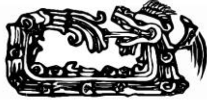

www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 16(1), Jan-April 2025, 151-163. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2025.v16i1.549

Dynamic Neural Network Optimization: A single agent neuroevolution algorithm based on hill climbing optimization for Neural Architecture Search

Yoqsan Angeles[1], Valeria Karina Legaria-Santiago[2], Hiram Calvo[1], Álvaro Anzueto[2]

[1] Computational Cognitive Sciences Laboratory, CIC-Instituto Politécnico Nacional, Av. Juan de Dios Bátiz, Mexico City, 07738, Mexico City, Mexico.
[2] Centro de Investigación en Computación (CIC), Instituto Politécnico Nacional, Av. Juan de Dios Bátiz, Mexico City, 07738, Mexico City, Mexico.
[3] Bionics Department, UPIITA-Instituto Politécnico Nacional, Insituto Politécnico Nacional, Mexico City, 07340, Mexico City, Mexico.
yangelesg2020@cic.ipn.mx; vlegarias2019@cic.ipn.mx; hcalvo@cic.ipn.mx; aanzuetor@ipn.mx;

Abstract. In the field of deep learning, the identification of optimal neural architectures requires not only profound expertise but also a substantial investment of time and effort in the evaluation of the outcomes generated by each proposed model. In this study, we introduce a Single Agent Neuroevolution algorithm, based on the Hill Climbing algorithm for Neural Architecture Search, named Dynamic Neural Network Optimization (DyNNO). This approach focuses on the evaluation of the performance of neural networks optimized for function approximation. Additionally, we have explored the minimization of the number of neurons within the neural network structures. The results demonstrated the feasibility of using this algorithm to automate the neural architecture search process. Furthermore, the reduction in the number of parameters improved the generalization capability of the networks. The findings also suggest that mutation in activation functions can be a factor to explore in achieving a more effective reduction in error rates.

Keywords: Artificial Neural Networks, Hill climbing, Metaheuristic, Neural Architecture Search, Function aproximation

Article Info
Received November 13, 2024
Accepted March 12, 2025

1 Introduction

Artificial Neural Networks (ANNs) are computational models that attempt to mimic the features of biological neurons. They are typically composed of interconnected nodes, or neurons, which can have one or more inputs and outputs. The basic model of a neuron in neural networks involves computing a weighted sum of the inputs, which is then passed through a typically nonlinear activation function, although linear functions can also be used. This output determines the response of the neuron. A single neuron computes this operation independently. When two or more nodes (neurons) compute such sums using the same inputs and are organized together, they form a structure commonly referred to as a perceptron. This model is widely applied in different fields for classification and regression problems (Campbell & Gear, 1995). If the perceptron has more than one layer, then it is a Multilayer Perceptron (MLP).

To build a MLP model is necessary to define its architecture, i.e., establish the number of layers, the number of neurons per layer, and the activation function that the model will have, although the process of finding the best parameters is usually time-consuming and requires a certain level of expert knowledge. The Neural Architecture Search (NAS) is a paradigm that emerged to optimize this process. In this way, different strategies have been proposed such as reinforcement learning, evolutionary algorithms, and gradient-based methods. However, reinforcement learning regularly requires large amounts of computational resources. Evolutionary algorithms may offer a viable alternative when resources are constrained, owing to their rapid convergence and ability to operate without prior knowledge. Despite the absence of such knowledge, their performance is at least comparable to other algorithms (Pan & Yao, 2021).

© Editorial Académica Dragón Azteca S. de R.L. de C.V. (EDITADA.ORG), All rights reserved.

In this context, metaheuristic algorithms, some of them inspired by optimization processes observed in nature, provide an efficient search method. Metaheuristic algorithms are computational optimization approaches to find high-quality solutions for complex and challenging problems. Unlike conventional methods, metaheuristic algorithms do not rely on specific problem information and can explore broad solution spaces in search of the optimal solution. These algorithms draw inspiration from natural processes, physical phenomena, or global search strategies to guide the exploration of solution spaces. Examples of metaheuristic algorithms include hill climbing, genetic algorithms, simulated annealing, tabu search, and particle swarm optimization. Their flexibility and ability to address various problems make metaheuristic algorithms powerful tools for tackling complex challenges in engineering, logistics, planning, and others.

In this study, we propose a single agent neuroevolution algorithm based on hill climbing named Dynamic Neural Network Optimization (DyNNO), which is employed for the exploration of neural architectures in a multilayer perceptron, applied to the task of function approximation. The structure of this paper is as follows: Section 2 presents previous work. Section 3 explains the algorithm of Dynamic Neural Network Optimization. Section 4 explains the experiments done with the functions to be approximated by the neural network. Section 5 showcases the obtained results. Section 6 provides the conclusions.

2 Previous work

According to (Elsken et al., 2019) Neural Architecture Search (NAS) can be divided into three fields: search space (to define which architectures to explore, initially), search strategy (to maximize ANN performance by exploring the search space of neural architectures), and performance estimation strategy (to estimate architecture's performance without training each architecture to be evaluated from scratch).

In the case of search strategy (the field where this work lay), some of the proposed methods have been Bayesian optimization (Rao et al., 2022), evolutionary methods (Klosa & Büskens, 2022; Niu et al., 2019) and reinforcement learning (Lyu et al., 2023). In the context of neuro-evolutionary methods, genetic algorithms are used as a weight-sharing strategy to speed up the evaluation of the architectures. Moreover, other methods, such as simulated annealing or tabu search, have also been proposed (Ludermir et al., 2006).

Nevertheless, most of the work on NAS revolves around the training or construction of Convolutional Neural Networks, with interest in reducing their design time and even resource use (Wu et al., 2022), complexity (DLW-NAS), or improving its performance in object classification, segmentation or identification tasks (Li et al., 2021), where data sets such as CIFAR-10 (Niu et al., 2019; Z. Chen & Li, 2020) or ImageNet (Wu et al., 2022) is used as a reference (DLW-NAS). However, each domain requires a specific approach (Wolpert & Macready, 1997). Few works focus NAS on other kinds of networks, for example, in (Ludermir et al., 2006) a NAS algorithm was proposed to optimize a multilayer perceptron (MLP) network weights and architectures, where the constraint was to generate topologies with few connections and high classification performance to solve classification tasks with tabular datasets and prediction tasks with time series. (Klosa & Büskens, 2022) propose an evolutionary NAS to predict traffic conditions training Graph Convolutional Networks (GCN), taking care of performance, robustness, and resource consumption.

3 Dynamic Neural Network Optimization (DyNNO)

The hill climbing algorithm is an optimization method used in machine learning. The algorithm aims to find the best solution to a problem through successive iterations. It starts at a random solution and makes small modifications to get a new solution. If the new solution is better, then it becomes the actual solution and repeats the process until no further improvements are found or other termination criteria are met. This approach is particularly useful in complex problems where finding the optimal solution is computationally difficult. This algorithm has been used in many applications (Tsamardinos et al., 2006; Guindon and Gascuel, 2003).

DyNNO is based on hill climbing; however, instead of selecting a random point in the search space, a neural network with a random architecture within the search space is created. The pseudocode is presented in Algorithm 1, and its explanation is as follows:

- First the neural network architecture is constructed using the random net function. This function takes as parameters the minimum and maximum values for the number of layers and the number of neurons per layer the network may have when it is initialized. Then the architecture is built with a random number of layers, within the defined range of layers. For each layer,

the number of neurons is chosen randomly within the defined range of neurons, and a matrix of random numbers between 0 and 1 is created to generate the synaptic weights. In this work all random choices were taken from a uniform distribution, the layers range was [2, 3] and the neurons range was [50,200]. Finally, all layers have a hyperbolic tangent activation function, except for the output layer, which has a linear function.

- Once the neural network is created, it is trained using the backpropagation algorithm, employing the mean squared error as the loss function. Additionally, a minibatch of size 100 is utilized to expedite the training process.
- With the initial network trained, the architecture search process begins. For this, a mutation is applied to the network, where the mutation process includes modifying the number of neurons in a layer, the number of layers, or the activation functions (see subsection 3.1. Modification of the number of neurons or layers and subsection 3.2 Modifying activation functions). The mutated network is also trained using the backpropagation algorithm. If the mutated network exhibits a lower error, the mutated network replaces the current one; otherwise, no changes are made, and the current network undergoes another mutation.
- This process continues for a specified number of epochs. In this work, 20 epochs were conducted for each function. The number of epochs was chosen solely based on execution time, ensuring the algorithm did not exceed one hour for each trial.

Algorithm 1. DyNNO Algorithm

DyNNO Algorithm
1: procedure DyNNO(layers range, neurons range, Input, Output, epochs)
2: net ← random net(layers range, neurons range)
3: net, error ← train net(net, Input, Output)
4: for epoch ← 1 to epochs do
5: new net ← mutate net(net)
6: new net, new error ← train(new net, Input, Output)
7: if new error < error then
8: net ← new net
9: error ← new error
10: end if
11: end for
12: end procedure

3.1 Modification of the number of neurons or layers

The mutation algorithm for modifying the number of neurons or layers automatically and randomly is shown in Algorithm 2. The process of mutation is realized every epoch, and the number of layers could decrease until 1, but increase without restriction (according to the modifications made in each epoch). The modification of the number of neurons or layers is generated according to the value of r, where r is a random number with uniform distribution with values from 0 to 1, and it is generated every training epoch. The mutation of the network is generated according to the value of r, with the following rules:

1. If $r < 0.1$, a layer is added at a random position in the architecture. The number of neurons in the new layer is randomly generated based on the neurons range defined previously. Their weights are also randomly generated with values between 0 and the weights of the previous layer and the next layer are adjusted to match the weights of the new layer.
2. If $r \geq 0.1$ and $r \leq 0.9$, the number of neurons is modified, excepting the last layer so as not to modify the desired output size of the network. Neurons are added with a probability of 0.5, otherwise, they are removed. The number of neurons to add or remove, number neurons, is chosen randomly in a range from 1 to 10.
3. $r > 0.9$, a random layer is removed. The weights of the next layer are adjusted to match the layer before the removed layer.
4. In cases 2 and 3, if only the output layer remains, a hidden layer is added.

Algorithm 2. Mutation algorithm for neurons and layers

Mutation algorithm for neurons and layers
1: procedure Network Mutation(neurons range)
2: r ← rand[0, 1]
3: if r < 0.1 then
4: net mut ← add layer(neurons range)
5: else if r > 0.1 and r < 0.9 then
6: net mut ← modify number of neurons(random layer, number neurons)
7: else
8: net mut ← delete layer
9: end if
10: end procedure

3.2 Modifying activation functions

Modifying activation functions consists of changing the type of function (sigmoid or tangent), or its characteristics (amplitude and slope). Eq. 1 shows the sigmoid activation function and eq. 2 shows the hyperbolic tangent activation function. Fig. 1 shows an example of two variants of the sigmoid function. The blue one has a slope and amplitude equal to one. The red one has a slope and amplitude equal to two.

$$f(x) = \frac{\alpha}{1 + e^{-\beta x}}. \quad (1)$$

$$f(x) = \alpha \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}}. \quad (2)$$

where
α is a factor for the slope
β is a factor for the amplitude

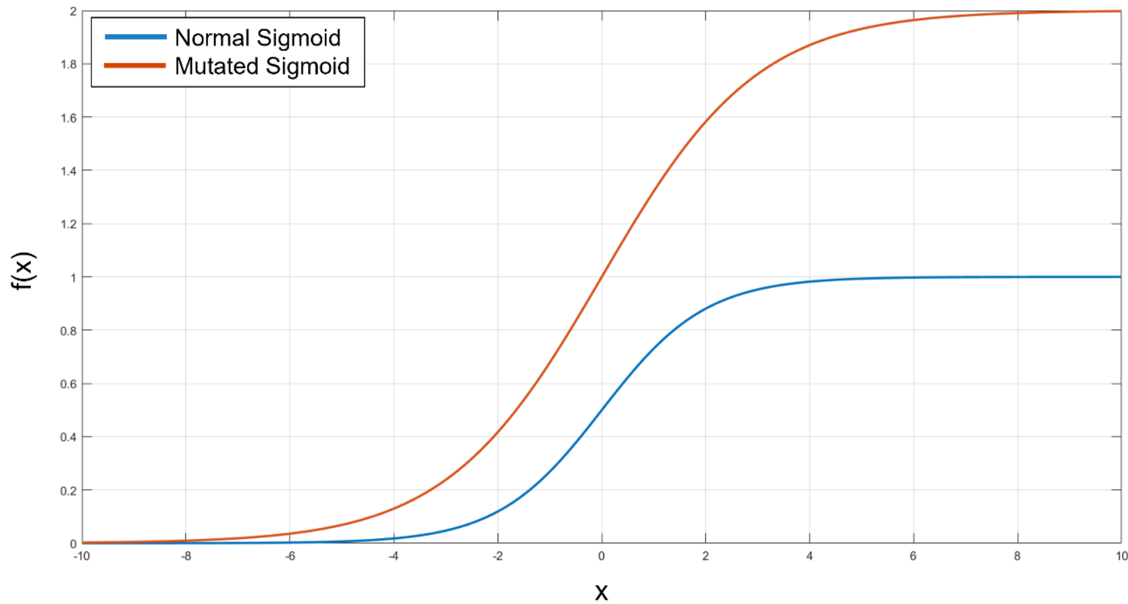

Fig. 1. Sigmoid function

Function modification is performed with a probability of 0.2 as long as there is at least one hidden layer, this due to the output layer is never modified, and its function always remains linear. This value to function modification was chosen because most of

the research utilizes standard activation functions, however, there is evidence suggesting that modifying these functions can enhance performance (Mercioni et al., 2019; Pedamonti, 2018). Whether the function modification is valid, a hidden layer n is randomly selected, and random numbers r1 and r2 are generated with values from 0 to 1. The function modification is carried out according to the following rules:

1. If r1 < 0.2 we proceed to the activation function mutation. Otherwise, no modification is made to the activation functions.
   (a) If r2 < 0.2, the type of activation function is modified. If the current function is sigmoid type it is changed to a hyperbolic tangent type function. Conversely, if the current function is a hyperbolic tangent it is changed to the sigmoid function.
   (b) If r2 ≥ 0.2, the parameters of the current activation function are randomly modified, increasing, or decreasing the amplitude and the slope of the function, each one with a different random value between -0.5 and 0.5.

Algorithm 1. Mutation algorithm for activation functions

Mutation algorithm for activation functions
1: procedure NetworkMutation(net)
2: r1 ← rand
3: if r1 < 0.2 then
4: if numLayers > 2 then
5: r2 ← rand
6: n ← randint(lenLayers)
7: if r2 < 0.2 then
8: Layers.f cn(n) ← change f cn(Layers.f cn(n))
9: else
10: m ← rand([−0.5, 0.5])
11: Layers.f cn(n).amplitude ← Layers.f cn(n).amplitude + m
12: p ← rand([−0.5, 0.5])
13: Layers.f cn(n).slope ← Layers.f cn(n).slope + p
14: end if
15: end if
16: end if
17: end procedure

3.3 Optimization Model

Considering a neural network with architecture x, it is applied the mutation function M to get the new neural architecture x′, defined by:

$$x' = M(x) \quad (3)$$

Then, the neural network error is defined as ϵ1 for the neural network with the architecture before the mutation, and ϵ2 for the neural network error after the mutation:

$$\epsilon 1 = \text{Error}(x) \quad (4)$$

$$\epsilon 2 = \text{Error}(x') \quad (5)$$

The neural network’s error is calculated with eq. 6:

$$\text{Error} = \frac{1}{n}\sum_{i=1}^{n} (y_i - y_i')^2 \quad (6)$$

where,
y_i is the output of the function (Sphere, Rastrigin or Griewank) and y'_i is the output of the neural network.

The new architecture x′ will be accepted if the following conditions are satisfied:

$$\text{Accept } x' \text{ if } (\epsilon 2 < \epsilon 1) \quad (7)$$

The algorithm with the acceptance condition given by eq. 7 is named Normal DyNNO, to refer to it only searches for the minimum error value. Additionally, a second proposal of the previous model has a variation of the acceptance condition by considering the possibility of a slightly worse performance but with fewer parameters in the neural network. The algorithm with the second acceptance condition is referred to as Min DyNNO. The acceptance condition of Min DyNNO is described in eq. 8.

$$\text{Accept } x' \text{ if } (\epsilon 2 - \epsilon 1 < 0.1) \text{ AND } (\text{NumParams}(x') < \text{NumParams}(x)) \quad (8)$$

where,
NumParams(x) and NumParams(x′) represent the number of parameters in the architectures x and x′, respectively.

4 Experiments

The approximation functions and the experiments carried out are described in this section. This work considers three different functions to be approximated by the neural networks: Sphere, given by eq. 9, Rastrigin, given by eq. 10 and Griewank, given by eq. 11. The functions were selected due to their use in testing optimization algorithms (Yang et al., 2013; Garcıa et al., 2023), and their form poses an interesting challenge for approximation with a neural network.

$$f(x) = \sum_{i=1}^{d} x_i^2 \quad (9)$$

$$f(x) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \quad (10)$$

$$f(x) = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)] \quad (11)$$

where,
d is the number of dimensions for the function, and X is a vector.

The experiments consider different dimensions of the same function, varying the number from 1 to 10, but maintaining a single output in each function. The output of the functions for dimensions 1 and 10 are shown in Fig. 2. The input data is in the range of [-1, 1], with increments of 0.1, which returns a vector size = 21. In scenarios involving two or more dimensions, a grid of uniformly distributed points is generated and organized into a matrix of size [dimension, $\text{vector_size}^{\text{dimension}}$]. However, if $\text{vector_size}^{\text{dimension}} > 10,000$, only the first 10,000 columns are used.

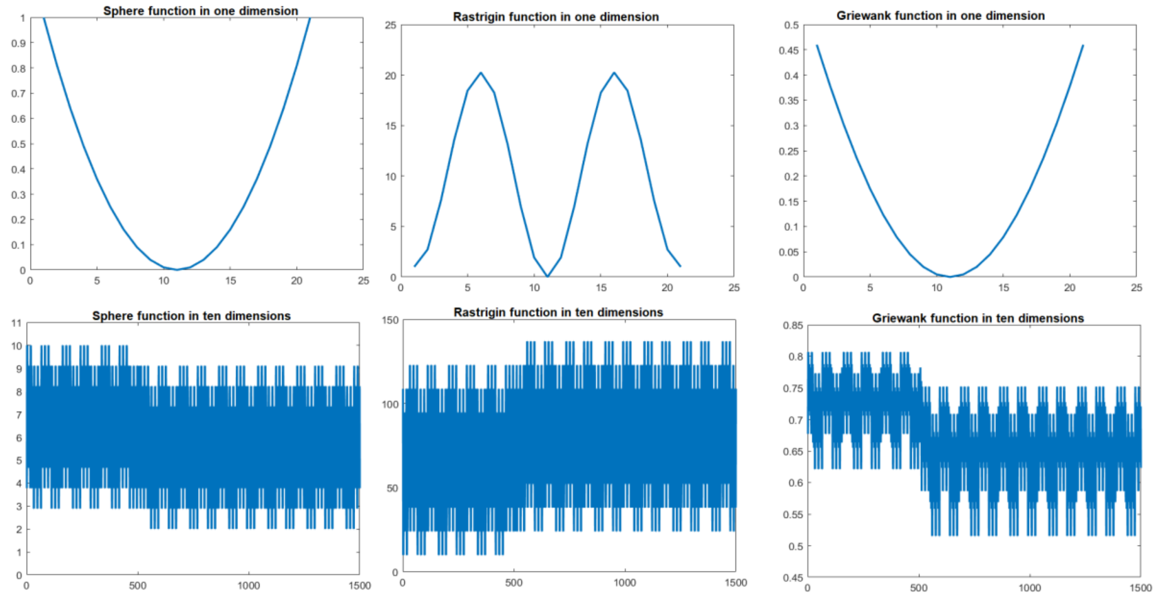

Fig. 2. Function outputs for dimensions 1 and 10

5 Results

A hundred experiments were conducted for each dimension, with statistical error values calculated. Two types of training were performed, first with Normal DyNNO and then with Min DyNNO.

The pseudocode for Min DyNNo was adjusted accordingly to Algorithm 4. After completing the trials, the following statistical data on the error was collected: minimum, maximum, average, median, standard deviation, total weights, and bias of the network with the minimum error, and average error across the 100 trials. Then, we also used the best neural network with validation data with the same length as the training input. The validation data is the same as the input but adds random numbers in the range of [-0.1,0.1]. The results are shown in the Tables 1,2,3,4,5,6. Based on the data obtained from the conducted tests, we proceeded to implement a Multi-Criteria Decision Making (MCDM) approach, a methodology used for decision-making involving multiple criteria (Hwang et al., 1981). We employed the TOPSIS (Technique for Order Preference by Similarity to Ideal Solution) method, developed and applied in numerous studies and practical applications (S.-J. Chen & Hwang, 1992; Behzadian et al., 2012). This technique is based on the idea that the chosen option should have the smallest Euclidean distance to the ideal solution and the greatest distance to the non-ideal solution. The criteria used are the statistical data in the tables.

Algorithm 4. Modification to prioritize fewer parameters

Modification to prioritize fewer parameters
1: if new error < error OR (new error −error) < 0.1AND new net.P arameters < net.P aramete 2: net ← new net 3: error ← new error 4: end if

Table 1. Results of Sphere function with normal algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	0.1078	168.44	3.4382	0.3479	17.5571	0.1098	7326
2	0.0123	0.2156	0.0672	0.0627	0.0372	0.0134	13440
3	0.0057	0.5428	0.1069	0.0453	0.1240	0.0062	13516
4	0.0137	0.6236	0.0962	0.0641	0.1120	0.0289	11342

5	0.0020	0.0140	0.0050	0.0041	0.0025	12.1131	8852
6	0.0033	0.0096	0.0054	0.0051	0.0018	4.5509	7093
7	0.0010	0.0175	0.0035	0.0026	0.0029	0.1402	8187
8	0.0030	0.0134	0.0054	0.0049	0.0019	8.5237	11964
9	0.0035	0.0171	0.0059	0.0049	0.0036	11.7938	6791
10	8.06e-05	0.0038	0.0011	9.98e-04	8.44e-04	0.2715	11684

Table 2. Results of Sphere function with minimization algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	0.1069	41.9907	1.3946	0.2660	5.0881	0.1132	7380
2	0.0023	0.1042	0.0210	0.0194	0.0172	0.0035	3794
3	0.0062	0.1097	0.0351	0.0372	0.0217	0.0060	3852
4	0.0006	0.0272	0.0022	0.0015	0.0033	0.0024	685
5	0.0027	0.0586	0.0077	0.0055	0.0068	0.0821	6051
6	0.0035	0.0969	0.0100	0.0072	0.0132	0.2087	3595
7	7.22e-04	0.0360	0.0072	0.0050	0.0081	0.2880	3273
8	2.15e-04	0.0252	0.0103	0.0081	0.0059	0.2704	8953
9	0.0025	0.0413	0.0074	0.0057	0.0060	0.1662	3901
10	1.43e-05	0.0071	0.0023	0.0020	0.0015	1.6687	5321

The factors were weighted to give more importance to some of them. To weigh the criteria, decimal numbers were assigned whose sum equals 1. The assigned weights were as follows: minimum error: 0.2, maximum error: 0.1, average error: 0.15, median error: 0.15, standard deviation error: 0.1, and test error: 0.3.

To apply TOPSIS, the statistical data obtained in all the dimensions were added up according to their function and the experiment under comparison (with Min DyNNO referred to as “Min” against Normal DyNNO, referred to as “Normal”). The results are shown in the first section of Table 7.

Table 3. Results of Rastrigin function with normal algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	42.1788	155.3428	62.0994	53.3792	83	42.0016	5079
2	0.0795	99.9939	2.3214	1.3431	9.9090	0.4928	4580
3	0.0130	37.3487	0.6232	0.0766	3.7232	0.3360	7480
4	0.0404	0.9145	0.2472	0.1934	0.1813	1.7927	13978
5	0.0677	110.7870	49.7869	55.0708	25.5513	6.6513	4098
6	0.0158	1.5043	0.3555	0.2385	0.3133	8.4582	6668
7	0.0008	41.4356	2.0495	0.0042	8.8448	24.5503	3469
8	0.0015	152.9829	30.8810	0.0062	58.2250	37.0374	7405
9	0.0010	63.2166	0.6354	0.0028	6.3213	25.4991	2670
10	1.19e-06	0.0035	4.89e-04	7.02e-05	7.66e-04	253.6776	14347

Table 4. Results of Rastrigin function with minimization algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	44.4816	188.4018	67.4475	54.8170	27.6903	46.8318	11139
2	0.1817	2.7307	0.7864	0.6116	0.5766	0.6607	14138
3	0.1026	1.1735	0.3705	0.2730	0.2571	0.5637	9830
4	0.0667	1.7757	0.2997	0.2609	0.2150	1.7412	12598
5	0.1720	1.8329	0.7367	0.5561	0.4650	9.7870	3619
6	0.1612	1.2311	0.4856	0.3424	0.2643	11.0513	5020
7	0.0078	0.1491	0.0419	0.0298	0.0323	20.5160	1418
8	0.0001	0.1106	0.0388	0.0313	0.0281	183.1473	6453
9	0.0010	77.6584	3.3909	0.0038	14.9381	64.0813	1918
10	1.01e-05	0.0332	0.0025	0.0018	0.0042	279.6284	5523

Table 5. Results of Griewank function with normal algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	0.0229	130.1647	2.1082	0.1100	13.1507	0.0228	2198
2	0.0009	0.0239	0.0057	0.0046	0.0044	0.0012	9895
3	0.0002	0.0063	0.0021	0.0018	0.0015	0.2158	6419
4	0.0006	0.0055	0.0022	0.0025	0.0008	0.0015	7112
5	0.0002	0.0043	0.0021	0.0024	0.0008	0.0201	8245
6	0.0003	0.0045	0.0016	0.0018	0.0006	0.0033	6049
7	0.0005	0.0066	0.0026	0.0030	0.0010	0.0072	3661
8	0.0003	0.0044	0.0024	0.0023	0.0011	0.0023	3103
9	0.0003	0.0048	0.0021	0.0019	0.0012	0.0042	3861
10	2.06e-6	0.0055	0.0018	0.0013	0.0013	5.3608	2492

Following that, the outcomes of the three functions derived from the Normal DyNNO algorithm were added, doing the same with the results obtained with the Min DyNNO algorithm. The results are shown in the second section of Table 7. Subsequently, they were normalized using the Euclidean norm by dividing all the data by the number c obtained with equation 12.

Table 6. Results of Griewank function with minimization algorithm

Dim	Min	Max	Avg	Med	Sd	Test error	Param. avg
1	0.0230	20.7805	1.0913	0.2056	2.9516	0.0266	5383
2	0.0056	0.1315	0.0305	0.0247	0.0188	0.0049	6720
3	0.0028	0.0693	0.0238	0.0227	0.0092	0.0030	3060
4	7.26e-4	0.0359	0.0040	0.0027	0.0048	0.0011	10975
5	5.65e-04	0.0196	0.0037	0.0027	0.0032	0.0038	6837
6	8.21e-04	0.0255	0.0028	0.0018	0.0036	0.0024	5789
7	4.28e-04	0.0178	0.0041	0.0032	0.0029	0.0024	4858
8	4.41e-04	0.0351	0.0042	0.0037	0.0036	0.0022	4932
9	6.23e-04	0.0083	0.0037	0.0040	0.0011	0.0039	1190
10	5.17e-04	0.0059	0.0039	0.0041	0.0009	0.0045	1864

Table 7. Results of Sphere function with normal algorithm

	Min	Max	Avg	Med	Sd	Test error	Param. avg
Sphere Normal	0.1524	169.9018	3.7348	0.5426	17.8438	37.5515	100200
Sphere Min	0.1257	42.4969	1.4978	0.3576	5.1718	2.8092	46805
Rastrigin Normal	42.3985	663.5298	149.0000	110.3149	196.0700	400.4970	69774
Rastrigin Min	45.1747	275.0970	73.6005	56.9277	44.4710	618.0087	61836
Griewank Normal	0.0262	130.2305	2.1308	0.1316	13.1634	5.6392	53035
Griewank Min	0.0355	21.1294	1.1720	0.2752	2.9997	0.0548	51608
Normal	42.5771	963.6621	154.8656	110.9891	227.0772	443.6877	223000
Min parameters	45.3359	338.7233	76.2703	57.5605 5	52.642	620.8727	160250

The sums of the statistical error data were divided by the maximum output value of the larger dimension of each function, except for Griewank, whose maximum value is less than 1. The maximum output value in the Sphere function was 10, in Rastrigin it was 151.11, and in Griewank it was 0.8068.

$$c = \sqrt{\sum_{i=1}^{n} Matrix_i^2} \quad \textbf{(12)}$$

The idea of TOPSIS is to calculate the distance between the actual solution and the ideal value, and the distance between the actual solution and the worst solution. In this case, the worst solution would be an infinite value, so the distance to the worst solution is irrelevant, and only the distance to the best solution is considered.

Ideal solution = 0.

To calculate the distance to the ideal solution we use eq. 13 being Matrix the values of the rows of the second section of Table 7.

W = [0.2, 0.1, 0.15, 0.15, 0.1, 0.3],

$$DistPos_i = \sqrt{\sum_{i=1}^{n} \left(w_i \left(Matrix_{j,i} - IdealPositive_i\right)\right)^2} \quad \textbf{(13)}$$

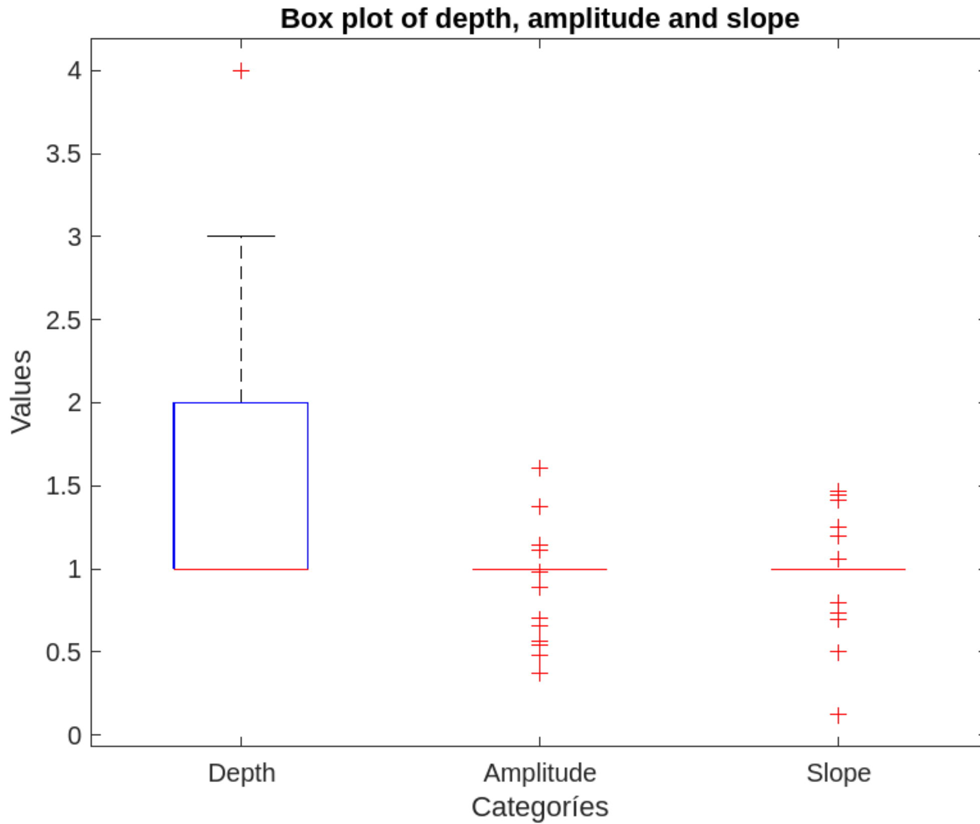

Fig. 3. Box-and-whisker plot of the final characteristics of the best neural architectures

Applying the TOPSIS method we obtain the following results:

Normal DyNNO: 1.1958
Min DyNNO: 0.6299

It is important to emphasize that MCDM does not inherently connote a multi-objective problem. MCDM is employed due to the presence of distinct data to calculate the best of the two proposed algorithms. Nevertheless, within the framework of Min DyNNO, the primary focus resides in error minimization, with the concurrent aspiration to minimize parameter count serving as a secondary objective.

Finally, figure 3 shows the box-and-whisker plot for the characteristics of the best neural architectures, considering all dimensions and functions. The characteristics shown are the number of hidden layers (depth), as well as the amplitude and slope of the activation functions of each hidden layer of the network.

In addition, we obtained the results shown in Table 8, using the data from the functions Sphere, Rastrigin, and Griewank in two and ten dimensions, following the methodology described in (Garcıa et al., 2023), where 5 metaheuristic algorithms were compared using an architecture of one hidden layer with 150 neurons. The results obtained by (Yang et al., 2013), where they trained a neural network for function approximation using BackPropagation (BP), Radial Basis Function (RBF) and Generalized Regression Neural Network (GRNN), are also shown in Table 8, but some cells remain empty because the lack of information reported.

Table 2. Comparisson of different optimization algorithms using data from functions Sphere, Rastrigin and Griewank in two and ten dimensions

	Sphere 2D	Rastrigin 2D	Griewank 2D	Sphere 10D	Rastrigin 10D	Griewank 10D
PSO	18.1777	15.1972	14.2934	144.7836	188.8615	136.6322

SA	14.6961	12.9845	13.1961	66.1094	103.6318	73.5678
DE	109.1617	295.3036	243.0766	737.33	1108.9818	594.8004
GA	447.3259	1158.2	979.2230	1258	3419.2	2155.5
ABC	11.4331	15.9049	11.5369	79.4692	180.6814	109.3722
BP	0.1793	—	—	—	—	—
RBF	23.59	0.1987	0.5601	—	—	—
GRNN	78900	0.5691	0.5600	—	—	—
DyNNO Normal	0.0672	2.3214	0.0057	0.0038	0.0005	0.0018
DyNNO Min	0.1042	0.7864	0.0305	0.0071	0.0332	0.0039

6 Conclusions

By using this neural architecture search algorithm, it is possible to avoid the need to manually supervise the training process and evaluate the improvement in error. The automation of neural architecture optimization not only simplifies the process but also has the potential to lead to more efficient performance by enabling the algorithm to systematically explore and propose network structures without the need for expert knowledge. This alleviates users from the laborious task of manually adjusting architectures and allows for a more effective approach in the quest for optimal neural models. An example of this was shown in Table 8, comparing the performance in function approximation using the same data across different algorithms that used manually defined architectures in other works. The results showed that the architecture found by applying the DyNNO algorithm had the best performance.

The optimization of neural architectures was carried out using the proposed metaheuristic DyNNO. In this algorithm, the training method was backpropagation but other optimization algorithms as Levenberg-Marquardt or metaheuristics can be used. While there is a possibility that a population-based algorithm may achieve superior results by expanding the search space, it is crucial to note that its execution time tends to increase with the size of the population. In this work, evolutionary approaches to population-based algorithms were implemented, choosing to consider a single individual to significantly reduce execution time. This approach effectively addressed the optimization of neural architectures, ensuring promising results while optimizing computational efficiency.

The experimental results reveal that, although Normal DyNNO algorithm shows a reduction in error when working with training data, this apparent advantage is counteracted when evaluating the model with validation data. Conversely, by using Min DyNNO, an enhancement in the model's generalization capacity was observed. These findings suggest that the strategy of reducing parameters strengthens the model's ability to generalize more effectively to unseen data during training.

According to the Multi-Criteria Decision Making (MCDM) analysis, the value of the distance to the positive ideal was lower in the algorithm with parameter minimization, indicating that it had a positive effect on the performance of the networks. Hence, it is advisable to follow this approach in future implementations. The algorithm can be applied to any problem that is amenable to solution by MLPs, extending beyond mere approximation function tasks. Regarding the mutation of activation functions, although the average of the best architectures remained within standard values, there were several instances of mutated activation functions. Therefore, it is recommended to explore mutation on activation functions for future work.

Acknowledgments. The authors wish to thank the support of the Instituto Politécnico Nacional (COFAA, SIP-IPN), and the Mexican Government (CONAHCyT, SNI).

References

Behzadian, M., Otaghsara, S. K., Yazdani, M., & Ignatius, J. (2012). A state-of-the-art survey of TOPSIS applications. *Expert Systems with Applications, 39(17)*, 13051–13069.

Campbell, S. L., & Gear, C. W. (1995). The index of general nonlinear DAES. *Numerische Mathematik*, 72(2), 173–196.

Chen, S. J., & Hwang, C. L. (1992). Fuzzy multiple attribute decision making methods. In *Fuzzy multiple attribute decision making: Methods and applications* (pp. 289-486). Berlin, Heidelberg: Springer Berlin Heidelberg.

Chen, Z., & Li, B. (2020). Efficient evolution for neural architecture search. In *2020 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–7). IEEE. https://doi.org/10.1109/IJCNN48605.2020.9207545

Elsken, T., Metzen, J. H., & Hutter, F. (2019). Neural architecture search: A survey. *Journal of Machine Learning Research*, *20*(55), 1-21. https://doi.org/10.1007/978-3-030-05318-5_3

García, Y. A., Calvo, H., & Ríos, A. A. (2023). Uso de metaheurísticas para entrenamiento de redes neuronales artificiales. *Research in Computing Science, 152(8)*, 127–139.

Guindon, S., & Gascuel, O. (2003). A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology, 52(5)*, 696–704.

Hwang, C. L., Yoon, K., Hwang, C. L., & Yoon, K. (1981). Methods for multiple attribute decision making. *Multiple attribute decision making: methods and applications a state-of-the-art survey*, 58-191.

Klosa, D., & Büskens, C. (2022). Evolutionary neural architecture search for traffic forecasting. In *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)* (pp. 1230–1237). IEEE. https://doi.org/10.1109/ICMLA55696.2022.00198

Li, Z., Xi, T., Zhang, G., & et al. (2021). Autodet: Pyramid network architecture search for object detection. *International Journal of Computer Vision, 129*, 1087–1105. https://doi.org/10.1007/s11263-020-01415-x

Ludermir, T. B., Yamazaki, A., & Zanchettin, C. (2006). An optimization methodology for neural network weights and architectures. *IEEE Transactions on Neural Networks, 17(6)*, 1452–1459. https://doi.org/10.1109/TNN.2006.881047

Lyu, B., Wen, S., Shi, K., & Huang, T. (2023). Multiobjective reinforcement learning-based neural architecture search for efficient portrait parsing. *IEEE Transactions on Cybernetics, 53(2)*, 1158–1169. https://doi.org/10.1109/TCYB.2021.3104866

Mercioni, M. A., Tiron, A., & Holban, S. (2019). Dynamic modification of activation function using the backpropagation algorithm in the artificial neural networks. *International Journal of Advanced Computer Science and Applications, 10(4)*.

Niu, R., Li, H., Zhang, Y., & Kang, Y. (2019). Neural architecture search based on particle swarm optimization. In *2019 3rd International Conference on Data Science and Business Analytics (ICDSBA)* (pp. 319–324). IEEE. https://doi.org/10.1109/ICDSBA48748.2019.00073

Pan, C., & Yao, X. (2021). Neural architecture search based on evolutionary algorithms with fitness approximation. *In 2021 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN52387.2021.9533986

Pedamonti, D. (2018). Comparison of non-linear activation functions for deep neural networks on MNIST classification task. *arXiv preprint arXiv:1804.02763*.

Rao, X., Xiao, S., Li, J., Wu, Q., Zhao, B., & Liu, D. (2022, December). LSBO-NAS: Latent Space Bayesian Optimization for Neural Architecture Search. In *2022 4th International Conference on Control and Robotics (ICCR)* (pp. 22-27). IEEE. https://doi.org/10.1109/ICCR55715.2022.10053904

Tsamardinos, I., Brown, L. E., & Aliferis, C. F. (2006). The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning*, *65*, 31-78.

Wolpert, D., & Macready, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation, 1(1)*, 67–82. https://doi.org/10.1109/4235.585893

Wu, B., Waschneck, B., & Mayr, C. (2022, September). Neural architecture search for low-precision neural networks. In *International Conference on Artificial Neural Networks* (pp. 743-755). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-15937-4_62

Yang, S., Ting, T., Man, K. L., & Guan, S.-U. (2013). Investigation of neural networks for function approximation. *Procedia Computer Science, 17*, 586–594.