ::=     16 | 32 | 64 | 128
<k_size>       ::=     (3,3) | (5,5) | (7,7)
<b_size>       ::=     32 | 64 | 128 | 256
<strides>      ::=     1 | 2
```

Figure 7: Evol Grammar.

improvements while highlighting important pitfalls with evolutionary algorithms. Both group have an associated control grammar to compare the performance. It is worth noting that after evolving their population's genomes, grammars return self-consistent performances. Meaning that although there is a highly non-linear relationship between the grammar and their score: a small modification in the source can make a big difference. If left untouched, the overall score of one grammar will convergence between runs. This non-trivial property makes the comparison of grammars possible. (See statistical testing)

In order to compare the performance of our proposed grammars – against our benchmark, the IEEE reference grammar, but also against each other – we have opted for frameworks that favor simplicity and code reproducibility.

The second iteration of PonyGE is a library that that allows the handling of BNF grammar in Python. The grammars, made of various building block, will yield networks of varying fitness thanks to this library. By abstracting the mapping process from phenotype to genetype, PonyGE enables its user to focus on analyzing the final models while retro-engineering the source grammar. The environmental settings that prevail during an evolutionary run can then be further refined. This is explained in more details in the next section.

We define the fitness of a model as the accuracy of its predictions. Out of several possible metrics that gauge the performance of CNNs, we chose accuracy for its important characteristics. Namely [put reasons why we chose accuracy]. It has proven versatile and allows us to seamlessly compare with the IEEE benchmark.

Let us turn our attention to the experimental application of our comparative approach.

5. EXPERIMENTATION

5.1 Dataset

The CIFAR-10 dataset [4] consists of 60,000 32x32 color images evenly distributed among 10 classes, with 6,000 images per class. There are 50,000 images for training and 10,000 images for testing.

5.2 Experimental Settings

In this study, the code was implemented using Python programming language. To run GE algorithm, we used PonyGE2 framework [3], while for the development and execution of the CNN, the Keras open source library [2] was used.

Table 1: Experimental parameters

Parameter	Value
Generations	20
Population size	30
Initialization	PI_grow
Selection	Tournament (Tournament size: 2)
Crossover	Variable-onepoint (Crossover probability: 0.75)
Mutation	Int-flip-per-codon

The experiments are conducted with two steps.

In the first step, for each proposed grammar, 5 independent runs were performed using a population of 30 individuals and 20 generations. For the evolutionary parameters, the default settings from PonyGE2 with tournament selection, variable-one-point crossover and int-flip-per-codon mutation were used as in Table 1. During the evolution, the individuals were evaluated by their fitness, which is calculated according to the accuracy obtained from testing the CNN model (after training for 50 epochs) against CIFAR-10 dataset. In the fitness function, the maximalization was set True, i.e, higher accuracy leads to higher fitness.

In the second step, we choose to execute 20 times for training and testing the best model obtained from 5 independent runs for each grammar in the first step, with 500 epochs.

In addition, we also run the experiments for grammar from reference paper [6] (here named IEEE Control Grammar) with the same settings to obtain the results for comparison in next section.

All experiments are run on GPU nodes [1] to accelerate the jobs. These GPU nodes are equipped with Tesla V100 chip, powered by a massive number of parallel processing cores with 16GB memory and 32 GB memory.

5.3 Results

Table 2 shows the results for best fitness and average fitness obtained from 5 runs for each grammar in the first step. The best models which scored the best fitness in each row are then selected for the execution in the second step. Detail results of each model are shown in Table 3.

Table 2: Comparison of grammars in the first step

Grammar	Best fitness	Average fitness
IEEE Control Grammar	0.779	0.7705
Evol Grammar	0.851	0.8373
Control Grammar	0.75	0.73
Inception v1	0.65	0.63
Inception v2	0.64	0.61

From the results, it can be observed that the proposed Evol Grammar has better results when compared to IEEE Control Grammar. It means that this approach was able to

Table 3: Comparison of best models in the second step

Grammar	Best fitness	Average fitness	Number of parameters
IEEE Control Grammar	0.78	0.78	180,554
Evol Grammar	0.8628	0.8584	2,290,354
Control Grammar	0.78	0.76	300,381
Inception v1	0.74	0.72	2,325,731
Inception v2	0.76	0.71	1,821,878

produce architectures for CNNs which can achieve higher accuracy for the given dataset.

5.4 The best CNN model

The best model from Evol. Grammar achieved an accuracy of 86.28% on the best case and 85.84% on average among 20 runs. Figure 8 shows the graph of this model. It has several convolutional layers, followed by batch normalization. Before the Softmax, there is a global average pooling layer. The feature maps for 10 corresponding classes from the last convolutional layer will be the input of this layer, then resulting vector is directly fed into the Softmax layer.

As our observation, the global average pooling layer has a big contribution to the final result. In contrast to traditional fully connected layers, which are prone to over-fitting, the global pooling layer has no parameter to optimize, hence it helps to provide more control over the over-fitting. It is also noticed that the global average pooling layer is better than the global max pooling layer. Indeed, in all 5 models which have the best fitness generated from 5 runs for Evol Grammar in the first step, the global average pooling layer is selected by the evolutionary algorithm.

6. CONCLUSIONS

In this study, we proposed the grammars for GE algorithm to automatically generate and optimize CNN architectures. Every generated CNN model was trained and tested on an image classification task using CIFAR-10 dataset.

Results showed that our proposed grammar was able to generate CNN models that gradually evolved during execution of the genetic algorithm and achieved promising results.

For the future works:

- Bias-variance Tradeoff (train bigger model, train longer and change the hyper-parameters to avoid bias|More data to generalize and regularization for variance)
- The grammars can be enriched to implement more complex types of layer for the CNN models (ie. combining Inception with ResNet according to Dr. Andrew founder of deeplearning.ai)
- Opt for Bottleneck layers to reduce n of operations.
- Use Multiple cores that can help us experiment different evolutionary parameters and fully manage our work and reliable results. (ie. to have more the possibility to get more than 1000 runs)
- Fitness caching: To minimise the model load operations - which are expensive - we implement a caching mechanism for the models, such that to make ready available those models that are most likely to be used. In other words, we try to optimise the prediction speed using an efficient memory allocation policy.

7. REFERENCES

[1] `https://hpc.uni.lu/systems/iris/`.

[2] F. Chollet et al. Keras. `https://keras.io`, 2015.

[3] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, and M. O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, page 1194–1201, New York, NY, USA, 2017. Association for Computing Machinery.

[4] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[5] B. Lewin and B. Lewin. *genes VII*, volume 1. Oxford University Press New York, 2000.

[6] R. H. R. Lima and A. T. R. Pozo. Evolving convolutional neural networks through grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 179–180, New York, NY, USA, 2019. Association for Computing Machinery.

[7] N. Lourenço, F. Assunção, F. B. Pereira, E. Costa, and P. Machado. Structured grammatical evolution: a dynamic approach. In *Handbook of Grammatical Evolution*, pages 137–161. Springer, 2018.

[8] N. Lourenço, F. B. Pereira, and E. Costa. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, 17(3):251–289, 2016.

[9] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[10] F. Rothlauf and M. Oetzel. On the locality of grammatical evolution. In *European Conference on Genetic Programming*, pages 320–330. Springer, 2006.

[11] C. Ryan, J. Collins, and M. O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Genetic Programming*, pages 83–96, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[12] D. Schweim, A. Thorhauer, and F. Rothlauf. On the non-uniform redundancy of representations for grammatical evolution: The influence of grammars. In *Handbook of Grammatical Evolution*, pages 55–78. Springer, 2018.

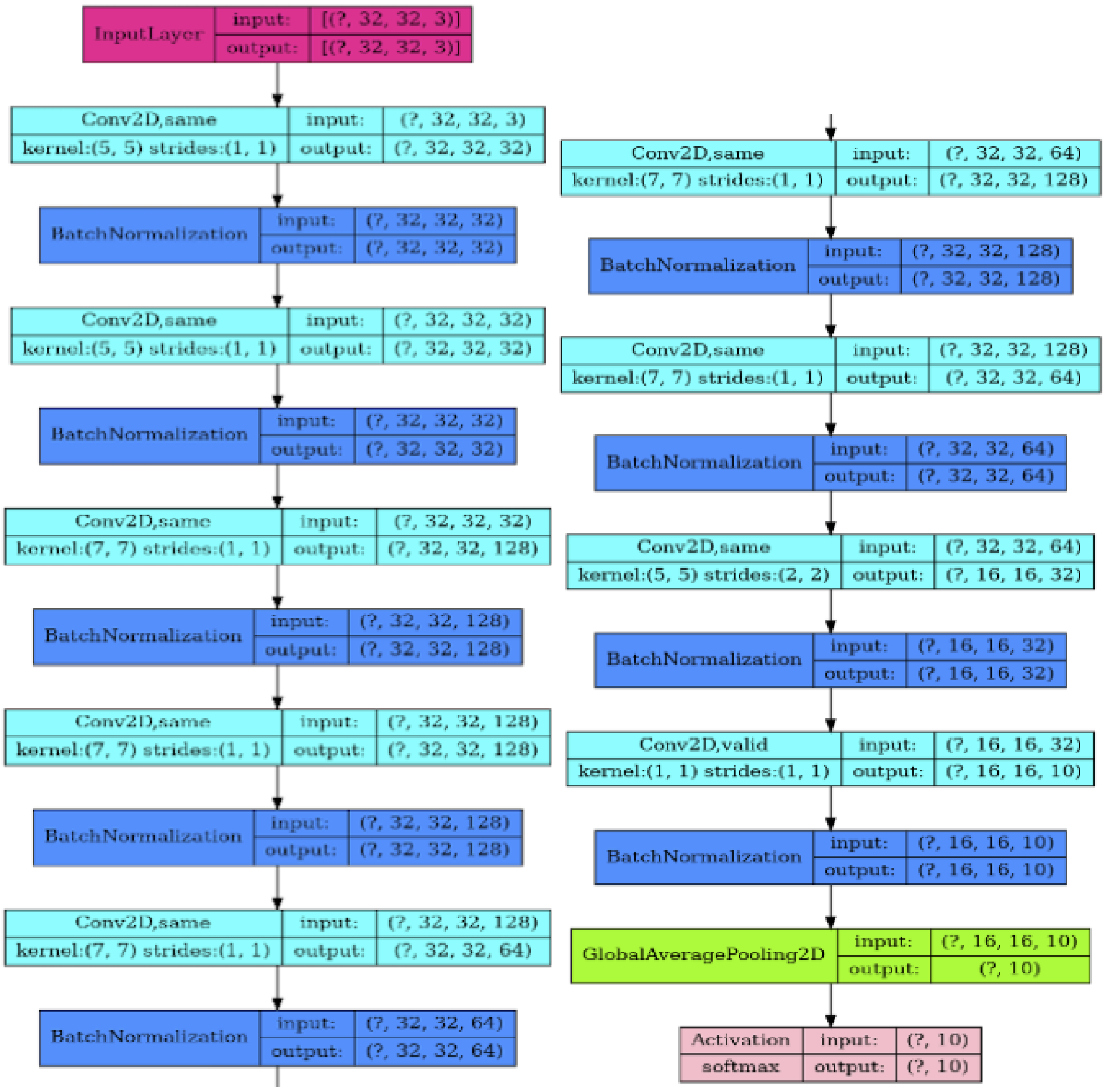


Figure 8: The best CNN model from Evol Grammar.

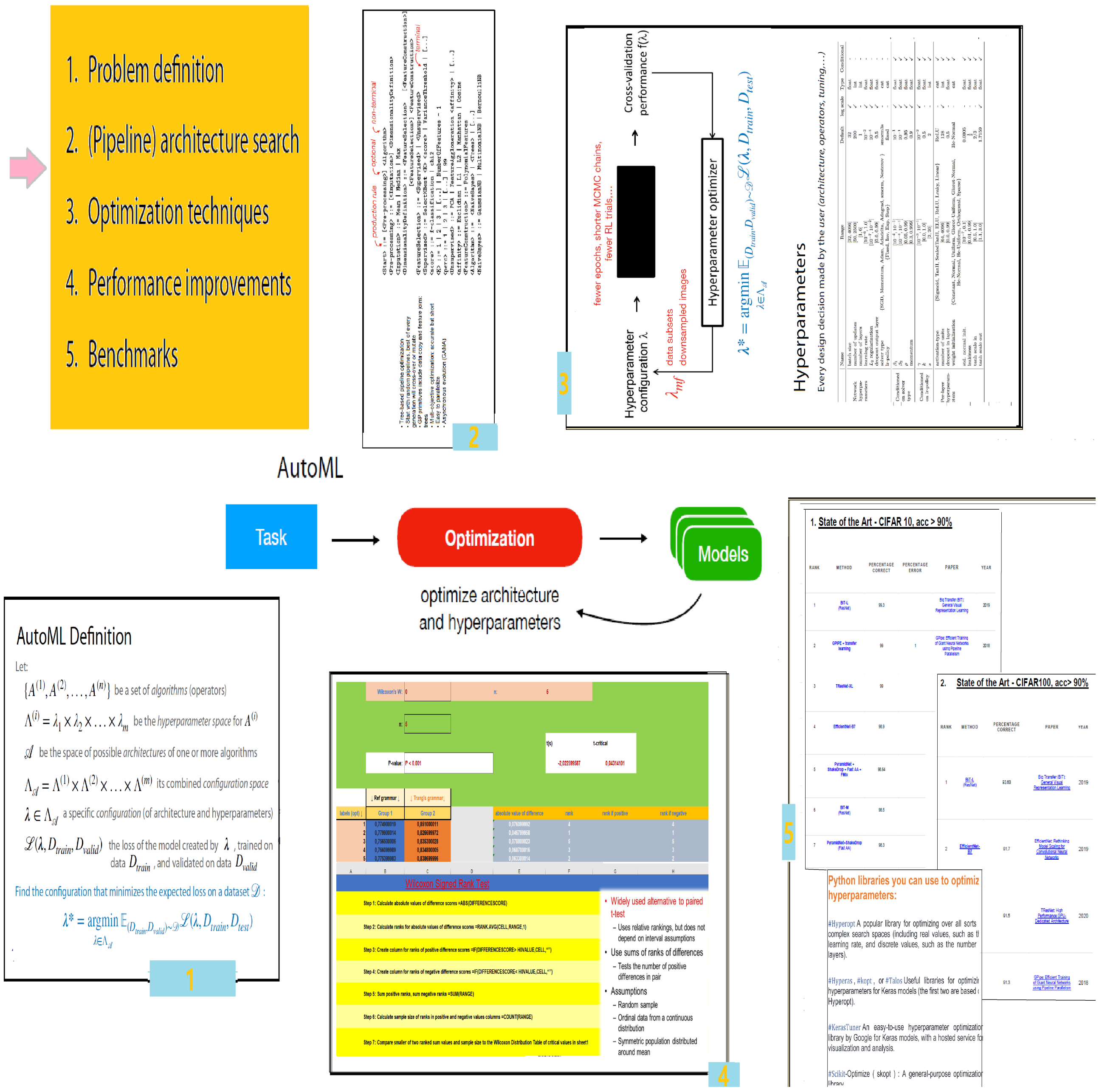

1. Problem definition
2. (Pipeline) architecture search
3. Optimization techniques
4. Performance improvements
5. Benchmarks
AutoML
Task
Optimization
Models
optimize architecture and hyperparameters
AutoML Definition
Let:
$\{A^{(1)}, A^{(2)}, \ldots, A^{(n)}\}$ be a set of algorithms (operators)
$\Lambda^{(i)} = \lambda_1 \times \lambda_2 \times \ldots \times \lambda_m$ be the hyperparameter space for $A^{(i)}$
$\mathcal{A}$ be the space of possible architectures of one or more algorithms
$\Lambda_{\mathcal{A}} = \Lambda^{(1)} \times \Lambda^{(2)} \times \ldots \times \Lambda^{(m)}$ its combined configuration space
$\lambda \in \Lambda_{\mathcal{A}}$ a specific configuration (of architecture and hyperparameters)
$\mathcal{L}(\lambda, D_{train}, D_{valid})$ the loss of the model created by λ, trained on data D_{train}, and validated on data D_{valid}
Find the configuration that minimizes the expected loss on a dataset $\mathcal{D}$:
$\lambda^* = \underset{\lambda \in \Lambda_{\mathcal{A}}}{\operatorname{argmin}} \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}} \mathcal{L}(\lambda, D_{train}, D_{test})$
Hyperparameter configuration λ
Cross-validation performance f(λ)
fewer epochs, shorter MCMC chains, fewer RL trials,...
data subsets downsampled images
Hyperparameter optimizer
Hyperparameters
Every design decision made by the user (architecture, operators, tuning,...)
production rule
optional
non-terminal
terminal
Wilcoxon Signed Rank Test
Step 1: Calculate absolute values of difference scores =ABS(DIFFERENCESCORE)
Step 2: Calculate ranks for absolute values of difference scores =RANK.AVG(CELL,RANGE,1)
Step 3: Create column for ranks of positive difference scores =IF(DIFFERENCESCORE>0,VALUE,CELL,"")
Step 4: Create column for ranks of negative difference scores =IF(DIFFERENCESCORE<0,VALUE,CELL,"")
Step 5: Sum positive ranks, sum negative ranks =SUM(RANGE)
Step 6: Calculate sample size of ranks in positive and negative values columns =COUNT(RANGE)
Step 7: Compare smaller of two ranked sum values and sample size to the Wilcoxon Distribution Table of critical values in sheet1
Widely used alternative to paired t-test
Uses relative rankings, but does not depend on interval assumptions
Use sums of ranks of differences
Tests the number of positive differences in pair
Assumptions
Random sample
Ordinal data from a continuous distribution
Symmetric population distributed around mean
1. State of the Art - CIFAR 10, acc > 90%
2. State of the Art - CIFAR100, acc> 90%
Python libraries you can use to optimize hyperparameters:
#Hyperopt A popular library for optimizing over all sorts complex search spaces (including real values, such as the learning rate, and discrete values, such as the number of layers).
#Hyperas, #kopt, or #Talos Useful libraries for optimizing hyperparameters for Keras models (the first two are based on Hyperopt).
#KerasTuner An easy-to-use hyperparameter optimization library by Google for Keras models, with a hosted service for visualization and analysis.
#Scikit-Optimize (skopt): A general-purpose optimization library
1
2
3
4
5

Figure 9: Contextualization and Workflow Chart