Ann Math Artif Intell manuscript No.
(will be inserted by the editor)

Multilayered neural architectures evolution for computing sequences of orthogonal polynomials

Dolores Barrios Rolanía · Guillermo Delgado Martínez · Daniel Manrique

Received: January 2018 / Accepted: date

Abstract This article presents an evolutionary algorithm to autonomously construct full-connected multilayered feedforward neural architectures. This algorithm employs grammar-guided genetic programming with a context-free grammar that has been specifically designed to satisfy three important restrictions. First, the sentences that belong to the language produced by the grammar only encode all valid neural architectures. Second, full-connected feedforward neural architectures of any size can be generated. Third, smaller-sized neural architectures are favored to avoid overfitting. The proposed evolutionary neural architectures construction system is applied to compute the terms of the two sequences that define the three-term recurrence relation associated with a sequence of orthogonal polynomials. This application imposes an important constraint: training datasets are always very small. Therefore, an adequate sized neural architecture has to be evolved to achieve satisfactory results, which are presented in terms of accuracy and size of the evolved neural architectures, and convergence speed of the evolutionary process.

Keywords Evolutionary computation · Grammar-guided genetic programming · Artificial neural networks · Orthogonal polynomials

Mathematics Subject Classification (2000) 68T05 · 68T20 · 68Q32 · 42C05 · 39A05

This work was partially supported by research grant MTM2014-54053-P of Ministerio de Economía y Competitividad, Spain.

D. Barrios Rolanía
ETS Ingeniería Civil,
Universidad Politécnica de Madrid, Spain
E-mail: dolores.barrios.rolania@upm.es

G. Delgado Martínez
Dept. de Inteligencia Artificial, ETSI Informáticos,
Universidad Politécnica de Madrid, Spain
E-mail: guillermo.delgado.martinez@alumnos.upm.es

D. Manrique
Dept. de Inteligencia Artificial, ETSI Informáticos,
Universidad Politécnica de Madrid, Spain
E-mail: daniel.manrique@upm.es

1 Introduction

Multilayer perceptrons are full-connected feedforward neural networks with one input layer, one output layer and several (at least one) hidden layers. This model is applicable within natural computing [22] and machine learning areas to solve a wide variety of real-world problems [36], thanks to its properties of learning by examples, function approximation and generalizing to unseen data [33]. Feedforward neural networks with a single hidden layer are universal approximators under some smooth conditions [20], but there are empirical evidence suggesting that neural networks with several hidden layers are better adapted to learn functions [31]. Nowadays, deep learning in neural networks is attracting widespread attention, mainly by outperforming alternative machine learning methods [37]. Despite the advantages and good results achieved by these biology-inspired intelligent systems, their success depend on finding an architecture to fit the task [34]. Therefore, a neural topology has to be specifically designed to solve a problem [27,15]. Designing a full-connected feedforward neural architecture involves adjusting the number of hidden layers and how many neurons are placed in each of them. This is not an easy task, which is typically governed by heuristics, previously gathered experience, trial and error methods or random search [5]. The optimal architecture is a network that is large enough to learn the problem, and is small enough to generalize well.

Several research works have focused on designing different search algorithms to find out adequate neural network architectures to solve a given problem, which is defined by a dataset of input-output vectors. Some of these approaches are called incremental, constructive or growing algorithms [45,21]. Complementary approaches are pruning or destructive methods [3], and it is also very common to combine these two strategies; namely, growing and pruning [17]. The snags with all these approaches are that obtaining a good solution usually depends on the initially chosen architecture, and may present premature convergence. Other solutions prefer to generate too large neural architectures and employ regularization methods (e.g. via more data) to avoid overfitting [19,40,44].

Promising studies originate from the identification of synergies between evolutionary algorithms and artificial neural networks throughout neuroevolution [29,41]. Evolutionary computation is used to search for neural network hyper-parameters [25], topologies [30,48], or as replacement or hybridization of the neural learning algorithm [12,43,42] to achieve a good enough network performance in a given task. Evolutionary algorithms are inspired by natural evolution and adaptation of species to achieve the prevalence or subsistence of a population of individuals. These algorithms are employed to solve search and optimization problems, where each (valid) individual represents a candidate solution. New generations of individuals are achieved throughout an evolutionary process based on genetic operators such as selection, reproduction, crossover, mutation and replacement [10,47,39]. This evolutionary process is expected to provide improved individuals; in such a way they represent better candidate solutions to solve the problem at hand. Evolutionary algorithms work in two different spaces at the same time: the search space, composed by all possible individuals that can be generated; and the solutions space, composed by candidate solutions to the problem at hand. Search space is mapped to solutions space by the so-called encoding scheme [13].

For any evolutionary algorithm chosen, the way in which solutions are encoded in the search space (encoding scheme) is the crucial step to successfully solve search and optimization hard real-world problems [13]. This is also an important issue in the case of evolutionary searching for neural architectures because of the size and complexity of the solutions space. Genetic algorithms are an evolutionary approach where solutions are encoded by vectors or strings of symbols (individuals) [1,24]. Several binary encoding schemes have been devel-

oped to codify neural architectures for such evolutionary algorithm. They can be divided into direct and indirect encoding methods. Direct encoding methods would be one of the first, where each bit in the binary string represents the presence or absence of a single connection [38,11]. Despite their simplicity, one of the major disadvantages of direct encoding methods is that they are unable to prevent infeasible individuals throughout the evolution process. On the other hand, indirect encoding methods [6], such as the basic architectures encoding method [27], overcome some of the disadvantages encountered in direct approaches. In any case, the lack of this last approach is that the size of the solutions space is delimited by the length of the strings that compose the individuals of the population.

Genetic programming evolves computer programs encoded by a tree structure [32]. Therefore, there is no size limit for the solutions that can be encoded, but infeasible individuals are likely generated. Grammar-guided genetic programming (GGGP) [28] tackles this so-called closure problem [23,46] by employing a context-free grammar (CFG), which establishes a formal definition of the syntactical restrictions of the individuals that encode candidate solutions [35]. GGGP features a population of parse or derivation trees that follow the production rules of the grammar. Each derivation tree represents a sentence or word that belongs to the language defined by the CFG. Either this language corresponds itself to the solutions space (direct encoding methods) or each word needs to be decoded to obtain the candidate solution (indirect methods) [14]. Indirect GGGP-based approaches have been studied to evolve neural architectures; namely, grammar-guided neural architecture evolution [9]. This evolutionary system proposes a CFG that generates a language of binary strings of variable length. A binary string encodes a one-hidden layer feedforward neural architecture with any number of neurons.

This paper presents a GGGP-based evolutionary algorithm to automatically construct full-connected feedforward multilayered neural architectures of any size in terms of number of hidden layers and number of neurons each. Activation functions for neurons in each layer may be optionally included in the evolution process. A CFG has been designed to generate a formal language that encode all valid, but only valid, neural architectures. A modified version of this CFG to encode the activation functions of a neural architecture is also presented. Both encoding schemes are indirect approaches, and there exists a one-to-one correspondence between the search space (derivation trees) and the solution space (neural architectures, including or leaving out the activation functions).

The proposed evolutionary system has been applied to the evolution of neural architectures to approximate the general term of some real sequences $\{a_n\}$, $n \in \mathbb{N}$. These sequences are the coefficients that define the three-term recurrence relation associated to an orthogonal polynomial sequence (OPS) in $\mathbb{R}$, e.g. Hermite or Lagerre [8]. The three-term recurrence relation is defined as:

$$P_n(x) = (x - b_{n-1})P_{n-1}(x) - a_{n-1}^2 P_{n-2}(x), \tag{1}$$

which can be represented using the Jacobi matrix

$$A = \begin{pmatrix} b_0 & a_1 & 0 & 0 & 0 & \dots & \dots \\ a_1 & b_1 & a_2 & 0 & 0 & \dots & \dots \\ 0 & a_2 & b_2 & a_3 & 0 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{n-2} & b_{n-2} & a_{n-1} & \dots \\ 0 & 0 & \dots & 0 & a_{n-1} & b_{n-1} & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

The polynomial $P_n(x)$ can be calculated from the principal minor A_n of A, the identity order n matrix I_n, and the relation:

$$P_n(x) = \det(xI_n - A_n). \tag{2}$$

The sequences of orthogonal polynomials play an important role in many areas of scientific computation such as least squares approximation, numerical integration [7] and applications in mathematics, physics and engineering. Despite the theory of orthogonal polynomials has their roots in the beginning of 20th century, the use of modern computers requires the continuous development of new and accurate algorithms, alternative to classical approaches [16], and methods in theirs applications. In particular, the study of approximation of the coefficients of the recurrence relations permits to obtain some conclusions about the location of zeros and other properties of this polynomials [4].

We begin the paper with a description of the main components involved in the proposed evolutionary neural architectures construction system, together with their inputs, intermediate and final results. Following this, we describe the CFG specifically designed to generate a formal language that encodes valid full-connected multilayered feedforward neural architectures. A variation of the proposed CFG is also presented to give the choice of including the activation functions for hidden and output neurons. Then, empirical results achieved by the proposed system in the evolution neural architectures, with the option of including the activation functions, to approximate general terms associated to OPS in $\mathbb{R}$ are shown. Finally, some concluding remarks, contributions and future lines of research are provided.

2 The evolutionary neural architectures construction system

Fig. 1 depicts the proposed evolutionary system. The dataset, composed by $k+m$ input and desired output (target) vectors, defines the problem to be solved by an artificial neural network with I inputs and O output neurons. This dataset is partitioned into k training and m testing patterns. A parameterized context-free grammar G_{IO} is designed for specific I and O values.

The GGGP-based evolutionary algorithm starts by the initialization of a population of derivation trees (individuals) randomly generated. Each individual represents a set of production rules of the grammar, starting from the axiom, that yields a sentence belonging to the language defined by G_{IO}. Each sentence is composed by substrings formed by the symbol n, separated by the symbol $/$. An important feature of this encoding scheme is that a sentence always encodes a valid (feasible) multilayered feed-forward neural architecture.

The individuals of the population are then evaluated by a fitness function and the evolutionary loop starts, by applying selection, crossover, mutation and replacement genetic operators until a stop condition is satisfied. The fitness evaluation of any new individual generated throughout the evolution process, either by the initialization process, crossover or mutation operators, involves reading the leaves of the derivation tree, from left to right, to extract the sentence. This is then decoded into its corresponding (valid) neural architecture to feed the training engine, employing the k training patterns from the dataset. The smaller the size of a network is, the better generalization capabilities are expected. However, if the neural architecture is too small, then its performance will decrease. Therefore, the size of the neural architecture along with the error achieved by the training process are evaluated to calculate the fitness of each new candidate solution. The goal is to obtain an adequate balance between size and performance of the final neural network solution that not only has

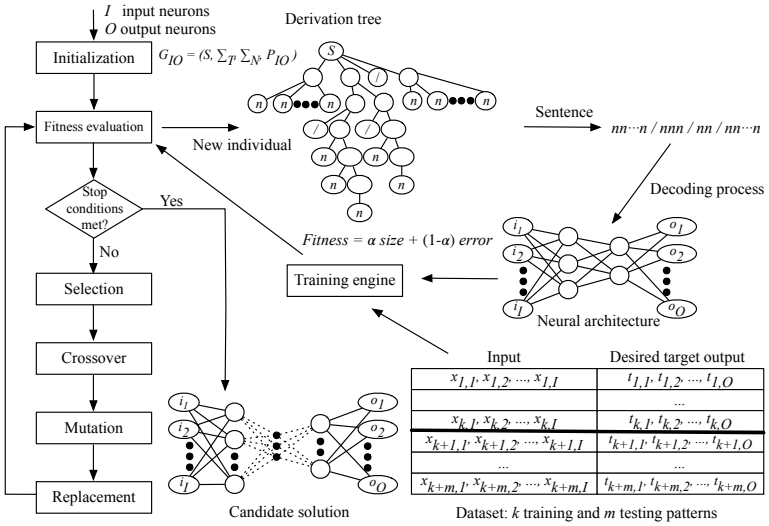


Fig. 1 Evolutionary neural architectures construction system. General scheme

learned the first k training patterns, but also achieves accurate results for the next unseen m patterns.

Obtaining an adequate size, neither too large or too small, of the final neural network structure is an important issue during the evolution process. This can not be accomplished by using a larger than necessary network and regularize via more data. This constrain comes from the application of the proposed evolutionary algorithm: for an unknown general term of a real sequence $\{a_n\}$ from which only the first k terms are available, the goal is to calculate the following m terms, with $m >> k$. Therefore, an arbitrary amount of data can not be considered.

3 The neural architectures encoding scheme

A context-free grammar has been designed to generate a formal language that only encodes all valid full-connected multilayered feedforward neural architectures of any size. The grammar defines the syntactical restrictions that must be satisfied by such neural architectures. This CFG-based indirect encoding scheme is shown to be effective and simple, without requiring any modification to standard genetic operators.

A context free grammar G is defined as a 4-tuple $G = (S, \Sigma_N, \Sigma_T, P)$, where $S \in \Sigma_N$ is the axiom or start symbol of the grammar. Σ_N is a finite set of variables or nonterminal symbols, from which new strings of symbols are produced. Σ_T is a finite set, with $\Sigma_N \cap \Sigma_T = \emptyset$, of terminal symbols or terminals from which sentences are composed. Finally, P is the finite set of production rules, rewriting rules or productions. If $A \in \Sigma_N$ and $s \in (\Sigma_N \cup \Sigma_T)^*$ is a string of terminal and nonterminal symbols, production rules are in the form $A \to s$, said as A yields, derives or produces s. If there is a series of production rules (derivation), starting

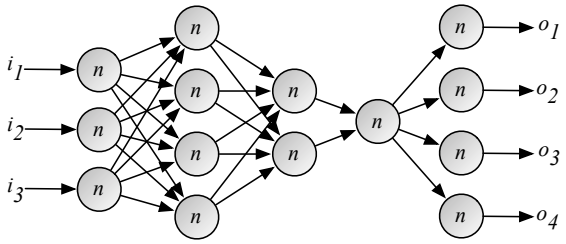


Fig. 2 Neural architecture 3-4-2-1-4, corresponding to the sentence $s = nnn/nnnn/nn/n/nnnn$, with $s \in L(G_{34})$

from A, that finally yields s, then this noted as $A \rightarrow^* s$. The language generated by a grammar G is $L(G) = \{s \in \Sigma_T^* : S \rightarrow^* s\}$.

A parameterized context-free grammar, G_{IO}, has been designed for the proposed encoding scheme. The parameters I and O represent the number of input and output neurons, respectively. They are obtained from the dataset, which defines the problem to be solved. G_{IO} is embedded into the GGGP-based evolutionary algorithm to search for neural architectures that solve the problem at hand. The language defined by the grammar $G_{IO} = (S, \Sigma_N, \Sigma_T, P_{IO})$ can be represented as the regular expression $L(G_{IO}) = \{n^I(/n^+)^+/n^O : n, / \in \Sigma_T\}$. Therefore, sentences $n^I(/n^+)^+/n^O$ start with a string of length I with the terminal n, followed by at least one substring beginning with the terminal $/$, and ending with at least one n. Finally, sentences end with the symbol $/$ followed by a string of length O with the terminal n. Symbol n is the terminal that represents either an input, output or hidden neuron. The terminal symbol $/$ represents a separator between layers for a better reading of sentences. As an example, the sentence $nnn/nnnn/nn/n/nnnn$ encodes the full-connected neural architecture of Fig. 2, with three input neurons; four, two and one hidden neurons subsequently distributed in three hidden layers, respectively; and four neurons in the output layer. This neural architecture is noted as 3-4-2-1-4.

The following considerations have been allowed for to design the context-free grammar $G_{I,O}$ to encode full-connected feedforward neural architectures with I inputs, O output neurons, and any number of hidden layers with any number of neurons each:

1. The set of terminal symbols of the grammar G_{IO} is $\Sigma_T = \{n, /\}$.
2. The input and output layers are defined with as many neurons as the number of input and output variables, respectively, within the dataset. The nonterminal symbol A has been chosen to derive a string of length I with the terminal n, representing I input neurons. The nonterminal symbol Z derives a string of length O with the symbol n, representing O output neurons. Consequently, $A \rightarrow n^I, Z \rightarrow n^O \in P_{IO}$.

   In the case of the example given in Fig. 2, with three input neurons ($I = 3$) and four outputs ($O = 4$), production rules $A \rightarrow nnn$ and $Z \rightarrow nnnn$ belong to P_{34} of the grammar G_{34}.
3. Being S the axiom of the grammar, the production rule $S \rightarrow A/Z$ encodes a neural architecture without hidden layers. For the example of $I = 3$ and $O = 4$, the production rule $S \rightarrow A/Z$, with $A \rightarrow nnn$, and $Z \rightarrow nnnn$, can generate only one derivation tree that represents the sentence $nnn/nnnn$. Fig. 3 shows this derivation tree, the corresponding sentence, and the encoded 3-4 neural architecture.
4. As the goal of this research is to encode neural architectures of at least one hidden layer, the nonterminal symbol H is defined for this purpose. If there would be only one hidden layer with one neuron, then the following production rules would be enough:

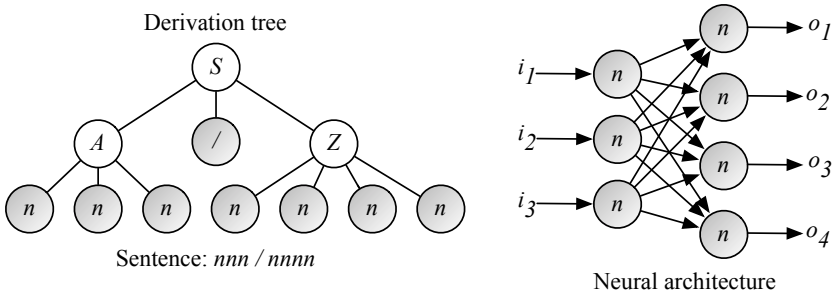


Fig. 3 Derivation tree that represents the sentence $s = nnn/nnnn \in L(G_{34})$, which encodes a 3-4 neural architecture

$S \to AH/Z; H \to /n$. In case of the example given, with $A \to nnn$, and $Z \to nnnn$, these production rules yield the sentence $nnn/n/nnnn$, which encodes a 3-1-4 neural architecture.

5. However, the interest lies on encoding any number of hidden layers with any number of neurons each. Therefore, a recursive production rule is needed. Starting from the nonterminal symbol H, it should be possible to add a new hidden layer or an undefined number of neurons to the current layer. The recursive production $H \to HH$ accomplishes the former, while $H \to /N$ the later, being N a nonterminal symbol to add any number of neurons to a hidden layer. Therefore, the productions of H are $H \to HH$ and $H \to /N$. Consequently, $S \to AH/Z; A \to n^I; Z \to n^O; H \to HH; H \to /N \in P_{IO}$, being S,A,H,Z and N nonterminal symbols.

   P_{IO} permits to configure neural architectures con I inputs, O output neurons and any number of hidden layers. Defining the production rules of the nonterminal N to set up how many neurons are in each hidden layer is left.

6. A string of any length with the symbol n describes a hidden layer, so the following recursive rule is needed for the nonterminal symbol $N : N \to nN$. A recursion stop rule is also included in the form $N \to n$, because, at least, one neuron must be placed in each hidden layer. Then, the production rules for N are $N \to nN|n$.

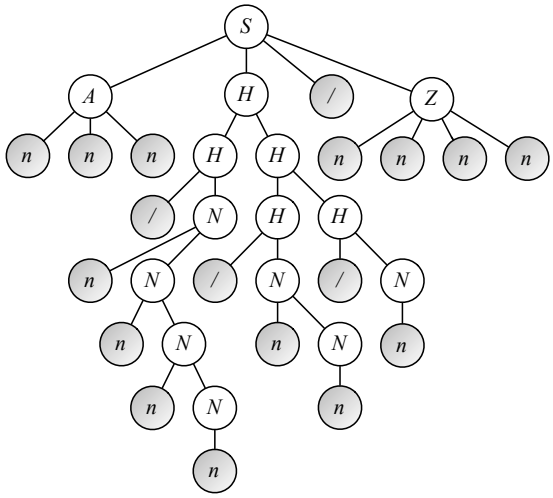
From the points above, the right recursive context-free grammar $G_{IO} = (S, \Sigma_N, \Sigma_T, P_{IO})$ is designed to indirectly encode feedforward full-connected multilayered artificial neural architectures, being:

- S, the start terminal symbol or axiom.
- $\Sigma_N = \{S,A,H,Z,N\}$, the set of nonterminal symbols.
- $\Sigma_T = \{n,/\}$, the set of terminal symbols.
- $P_{IO} = \{S \to AH/Z; A \to n^I; Z \to n^O; H \to HH|/N; N \to nN|n\}$, the set of production rules or productions.

Fig. 4 shows a derivation tree that encodes the neural architecture 3-4-2-1-4 of Fig. 2. On the right, the applied production rules in the derivation are also shown.

3.1 Encoding activation functions

A variation of the proposed CFG G_{IO}, noted as $G_{IO\mathbb{F}}$, is also presented to encode the activation functions of the evolved neural architectures, e.g. sigmoidal, hyperbolic tangent, pure



1: $S \rightarrow A\ H\ /\ Z$
2: $A \rightarrow nnn$
3: $H \rightarrow HH$
4: $H \rightarrow /N$
5: $N \rightarrow nN$
7: $N \rightarrow nN$
8: $N \rightarrow nN$
9: $N \rightarrow n$
10: $H \rightarrow HH$
11: $H \rightarrow /N$
12: $N \rightarrow nN$
13: $N \rightarrow n$
14: $H \rightarrow /N$
15: $N \rightarrow n$
16: $Z \rightarrow nnnn$

Sentence: nnn / nnnn / nn / n / nnnn

Production rules

Fig. 4 A derivation tree that encodes a 3-4-2-1-4 neural architecture and its productions

linear, softmax, (leaky) rectifier linear or any others [26]. While I and O are still the number of input and output neurons, respectively, of the neural architectures to evolve, $\mathbb{F}$ is the set of activations functions to be considered. $G_{IO\mathbb{F}}$ replaces the separator terminal symbol / included in G_{IO} by any terminal symbol representing an activation function. This way, the most often scenario in neural networks design is covered: neurons within a layer employ the same activation function, which may be different in each layer.

Let be $\mathbb{F} = \{f_1, f_2, ..., f_\phi\}$ a set of activation functions. The language generated by $G_{IO\mathbb{F}}$ encodes, similarly to $L(G_{IO})$, the set of all valid full-connected multilayered feedforward neural architectures of any size with I input neurons, O outputs and any $f_i \in \mathbb{F}$ in each hidden and output layers. $G_{IO\mathbb{F}} = (S, \Sigma_N, \Sigma_T, P_{IO\mathbb{F}})$, being:

- S the axiom.
- $\Sigma_N = \{S, A, H, Z, N, F\}$, the set of nonterminal symbols. The new nonterminal F represents a activation function.
- $\Sigma_T = \{n, f_1, f_2, ..., f_\phi\}$, the set of terminal symbols. The separator / in G_{IO} is replaced by the activation functions to consider in the neural architectures.
- $P_{IO\mathbb{F}} = \{S \rightarrow AHFZ; A \rightarrow n^I; Z \rightarrow n^O; H \rightarrow HH|FN; N \rightarrow nN|n; F \rightarrow f_1|f_2|...|f_\phi\}$, the set of productions. The separator / in the right hand side of productions has been replaced by the nonterminal F, which produces any of the ϕ considered activation functions.

Both CFG G_{IO} and $G_{IO\mathbb{F}}$ have been specifically designed to be ambiguous, since this feature may be beneficial for the searching process [9]. After applying the production $H \rightarrow HH$, the first H can be rewritten as either $/N$ or HH. In the case of $G_{IO\mathbb{F}}$, FN or HH. Therefore, as long as there are more than one hidden layer, a sentence may be produced by more than one derivation.

Table 1 General terms of sequences $\{a_n\}$ and $\{b_n\}$, $n \in \mathbb{N}$, that define some classical OPS

OPS	a_n	b_n
Tchebichef	$\frac{1}{4}$	0
Hermite	$\frac{n}{2}$	0
Charlier	an	$n+a$
Laguerre	$n(n+\alpha)$	$2n+\alpha+1$
Jacobi elliptic	$4n^2(2n-1)^2r^2$	$(2n+1)^2+(2n)^2r^2$
Modified Lommel	$\frac{1}{4(n+v)(n+v-1)}$	0
Legendre	$\frac{n^2}{(2n-1)(2n+1)}$	0

4 Results

The proposed evolutionary neural architectures construction system has been applied to evolve feedforward multilayered artificial neural networks. Tests have been accomplished to show the results achieved by the evolved neural networks approximating the known general terms related to the classical OPS shown in Table 1. These results are presented in terms of accuracy and size of the neural networks solution.

4.1 Experimental setup

For each general term, a_n or b_n from those included in Table 1, a dataset of $k+m$ input, output patterns $\{(i,a_i)\}_{i=1}^{k+m}$, has been built to train and test the artificial neural networks evolved by the proposed GGGP-based system (general scheme of Fig. 1). Therefore, neural architectures with one input, corresponding to the index $i \in \mathbb{N}$ (denoted as x in the general scheme of Fig. 1), and one desired target output to predict the i^{th} term, $a_i \in \mathbb{R}$ (denoted as t in the same figure), of the sequence need to be encoded. A pure linear and non-linear, such as hyperbolic tangent, activation functions are, at least, necessary for the neural networks to successfully accomplish the proposed experiments. Instead of manually assigning the activation functions to the neurons of hidden and output layers, and for the sake of completeness, $G_{IO\mathbb{F}}$ has been employed. This scenario is covered assigning $I=1$, $O=1$, and $\mathbb{F}=\{f_{\text{purelin}}, f_{\tanh}\}$; with $\phi=2$, $f_{\text{purelin}}(x)=x$ the pure linear activation function, and $f_{\tanh}(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$ the hyperbolic tangent.

Each dataset $\{(i,a_i)\}_{i=1}^{k+m}$ is partitioned into two subsets: the first k elements as the training patterns, $\{(i,a_i)\}_{i=1}^{k}$; and the subsequent m terms, $\{(i,a_i)\}_{i=k+1}^{k+m}$, for exclusively testing purposes. This way, it is possible to evaluate the accuracy for unseen data of the evolved and trained neural architectures.

The proposed GGGP-based evolutionary algorithm employs a standard initialization method, which randomly generates a population of derivation trees. This way, results achieved are not biased by the initial population. Whigham's crossover operator [46] has been chosen since it is widely tested, assures a valid offspring and has a balanced exploration and exploitation (local search) capabilities, what usually achieves satisfactory results [2]. No mutation has been employed in the experiments to avoid, again, biasing.

Experimental pre-runs were conducted with datasets of the form $\{(i,a_i)\}_{i=1}^{k+m}$, taking $k=20$ and $m=60$, to set up other configuration parameters of the GGGP-based evolutionary neural architectures construction system. The first experimental runs reported a clear superiority of tournament selection method. It achieved higher convergence speed and better

average accuracy in approximating the general terms included in Table 1. Additional experimental runs were accomplished to set up the number of generations to stop. This was set to 500 after observing an adequate trade-off between computational cost and accurate results. A small-sized population of 30 individuals was chosen to show the exploration capabilities of the proposed evolutionary algorithm.

Fitness evaluation of any individual (derivation tree) involves two steps: training and size calculation of the encoded neural architecture. The training step is accomplished by the back-propagation learning algorithm with momentum factor [18] on the set of training patterns: the first k terms of the dataset $\{(i,a_i)\}_{i=1}^{k+m}$, with $k = 20$. This learning process stops when any of the two following stop conditions are met: the mean square error for the set of training patterns is less than a threshold, or the maximum number of back-propagation epochs has been reached. Then, the training error is calculated as:

$$E = 1 - \frac{1}{k}\sum_{i=1}^{k}\left|\frac{a_i'}{a_i}\right|, \tag{3}$$

being a_i' the output computed by the neural network taking index i as input, and a_i the desired target output taken from the training patterns.

Again, experimental pre-runs were carried out with several datasets to estimate the best parameters to achieve an adequate trade-off between computational cost and accuracy. In the case of our study, we adopted a mean square error threshold of 0.01 and a maximum number of 500 epochs.

The second phase of fitness evaluation calculates an estimation of the size of the neural architecture as a weighted sum between the total amount of neurons N, and the number of layers C:

$$T = \beta C + (1-\beta)N \tag{4}$$

Finally, individual's fitness is a convex combination of the training error and the size of the encoded neural architecture as a regularizer:

$$F = \alpha T + (1-\alpha)E \tag{5}$$

Fitness calculation involves two parameters that were set, by random search [5], to $\alpha = 0.3$ and $\beta = 0.5$ to carry out the tests. Average results are presented after 70 executions were run.

4.2 Accuracy and size of the neural networks

Table 2 shows descriptive statistics from 70 different runs for each experiment of the proposed evolutionary neural architectures construction system. Results gathered are concerned to the error achieved by the evolved artificial neural networks during the testing phase. Error values are calculated according to (3), involving the unseen m terms $\{(i,a_i)\}_{i=k+1}^{k+m}$, with $k = 20$ and $m = 60$, of a general term a_n dataset related to an OPS. For each orthogonal polynomial sequence included in Table 1, results are shown for both a_n and b_n general terms. Average, standard deviation, minimum and maximum error values, and lower and upper bounds of the 95% confidence interval are presented for the best evaluated neural network (5) and the average error of the neural architectures within the final population in each execution. The results shown in the *Best* rows of the table, two rows per OPS, one per general term, correspond to the artificial neural networks provided as solution by the proposed evolutionary algorithm in each of the executions run.

Table 2 Descriptive statistics, obtained after 70 runs, of the error achieved in the testing phase by the best evolved artificial neural networks, and the average errors of the neural architectures within the final populations, while approximating general terms related to OPS

OPS	Gen. term	ANN	Avg.	SD	Min.	Max.	Confidence interval 95%	
							Inf.	Sup.
Tchebichef	a_n	Best	0	0	0	0	0	0
		Average	0	0	0	0	0	0
	b_n	Best	0	0	0	0	0	0
		Average	0	0	0	0	0	0
Hermite	a_n	Best	$8.6112e^{-15}$	$1.3232e^{-14}$	0	$4.6851e^{-14}$	$2.1275e^{-15}$	$1.5095e^{-14}$
		Average	0.0017	0.0052	$5.9639e^{-4}$	0.0033	$-8.0908e^{-4}$	0.0043
	b_n	Best	0	0	0	0	0	0
		Average	0	0	0	0	0	0
Charlier	a_n	Best	$4.0046e^{-4}$	$1.0416e^{-4}$	$3.2981e^{-4}$	$5.5310e^{-4}$	$2.9839e^{-4}$	$5.0254e^{-4}$
		Average	0.0011	$8.7076e^{-4}$	0.0010	0.0012	$2.4102e^{-4}$	0.0019
	b_n	Best	$2.7212e^{-5}$	$1.0031e^{-4}$	0	$4.9816e^{-4}$	$3.7138e^{-6}$	$5.0711e^{-5}$
		Average	$9.2969e^{-5}$	$6.4658e^{-4}$	0	0.0019	$5.8502e^{-5}$	$2.4444e^{-4}$
Laguerre	a_n	Best	0.0069	0.0051	$1.1138e^{-4}$	0.0295	0.0056	0.0082
		Average	0.0322	0.0726	0.0160	0.0567	0.0140	0.0504
	b_n	Best	$4.8533e^{-15}$	$1.2645e^{-14}$	0	$3.3529e^{-14}$	$4.5144e^{-15}$	$1.4221e^{-14}$
		Average	0.0020	$0.0075e^{-05}$	0.0010	0.0039	0.0035	0.0076
Jacobi elliptic	a_n	Best	0.0508	0.0418	$4.5069e^{-4}$	0.1479	0.0363	0.0653
		Average	0.2851	0.1156	0.2246	0.3266	0.2450	0.3251
	b_n	Best	0.0073	0.0056	$1.6252e^{-4}$	0.0220	0.0060	0.0087
		Average	0.0341	0.0775	0.0179	0.0556	0.0159	0.0522
M. Lommel	a_n	Best	0.0015	0.0011	$1.0823e^{-5}$	0.0059	0.0013	0.0018
		Average	0.0040	0.0065	0.0023	0.0061	0.0025	0.0055
	b_n	Best	0	0	0	0	0	0
		Average	0	0	0	0	0	0
Lengendre	a_n	Best	$2.3411e^{-4}$	$5.7965e^{-8}$	$9.1624e^{-6}$	$9.2808e^{-4}$	$2.7290e^{-4}$	$3.8259e^{-4}$
		Average	$4.7117e^{-4}$	$4.4986e^{-4}$	$3.4181e^{-4}$	$5.8311e^{-4}$	$3.6601e^{-4}$	$5.7678e^{-4}$
	b_n	Best	0	0	0	0	0	0
		Average	0	0	0	0	0	0

From these results, it is clear that the artificial neural networks evolved by the proposed GGGP system show high accuracy for the testing datasets. Given a set of $k = 20$ training patterns $\{(i, a_i)\}_{i=1}^{20}$, corresponding to the first 20 terms of a sequence related to an OPS included in Table 2, according to the *Best* rows of this table, a typical execution of the evolutionary algorithm provides an artificial neural network that is able to accurately approximate, at least, the next $m = 60$ unseen terms. Moreover, according to the lower and upper bounds of the 95% confidence interval in *Average* rows of Table 2, any artificial neural network, not only that one with the best fitness, have accuracy enough to calculate the next terms with an error close to zero.

Results in terms of accuracy are presented more in detail for the general terms a_n related to Lagerre, Modified Lommel and Legendre OPS, and b_n in the case of Jacobi elliptic. They have been selected because their complexity: second degree polynomial functions that do not converge in the cases of Laguerre a_n and Jacobi elliptic b_n; and rational functions in the cases of modified Lommel and Legendre a_n general terms (Table 1). In addition, it can be also observed from Table 2 that the best evolved artificial neural networks achieve the highest testing errors in these cases, excepting Legendre. Jacobi elliptic a_n is also an interesting case of study, since it is a fourth degree polynomial that does not converge. However, its detailed results are not presented since they are similar to those for the general term Jacobi elliptic b_n.

Fig. 5, 6, 7 and 8 show the errors in the testing phase achieved by the evolved artificial neural networks within the final population for each of the 70 executions run with the proposed evolutionary algorithm. Each of these figures shows results related to the general terms under more in detail study; namely, Lagerre, modified Lommel and Legendre a_n gen-

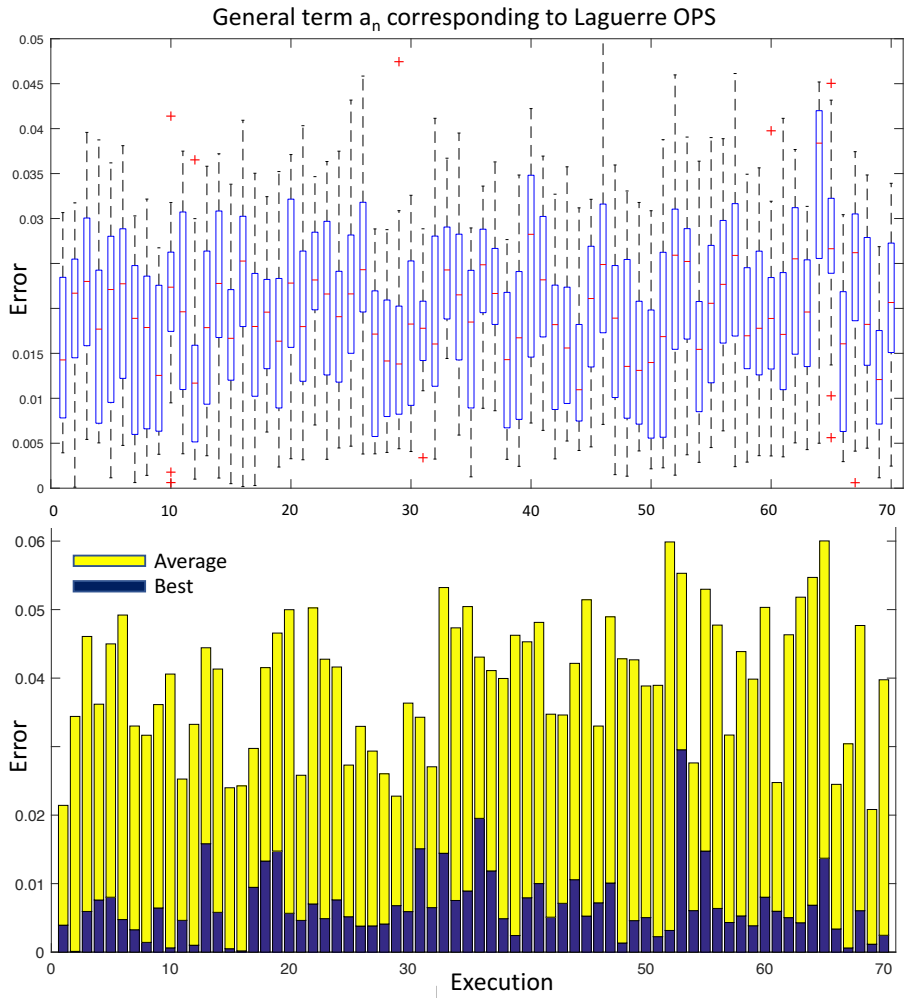


Fig. 5 Testing errors achieved by ANNs in the final population for each execution. Above, error distributions. Below: Best and average errors

eral terms, and b_n in the case of Jacobi elliptic OPS. The graph above, in each figure, shows box-and-whisker plots representing the 70 error distributions of each execution, while a bar chart below reports the errors achieved by the best evolved neural architectures, in dark color, and the average error of each final evolved population in light. Horizontal axis represents each of the 70 executions run, with a box-and-whisker plot each in the upper side, and stacked bar charts below (best and average). Error values appear in the vertical axis. Note, graphically, that error distributions are closer to zero in the cases of modified Lommel and Legendre a_n, both rational general terms that converge, with error averages of 0.0015 and $2.3411e^{-4}$, respectively, for the best evolved neural architectures.

Fig. 9 compares the convergence speed of the proposed evolutionary system searching for the neural architectures that better calculate the 60 following unseen terms (testing phase) for the four general terms under deeper study. The average error achieved by the

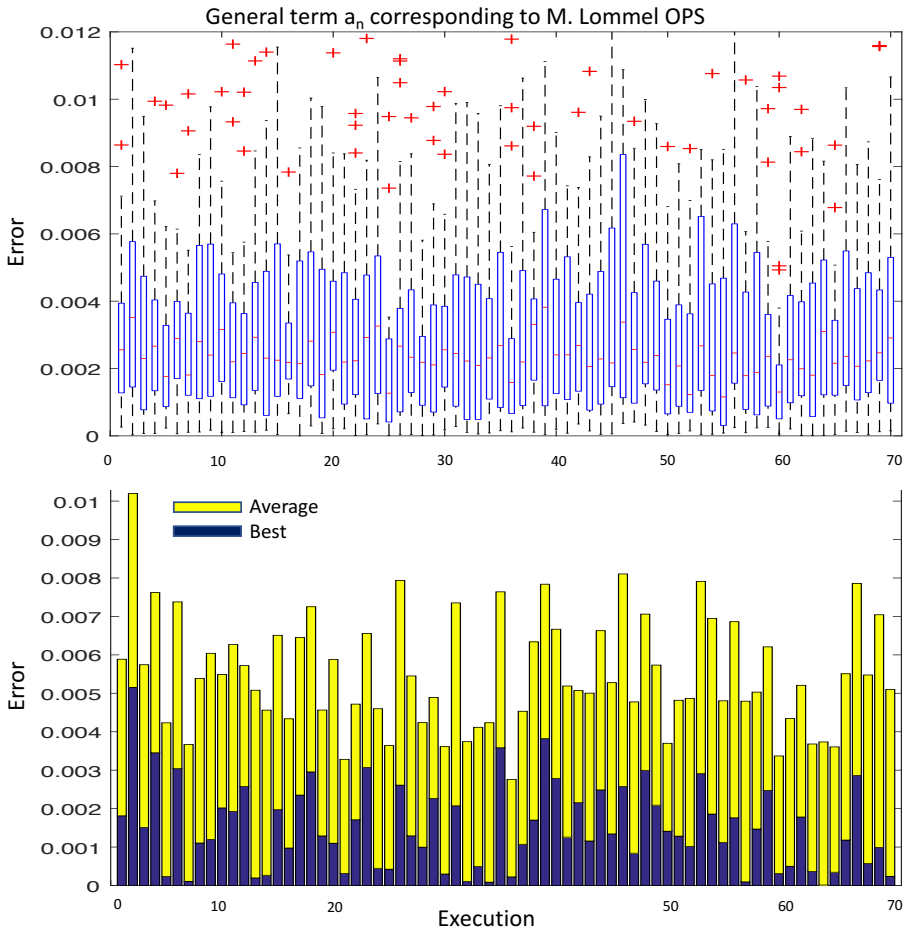


Fig. 6 Testing errors achieved by ANNs in the final population for each execution. Above, error distributions. Below: Best and average errors

best neural network of the population in each experiment is plotted against the generation. Average errors are calculated from 70 executions run. 500 generations are displayed since this value was set as a stop condition for the evolutionary algorithm. The best evolved neural networks calculate the 60 unseen terms included in the testing datasets with a very low error, in average, for the cases of a_n general terms related to Legendre and modified Lommel OPS. In fact, low errors are achieved at the very start of the evolution. On the contrary, Jacobi elliptic and Laguerre experiments start from higher testing errors, but the evolutionary algorithm leads the population to better points of the search space that represent more accurate neural networks calculating the unseen terms of a sequence. Starting from errors nearby 0.045 and 0.035 in average, the evolved solutions reach values of 0.0073 and 0.0069, respectively; very low considering the complexity of these general term functions.

Results in terms of the neural architectures achieved as solution by the proposed evolutionary system are also reported in Table 3 for each of the 70 executions run in each experiment. The table shows the most frequent neural topologies obtained. Therefore, the

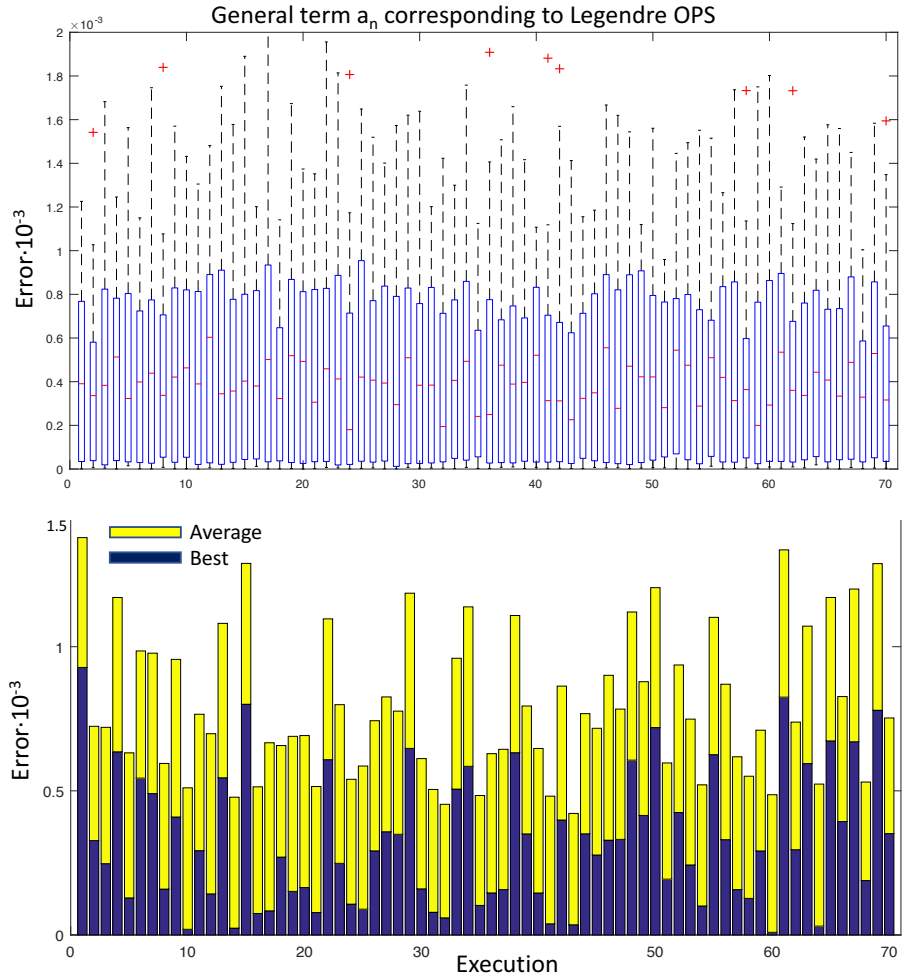


Fig. 7 Testing errors achieved by ANNs in the final population for each execution. Above, error distributions. Below: Best and average errors

percentages column do not always add up to 100%, nor does the frequencies column add up to 70 for each experiment. The notation employed is $1-y_1-y_2-...-y_n-1$, which represent a full-connected feedforward neural network with 1 neuron in the input layer, y_1, y_2, ..., y_n neurons in each of the n hidden layers, and 1 neuron in the output layer. In case of both general terms, a_n and b_n, of Tchebichef, Hermite and Charlier, $1-1$ neural architectures are always obtained as solution of the evolutionary system. The same occurs with the general terms b_n of Laguerre, modified Lommel and Legendre. This is a particular interesting result since these general terms are linear or constant, and the smallest neural network, with just an input and output layers, should be enough to calculate the terms of their sequences.

As it can be observed from Table 3, neural architectures explicitly represented have a reduced size, never surpassing five layers. This is noteworthy because small neural networks that learn the training patterns have better generalization capabilities, what can explain the

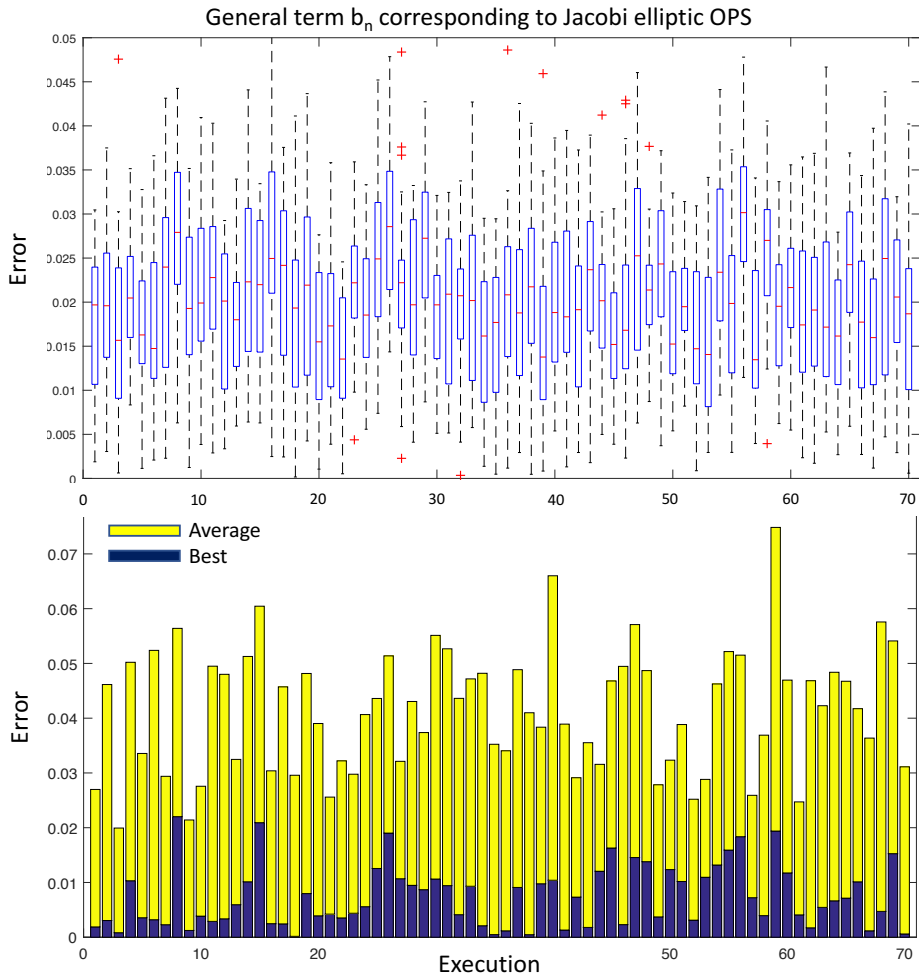


Fig. 8 Testing errors achieved by ANNs in the final population for each execution. Above, error distributions. Below: Best and average errors

accurate results previously shown for the testing phase. The neural architectures that do not appear in this table have a very low frequency, less than three times were evolved as solution. However, these very low frequent architectures may be a numerous group. Such are the cases of Laguerre a_n, and Jacobi elliptics a_n and b_n general terms. Results gathered confirm that even these three groups contain small neural topologies: 97.14% (68/70) of the architectures evolved as solution for Laguerre a_n experiment have a maximum of five layers; 68.57% (48/70) have a maximum of five layers, 91.43% (64/70) in case of six, for Jacobi elliptics a_n; and 94.29% (66/70) of the solutions evolved have a maximum of five layers in case of Jacobi elliptics b_n general term.

The context-free grammar employed in the experiments, $G_{IO\mathbb{F}}$, not only permits to encode the neural topologies throughout the evolutionary process, but also the activation functions assigned to hidden and output layers. The case of linear or constant general terms are, again, interesting to analyze because a linear activation function would accurately calculate

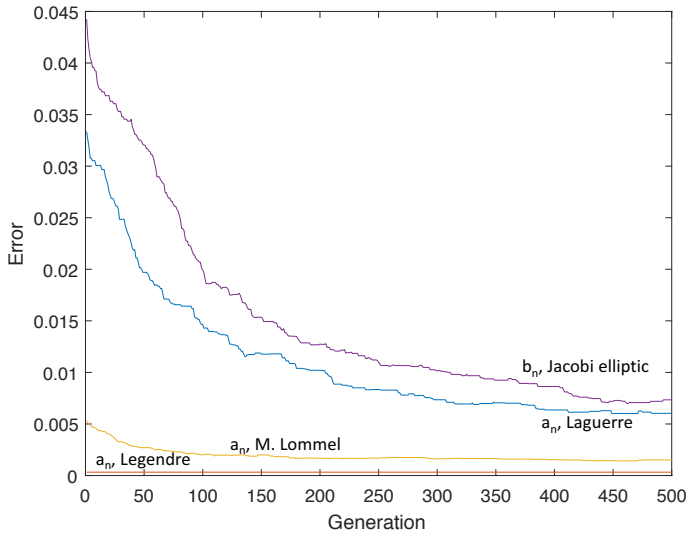


Fig. 9 Convergence speed comparison evolving artificial neural networks

Table 3 Frequencies and percentages of neural topologies obtained as solution after 70 runs

OPS	Gen. term	Topology	Frequency	Percentage
Laguerre	a_n	1-2-1	17	24.28%
		1-3-1-1-1	9	12.85%
		1-3-1	7	10%
		1-2-2-1	6	8.6%
		1-3-2-1-1	4	5.71%
	b_n	1-1	70	100%
Jacobi elliptics	a_n	1-4-1	6	8,57%
		1-8-1	4	5.71%
		1-3-3-1	4	5.71%
		1-5-7-1	3	4.29%
		1-3-1	3	4.29%
		1-2-5-1	3	4.29%
	b_n	1-2-1	18	25.71%
		1-2-2-1	4	5.71%
		1-4-2-1	4	5.71%
		1-4-1	3	4.29%
M. Lommel	a_n	1-1-1	40	57.14%
		1-2-1	20	28.57%
		1-3-1	3	4.29%
	b_n	1-1	70	100%
Lengendre	a_n	1-1-1	70	100%
	b_n	1-1	70	100%

the terms of their sequences. Effectively, the evolutionary system successfully evolves solutions with a pure linear activation function in the output neuron in the 100% of the executions run. This is the simplest solution since the topologies evolved to solve these problems are of the type $1-1$ (two layers), and the input neuron lacks the activation function. The rest of experiments are not linear, and hyperbolic tangent activation functions are always selected by the evolutionary algorithm as they are required to accurately solve these problems.

5 Conclusion

An evolutionary algorithm based on grammar-guided genetic programming is presented to autonomously construct multilayered feedforward neural architectures of any size. The proposed evolutionary system has been successfully applied to approximate the sequences of coefficients that define the three-term recurrence relation associated to well-known OPS. Two different CFG, $G_{IO\mathbb{F}}$ and G_{IO}, have been presented to encode only valid neural architectures, including the activation functions in each layer, or leaving them out; respectively.

Satisfactory results are presented in terms of accuracy and size of the evolved neural architectures approximating sequences of coefficients related to the OPS considered. Since these coefficients are the fundamental quantities in the constructive theory of orthogonal polynomials, the new algorithm provides an interesting contribution. The general terms, a_n and b_n, involved in the experiments cover constant, linear and non-linear functions. Moreover, their sequences, $\{a_n\}$ and $\{b_n\}$, may converge or not. $G_{IO\mathbb{F}}$ has been employed in the experiments, instead of G_{IO}, for the sake of completeness. A pure linear activation function is chosen by the evolutionary algorithm when the neural architecture solution is $1-1$ to solve linear or constant problems. Otherwise, hyperbolic tangent activation functions are also selected.

The accuracy results gathered show that the highest testing error achieved by the neural networks solution within the 95% confidence interval is 0.0653, corresponding to Jacobi elliptic a_n general term, a fourth grade polynomial whose sequence does not converge. This means very accurate results, even when the testing error average of the final populations is considered. Convergence speed results evolving neural networks also confirm this problem as the hardest. Box-and-whisker plots and bar charts show graphically the very small and homogeneous errors achieved by the evolved neural networks within the final population approximating terms of the sequences defined by non-linear general terms.

The size of the evolved neural architectures is another important feature that has been also considered. The smaller the neural networks that can learn the training patterns, the better is their generalization capabilities. This issue has to be solved during the evolutionary process since evolved neural networks can not be regularized via more data. Only the first k terms of a sequence are available to approximate the following m. Results gathered show that most neural networks solution never surpass five layers in the case of non-linear problems, where at least one hidden layer with a non-linear activation function is required. The generalization capability of the neural networks solution is observed from the accuracy results obtained calculating unseen terms of a sequence; namely, the testing dataset.

Future work includes the design of a new neural model capable of approximating also the first k terms of a sequence, those employed as training patterns in this research, from two points in $\mathbb{R}$ and their corresponding images of orthogonal polynomials belonging to sequence up to k degree. By combining both neural and the evolutionary system presented in this work, it will be possible to calculate the two sequences $\{a_n\}$ and $\{b_n\}$ that define an OPS up to $k+m$ degree, with $m >> k$.

The proposed GGGP-based evolutionary algorithm can be also adapted to deep neural networks. As deep learning scales up neural architectures, finding topology together with its hyper-parameters to fit a task is also hard to be accomplished by current approaches, mainly, by hand and patience. Not only deep learning is benefited from the increase of computer power, but also evolutionary algorithms and the hybridization of these two disciplines can do. Nowadays, deep neural networks evolution is a promising approach to solve more challenging problems.

References

1. Abdul-Rahman, O., Munetomo, M., Akama, K.: An improved binary-real coded genetic algorithm for real parameter optimization. In: Proceedings of 3rd World Congress on Nature and Biologically Inspired Computing (2011)
2. Alonso, F., Martínez, L., Pérez, A., Santamaría, A., Valente, J.: Modeling medical time series using grammar-guided genetic programming. In: Proceedings of the 8th industrial conference on Advances in Data Mining: Medical Applications, E-Commerce, Marketing, and Theoretical Aspects (2008)
3. Augasta, M., Kathirvalavakumar, T.: Pruning algorithms of neural networks - a comparative study. Cent. Eur. J. Comput. Sci. **3 (3)**, 105–115 (2013)
4. Barrios, D., López, G., Torrano, E.: Location of zeros and asymptotics of polynomials satisfying three-terms recurrence relations with complex coefficients. Russ. Acad. Sci. Sb. Math. **80 (2)**, 309–333 (1995)
5. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**, 281305 (2012)
6. Bernardos, P., Vosniakos, G.: Optimizing feedforward artificial neural network architecture. Eng. Appl. Artif. Intell. **20**, 365–382 (2007)
7. Cheney, E.: Introduction to Approximation Theory. Providence, Rhode Island: AMS Chelsea Pub. (2000)
8. Chihara, T.: An Introduction to Orthogonal Polynomials. Gordon and Breach (1978)
9. Couchet, J., Manrique, D., Porras, J.: Grammar-guided neural architecture evolution. Lect. Notes Comput. Sc. **4527**, 437–446 (2007)
10. De Jong, K.: Evolutionary Computation: A Unified Approach. Cambridge, MA: MIT Press (2006)
11. Dorado, J.: Cooperative Strategies to Select Automatically Training Patterns and Neural Architectures with Genetic Algorithms. University of La Corua, Spain: PhD Thesis (1999)
12. Fernando, C., Banarse, D., Besse, F., Jaderberg, M., Pfau, D., Reynolds, M., Lactot, M., Wierstra, D.: Convolution by evolution: Differentiable pattern producing networks. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO 2016, Denver, CO, USA (2016)
13. Font, J., Manrique, D., Ramos Criado, P., del Río, D.: Partition based real-valued encoding scheme for evolutionary algorithms. Nat. Comput. **15 (3)**, 477–492 (2016)
14. Font, J., Manrique, D., Ríos, J.: Redes de neuronas artificiales y computación evolutiva. Madrid, Spain: Fundación General de la Universidad Politécnica de Madrid (2009)
15. Garro, B., Sossa, H., Vázquez, R.: Artificial neural network synthesis by means of artificial bee colony (abc) algorithm. In: Proceedings of IEEE Congress on Evolutionary Computation (2011)
16. Gautschi, W.: On generating orthgonal polynomials. SIAM J. Sci. Comput. **3 (3)**, 289–317 (1982)
17. Han, H., Qiao, J.: A structure optimisation algorithm for feedforward neural network construction. Neurocomputing **99**, 347–357 (2013)
18. Haykin, S.: Neural Networks and Learning Machines. Upper Saddle River, NJ: Pearson (2010)
19. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. arXiv: 1207.0580v1 (2012)
20. Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural Netw. **4 (2)**, 251–257 (1991)
21. Huang, G., Chen, L., Siew, C.: Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Trans. Neural Netw. **17 (4)**, 879–892 (2006)
22. Kari, L., Rozenberg, G.: The many facets of natural computing. Commun. ACM **51 (10)**, 72–83 (2008)
23. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press (1992)
24. Loiacono, D., Cardamone, L., Lanzi, P.: Automatic track generation for high-end racing games using evolutionary computation. IEEE Trans. Comput. Intell. AI Games **3 (3)**, 245–259 (2011)
25. Loshchilov, I., Hutter, F.: CMA-ES for hyperparameter optimization of deep neural networks. arXiv: 1604.07269v1 (2016)

26. Maas, A., Hannun, A., Ng, A.: Rectifier nonlinearities improve neural network acoustic models. In: Proceedings of ICML Workshop on Deep Learning for Audio, Speech and Language Processing (2013)
27. Manrique, D., Ríos, J., Rodríguez-Patón, A.: Artificial Neural Networks in Real Life Applications, ch. Self-Adapting Neural Intelligent Systems Using Evolutionary Techniques. Barcelona, Spain: Idea Group (2006)
28. McKay, R., Hoai, N., Whigham, P., Shan, Y., ONeill, M.: Grammar-based genetic programming: A survey. Genet. Program. Evol. Mach. **11 (3)**, 365–396 (1988)
29. Miikkulainen, R.: Encyclopedia of Machine Learning, chap. Neuroevolution. Springer, New York (2010)
30. Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., Hodjat, B.: Evolving deep neural networks. arXiv: 1703.00548v2 (2017)
31. Nielsen, M.: Neural Networks and Deep Learning. Determination Press (2015)
32. Poli, R., Langdon, W., McPhee, N., Koza, J.: A Field Guide to Genetic Programming. UK: Lulu.com (2008)
33. Principe, J.C., Euliano, N.R., Lefebvre, W.C.: Neural and Adaptive Systems, Fundamentals through Simulations. New York: Wiley & Sons (2000)
34. Pulina, L., Tacchella, A.: NeVer: a tool for artificial neural networks verification. Ann. Math. Artif. Intell. **62**(3), 403–425 (2011)
35. Ramos Criado, P.: New techniques for grammar guided genetic programming: dealing with large derivation trees and high cardinality terminal symbol sets. Ph.D. thesis, Universidad Politécnica de Madrid, doi: 10.20868/UPM.thesis.48795 (2017)
36. Samarasinghe, S.: Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition. Florida: CRC Press (2017)
37. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Netw. **61**, 85–117 (2015)
38. Siddiqi, A., Lucas, S.: A comparison of matrix rewriting versus direct encoding method for evolving neural networks. In: Proceedings of 1998 IEEE International Conference on Evolutionary Computation (1998)
39. Simon, D.: Evolutionary Optimization Algorithms. NY: Wiley (2013)
40. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
41. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**, 99–127 (2002)
42. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. J. Artif. Intell. Res. **21**, 63–100 (2004)
43. Tapas, S., Simanta, H., Jana, N.: Artificial neural network training using differential evolutionary algorithm for classification. Adv. Intel. Soft Comp. **132**, 769–778 (2012)
44. Wan, L., Zeiler, M., Zhang, S., Cun, Y.L., Fergus, R.: Regularization of neural networks using dropconnect. In: S. Dasgupta, D. McAllester (eds.) Proceedings of the 30th International Conference on Machine Learning, pp. 1058–1066. PMLR, Atlanta, Georgia, USA (2013)
45. Wang, J., Wang, H., Chen, Y., Liu, C.: A constructive algorithm for unsupervised learning with incremental neural network. J. Appl. Res. Technol. **13 (2)**, 188–196 (2015)
46. Whigham, P.: Grammatically-based genetic programming. In: Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (1995)
47. Yu, X., Gen, M.: Introduction to Evolutionary Algorithms. London, UK: Springer (2010)
48. Zoph, B., Le., Q.V.: Neural architecture search with reinforcement learning. arXiv:1611.01578v2 (2016)