Applied Soft Computing Journal 172 (2025) 112832

Contents lists available at ScienceDirect

Applied Soft Computing

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

A holistic approach for resource-constrained neural network architecture search

M. Lupión [a,*], N.C. Cruz [b], E.M. Ortigosa [b], P.M. Ortigosa [a]

[a] Department of Informatics, University of Almería (UAL), ceIA3, Sacramento Road, Almería, 04120, Andalucía, Spain
[b] Department of Computer Engineering, Automation and Robotics, University of Granada, Periodista Daniel Saucedo Street, Granada, 18071, Andalucía, Spain

ARTICLE INFO

Keywords:
Artificial neural networks
Neural architecture search
Meta-heuristic
TLBO
Neural network encoding
Performance predictor

ABSTRACT

The design of Artificial Neural Networks (ANN) is critical for their performance. The research field called Neural Network Search (NAS) investigates automated design strategies. This work proposes a novel NAS stack that stands out in three facets. First, the representation scheme encodes problem-specific ANN as plain vectors of numbers without needing auxiliary conversion models. Second, it is a pioneer in relying on the TLBO meta-heuristic. This optimizer supports large-scale problems and only expects two parameters, contrasting with other meta-heuristics used for NAS. Third, the stack includes a new evaluation predictor that avoids evaluating non-promising architectures. It combines several machine learning methods that train as the optimizer evaluates solutions, which avoids preliminary preparing this component and makes it self-adaptive. The proposal has been tested by using it to build a CIFAR-10 classifier while forcing the architecture to have fewer than 150,000 parameters, assuming that the resulting network must be deployed in a resource-constrained IoT device. The designs found with and without the predictor achieve validation accuracies of 78.68% and 80.65%, respectively. Both outperform a larger model from the recent literature. The predictor slightly constraints the evolution of solutions, but it approximately halves the computational effort. After extending the test to the CIFAR-100 dataset, the proposal achieves a validation accuracy of 65.43% with 478,006 parameters in its fastest configuration, competing with current results in the literature.

1. Introduction

Artificial Neural Networks (ANN) were conceived to emulate the human brain's ability to learn and solve complex tasks [1,2]. However, designing a neural network that works well for a particular problem is challenging. This process requires expertise as well as trial and error [3,4].

This section starts by giving the reader an overview of automating the design of ANN. After that, it contains a detailed literature review on this topic. Finally, it concludes by highlighting the contributions of the present work.

1.1. Design of ANN through computational optimization

Over time, different neural networks have been specially designed for specific tasks. For instance, YOLOv10 [5] and DETR [6] are prominent in object recognition. Nevertheless, even application-specific networks must be adapted to particular datasets [7]. Thus, the research field called Neural Architecture Search (NAS) [3,4] studies strategies to find optimal application-specific neural networks automatically. Automation is highly relevant because real-world problems rapidly increase in complexity and size, so human experts cannot obtain the most effective neural network designs [4]. Besides, NAS also allows controlling the size and complexity of architectures. This approach makes it possible to deploy them in low-spec devices [8–10], such as Arduino micro-controllers and Raspberry Pi micro-computers, which are popular in the emerging field of the Internet of Things (IoT) [11].

As with the fitting of biologically realistic neural models [12], it is natural to approach the design of ANN as an optimization problem [3,4]. In this context, the fundamental design levels of a NAS methodology are the following [4]: (i) a representation of candidate solutions, (ii) an objective function to assess them, (iii) constraints, and (iv) a search method or optimizer to guide the process and find the best design.

Unfortunately, the optimization problems that arise in NAS are challenging due to several factors [3,4]: First, the search space is vast and may combine continuous and discrete variables with complex non-linear constraints. Second, there are multiple options to define evaluation criteria, i.e., objective functions, which cannot be directly optimized. Third and last, evaluating solutions generally involves high

* Corresponding author.
E-mail addresses: marcoslupion@ual.es (M. Lupión), ncalvocruz@ugr.es (N.C. Cruz), ortigosa@ugr.es (E.M. Ortigosa), ortigosa@ual.es (P.M. Ortigosa).

https://doi.org/10.1016/j.asoc.2025.112832
Received 27 May 2024; Received in revised form 6 January 2025; Accepted 23 January 2025
Available online 10 February 2025

computational cost and uncertainty. For these reasons, NAS approaches usually rely on heuristic and meta-heuristic methods [3,4,13]. These optimization strategies obtain acceptable results with a reasonable computational effort for problems in which it is impossible to certify the optimality of solutions. This effectiveness not only makes meta-heuristics highly appreciated in Engineering [14,15] but also in sub-fields of machine learning and artificial intelligence, such as feature selection [16].

Natural processes usually inspire this kind of optimizer [13,15,17]. Among them, the interaction of living beings is one of the most emulated principles and results in population-based meta-heuristics. They stand out because working with multiple candidate solutions improves search space exploration and supports parallel computing [18]. Population-based meta-heuristics are also highly modifiable. Furthermore, in general, they only need to evaluate candidate solutions. Hence, they can also be classified as direct or black-box optimizers [12,19,20]. As these properties fit perfectly with NAS, population-based meta-heuristics are widespread in this field [3,4,7].

1.2. Related works

The work in [7] is a canonical example of designing a complete NAS solution relying on a population-based optimizer. The authors propose to represent the different configurations of each possible layer as an IP address in its particular range or hypothetical IP network [21]. This innovative approach includes defining a network for disabled layers, which allows the search method to work with variable-length solutions. Based on this, the authors adapt for NAS for the first time Particle Swarm Optimization (PSO) [13,17], a widespread swarm intelligence method inspired by fish schools and bird flocks. The optimizer handles constraints by fixing solutions violating them. The evaluation of candidate solutions considers the average performance on parts of the dataset for accelerating the process.

Another descriptive example with the same scope and context is the work in [22]. The authors design a solution representation that uses variable-length vectors combining structures of different types depending on the type of layer. The encoded information includes not only architecture parameters but also statistically defined weight initialization schemes. Based on this format, the authors design a genetic algorithm [13,17], a meta-heuristic based on Darwinian evolution. They adapt the operations to handle their solution representation scheme. It is critical to maintain the overall division between the convolution and pooling layers (header) and the fully connected ones (tail) while allowing combining solutions (crossover) and search mobility.

As can be seen, solution representation is arguably the cornerstone of any NAS methodology, as it determines the applicable search strategies. For instance, the network-inspired representation previously mentioned uses integer numbers, which results in a discrete search space in terms of computational optimization. However, a discrete search space is highly inefficient because it grows exponentially with the number of choices [23]. Instead, one might be interested in exploring a continuous search space, especially considering the rich set of effective meta-heuristics for this kind of search space [13]. Furthermore, depending on the approach, it can even allow opting for efficient gradient-based methods instead of plain black-box optimizers [24]. Nevertheless, since NAS generally involves dealing with classes and integer parameters [3], defining a continuous representation is not straightforward. Still, several authors have tried to address NAS through continuous optimization, which also applies to the present work.

According to [4], representation schemes can be either direct or indirect. The first type refers to encoding schemes specifying complete neural networks, including their architecture and hyperparameters. Conversely, the second class refers to those that do not encode complete architectures and need advanced decoding, even through other neural networks and non-deterministic processes. This approach aims to offer more compact representations and attenuate the scalability problems of direct ones as neural networks increase in size. There exist examples of defining continuous search spaces for NAS problems in both direct and indirect representation schemes.

Concerning direct representations, it is possible to highlight the work in [25]. The authors encode each operation as a set of weighted candidate ones. Since weights can be represented by a continuous vector, every choice results in a differentiable Softmax operation covering the different options. They also work with cells [26] instead of letting the search form architectures from scratch. Cells are predefined blocks of adequate or structurally alike layer combinations that can be stacked [27,28]. Their use simplifies the search space at the expense of reducing the exploration capabilities.

In this context, the authors apply a gradient descent method that finds and configures the operations between the cell components. Despite reducing the search space with cells and using a local search optimizer, their proposal finds competitive architectures with significantly fewer solution evaluations than derivative-free strategies. The authors emphasize this particularity by naming their method DARTS, which comes from Differentiable ARchiTecture Search. A notable variation can be found in [29], where the authors design a method called SWD-NAS focused on improving how DARTS chooses operations and adjusts cells depending on their position. They rely on defining a two-level attention system and weight normalization.

The work in [30] also deals with gradient-based optimization and using a reference architecture. However, the authors focus on making the continuous search space equivalent to the original discrete one, which is not granted by strategies such as DARTS. For this purpose, they use a graph auto-encoder that maps the discrete search space of NAS onto a continuous one. Besides, their gradient formulation includes a measure of the novelty of new configurations to escape from local optima. Finally, the authors of [31] try to harness the benefits of evolutionary optimization and gradient-based search in their GENAS method. It integrates DARTS as the local search component method of an evolutionary optimizer. The latter attenuates the reduced mobility and undesired dependencies of the DARTS's search, while DARTS still provides a fast way to improve candidate designs.

Regarding indirect representations, remarkable contributions also rely on a continuous search space. The work in [8] defines a neural network architecture optimizer based on reinforcement learning. The method takes a given design and simplifies it while maintaining its effectiveness. This proposal has two components implemented with auxiliary long short-term memory neural networks. One focuses on removing layers from the given network, and the other shrinks the remaining ones. The work in [9] has the same goal, with its objective function also considering the compression rate and the validation performance. However, the authors use Bayesian Optimization. They allow omitting connections along with removing and reducing layers, which improves exploration. The search space depends on the given network and how it can change. Specifically, the authors map the different architectures that can be achieved onto a continuous search space. The mapping function is incrementally learned using bidirectional long short-term memory neural networks. This embedding space allows defining a kernel function for the Bayesian Optimization process that prioritizes the most promising architectures.

The authors of [23] propose a NAS method that uses cells and combines three components, i.e., an encoder, a performance predictor, and a decoder. The encoder maps network architectures onto a continuous space. The performance predictor guides a gradient descent method by estimating the accuracy of candidate architectures. The decoder interprets the structure of the designs found. Encoding and decoding are implemented using long short-term memory neural networks, while the performance predictor is a multi-layer perceptron. Given a set of candidate designs, these components are jointly trained. Then, the gradient-based optimization starts from a promising design.

Regardless of solution representation schemes, the underlying problem of NAS is computationally challenging [24,32,33]. The optimizers

used must evaluate multiple candidate solutions, and this process is demanding because it involves building and evaluating many neural networks. Thus, different proposals exist to cope with this situation. As highlighted in [7,22], a simple yet powerful strategy is to train candidate networks with partial datasets and fewer iterations. This is considered enough to identify promising neural network architectures. Nevertheless, a recent trend is to develop performance predictors that estimate the quality of candidate networks based on their properties [23,24,32].

For instance, in [32], the authors study how learning curves evolve along with some properties of the networks, such as the number of layers and weights. This information allows training a set of support vector machine regressions that can predict the validation accuracy of partially trained networks. It is effective with a small training set of fully trained curves and more computationally efficient than other approaches based on Bayesian models. The work in [27] has a more ambitious goal and aims to estimate the performance of a network without even starting to train it. For this purpose, the authors design a layer encoding method coupled to a long short-term memory network to integrate the layer representations and handle them independently of the topology. A multi-layer perceptron ultimately predicts the accuracy of architectures after a given number of training epochs. Finally, the work in [28] also aims to estimate the quality of candidate neural networks without training them. However, the authors use a set of multi-layer perceptrons. It is compatible with their constructive cell-based approach and evolves through the search.

The interested reader can extend this literature review with the works in [3,4], which provide a complete overview of different NAS approaches. Besides, the recent review in [34] pays special attention to using population-based optimizers. It covers every aspect of working with ANN, from data preparation to model deployment, including the sub-fields of NAS.

1.3. Contributions

This work proposes a complete stack for NAS containing three main and decoupled components. The first is the compact representation of neural network architectures that we outlined in [33].

The proposed representation treats the space of parameters of each layer as a multi-dimensional matrix projected onto a single-dimensional one (vector), like matrices stored in computers' memory. The position in the vector linked to every layer encodes the corresponding set of parameters. This approach allows mapping the search space of the overall neural network architecture and its specific parameters onto a continuous search space with a single dimension per layer. It is also flexible enough to handle discrete and continuous parameters alongside layer removal, which is highly relevant in searching for reduced network sizes.

According to the taxonomy in [4], since the proposed representation encodes complete architectures, it can be classified as direct, like the network-inspired proposal in [7]. However, compared to other approaches, such as emulating networks [7], our method always uses a search dimension per layer and implicitly spreads the possibilities through the numerical space. It also offers significant search mobility compared to structured [22] and cell-based approaches [31]. Furthermore, in contrast to other methods for defining a continuous search space, such as [8,9,23], ours is simpler and does not need any auxiliary conversion model.

The second component is the optimization method. Our proposal relies on the Teaching-Learning-Based Optimization (TLBO), which is a widely used population-based meta-heuristic [12,35]. Although using a nature-inspired optimizer for NAS is not innovative [3], we are pioneers in applying this method for this purpose [33]. The main previous works relating TLBO and ANN were [36], which deals with defining a new optimization method, and [37,38], where TLBO only adjusts the connection weights of a given architecture.

Moreover, selecting TLBO for NAS is a founded decision. First, in contrast to other population-based methods, such as PSO and ACO, TLBO only has two parameters. Second, as a large-scale optimizer [18], it is inherently compatible with handling problems involving neural networks with multiple layers. Conversely, the performance of methods like PSO and ACO significantly decreases with the number of variables [38]. Third and last, TLBO is simple to implement and compatible with high-performance computing.

The third and last component, which is optional, is an adaptive solution-assessment strategy or evaluation predictor. It relies on an auxiliary neural network to avoid starting complete neural network architecture evaluations of non-promising solutions. This model continuously improves with the knowledge extracted from the evaluated candidate solutions.

Avoiding computationally expensive evaluations has already been considered at different scopes [23,27,32]. Previously trained components, such as those in [27,32], offer higher and more stable performance at optimization. However, preliminary training is an extra stage requiring access to data and thinking in reasonable network samples. Conversely, our predictor is simpler and self-adapted to every problem. Moreover, its lower effectiveness at the beginning, i.e., before acquiring knowledge, should affect a minimal part of the search. We share the goal in [28] but do not need to support their cell-based representation, which increases complexity.

However, like other NAS tools, the present proposal has several limitations. First, its results are limited by the data modeling context. This aspect means the proposal may not overcome poor datasets and unpredictable variables. Second, exploring numerous candidate designs in a reasonable time requires enough computational resources. Third, the proposal targets expert users to enhance their neural network design exploration capabilities. Non-expert users will have difficulties in aspects such as defining the layers to consider. Fourth and last, using meta-heuristics implies renouncing the certainty of finding optimal solutions.

The rest of the paper is structured as follows: Section 2 describes the components of our NAS proposal. Then, Section 3 explains its implementation and the experimentation carried out. Finally, Section 4 draws the conclusions and future work.

2. Methodology

This section begins with the definition of NAS as an optimization problem. After that, it describes the three components forming our proposal, i.e., the solution representation scheme, the TLBO optimization algorithm, and the evaluation predictor.

Fig. 1 provides the reader with an overview of the architecture of the proposed NAS stack. The general idea is that the chosen optimizer creates and manages a list of candidate solutions. Each is a vector encoding a neural network architecture for the target problem. The optimizer only sees the different vectors and their fitness values. The latter computation involves decoding every candidate solution. Although infeasible ones are partially evaluated quickly, evaluating feasible solutions requires building and training the corresponding neural network. Since this process is computationally demanding, the evaluation predictor tries to avoid it. The details are explained next.

2.1. Problem formulation

Conceptually, NAS focuses on finding the most appropriate neural network for a given dataset. The particular problem formulation depends on the scope and the abstraction level. According to [4], a NAS problem may cover one of three targets: (i) network topology, (ii) hyper-parameter optimization, and (iii) joint optimization.

The first, also called NAS in that work but in a more restricted sense than herein, is limited to aspects such as the number of layers, their type, and connections. Other parameters must be tuned in advance.

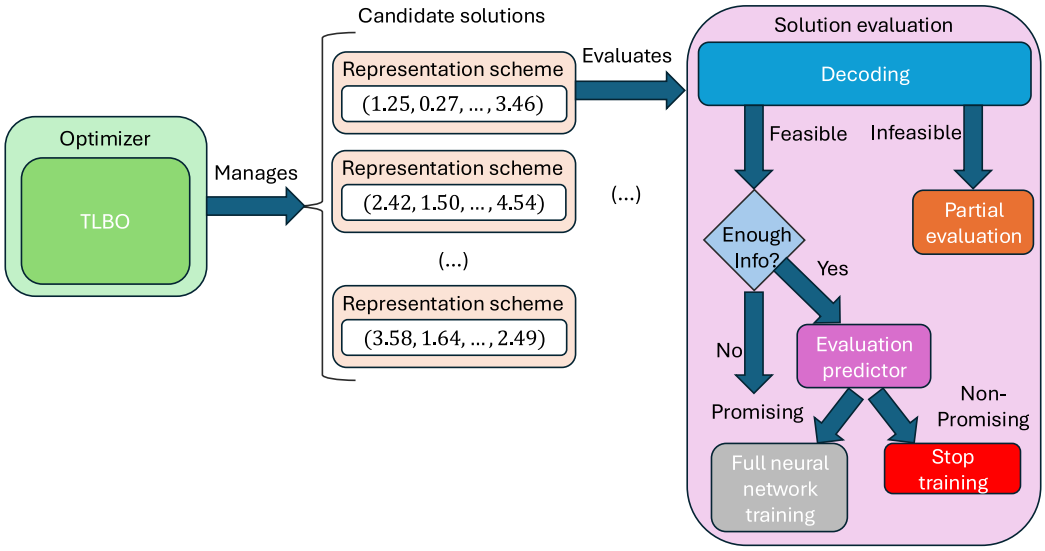


Fig. 1. Architecture of the proposed methodology.

The second, which implies having defined an architecture, refers to a wide range of parameters covering from operational (e.g., filter sizes in convolutional layers) to characteristic ones, such as learning rates and batch sizes. The third and last one considers the network topology and its parameters because separating their configuration is potentially sub-optimal. It represents the most general formulation. The interested reader can find in [4] a detailed problem statement.

The problem addressed in this work belongs to joint optimization. However, aspects such as batch sizes and training conditions are fixed depending on the application. For this reason, we opt for the formulation approach proposed in [3], which is concise and flexible due to its abstract conception. Accordingly, the target problem is defined as follows:

$$\begin{cases} \arg\min\limits_{a} & C(a, \mathcal{D}_{train}, \mathcal{D}_{fitness}) \\ \text{subject to} & a \in \mathcal{A} \end{cases} \tag{1}$$

where C is the objective or cost function in optimization terms. Given its abstract definition, this function is arbitrary. However, this formulation has some general implications.

First, the function takes a candidate network design, a, which has been trained and evaluated with datasets $\mathcal{D}_{train}$ and $\mathcal{D}_{fitness}$, respectively. While the first parameter is under optimization, the two datasets depend on the application.

Second, being a minimization problem, the objective function must measure some error metric registered after processing $\mathcal{D}_{fitness}$. Regardless, turning a minimization problem into a maximization one is straightforward, i.e., minimizing C is equivalent to maximizing $-C$.

Third and last, every candidate design or solution, a, must belong to $\mathcal{A}$, the set of feasible neural network designs. The conditions making network architectures infeasible might come from two sources. One is that some configurations can be conceptually impossible, e.g., putting a particular layer before another that expects a different input (implicit constraints). The other is that some feasible configurations can be out of the search space, for instance, when they result in too many parameters (explicit constraints).

Turning this formulation into a particular application case involves particularizing every element. As introduced, this process implies defining the following aspects [4]: (i) a candidate solution representation, (ii) an objective function, (iii) constraints, and (iv) an optimizer. The proposed NAS stack only sets the following conditions:

First, candidate solutions (neural network architectures) must be represented by vectors of user-given length, n, belonging to $\mathbb{R}^n$, and with n being the number of layers. Hence, there is a real number per layer defining all its properties. The meaning of the numerical values and their valid ranges depend on the context of the user-defined problem, i.e., the allowed layer types and their configuration. Besides, since 'disabled' may be a valid layer type, n can be considered an upper limit to the number of layers.

Fig. 2 illustrates the main idea of representing neural networks with vectors of real numbers. Notice that the flattening operation needed to connect the pooling layer with the fully connected one is implicitly added when interpreting the solution. Including this kind of consideration as part of the decoding logic avoids the necessity of including such operations as part of the search space and simplifies the problem.

Second, although the objective function is arbitrary, the chosen representation requires it to be a function of the form $\mathbb{R}^n \rightarrow \mathbb{R}$. According to Eq. (1), the smaller its value for a candidate network architecture, the better the choice. However, as previously explained, taking the opposite approach is trivial, so this is not a strict requirement. This function encapsulates the processes of building and assessing the network design. Its computation should involve interacting with a framework such as TensorFlow [39], which is the choice in this work. Regarding $\mathcal{D}_{train}$ and $\mathcal{D}_{fitness}$, these datasets are defined by the user and must be available during the process. The same applies to any contextual information and fixed parameters, such as the maximum number of layers and their type.

The solution representation scheme, the associated decoding process, and the bound checking of the optimizer avoid violating explicit constraints. However, it is necessary to handle the implicit ones, such as concatenating incompatible layers. A variation of the cost function is defined for this purpose. Let C' be this alternative function. Since evaluating such infeasible architectures does not make sense, the output of C' must be a value that worsens with the amount and degree of constraint violation. This approach is equivalent to defining a penalty function, a widespread way to handle constraints in population-based meta-heuristics [40].

Third and last, the proposed NAS stack only expects the selected optimizer to support continuous optimization problems with box constraints, i.e., bounded variables. The reason is that the solution representation hides the underlying complexity and enables us to define an objective function of the form $\mathbb{R}^n \rightarrow \mathbb{R}$. This particularity makes it possible to choose among multiple continuous optimization algorithms and is one of the most remarkable properties of the proposed framework. In this context, the underlying problem complexity makes meta-heuristic

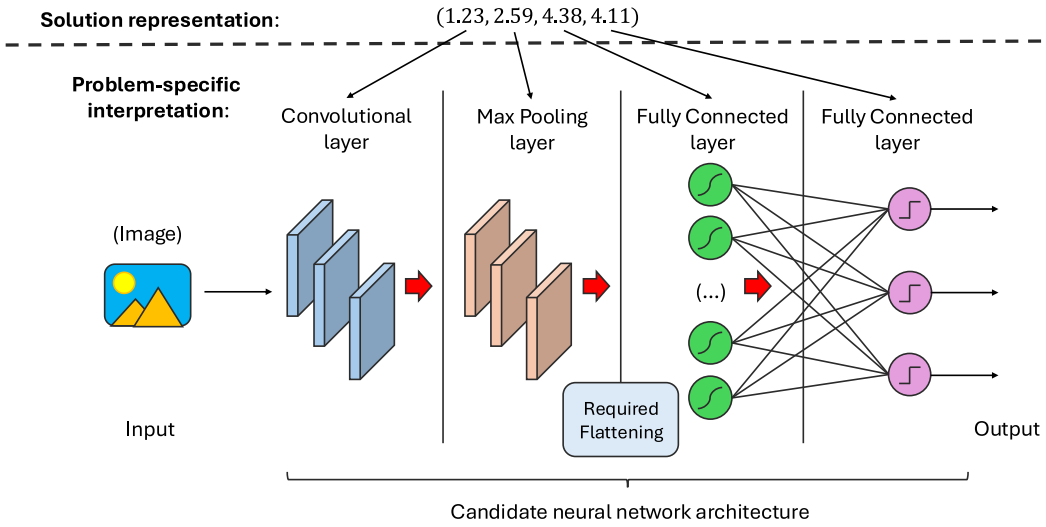


Fig. 2. Representation of a neural network as a vector of real numbers.

search methods a reasonable choice [3,4]. As introduced, the proposed NAS stack also opts for this kind of method, i.e., TLBO, described in this section.

According to the previous description, the solution representation scheme and the optimizer define the core of the proposed NAS stack. Hence, the evaluation predictor is not a requirement for Eq. (1). Nevertheless, this component is highly beneficial in practical terms due to the computational cost of the kind of objective functions considered. Regardless, it can be either considered an extra module of the optimizer or another variant of the cost function for specific candidate solutions.

2.2. Architecture representation

Every layer features two fundamental pieces of data to represent: the type of layer and its configuration. Storing this information in a single value relies on reserving the integer part for the layer type while the decimal one encodes its configuration. The user must define the types of layers to consider depending on the target problem. For instance, it seems reasonable to include bi-dimensional convolutional layers when addressing an application using images. The same occurs with one-dimensional convolutional layers and time series. It is also necessary to decide the number of layers to use. In this context, reserving a type for disabled layers, as done in [7] and previously mentioned, enables the search of architectures of different sizes. This approach gives the search an extra degree of freedom and is convenient when looking for compact designs. However, apart from deciding the layers to use and the parameters to tune, this component self-adapts and does not require further configuration.

In the proposed representation, relating any vector component with a type of layer is trivial. For example, let us assume three possible classes of layers. The integer parts 0, 1, and 2 correspond to the first, second, and third types, respectively. Nonetheless, this assignation order is arbitrary and can be changed if decoding is updated accordingly. For instance, provided three classes of layers, working in [0, 2] is the same as doing so in [6, 8]. Regardless, it is advisable to work in consecutive ranges defining contiguous regions. Otherwise, standard optimizers become incompatible.

Interpreting the decimal part of a particular component is more sophisticated. The meaning of the decimal part, always in the range [0, 1), significantly varies between different types of layers. As introduced, the strategy for representing a set of parameters using a single decimal value is inspired by how a computer stores a multi-dimensional matrix as a plain vector. This vector is as long as the product of all the dimension sizes. In this context, one can think of that vector as a segment and consider any decimal part a percentage of its total length. With 0% being one extreme and 100% being the other, scaling percentages to the range [0, 1), like decimal parts, is straightforward.

Logically, the upper bound cannot be included because it refers to the next integer. Hence, the corresponding part in the parametric space is negligibly underrepresented. Besides, since looking for the position of a scaled value might require rounding, nearby values could result in the same decoded value. Nonetheless, this aspect is irrelevant because the meta-heuristics generally considered for NAS, like the one used in this work, are derivative-free. This property makes them unaffected by micro-plateaus in the search space. That said, it is advisable to work with double precision, especially when considering layers with multiple parameters. This way, every feasible configuration of the parametric search space is reachable after being projected onto a decimal part. Otherwise, some intermediate values may become unreachable due to rounding errors.

Fig. 3 shows an example of interpreting the codification for a hypothetical layer defined by the value 4.58. From top to bottom, the decoding approach starts by processing the integer part, which indicates how to interpret the decimal one. The corresponding layer type expects two discrete parameters. They are the activation function and the number of neurons. The example assumes three activation functions to choose and a limit of four hundred neurons for the layer. In this context, representing the parametric space needs a 3×400 matrix, which is equivalent to a vector of 1200 positions. The hypothetical decimal part of 0.58 ultimately means defining a layer of 296 neurons with the second activation function (TanH).

Notice that the proposed approach is more versatile than it might seem. Decoding can consider the matrix as a look-up table containing arbitrary values. One option is storing non-consecutive numbers. It is also possible to sample continuous ranges arbitrarily. For example, the first row in Fig. 3 could contain 3, 7, and 18 for a particular layer class and three different integers for another type. Analogously, if the parameter is continuous, the previous numbers can be sampled decimal values, e.g., 6.17, 8.78, and 15.33. Nevertheless, this approach requires loading the decoder with potentially high quantities of values instead of defining them implicitly from their index.

2.3. Teaching-learning-based optimization algorithm

The Teaching-Learning-Based Optimization (TLBO) algorithm is a meta-heuristic for continuous large-scale optimization problems [18]. It simulates a class of students who learn by interacting with each

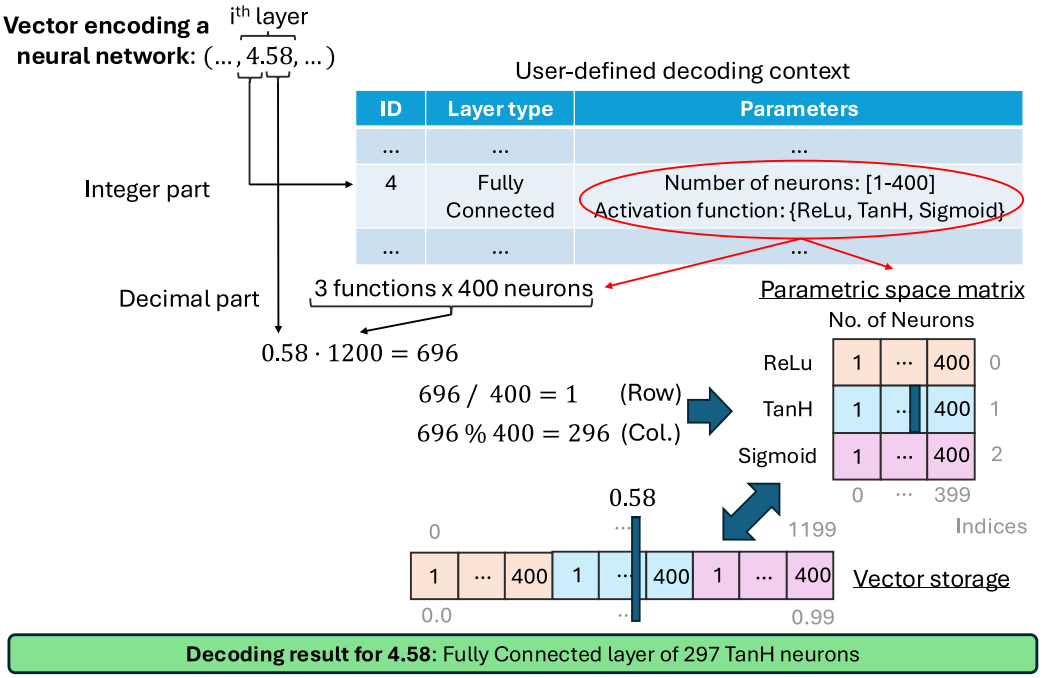


Fig. 3. Example of decoding one of the layers of a hypothetical vector encoding a neural network architecture.

other, which makes it a swarm intelligence population-based metaheuristic [12]. Among them, TLBO stands out because of its simplicity, performance, and lack of metaphor-specific parameters. These properties have motivated its successful application to numerous problems [12,35,41] and its selection for this work.

The configuration of TLBO only relies on two parameters, i.e., the population size and the number of cycles. Every global search method combines two principles [20]: exploration and exploitation. The former refers to visiting as many regions of the search space as possible. The latter means exploring every zone in depth. In this context, relating the two parameters of TLBO to these ideas arises naturally: Large populations increase the probability of finding a promising solution (exploration), and more cycles let the method improve its solutions (exploitation) [35].

The procedure of TLBO starts by randomly generating a set of individuals or candidate solutions. According to the representation scheme, each candidate solution is a vector, and every i component is a real number linked to the ith potential layer of the corresponding neural network architecture. However, not all the initial solutions need to be random. Including some prearranged architectures is a valuable way to provide this general-purpose optimizer with problem-specific knowledge [4].

Every solution must be evaluated with the defined objective function. This process implies building the neural network that it encodes, training the network with the training set, and registering its performance with the evaluation one. However, as introduced, not every architecture is feasible, and evaluating unfeasible ones is simpler and faster. Similarly, the optimizer will be linked to a component that tries to avoid demanding evaluations.

After initialization, the optimizer executes the requested number of iterations. Each consists of two consecutive stages: the teacher and the learner phases. The teacher phase tries to improve the solutions by shifting them towards the best one, treated as the teacher, T. For this purpose, TLBO calculates a vector with the mean of each component from the solutions in the current population. Then, the algorithm computes the modified (shifted) version, S', from every student, S. This computation follows Eq. (2), defined in terms of every vector component, i. The term rnd_i refers to a random real number between 0 and 1 linked to component i. The term T_F, known as the teaching factor, is a random integer that can be either 1 or 2. Both random factors are global for the current step. Finally, every altered solution, S', is evaluated. Those outperforming their original versions replace them, while the rest are discarded.

$$S'_i = S_i + rnd_i \left(T_i - T_F M_i\right) \tag{2}$$

The learner phase simulates the interaction between students as if they explained to each other. This part pairs every student, S, with a different one, W. Again, the aim is to create a variant S' from every S to replace the original if the modified one outperforms it. Nevertheless, the computation of each vector component, i, follows Eq. (3) in this stage. The term rnd_i is a random real number between 0 and 1 linked to component i. It is globally computed for the current step. This operation tries to move each individual toward the best student of the pair.

$$S'_i = \begin{cases} S_i + rnd_i \left(S_i - W_i\right) & \text{if S is better than W} \\ S_i + rnd_i \left(W_i - S_i\right) & \text{otherwise} \end{cases} \tag{3}$$

After executing all the cycles, TLBO returns the best solution in the population. Notice that the search has stochastic procedures from the beginning, i.e., the random generation of solutions. Therefore, this algorithm is a stochastic method, like most population-based metaheuristics [4]. This property means that its result can vary from one run to another.

2.4. Evaluation predictor

As introduced, evaluating candidate architectures is potentially demanding and uncertain, as training does not necessarily have a fixed duration. For instance, training the networks considered in this work can take from minutes to hours. However, our proposal tries to avoid wasting resources on designs exhibiting low potential for improvement after the first training epochs. For this purpose, we propose a machine learning model or predictor that studies the behavior of individuals for a user-given number of epochs and a reference value.

As a machine learning method, this component needs a dataset to train. However, instead of gathering it separately, we build it dynamically as the optimizer evaluates solutions. This approach makes the method self-adaptive at the expense of being initially unavailable.

During the first optimization cycles, every candidate solution is fully trained with the corresponding method (e.g., typically, the backpropagation algorithm). Then, after a user-given number of cycles to train the predictor, the optimization algorithm starts using it to discard non-promising architectures quickly. Additionally, this predictor re-trains at the end of each cycle, which improves its reliability as the execution advances.

Two main concerns affect the information gathered to form the dataset. First, provided that every candidate design will result from varying an existing solution after initialization, both the original and the current one are relevant. Second, the focus is on feasible architectures coming from feasible ancestors only. Based on these ideas, the following information is registered:

- *Original's value*: Score of the candidate solution from which the current one originates. This value will be relevant to decide if new solutions are promising enough.
- *Value*: Score registered for the current solution after being completely trained and evaluated.
- *Number of parameters*: Number of trainable parameters of the neural network defined by the current solution. This piece of data is also considered for the predictor proposed in [32].
- *Number of epochs*: Number of epochs that the training of the current neural network lasted before converging.
- *Total time*: Time that it took to train the encoded neural network.
- *Layers*: Layers used and their arrangement.
- *Validation loss per epoch*: Validation loss at the end of every epoch.

This information is processed further. First, a derived field called "Time per epoch" is calculated. It registers the average time an epoch needs to be executed and results from dividing the field "Total time" by the one "Number of epochs", being both ultimately removed. As it will be impossible to know how many epochs a network needs to converge in advance, dispensing with the latter is necessary. Otherwise, this information will not be available at inference.

Second, the validation loss at the first epoch is discarded because values at that point are not generally descriptive. Besides, the maximum loss value is set to a user-given value (e.g., 20) to avoid outliers in which the model training saturates and produces abnormally high loss values. This truncation preceding normalization avoids handling distorted ranges that bias the models. Alongside the truncation threshold, the number of epochs to consider also depends on a user-defined parameter. For example, assuming this parameter is 5, the validation losses from the second to the sixth epoch will be part of the dataset. Aside from the validation loss values, several aggregated metrics derived from them are also included. They include the maximum, the minimum, their difference, the average, and the standard deviation.

Third, the number of parameters might be several orders of magnitude higher than the rest. However, it is also less descriptive than the value of the original solution and the loss evolution for the classification. Thus, this value is normalized to be between 0 and 1, considering the maximum number of parameters allowed. It is also necessary to define the "Improve" class attribute, which registers whether the new individual outperforms its ancestor. Its value is 1 when the new candidate solution is better than the original one. Otherwise, it is 0.

Accordingly, the resulting dataset contains the following fields: "Original's value", "Number of parameters", "Time per epoch", from "Encoded Layer 0" to the maximum number of layers, from "Loss value of epoch 1" to the total number of observed epochs, "Maximum loss", "Minimum loss", "Difference loss", "Average loss", "Standard deviation loss" and "Improve". The latter is the class to predict.

Notice that the dataset is potentially unbalanced. More samples of non-improving derived solutions will be available. Thus, any machine learning model trained with it will tend to choose the predominant class. However, this behavior may cause some promising architectures to be discarded. To overcome this situation, solutions from class 0 are subsampled to have equal representatives of each class.

After preparing the dataset, the machine learning predictor is trained. For this purpose, we propose using an ensemble of algorithms. According to preliminary experimentation, this approach increases the prediction accuracy because decisions rely on several independent models. Specifically, the proposed ensemble consists of a Random Forest [42], a k-Nearest Neighbors [2], and a Gradient Boosting Classifier [43] model. The output is the most repeated class.

Based on the previous comments, one can identify two kinds of parameters. The first group, which mainly affects the definition of the dataset, consists of three generic parameters. The first is the number of optimization cycles to record samples for the dataset at the beginning. The second is the maximum loss threshold to avoid outliers. The third and last is the number of epochs that every candidate design will be trained before being classified. The configuration of the first must consider whether the total number of observations is enough for the methods included in the ensemble. Once it is, the smaller this parameter is, the sooner the predictor is available. The two remaining parameters in the first group can be tuned by observing the training behavior of some candidate architectures for the target problem.

The second group of parameters contains those required by every model selected to be part of the evaluation predictor. Therefore, they inherit the configuration guidelines linked to every chosen method.

Finally, once the predictor is ready, it is activated. Then, it alters the evaluation workflow for feasible solutions. At that point, every new candidate solution results from mutating an existing one. Therefore, the modified evaluation procedure that includes the predictor considers the solution to evaluate and its ancestor. Fig. 4 depicts this process. As shown, it starts by checking the feasibility of the original solution. If the ancestor was infeasible, the training does not end prematurely, as there is no background information. If the original architecture was feasible, the new one trains for a user-given limit of epochs. Then, if the new solution outperforms the original one, the architecture is fully trained. Otherwise, the machine learning predictor determines whether the training must continue. The decision depends on the behavior shown by the solution so far. If the training stops, the new solution is discarded in favor of the original. Conversely, when the predictor suggests continuing, the candidate design trains until converging, and its real performance is ultimately registered.

3. Experimentation and results

This section starts by describing the implementation of the proposed methodology. After that, it explains the target application. This description covers the dataset used, the layer availability, the constraints, the configuration of the optimizer, including the prearranged initial solutions, and how to assess the performance predictor. The results obtained are shown and compared to other approaches. Finally, the section concludes by addressing a more challenging dataset.

3.1. Implementation and infrastructure

The proposed NAS stack has been implemented by combining different technologies. The TLBO algorithm has been adapted from our public general-purpose parallel implementation written in the C programming language and published in [18]. The construction and evaluation of feasible solutions is performed with TensorFlow [39], which is controlled by auxiliary Python scripts.

The experimentation platform consists of three nodes from the Bull cluster owned by the Supercomputing – Algorithms Research Group of the University of Almería, Spain. The one executing the optimization algorithm and commanding the execution has 2 processors AMD EPYC Rome 7642 of 48 cores, i.e., 96 cores in total, and 512 GB of DDR4 RAM. Each of the remaining two features 2 processors AMD EPYC 7302 with 16 cores, i.e., 32 cores in total, 512 GB of DDR4 RAM, and 2 GPUs NVIDIA TESLA V100. The former launches the evaluation of feasible solutions in the latter, as GPU computing is required to complete this task in a reasonable time. The communication between nodes relies on the widespread Message-Passing Interface (MPI) [18] library.

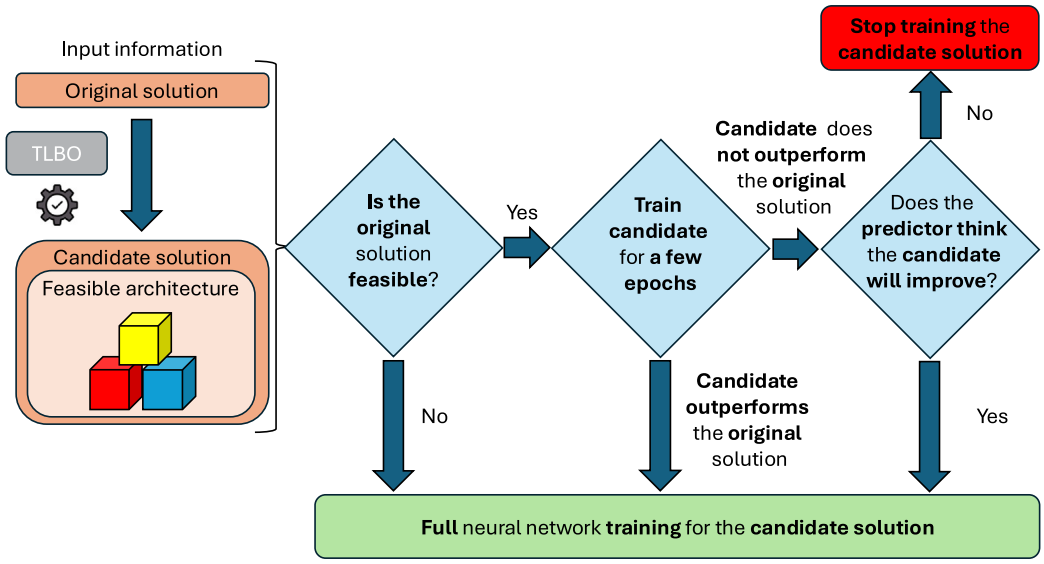


Fig. 4. Workflow of the evaluation predictor.

3.2. Problem setup

This section describes the target application used for testing purposes. It covers the dataset, the configuration of our proposal, and the metrics for the evaluation predictor.

3.2.1. Dataset

The proposed NAS framework has been used to design a neural network with the highest classification accuracy on the CIFAR-10 [44] dataset. It was created in 2019 and contains 60,000 32 × 32 RGB images. It is divided into 50,000 training and 10,000 test images. They exclusively belong to one out of ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

3.2.2. Considered layers and configurations

Although the proposed method can handle any set of layers and configurations, the user must define a working context depending on the target problem and his/her expertise. Accordingly, the NAS framework has been configured to work with the layers shown in Table 1.

First, disabled layers do not take any parameters and allow the method to work with designs of different lengths for flexibility. The user-given number of layers becomes an upper bound instead of a requisite.

Second, convolutional layers have five parameters. The first is the number of filters or kernels. The second defines their size. The third is the interleaving step in the filter application, known as the stride in this field. The fourth refers to the activation function linked to the output of filtering. The fifth is a binary value that indicates whether the weights go through a batch normalization process. In most frameworks, batch normalization would correspond to a different layer. However, having batch normalization as a separate layer increases the probability of optimizers trying to concatenate several, which can result in over-normalization and lower performance.

Third, the pooling layer can perform the "Average" or "Maximum" operation with an adjustable stride. Although these operations are usually seen separately, we incorporate them in the pooling layer. Since they share the fundamental goal, merging them allows the optimization algorithm to find the right place for the pooling layer first. After that, deciding the operation is easier.

Fourth, the dropout operation corresponds to a specific layer that only expects the number of hidden units. Dropout layers, which can follow any layer containing weights, can help the final network not to overfit.

Table 1
Types of layers and parameters for the application example.

ID	Layer type	Configurations	
		Name	Parameter ranges
0	Disabled	–	–
1	Convolutional	N. Filters	[1–512]
		Filter size	[1–8]
		Strides	[1–4]
		Activation	[ReLu, Sigmoid, Tanh, SeLU]
		Batch Norm.	[True, False]
2	Pooling	Type	[Max, Average]
		Strides	[2–4]
3	Dropout	% Hidden units	[1–90]
4	Fully connected	Neurons	[1–500]
		Activation	[ReLu, Sigmoid, Tanh, SeLU]

Finally, the fully connected layer contains the number of neurons and the activation function as parameters. The maximum number of neurons is 500. This configuration helps to avoid exceeding the maximum number of parameters set.

3.2.3. Constraints

As introduced, generating feasible neural network architectures is challenging for population-based meta-heuristics. The problem at hand deals with convolutional neural networks, and the following constraints apply:

- The first layer must be convolutional: The network begins with a convolutional layer to extract spatial information from the input image, which is essential for identifying patterns and shapes.
- Fully connected layers cannot precede convolutional or pooling ones: Convolutional and pooling layers are designed to extract spatial features, which fully connected layers use for classification or regression.
- The input dimensions for a convolutional layer must be equal to or larger than the filter dimensions: Convolutional layers require input images that are larger than the filter to perform the convolution operation effectively.
- The architecture must include at least one active layer: A neural network requires active layers to process information. Completely disabling all layers would make the network non-functional.

Table 2
Probability of selecting each type of layer depending on the position when randomly generating feasible solutions.

Layer type	Probabilities (%)	
	Initial layers	Last layers
Disabled	30	10
Convolutional	35	5
Pooling	20	5
Dropout	10	10
Fully connected	5	70

These constraints are handled by adaptively penalizing the value of infeasible solutions, which is called adaptive penalization [40]. Accordingly, the value of infeasible solutions is 1000 (a high number in this context) minus the number of layers placed adequately. This approach gives the optimizer more comparison resolution and improves convergence.

Another constraint limits the number of parameters of the neural network. As introduced, this approach allows considering the resources available to train and deploy the resulting network. For the studied problem, neural networks are limited to 150,000 parameters. If any candidate architecture is feasible yet exceeds this value, its value is 950 plus the number of parameters divided by 150,000. This way, although the optimizer penalizes both aspects, excessively big yet feasible are preferred over infeasible ones.

3.2.4. Initial population

As explained, adding some prearranged solutions facilitates and accelerates convergence. Consequently, the TLBO algorithm has been provided with some predefined neural networks.

First, the initial population contains two neural network architectures selected by the authors. One was tailored by the authors after trial and error. The other is the LeNet5-CNN architecture [45]. Second, some random yet feasible candidate solutions are included. For this purpose, we have developed an external module that builds them in advance, which resembles the proposal in [27].

Specifically, the proposed method for randomly generating feasible individuals is as follows: Provided that the first layer must be convolutional, there are two groups, the initial and the last layers. In this case, the initial layers are from the second to the seventh. For them, the most likely layer types are convolutional, disabled, and pooling. The second group contains from the eighth to the tenth layer, and the highest apparition probability corresponds to fully connected layers. Table 2 shows how likely a layer type is depending on the depth of the network.

Finally, the remaining individuals are generated completely randomly. Some feasible architectures may appear this way, but most will be infeasible.

3.2.5. Evaluation predictor metrics

As a machine learning component, precision and recall are relevant metrics for assessing the evaluation predictor's performance. Precision measures the proportion of all the model's positive classifications that are actually positive. Recall, also known as the true positive rate, reflects the proportion of all actual positives correctly classified as positives.

Eqs. (4) and (5) show the mathematical formulation of precision and recall, respectively. TP is the number of true positives, FP corresponds to the number of false positives, and FN refers to the number of false negatives. In this context, a false positive is a candidate solution that the predictor classified as improving, and it did not. A false negative is a candidate solution whose value would have improved, but the predictor discarded it. Analogously, a true positive is a solution classified as improving and meeting this expectation, while a true negative is a non-improving one correctly discarded.

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

Table 3
Results of the proposed framework with and without the evaluation predictor.

Configuration	Cost	Val. Accuracy	Cycle	Time (days)
Proposed	0.7550	74%	242	59
Proposed + Pred.	0.7381	75%	101	23

3.3. Optimization results

TLBO has been configured to work with 1000 individuals for 400 cycles. This configuration was tuned after preliminary testing, considering the comments made in Section 2.3.

We have tested two different configurations of our stack with the same seed. One of them runs without activating the evaluation predictor proposed in Section 2.4, while the other uses it. This ablation study of the only optional component of the proposal aims to confirm its effectiveness under the same circumstances.

When used, the evaluation predictor is configured as follows. The initial dataset consists of the first 5 optimization cycles. The information gathered is sufficient considering that the predictor is re-trained with new data at the end of each cycle. The maximum loss threshold is 20, and candidate designs train for 6 epochs before being evaluated. Both values were tuned by studying the training behavior of several candidate networks. Concerning the parameters required by every core model included in the predictor, the default configuration from the *scikit-learn* library has been successfully used.

Table 3 contains the results for the proposed stack with and without the evaluation predictor. As shown, the one including the predictor outperforms the other. The cost value is slightly lower (better) in the minimization problem (0.7381 compared to 0.7550), which results in higher validation accuracy for the resulting network (75% versus 74%). Using the predictor causes the best result to appear before, i.e., cycle 101 versus cycle 242.

As intended, the execution time of the proposal is also reduced to less than half when using the evaluation predictor, i.e., 23 days vs. 59 days. Specifically, this component avoided evaluating 56,220 individuals out of 111,511 (50.41%).

Fig. 5 shows the evolution of the average cost of the population for each configuration. Using the predictor results in a better, i.e., lower, average cost value in the beginning. However, dispensing with this component allows the optimizer to achieve better average costs as the population evolves. The reason is that enabling the predictor always opens the possibility of wrongly discarding promising candidate solutions. Therefore, its use comes at the expense of potentially limiting the evolution of populations.

A key point is that the predictor must minimize the number of promising candidate solutions or individuals incorrectly discarded. Fig. 6 shows how recall evolves when creating the dataset after each cycle. The higher the recall, the fewer false negatives the predictor produces. Given the individuals and their final cost, the predictor was retrained and evaluated at the end of each cycle. The recall increases rapidly during the first 15 epochs, going from 0.70 to 0.99. Then, the value stabilizes and decreases to 0.95.

A value above 0.95 means that 5% of the promising individuals are wrongly discarded. This loss of good individuals is the principal cause of the stabilization of the average cost of the population depicted in Fig. 5. Regardless, given the overall time reduction, although the predictor may overlook some promising solutions, it allows the optimizer to execute significantly more cycles. Besides, the amount

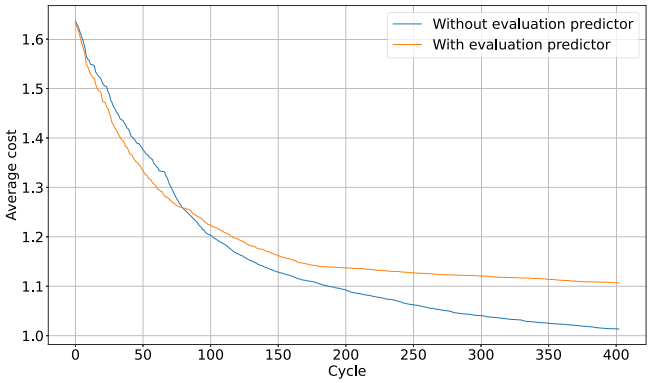


Fig. 5. Evolution of the average cost in the population using the configuration with (orange) and without (blue) the evaluation predictor.

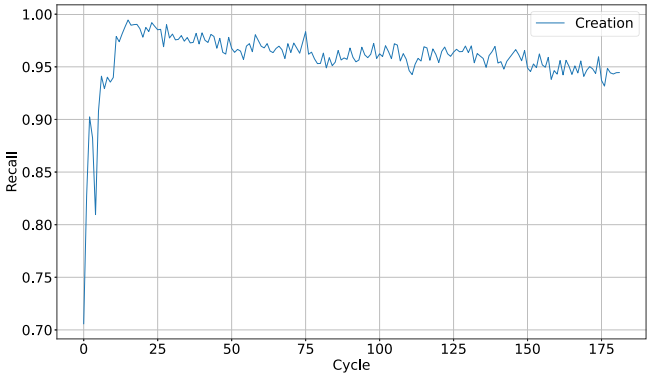


Fig. 6. Evolution of the recall in the dataset creation.

of overlooked solutions tends to decrease. As the execution advances, improving individuals becomes harder, which results in an imbalanced dataset. Thus, the model defining the predictor is prone to choosing the predominant class, i.e., non-improving solution.

These results confirm the effectiveness of including an evaluation predictor to discard non-promising solutions before assessing them. Aside from saving time and computational resources, it allows focusing on promising solutions. Although this approach may degrade the long-term performance, it increases the chances of achieving better results faster.

Finally, explainability is one of the most relevant aspects when evaluating a machine learning model [46]. This term refers to understanding why the model made a particular decision for a given input. It involves identifying the features that most affect the model's behavior.

For the proposed evaluation predictor, the variables considered were chosen intuitively. Thus, the Random Forest [42] technique, widespread for explainability purposes, has been used to study the relevance of the different features.

Fig. 7 exhibits the results of the explainability study carried out. The Y-axis contains the different features, while the X-axis represents their importance. As shown, the cost of the original solution is the most relevant feature. If its cost is low, it will be hard for the derived or candidate solution to outperform it. Thus, the predictor suggests stopping the evaluation. The opposite occurs otherwise. Then, the minimum loss, the average cost, and the loss values for the last two epochs, i.e., 4 and 5, have a high impact. Since the evolution of the loss curve is decisive for predicting whether the candidate solution can outperform the original one, this result makes sense. On the contrary, the encoded layers show a limited impact on their own.

3.4. Comparison to a state-of-the-art method

The results show that the proposed methodology progressively improves the initial solutions in either configuration. However, it is logical to wonder if these results are competitive compared to an alternative from the recent literature.

In [10], the authors develop a NAS solution to design neural networks small enough to fit into IoT devices without significantly degrading their inference performance. They build a multi-objective evolutionary algorithm based on Structured Grammatical Evolution and NSGA-II to optimize the model size and accuracy. Their experimentation limits the parameters to 500,000, while our threshold is 150,000. However, aside from the architecture, they also optimize the training parameters, such as the optimizer, early stopping criteria, and batch size.

These authors obtain a neural network architecture with over 153,000 parameters featuring a validation accuracy of 81% on the CIFAR-10 dataset. They compare their solution to SqueezeNet [47], DeepMaker [48], and NSGA-Net [49]. First, although SqueezeNet has fewer parameters, it achieves an accuracy of 80%. Besides, that network was built by hand, including specific operations instead of applying a NAS framework. Regarding DeepMaker, it obtained higher accuracy (over 85%) after being adapted with a NAS framework and consisted of fewer parameters. However, the solution used multiple blocks based on well-known architectures, requiring deep human knowledge to design them beforehand. Similarly, NSGA-Net relies on pre-defined blocks to obtain an optimal neural network architecture. It reached an accuracy of 96% but required 3.3 million parameters, which does not fit into regular IoT devices.

Our proposal obtains architectures that achieve validation accuracies of approximately 75%. However, these values directly come from their training as solutions under optimization, which includes an early-stopping condition of 5 cycles for practical reasons. This aspect means that training ends when the value does not improve after five consecutive epochs. Thus, these architectures found by our NAS method have been ultimately trained for 1000 epochs without early stopping to develop their full potential.

Unfortunately, comparing our results to the neural network designed in [10] is not straightforward. Although the authors provide the main design details of their architecture, they omit aspects such as the batch size, optimization algorithm, and learning rate. For this reason, after replicating that neural network design with the Tensorflow framework, we have trained and evaluated it with our training script to compare that solution to ours. It will be referred to as the reference solution in what follows.

Table 4 shows how the reference solution and our two networks (generated with and without the evaluation predictor) have evolved after training. The "Num. Params" column contains the number of parameters of each architecture. The "Val. Accuracy" column shows the validation accuracy that each neural network reached. Finally, the "Epoch" column contains the epoch at which the network reached its performance peak.

As can be seen, all the architectures have between 140,000 and 153,000 parameters. These values are compatible with their use in IoT devices. The most accurate architecture is the one obtained by our method without using the evaluation predictor (accuracy of 80.65%). It is followed by the one also designed with our proposal and the predictor (accuracy of 78.68%). The third one in terms of accuracy is the reference architecture (accuracy of 78.49%). The latter is also the fastest to reach its peak, getting to it after 174 epochs.

As shown, there is room for improving the solutions found automatically. For instance, the accuracy of our best design raised from 74% to 80.65% without early stopping. This uncertainty during the search is unavoidable for practical reasons, but users should understand this situation and try to refine the promising designs found.

Furthermore, it is also remarkable that our best design outperforms the designs by [10] and SqueezeNet, created by human experts by

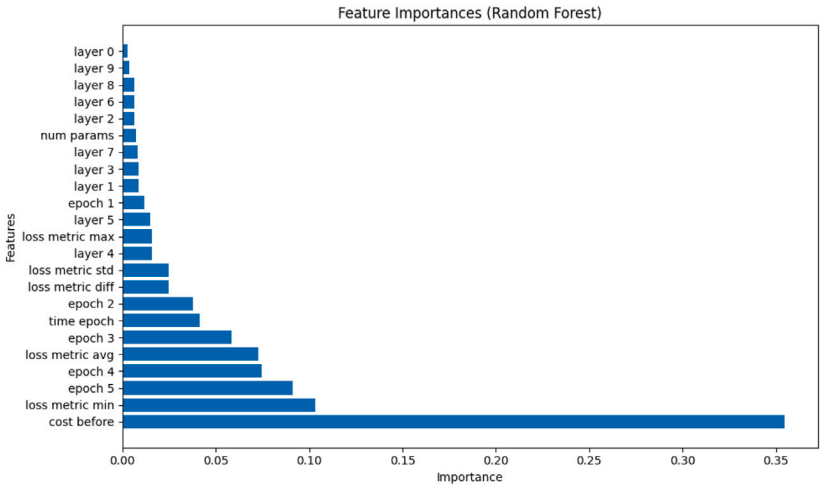


Fig. 7. Importance of the different attributes of the evaluation predictor.

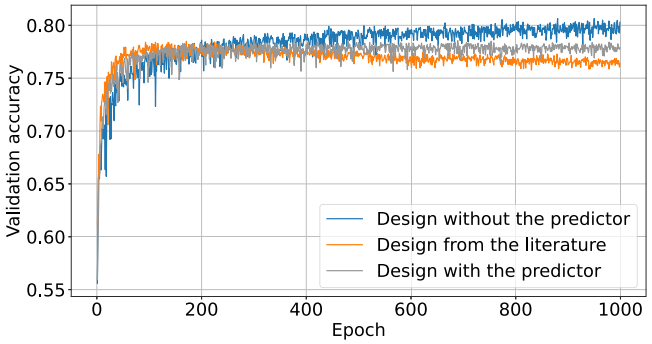


Fig. 8. Long-term evolution of the complete training of the three compared neural networks.

Table 4
Comparison between the networks found by our proposal and the one in [10].

Configuration	Num. Params.	Val. Accuracy	Epoch
Proposed without predictor	147,240	**80.65%**	953
Proposed with predictor	**140,144**	78.68%	300
Design in [10]	153,290	78.49%	**174**

hand. While our best architecture obtained a validation accuracy of 80.65%, the proposals in [10] and SqueezeNet reached 78.49% and 80%, respectively.

Fig. 8 shows the evolution of the validation accuracy of the three networks compared in Table 4 during a complete training of 1000 epochs. The reference, i.e., the one designed in [10], is represented in orange. Our proposal without using the predictor is in blue, and the one with it is in gray.

The reference network performs better during the first epochs, but its accuracy decreases as training lasts. It only contains two dropout layers and six convolutional ones, so it cannot handle overfitting adequately. This situation might have been caused by defining a fixed time limit for evaluation when optimizing the design [10]. As for our architectures, the one found without the predictor shows a continuous trend of improvement without overfitting and reaches the highest accuracy. It outperforms the design achieved with the predictor, which stays between the other two.

At this point, it is interesting to see the neural network designs that our proposal found for the two configurations considered. Figs. 9(a) and 9(b) show the architectures found without and with the evaluation predictor, respectively.

The neural network designed without the predictor consists of 7 layers (obviating the "Batch Normalization" and "Flatten" ones). It starts with three "Convolutional" layers, with the first and third followed by a "Batch Normalization" one. Before flattening, there is an "Average Pooling" layer followed by a "Dropout" one. It ends with a "Fully Connected" layer.

The architecture obtained using the performance predictor slightly varies. It contains only two "Convolutional" layers (including a "Batch Normalization" layer in the first one). However, it incorporates two "Average Pooling" layers with a "Dropout" in between. After flattening, there are two "Fully Connected" layers with fewer neurons than before, a "Dropout" layer and another "Fully Connected" one.

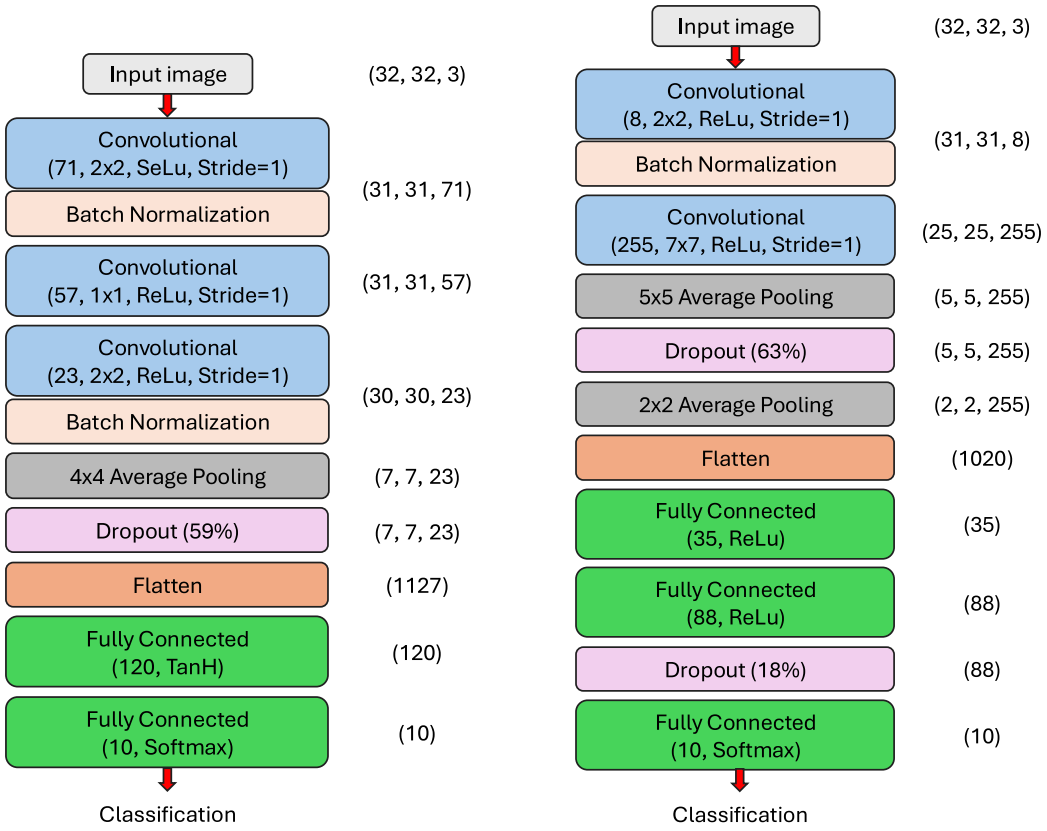
3.5. Test with the CIFAR-100 dataset

Given the positive results obtained with the CIFAR-10 dataset, the CIFAR-100 [44] has been selected for an extended analysis due to its increased complexity. While both contain the same number of images, CIFAR-100 includes 100 distinct classes, each with significantly fewer examples per class. This characteristic makes CIFAR-100 a more challenging benchmark, especially for methods aiming to design compact neural network architectures with limited computational resources. Hence, this experiment demonstrates the feasibility of applying the proposal to datasets featuring more variability and complexity.

For the test, we employed the configuration excluding the evaluation predictor. This decision is based on prior results showing that the design found in this way reached better validation accuracy in the long term. The experimental setup maintained the same execution environment, configuration, and constraints as in the CIFAR-10 evaluation. Like in the previous case, we included different feasible solutions in the population to accelerate convergence. However, considering the increased problem difficulty, the maximum number of layers and parameters were set to 12 and 500,000, respectively.

After 26 days of execution and 420 cycles, we took the best candidate design in the population. Fig. 10 shows the obtained neural network. After training it for an extended period, like in the previous case, and applying data augmentation to that process, the architecture achieved a remarkable accuracy of 65.43% on CIFAR-100 while adhering to the tight parameter limit. Conversely, our best hand-tuned design within this size limit only reached an accuracy of 59.64% in the same enhanced training context.

Apart from comparing the hand-tuned result with the one found using our proposal, the lack of well-established benchmarks for efficient neural networks on CIFAR-100 complicates direct comparisons. In the realm of small architectures with less than several million parameters,



(a) Without the evaluation predictor. (b) With the evaluation predictor.

Fig. 9. Neural networks found for the CIFAR-10 dataset with the proposal.

Table 5
Comparison of our method with related works on CIFAR-100 under parameter constraints.

Method	CIFAR-100 Acc.	# Parameters	NAS
Grouped Pointwise Conv. [50]	71.36%	520k	No
ThriftyNet [51]	74.37%	600k	No
Wide-ResNet [52]	69.11%	600k	No
Our method	**65.43%**	**478k**	**Yes**

CIFAR-100 is not widely studied. Nevertheless, a few relevant works exist, providing useful reference points for comparison.

Table 5 compares our result and several state-of-the-art approaches designed for compact neural network architectures on the CIFAR-100 dataset. All of them propose specific strategies for obtaining tiny CNN architectures. As shown, these methods achieve higher accuracy values, i.e., 71.36% for [50], 74.37% for [51], and 69.11% for [52], while ours reaches 65.43%. However, all have been hand-tuned and consist of significantly more parameters, ranging from 520,000 to 600,000. In contrast to them, our design relies on NAS, outperforms our manual design process, and features less than 500,000 parameters.

Accordingly, our proposal stands out for adhering to the 500,000-parameter limit, offering a compact and resource-efficient solution for scenarios like IoT devices. Unlike the other approaches, which heavily depend on manual architecture design and focus on convolutional networks, our method explores the given design space autonomously once launched.

4. Conclusions

This work has proposed and studied a complete stack of methods for designing neural networks automatically. It supports any layer type, candidate architectures of different lengths, and constraints on design sizes for low-specification computing environments. This proposal combines three components, i.e., an original solution representation scheme, a standard continuous optimization algorithm, and a new performance predictor.

The representation scheme encodes neural network architectures as vectors of real numbers, which allows neural network design to be approached as a continuous optimization problem. The chosen optimizer is Teaching-Learning-based Optimization (TLBO), a population-based meta-heuristic. We are pioneers in opting for this method in this context. This choice is based on its minimalist set of parameters, i.e., population size and cycles, and its compatibility with large-scale problems. The performance predictor is a self-adaptive machine learning ensemble that avoids evaluating non-promising candidate architectures.

The effectiveness of this proposal has been studied by building an image classifier using the CIFAR-10 dataset. The method has been allowed to work with five different layer types. Besides, the designed neural networks have been limited to contain up to 150,000 parameters. This aspect shows the ability of our proposal to work with this kind of constraint and its compatibility with designing neural networks to run in low-specification computing platforms, such as IoT devices.

In this context, the experimentation first focused on studying the performance predictor, as it has a critical role in obtaining neural network designs in a reasonable time. According to the results, this component avoids evaluating multiple candidate solutions and reduces the run time to less than half. It also quickly centers the attention on the most promising solutions. However, its usage comes at a price: the average value of the population stagnates. Accordingly, without any time constraint, it seems advisable not to activate the evaluation predictor to avoid premature convergence.

In the sample application with the CIFAR-10 dataset, our method takes two months without the evaluation predictor and less than one

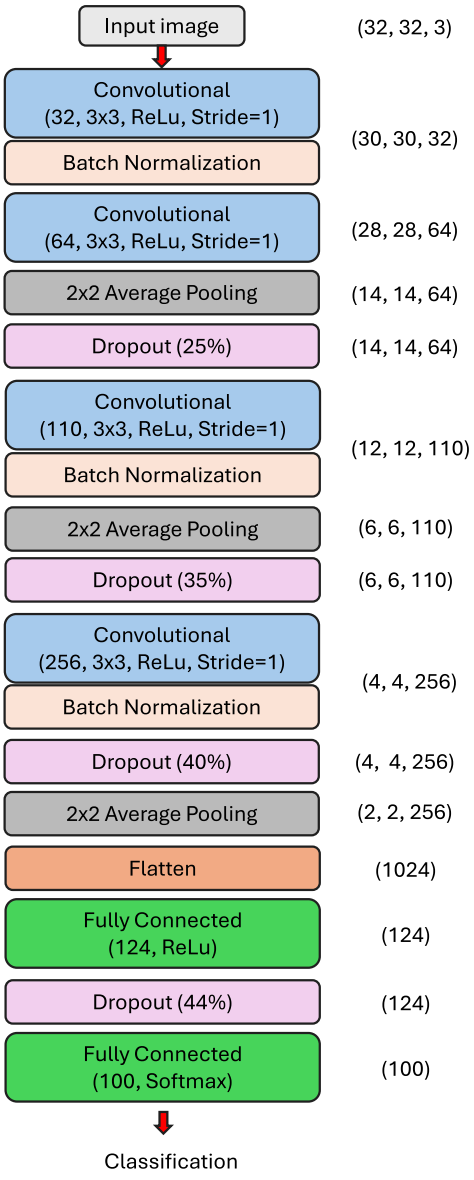


Fig. 10. Best architecture found for the CIFAR-100 dataset.

with this component. However, it finds neural networks outperforming a state-of-the-art solution in both configurations. Specifically, the reference design consists of 153,290 parameters and achieves an accuracy of 78.49%. The one found by our proposal when using the performance predictor only needs 140,144 parameters and reaches an accuracy of 78.68%. Without this component, the resulting architecture has 147,240 parameters, still within the specifications, and the accuracy is higher, i.e., 80.65%. The only magnitude in which the reference design remains a better option is the number of training epochs taken to converge (174 versus 300 for our fastest configuration). Therefore, our proposal is competitive in problem terms as long as the time window for designing the desired network is relatively flexible. This aspect is reasonable considering that, in general, neural networks are designed once and run while the resulting application is at exploitation.

Furthermore, we have also tried the proposal with a more challenging problem, i.e., the CIFAR-100 dataset and limiting the designs to 500,000 parameters. In the literature, there exist relatively small neural networks with accuracy values ranging from 69 to 74% for this problem. However, all have been designed by the direct intervention of human experts and consist of more parameters, from 520,000 to 600,000. Conversely, our proposal found a neural network featuring an accuracy of 65.43% while sticking to a strict limit of 500,000 parameters, and with the best initial solution provided reaching only 59.64% under the same training conditions. Thus, the proposal seems interesting for dealing with challenging problems under strict resource limitations.

In future works, the proposed neural architecture search solution will be tested with different applications. Besides, given the modularity of our stack and the convenience of wrapping neural architecture search as a continuous optimization problem, we will study different optimization techniques. Finally, we will try to improve the performance predictor.

CRediT authorship contribution statement

M. Lupión: Writing – original draft, Validation, Software, Methodology, Conceptualization. **N.C. Cruz:** Writing – review & editing, Software, Methodology, Formal analysis, Conceptualization. **E.M. Ortigosa:** Writing – review & editing, Software, Methodology. **P.M. Ortigosa:** Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been funded by the projects ECSEL Joint Undertaking in IMOCO4.E (EU H2020 RIA-101007311) (PCI2021-1219 257) from MCIN/AEI/10.13039/501100011033 and EU NextGenerationEU/PRTR; R+D+i PID2021-123278OB-I00 and PDC2022-133370-I00 from MCI-N/AEI/10.13039/501100 011033 and ERDF funds; and the Department of Informatics of the University of Almería. M. Lupión is a fellowship of the FPU program from the Spanish Ministry of Education (FPU19/02756). N.C. Cruz is supported by the Ministry of Economic Transformation, Industry, Knowledge and Universities from the Andalusian government (PAIDI 2021: POSTDOC_21_00124). The authors would also like to thank J.J. Moreno from the University of Almería for his technical support with the computing platform.

Data availability

Data will be made available on request.

References

[1] O.I. Abiodun, A. Jantan, A.E. Omolara, K.V. Dada, N.A. Mohamed, H. Arshad, State-of-the-art in artificial neural network applications: A survey, Heliyon 4 (11) (2018) http://dx.doi.org/10.1016/j.heliyon.2018.e00938.
[2] W. Ertel, Introduction to Artificial Intelligence, Springer Nature, 2024.
[3] Y. Liu, Y. Sun, B. Xue, M. Zhang, G.G. Yen, K.C. Tan, A survey on evolutionary neural architecture search, IEEE Trans. Neural Netw. Learn. Syst. 34 (2) (2021) 550–570, http://dx.doi.org/10.1109/TNNLS.2021.3100554.
[4] E. Talbi, Automated design of deep neural networks: A survey and unified taxonomy, ACM Comput. Surv. 54 (2) (2021) 1–37, http://dx.doi.org/10.1145/3439730.
[5] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, G. Ding, YOLOv10: Real-time end-to-end object detection, 2024, http://dx.doi.org/10.48550/arXiv.2405.14458, arXiv:2405.14458. URL https://arxiv.org/abs/2405.14458.
[6] Z. Zong, G. Song, Y. Liu, DETRs with collaborative hybrid assignments training, 2023, arXiv:2211.12860. URL https://arxiv.org/abs/2211.12860.
[7] B. Wang, Y. Sun, B. Xue, M. Zhang, Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification, in: 2018 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2018, pp. 1–8, http://dx.doi.org/10.1109/CEC.2018.8477735.

[8] A. Ashok, N. Rhinehart, F. Beainy, K.M. Kitani, N2N learning: Network to network compression via policy gradient reinforcement learning, 2017, http://dx.doi.org/10.48550/arXiv.1709.06030, arXiv preprint arXiv:1709.06030.

[9] S. Cao, X. Wang, K.M. Kitani, Learnable embedding space for efficient neural architecture compression, 2019, http://dx.doi.org/10.48550/arXiv.1902.00383, arXiv preprint arXiv:1902.00383.

[10] I. Cardoso-Pereira, G. Lobo-Pappa, H.S. Ramos, Neural architecture search for resource-constrained internet of things devices, in: 2021 IEEE Symposium on Computers and Communications, ISCC, IEEE, 2021, pp. 1–6, http://dx.doi.org/10.1109/ISCC53001.2021.9631535.

[11] F. Oliveira, D.G. Costa, F. Assis, I. Silva, Internet of intelligent things: A convergence of embedded systems, edge computing and machine learning, Internet Things (2024) 101153, http://dx.doi.org/10.1016/j.iot.2024.101153.

[12] N.C. Cruz, M. Marín, J.L. Redondo, E.M. Ortigosa, P.M. Ortigosa, A comparative study of stochastic optimizers for fitting neuron models. application to the cerebellar granule cell, Inf. 32 (3) (2021) 477–498, http://dx.doi.org/10.15388/21-INFOR450.

[13] S. Salhi, Heuristic Search: The Emerging Science of Problem Solving, Springer, 2017.

[14] F.S. Gharehchopogh, M.H. Nadimi-Shahraki, S. Barshandeh, B. Abdollahzadeh, H. Zamani, CQFFA: A chaotic quasi-oppositional farmland fertility algorithm for solving engineering optimization problems, J. Bionic Eng. 20 (1) (2023) 158–183, http://dx.doi.org/10.1007/s42235-022-00255-4.

[15] F.S. Gharehchopogh, S. Ghafouri, M. Namazi, B. Arasteh, Advances in manta ray foraging optimization: A comprehensive survey, J. Bionic Eng. 21 (2) (2024) 953–990, http://dx.doi.org/10.1007/s42235-024-00481-y.

[16] A. Hosseinalipour, R. Ghanbarzadeh, B. Arasteh, F.S. Gharehchopogh, S. Mirjalili, A metaheuristic approach based on coronavirus herd immunity optimiser for breast cancer diagnosis, Clust. Comput. (2024) 1–25, http://dx.doi.org/10.1007/s10586-024-04360-3.

[17] G. Lindfield, J. Penny, Introduction to Nature-Inspired Optimization, Academic Press, 2017.

[18] N.C. Cruz, J.L. Redondo, J.D. Álvarez, M. Berenguel, P.M. Ortigosa, A parallel teaching–learning-based optimization procedure for automatic heliostat aiming, J. Supercomput. 73 (1) (2017) 591–606, http://dx.doi.org/10.1007/s11227-016-1914-5.

[19] N.C. Cruz, J.L. Redondo, E.M. Ortigosa, P.M. Ortigosa, On the design of a new stochastic meta-heuristic for derivative-free optimization, in: International Conference on Computational Science and Its Applications, ICCSA, Springer, 2022, pp. 188–200, http://dx.doi.org/10.1007/978-3-031-10562-3_14.

[20] D.R. Jones, J.R.R.A. Martins, The DIRECT algorithm: 25 years later, J. Global Optim. 79 (3) (2021) 521–566, http://dx.doi.org/10.1007/s10898-020-00952-6.

[21] A. Elahi, A. Cushman, Computer Networks: Data Communications, Internet and Security, Springer Nature, 2023.

[22] Y. Sun, B. Xue, M. Zhang, G.G. Yen, Evolving deep convolutional neural networks for image classification, IEEE Trans. Evol. Comput. 24 (2) (2020) 394–407, http://dx.doi.org/10.1109/TEVC.2019.2916183.

[23] R. Luo, F. Tian, T. Qin, E. Chen, T.Y. Liu, Neural architecture optimization, Adv. Neural Inf. Process. Syst. 31 (2018) URL https://proceedings.neurips.cc/paper_files/paper/2018/file/933670f1ac8ba969f32989c312faba75-Paper.pdf.

[24] P. Ye, B. Li, Y. Li, T. Chen, J. Fan, W. Ouyang, β-DARTS: Beta-decay regularization for differentiable architecture search, 2022, arXiv:2203.01665. URL https://arxiv.org/abs/2203.01665.

[25] H. Liu, K. Simonyan, Y. Yang, DARTS: Differentiable architecture search, 2018, http://dx.doi.org/10.48550/arXiv.1806.09055, arXiv preprint arXiv:1806.09055.

[26] B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710, http://dx.doi.org/10.48550/arXiv.1707.07012.

[27] B. Deng, J. Yan, D. Lin, Peephole: Predicting network performance before training, 2017, http://dx.doi.org/10.48550/arXiv.1712.03351, arXiv preprint arXiv:1712.03351.

[28] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.J. Li, L. Fei-Fei, A. Yuille, J. Huang, K. Murphy, Progressive neural architecture search, in: Proceedings of the European Conference on Computer Vision, ECCV, 2018, pp. 19–34, http://dx.doi.org/10.48550/arXiv.1712.00559.

[29] Y. Xue, X. Han, Z. Wang, Self-adaptive weight based on dual-attention for differentiable neural architecture search, IEEE Trans. Ind. Inf. (2024) http://dx.doi.org/10.1109/TII.2023.3348843.

[30] M. Zhang, H. Li, S. Pan, X. Chang, Z. Ge, S. Su, Differentiable neural architecture search in equivalent space with exploration enhancement, Adv. Neural Inf. Process. Syst. 33 (2020) 13341–13351, URL https://proceedings.neurips.cc/paper_files/paper/2020/file/9a96a2c73c0d477ff2a6da3bf538f4f4-Paper.pdf.

[31] Y. Xue, X. Han, F. Neri, J. Qin, D. Pelusi, A gradient-guided evolutionary neural architecture search, IEEE Trans. Neural Netw. Learn. Syst. (2024) http://dx.doi.org/10.1109/TNNLS.2024.3371432.

[32] B. Baker, O. Gupta, R. Raskar, N. Naik, Accelerating neural architecture search using performance prediction, 2017, http://dx.doi.org/10.48550/arXiv.1705.10823, arXiv preprint arXiv:1705.10823.

[33] M. Lupión, N.C. Cruz, J.F. Sanjuan, B. Paechter, P.M. Ortigosa, Accelerating neural network architecture search using multi-GPU high-performance computing, J. Supercomput. (2022) 1–17, http://dx.doi.org/10.1007/s11227-022-04960-z.

[34] N. Li, L. Ma, G. Yu, B. Xue, M. Zhang, Y. Jin, Survey on evolutionary deep learning: Principles, algorithms, applications, and open issues, ACM Comput. Surv. 56 (2) (2023) 1–34, http://dx.doi.org/10.1145/3603704.

[35] J.L. Torres-Moreno, N.C. Cruz, J.D. Álvarez, J.L. Redondo, A. Giménez-Fernandez, An open-source tool for path synthesis of four-bar mechanisms, Mech. Mach. Theory 169 (2022) 104604, http://dx.doi.org/10.1016/j.mechmachtheory.2021.104604.

[36] Y. Zhang, Z. Jin, Y. Chen, Hybrid teaching–learning-based optimization and neural network algorithm for engineering design optimization problems, Knowl.-Based Syst. 187 (2020) 104836, http://dx.doi.org/10.1016/j.knosys.2019.07.007.

[37] M. Kankal, E. Uzlu, Neural network approach with teaching–learning-based optimization for modeling and forecasting long-term electric energy demand in Turkey, Neural Comput. Appl. 28 (1) (2017) 737–747, http://dx.doi.org/10.1007/s00521-016-2409-2.

[38] Z. Yang, K. Li, Y. Guo, H. Ma, M. Zheng, Compact real-valued teaching-learning-based optimization with the applications to neural network training, Knowl.-Based Syst. 159 (2018) 51–62, http://dx.doi.org/10.1016/j.knosys.2018.06.004.

[39] G. Zaccone, M.R. Karim, A. Menshawy, Deep Learning with Tensorflow, Packt Publishing Ltd, 2017.

[40] Z.H. Zhan, L. Shi, K.C. Tan, J. Zhang, A survey on evolutionary computation for complex continuous optimization, Artif. Intell. Rev. 55 (1) (2022) 59–110.

[41] R. Xue, Z. Wu, A survey of application and classification on teaching-learning-based optimization algorithm, IEEE Access 8 (2019) 1062–1079, http://dx.doi.org/10.1109/ACCESS.2019.2960388.

[42] R. Genuer, J.M. Poggi, Random Forests, Springer, 2020.

[43] C. Bentéjac, A. Csörgő, G. Martínez-Muñoz, A comparative analysis of gradient boosting algorithms, Artif. Intell. Rev. 54 (2021) 1937–1967, http://dx.doi.org/10.1007/s10462-020-09896-5.

[44] A. Krizhevsky, Learning Multiple Layers of Features from Tiny Images, Tech. Rep., University of Toronto, 2009.

[45] M. Cho, Y. Kim, Implementation of data-optimized FPGA-based accelerator for convolutional neural network, in: 2020 International Conference on Electronics, Information, and Communication, ICEIC, IEEE, 2020, pp. 1–2, http://dx.doi.org/10.1109/ICEIC49074.2020.9050993.

[46] A.B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins, Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI, Inf. Fusion 58 (2020) 82–115, http://dx.doi.org/10.1016/j.inffus.2019.12.012.

[47] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size, 2016, http://dx.doi.org/10.48550/arXiv.1602.07360, arXiv preprint arXiv:1602.07360.

[48] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, M. Sjödin, DeepMaker: A multi-objective optimization framework for deep neural networks in embedded systems, Microprocess. Microsyst. 73 (102989) (2020) http://dx.doi.org/10.1016/j.micpro.2020.102989.

[49] Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, W. Banzhaf, NSGA-net: neural architecture search using multi-objective genetic algorithm, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 419–427, http://dx.doi.org/10.1145/3321707.3321729.

[50] S. Schuler, J. Paulo, S. Romani, M. Abdel-nasser, H. Rashwan, D. Puig, Grouped pointwise convolutions reduce parameters in convolutional neural networks, Mendel 28 (2022) 23–31, http://dx.doi.org/10.13164/mendel.2022.1.023.

[51] G. Coiffier, G.H. Boukli, V. Gripon, ThriftyNets: convolutional neural networks with tiny parameter budget, IoT 2 (2) (2021) 222–235, http://dx.doi.org/10.3390/iot2020012.

[52] S. Zagoruyko, N. Komodakis, Wide residual networks, 2016, arXiv abs/1605.07146. URL https://api.semanticscholar.org/CorpusID:15276198.