Accelerating neural network architecture search using multi-GPU high-performance computing

Marcos Lupión[1*], N.C. Cruz[2], Juan F. Sanjuan[1], B. Paechter[3] and Pilar M. Ortigosa[1]

[1]Department of Informatics, University of Almería, CEiA3, Sacramento Road, Almería, 04120, Spain.
[2]Department of Computer Engineering, Automation and Robotics, University of Granada, Journalist Daniel Saucedo Aranda Street, Granada, 18071, Spain.
[3]School of Computing, Edinburgh Napier University, 10 Colinton Rd, EH10 5DT, Edinburgh, Scotland, UK.

*Corresponding author(s). E-mail(s): marcoslupion@ual.es;
Contributing authors: ncalvocruz@ugr.es; jsanjuan@ual.es;
B.Paechter@napier.ac.uk; ortigosa@ual.es;

Abstract

Neural networks stand out from Artificial Intelligence because they can complete challenging tasks, such as image classification. However, designing a neural network for a particular problem requires experience and tedious trial and error. Automating this process defines a research field usually relying on population-based meta-heuristics. This kind of optimizer generally needs numerous function evaluations, which are computationally demanding in this context as they involve building, training, and evaluating different neural networks. Fortunately, these algorithms are also well suited for parallel computing. This work describes how the Teaching-Learning-based Optimization (TLBO) algorithm has been adapted for designing neural networks exploiting a multi-GPU high-performance computing environment. The optimizer, not applied before for this purpose up to the authors' knowledge, has been selected because it lacks specific parameters and is compatible with large-scale optimization. Thus, its configuration does not result in another problem and could design architectures with many layers. The

parallelization scheme is decoupled from the optimizer. It can be seen as an external evaluation service managing multiple GPUs for promising neural network designs, even at different machines, and multiple CPU's for low-performing solutions. This strategy has been tested in designing a neural network for image classification based on the CIFAR-10 dataset. The architectures found outperform human designs, and the sequential process is accelerated 4.2 times with 4 GPUs and 96 cores thanks to parallelization, being the ideal speed up 4.39 in this case.

Keywords: Artificial neural networks, Neural network design, HPC, TLBO, Multi-GPU

1 Introduction

Neural networks simulate the behavior of the human brain to perform different tasks, such as image classification, object recognition, language processing, and anomaly detection [1]. However, designing a neural network for a particular problem is not trivial. It is necessary to select the type of network and its configuration or architecture, which requires experience and implicitly needs trial and error.

Throughout the years, there have been neural networks designed for specific problems. For instance, Yolo [2] aims at object recognition, while ResNet [3] and VGG [4] have been conceived for image processing. Regardless, prearranged designs do not always adapt well to datasets. Thus, finding the most appropriate architecture of the neural network used for a particular application remains a problem-specific task. This situation led to the definition of an active research field known as Neural Architecture Search (NAS) [5]. It studies the automation of finding the optimal neural network architecture for a problem and the associated dataset. Google was a pioneer in designing algorithms for this purpose [6].

In this context, population-based algorithms are widely used for addressing the underlying optimization problem [5, 7]. As these methods are metaheuristic and simultaneously work with multiple solutions, the computational demand is generally high, and the results might be sub-optimal. Nevertheless, their exploration capabilities significantly outperform human experts, whose search process is tedious and potentially biased. Besides, the benefits of such methods increase with the problem size because the possible configurations to consider rapidly become immeasurable for human beings.

Population-based algorithms work with vast sets of candidate solutions that interact with each other to improve their quality. For the problem at hand, each solution represents a neural network architecture, and its associated fitness indicates its goodness after being evaluated. Some of the population-based

optimizers considered for this purpose so far are Particle-Swarm Optimization (PSO) [8] and Ant-Colony Optimization (ACO) [9]. They have been demonstrated to find designs comparable to those obtained by human experts.

This work proposes using Teaching-Learning-based Optimization (TLBO) to optimize neural network architectures. This algorithm is also a population-based meta-heuristic proposed in [10] and has gained popularity. This choice, innovative in this field up to the authors' knowledge, is based on two fundamental aspects with respect to the population-based optimizers previously mentioned. Firstly, TLBO was designed for large-scale optimization problems, i.e., with many variables. This aspect would allow its use for designing complex neural networks, as required for most real applications. In contrast, the performance of methods such as PSO and ACO decreases with the number of variables [11]. Secondly, TLBO only needs as input the number of cycles to run and the population size, while most population-based optimizers, such as PSO and ACO, must be carefully configured according to their working metaphor.

In general, meta-heuristic optimizers need to evaluate numerous candidate solutions (individuals) through their search, and this assessment uses what is called the 'cost' or 'objective' function. For the problem at hand, evaluating a solution involves building, training, and assessing the neural network architecture that it represents. The training is done using the widely-used back-propagation algorithm. It is responsible for tuning the weights of the neural network connections to minimize the error that the network commits according to the associated gradient vector. Some years ago, implementing this method was not feasible, especially for big networks, due to hardware limitations. However, the evolution of Graphical Processing Units (GPUs) as general-purpose accelerators has significantly attenuated this problem. Current GPUs are even optimized for this kind of task.

In some Internet of Things (IoT) systems, the datasets constantly change due to the large amount of data generated per second. Therefore, the performance of models decreases over the time if they are not updated with the new data [12]. Having neural networks adapted to the datasets requires finding their most appropriate architecture as fast as possible. Therefore, it is critical to parallelize neural network architecture search algorithms to accelerate their execution and benefit from the computing platforms used.

Fortunately, population-based meta-heuristics are suitable for parallel computing. The parallelization strategies generally distribute the evaluation of candidate solutions among the available computing units. In shared-memory environments, PThreads and OpenMP are widely used. However, in a distributed-memory environment, such as a cluster of several nodes, it is necessary to apply message passing, e.g., using MPI. It can be implemented either exclusively or coupled with a shared-memory implementation at each node.

Regardless, current high-performance computing platforms do not only consist of multiple machines with several processors. They also have several GPUs for general-purpose computing, and as mentioned, they are well adapted to work with artificial neural networks. The authors of this work are aware of

this situation, so the proposed parallelization scheme focuses on the number of available CPU and GPU resources in the cluster nodes. The implementation developed must face several difficulties of general interest. Firstly, it is necessary to do dynamic load balancing because not every candidate architecture is promising enough (or even feasible) to be fully evaluated. Secondly, a two-stage evaluation of the individual is proposed. It uses CPU and GPU resources to evaluate each type of neural network depending on its architecture's feasibility. Thirdly, GPUs can be distributed among different machines, and their availability might vary. For these reasons, the optimization algorithm is decoupled from the specific evaluation process (abstraction). Hence, it has been implemented as an external module that could be linked to any other optimizer.

The main contributions of this work are the following:

- The description of how to adapt a continuous optimization algorithm, TLBO in our case due to its properties, to design neural network architectures using a new solution encoding.
- The design of an external module called 'oracle' that processes the individuals in two phases, depending on their architecture feasibility. The oracle avoids execution errors when evaluating non-feasible individuals in GPUs and dynamically distributes individuals to different nodes using MPI.

The rest of this paper is structured as follows: Section 2 discusses some high-performance computing (HPC) approaches for distributing the processing of candidate solutions in NAS. Section 3 explains the proposed methodology for automatically optimizing neural networks. Section 4 describes the experimentation carried out and the results obtained. Finally, Section 5 draws the conclusions and states future research lines.

2 Related Works

As introduced above, evaluating individuals of population-based metaheuristics is suitable for parallelization. If the computational time of tasks is known in advance, static load balancing approaches are enough. There exist solutions for managing a set of computational jobs in a given HPC platform optimally, for example, using genetic algorithms [13]. However, timing is not always predictable, and dynamic load balancing strategies are required to minimize idle periods of the execution units. This situation occurs when working with different individuals representing neural network architectures to evaluate.

In the literature, evaluating alternative neural network architectures generally relies on working with a heterogeneous platform using dynamic load balancing. The computing environment usually combines multiple CPUs and GPUs and requires tuning and supervision. Some frameworks, such as StarPU [14], Qilin [15] and Scout [16], facilitate load sharing in such environments. Their main limiting factor is that they are not designed as external modules, so

the user must implement his/her algorithm within them, using their directives and APIs.

In [17], the authors propose a genetic algorithm for NAS that uses a Galaxy 2, an HPC simulator and management platform developed by Stellar Science, to perform asynchronous load sharing. A Galaxy server is created with a master process and different workers that evaluate individuals. This structure is similar to what it is pursued in this work. However, Galaxy 2 is not open-source software, and the programmer is restricted to use the functionalities defined in it.

In [18], the authors design a NAS framework for building accurate yet fast neural networks. The evaluation of individuals rely on MENNDL, a software package written in PyTorch, a deep learning framework. It uses MPI for asynchronous communication between nodes. Their goal is similar to ours, i.e., distributing the evaluation of neural networks among GPUs and CPUs. However, their management scheme is embedded in the optimization algorithm, which makes it hard to apply their method to other optimizers.

In [19], the authors propose a framework for implementing and studying NAS methods. It is called DeepHyper and deals with limitations in parallel executions and scalability. It uses Balsam [20], a general-purpose framework for managing workflows in HPC environments. Balsam relies on MPI to distribute the load between nodes, maintaining a PostgreSQL database with the status of the different tasks. In theory, applications require no modification to run inside Balsam. However, evaluating a candidate design does not necessarily need any GPU. It is desirable to know a priori the feasibility of architectures and allocate the resources accordingly. Since a pre-processing stage is needed, it would be linked to the optimizer, and load distribution would have to be managed separately from Balsam. Therefore, despite being a system external to the application, it is not completely adapted to the problem at hand. Moreover, the generality of this solution introduces unnecessary overhead.

In the present work, the proposed solution differs from the previous ones due to its simple design that does not depend on the optimizer. Hence, it can be easily associated with any other optimization algorithm. Furthermore, it is built using only OpenMP and MPI with the widely-used Slurm queuing system, which eliminates the extra overhead that would be caused by creating databases and auxiliary data structures. The proposed solution can be easily configured to handle different GPUs and CPUs and only uses open-source technologies. Finally, the proposal can become a standard solution to NAS methods considering different types of candidate neural network designs. It allows balancing the evaluation of viable and non-viable architectures efficiently, saving resources and only allocating them for the tasks required.

3 Methodology

This section explains in detail the developed tool for automatically designing neural networks. This description includes the optimization algorithm, the

architecture encoding scheme, the cost function, and the parallel multi-GPU implementation.

3.1 Teaching-Learning-based Optimization (TLBO)

TLBO is a population-based meta-heuristic proposed in [10] for continuous large-scale optimization. As a meta-heuristic, it is independent of the objective function, which is mainly seen as a black box. This algorithm is widely used due to its applicability, simplicity of implementation and configuration, and effectiveness [21–23].

TLBO simulates a class of students that interact with each other to improve their skills. In practical terms, each student or individual is a candidate solution for the problem at hand. As introduced, this method lacks metaphor-specific parameters. It only expects the population size and the number of cycles to run. This aspect is highly appreciated because most population-based meta-heuristics have numerous parameters to tune for balancing the effort in exploring new regions of the search space and exploiting the known ones, which makes their use more difficult [10, 24, 25]. For TLBO, one can directly enhance the access to new zones with larger populations and let them be studied in depth with more cycles [23]. Provided with these two parameters, TLBO starts by creating and evaluating as many solutions as required. Once the initial population has been created and evaluated, TLBO executes its main loop, which consists of the teacher and the student stages.

The teacher stage simulates how students learn from their professor. At this phase, the best solution in the population becomes the professor, T. Then, TLBO attempts to shift every candidate solution towards it in the search space. More specifically, after identifying T, the method computes a vector M with the average value of the current population for each dimension of the search space. Next, for each solution candidate solution S, TLBO computes a modified (shifted) version of it, S'. It follows Eq. (1), which is expressed in terms of each vector component or dimension of the search space, i. T_F, known as the 'teaching factor', is a random integer that can be either 1 or 2. r_i is a random number in the real range [0, 1]. Both r_i and T_F are fixed for the current iteration and stage. After computing each modified candidate solution, those that outperform their original version replace it, while the rest are discarded.

$$S'_i = S_i + r_i \left(T_i - T_F M_i\right) \tag{1}$$

The learner stage models how students learn from their partners. For this purpose, TLBO pairs each candidate solution, S, with another different from it, P_S. The aim is to create a modified version of it, S', according to Eq. (2). It is also expressed in terms of vector components, so r_i corresponds to a real random number in the range [0, 1] linked to component i. Again, these random factors are fixed for the current iteration and stage. As can be seen, this stage attempts a local shift for each candidate solution, S, in the (improving) direction defined with its pair. At the end of this stage, the altered solutions that outperform their initial version replace them.

$$S'_i = \begin{cases} S_i + r_i \left(S_i - P_{S_i}\right) & \text{if } S \text{ is better than } P_S \\ S_i + r_i \left(P_{S_i} - S_i\right) & \text{otherwise} \end{cases} \quad (2)$$

After executing the requested number of cycles, TLBO returns the best candidate solution in the population.

3.2 Representation of candidate solutions

TLBO aims at continuous optimization problems, so its candidate solutions must be vectors in $\mathbb{R}^N$, where N is the number of dimensions of the search space. Accordingly, as explained in the next section, the objective function ranking the candidate solutions, f, must be of the form $f : \mathbb{R}^N \to \mathbb{R}$.

In this context, each candidate solution has to represent a neural network architecture. For this purpose, and considering the previous requirements, neural network architectures are encoded as vectors of real numbers. The definition of a neural network architecture consists of a sequence of layers. Each layer is a set of artificial neurons of a particular type and configuration. Hence, there are two fundamental pieces of data to represent, i.e., the type of layer and its configuration. The proposed encoding stores this information using a single real value per layer, i. The integer part, I_i, defines the type of layer, while the decimal part, D_i, is mapped into its corresponding space of possible configurations. The user is responsible for defining the number of layers to consider, which ultimately defines the problem dimension, i.e., N, for TLBO (or any other continuous optimizer, as the representation of solutions is not linked to this method).

When decoding, relating any vector component with a particular type of layer is straightforward. For instance, if 3 types of layers are considered, the encoded value will correspond to the first, the second, or the third if the value is in the range [0,1), [1,2), or [2, 3), respectively. As can be seen, the type is ultimately defined by the integer part of the corresponding vector because the decimal part will always be between 0 and 0.999... Logically, the user is responsible for selecting the appropriate types of layers depending on the application. However, it is advisable to reserve a special type for disabled layers. By proceeding this way, the search is more flexible because the optimizer can find simpler architectures, i.e., with fewer layers than N, which becomes an upper bound in reality. This approach overcomes the assumption of fixed-length solutions of traditional optimizers, including TLBO. A similar strategy is applied in [26], where the authors look for the most appropriate subset of heliostats to activate in a solar power tower plant. Namely, they assign the value of 0 to disabled heliostats and 1 to the enabled ones so that the number of dimensions is fixed from the perspective of the population-based optimization used. Similarly, the authors of [8] also consider a type for disabled layers to hide some dimensions of the candidate solutions to their PSO method for neural network architecture optimization.

Interpreting the decimal part, D_i, linked to a particular integer part, I_i, is more sophisticated. D_i encodes the configuration of the type of layer defined

by I_i, but the number of parameters (and their range) significantly varies from one type to another. The strategy proposed to map the space of parameters into a single decimal value is inspired by how multidimensional matrices can be represented in a single vector in Informatics. In this situation, the length of the vector results from multiplying the size of each dimension. This length can be ultimately scaled as a value in the range [0, 1]. Notice that reverting the scaling can require rounding, so nearby values might result in the same decoded value. Nevertheless, TLBO, like most population-based metaheuristics, is derivative-free, so it is not affected by occasional plateaus in the search space. Furthermore, it is necessary to work with double precision to ensure that all the possible configurations of complex layers can be thoroughly represented. Otherwise, some intermediate values could be lost at rounding. Regardless, that situation would not be critical either since this kind of metaheuristic renounces to the certainty of finding the optimal solution due to practical limitations. In other words, not being able to achieve many configurations with few variations between them would not make a big difference in the results achieved.

For illustrating the previous explanation, one can think in a particular type of layer that expects two integer parameters. Let the first be the activation function considering 4 possible values. Let the second be the number of neurons to use, with a limit of 500. In this situation, the space of configurations can be represented by a 4×500 matrix. The resulting vector would have 2000 positions. Figure 1 depicts this context and the interpretation of a hypothetical decimal part of 0.85. Notice that the integer part would have previously defined the defined type of layer. It is also relevant to mention that this kind of matrix of possible configurations can be implemented as a look-up table if necessary, so the values contained do not need to be always consecutive intervals. Furthermore, if any of the layer-specific parameter were not integer, it would need to be sampled accordingly.

3.3 Cost function

As mentioned above, the cost or objective function used to evaluate and rank any candidate solution must be of the form $f : \mathbb{R}^N \to \mathbb{R}$. However, before defining it, it is necessary to determine if there are feasible and unfeasible neural network architectures. If every architecture is feasible, the cost function can be the average quality of the corresponding network with a validation dataset after having been built and trained. Both the dataset and training conditions, such as the stopping criteria, must common for every solution.

If not every architecture is feasible, those that are can be evaluated as described above. However, it is also necessary to define how to mark unfeasible ones, i.e., undesirable solutions, while also being able to distinguish which are the most problematic ones. This approach is one of the most popular for handling constraints with population-based meta-heuristics [27].

For the problem used later for testing, not every layer combination is possible. For instance, after 3 convolutional layers, the resulting images have a size

of 5 x 5. In this situation, it is not possible to apply another convolutional layer with a filter size of 7, as the size of the obtained images would be lower than 0.

In this case, unfeasible solutions obtain a very high value compared to feasible solutions. The value is 1000 (a high number never reached by feasible solutions) minus the number of correct layers. This is useful because, for example, a neural network containing a fully connected layer as the initial layer is worse than another neural network producing an error in the last layer. Hence, it is possible not only to distinguish between feasible and unfeasible ones but also to compare and rank unfeasible ones. This latter aspect gives the optimizer a higher resolution when working with that kind of solution and trying to improve it.

As explained, the more inconsistencies that a candidate solution has, the less value that it receives. Notice that it is convenient to include some feasible solutions in the initial population of the optimizer. Otherwise, the computational effort to converge to promising architectures is higher.

Concerning the evaluation of feasible candidate solutions, the associated neural networks are trained using the backpropagation algorithm and the Adam optimizer with a batch size of 16 and a learning rate of 0.0004. The training does not stop after a fixed number of iterations. Instead, it concludes when the evaluated neural network is unable to reduce the error committed with the validation dataset after five consecutive iterations. This approach implicitly adapts to the quality of each solution. It is also well aligned with the current trends in neural network training to avoid over-fitting. Thus, the cost function to minimize, can be expressed as follows:

$$cost = Cross\_Entropy \tag{3}$$

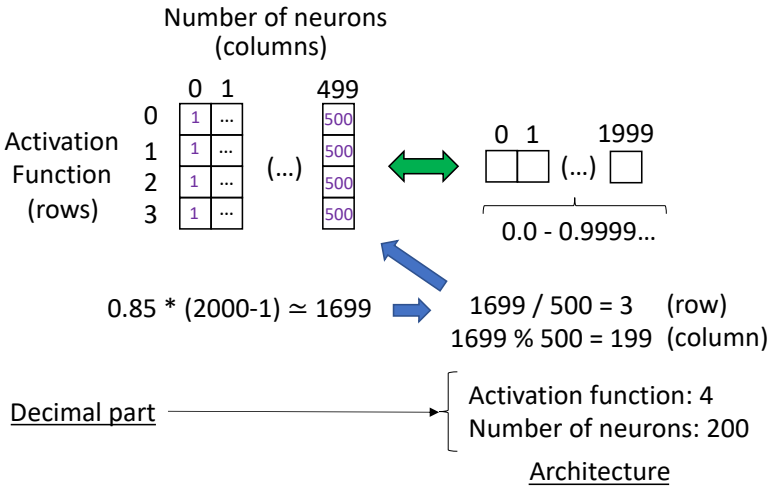

Fig. 1: Example of the decoding of the decimal part linked to a type of layer.

where *Cross_Entropy* is the error of the network after being trained with the training dataset and evaluated with the validation dataset.

3.4 Sequential evaluation of candidate solutions

As explained, TLBO starts with an initial population of user-defined size. It consists of 2 candidate solutions representing architectures known to perform well with the given dataset, such as LeNet-CNN [28]. 90% of the initial solutions are randomly generated, but their generation avoids violating constraints, such as having a filter size in a convolutional layer greater than the expected image size. Finally, the remaining solutions are totally random, which might include unfeasible architectures with poor values.

Unfeasible architectures can be evaluated on CPUs because they do not need to be trained with backpropagation. On the contrary, feasible ones should be assessed on GPUs. However, their feasibility is unknown before being evaluated, so it is impossible to divide the computational load between CPUs and GPUs in advance. Therefore, every candidate solution is preliminarily assessed on a CPU, and those requiring a GPU are put in a queue to access this kind of computational resource sequentially. This scheme remains unaltered at every stage of the optimizer.

3.5 Parallel evaluation of candidate solutions

The evaluation is decoupled from the optimizer and implemented as an external module known as the 'oracle'. The oracle is synchronously called by the optimizer to evaluate the set of candidate solutions involved in a particular stage. Hence, it is responsible for knowing how to evaluate the different types of candidate solutions and the available resources. The communication between the optimizer and the oracle is through text files.

The oracle runs two phases (see Figure 2). The first runs on a CPU and checks the feasibility of the given solutions. Unfeasible ones obtain their final value at that point (a penalized value, as explained). As this stage does not require any GPU, it runs exploiting the number of cores of the machine where it runs using OpenMP and a shared-memory model. More specifically, each thread processes one of the encoded neural network architectures. It is important to remark that the evaluation of individuals in the first phase cannot directly access GPUs. In that case, all the cores would try to access GPU resources to evaluate their individuals, which is unsupported by the configuration of the underlying deep learning framework.

The second phase of the oracle is reserved for solutions encoding feasible architectures. However, their complexity and the required run-time might vary significantly between them. Thus, they cannot be statically distributed among the available GPUs in advance. It is necessary to opt for dynamic load balancing. Since the GPUs can be in different machines, the implementation of the oracle ultimately follows a distributed-memory model using MPI. There will be a worker process per GPU, and if a certain node features several GPUs, it will execute the same number of workers.

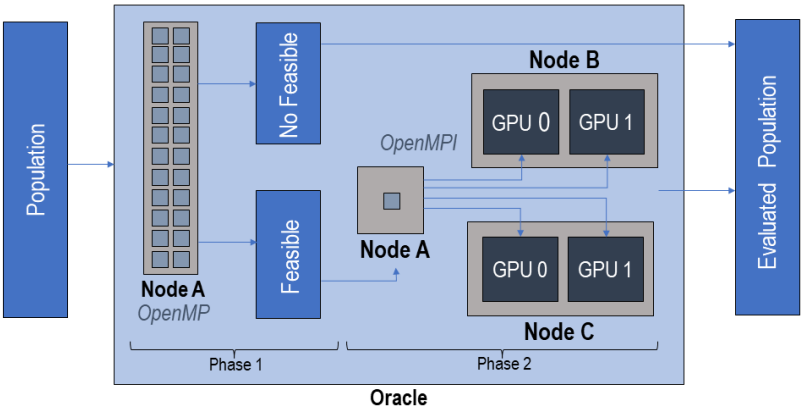

Fig. 2: Oracle phases

The oracle is integrated within a system of queues powered by Slurm. It generates a master process and as many workers as GPUs on the available nodes. The processes are distributed among the available nodes using the RRS expressions[1].

Accordingly, after executing the first phase, the master process distributes the workload among the different worker nodes. Firstly, each candidate solution is assigned to a worker process with its own GPU. When a worker ends evaluating a solution, it sends back the obtained value for the master to store this information. This worker will keep waiting for new tasks either until receiving a new solution to evaluate or the termination signal. Algorithm 1 shows the pseudo-code describing the second stage of the oracle and how it manages the different GPU nodes.

4 Experimentation and results

4.1 Implementation details

The optimization algorithm has been adapted from the public C implementation published in [21]. It has been necessary to externalize the evaluation of solutions to make TLBO use the oracle by passing and parsing text files. The oracle has also been implemented in C within an MPI structure of division of tasks. There is a master process responsible for the input and output with the calling optimizer. It also executes the initial evaluation of the input solutions (first stage) using OpenMP to benefit from running in a multi-core machine. Besides, the master process is designed to request the evaluation of solutions requiring a GPU (second stage) to worker processors. The worker processors limit to wait for new solutions to evaluate with the GPU that they are attached to and send the result to the master. This kind of evaluation internally launches

[1] https://docs.oracle.com/cd/E19061-01/hpc.cluster30/806-0296-10/6j9llte66/index.html

Algorithm 1 Behavior of the oracle at stage 2 using MPI

```
function MASTER(numWorkers,total)
    Distribute the solutions among the available workers
    sent ← numWorkers
    while processed < total do
        Receive the answer from worker i
        processed++;
        if sent < total then
            Sent solution to worker i
            sent++
        else
            Send termination request to workers
            Finish the loop
        end if
    end while
end function
function WORKER(rank)
    while 1 do
        Receive message from the master
        if message contains a solution then
            Evaluate the solution using my GPU
            Send the result to the master
        else
            Terminate execution
        end if
    end while
end function
function MAIN
    if rank is 0 then
        master(numWorkers,total)
    else
        worker(rank)
    end if
end function
```

a Python script that uses Tensorflow, a deep learning framework, to build the corresponding network and assess it.

4.2 HPC infrastructure

The experimentation has been carried out in the Bull cluster of the Supercomputing-Algorithms research group of the University of Almería, Spain. In this cluster, 2 nodes featuring 2 GPUs NVIDIA TESLA V100, 2 processors AMD EPYC 7302 with 16 cores, and 512 GB of RAM DDR4 each

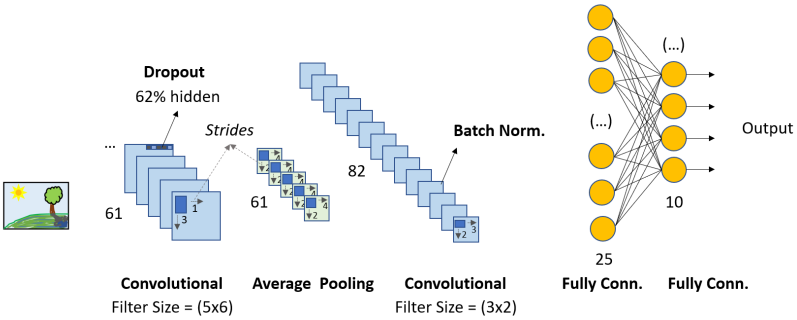

Fig. 3: Resulting Neural Network Architecture

have been used for the GPU workers of the oracle. Another node with 2 processors AMD EPYC Rome 7642 of 48 cores and 512 GB of RAM DDR4 have been reserved for the TLBO algorithm and the master process of the oracle.

4.3 Experiments

4.3.1 Evaluation of the optimization method

For evaluating the proposal made, this work tries to obtain a neural network architecture with less than 10 layers for the CIFAR-10 dataset, restricted to 150,000 parameters. The number of parameters is of great importance in IoT systems, as the neural networks have to be as small as possible due to memory constraints in microcontrollers. For this purpose, the proposed neural network architecture optimizer has been executed with a population size of 1000 individuals and 50 cycles. The achieved result has 7 layers with 71,859 parameters in total. It obtains a top-1 accuracy level of 65%, which outperforms the initial solutions provided to the system (defined by human experts), whose accuracy was 55% and 61%, respectively. This network (Figure 3) consists of the following layers: One convolutional layer with 61 coordinate filters of size 5 x 6 and strides of size 3 x 1 and sigmoid as activation function, 1 dropout layer of 62% hidden neurons, 1 Average Pooling layer with 2 x 4 as strides dimensions, one convolutional layer with 82 coordinate filters of size 3 x 2 and strides of 2 x 3 and ReLu as activation function, one Batch Normalization layer, and 2 fully-connected layers of 25 and 10 neurons, with ReLu and Softmax as activation functions respectively. Regardless, the proposed parallelization scheme is transparent to the optimizer. It also allows exploring more solutions per unit of time, which would benefit this or any other population-based meta-heuristic by making it possible to work with more individuals and reducing design times.

4.3.2 Evaluation of the performance

Table 1 shows the execution time for 3 different configurations. The first one executes a single optimization cycle using a single CPU thread and a GPU.

		Conf 1	Conf 2	Conf 3
Threads		1	96	96
GPUs		1	1	4
Time (s)	**CPU**	630	7	8
	GPUs	6140	6117	1603
	Total	6770	6124	1611
Speed Up		1	1,10	4,20

Table 1: Execution time and speedup of the different configurations.

This configuration corresponds to the sequential version of the system, and it takes 6 770 seconds for evaluating all the individuals up to the first cycle.

The second configuration shows the results of parallelizing only the first stage of the oracle with OpenMP. More specifically, it covers the initial evaluation of all the candidate solutions to tag them as feasible or unfeasible using a CPU. After that, the oracle has a single worker with a GPU for composing and assessing the feasible neural networks encoded. Thus, this configuration cannot exploit the availability of multiple GPUs. In this case, as can be seen, the first stage of evaluation takes 7 seconds instead of 630, which represents an speedup of 90. This value is near to the ideal speedup of the system, which would be 96 as the machine has 96 cores in total. In general terms, considering the CPU and GPU processing time, the overall speedup achieved is 1.10 due to the prominent amount of GPU time compared to the CPU one, which is well-aligned with Amdahl's law.

Finally, the third configuration combines the 96 threads for the first stage of the oracle along with 4 GPUs for the second one. In this case, the GPU time consumes 1603 seconds in total, which represents a speedup of 3.83. Again, this acceleration factor is near to the ideal one, which would be 4 for this part of the system as it has 4 equal GPUs. Considering both stages of the oracle, the overall speedup is 4.20, and the ideal one would be 4.39.

5 Conclusions and future work

This work has described how to use the widely-used Teaching-Learning-based Optimization (TLBO) method for designing neural network architectures for specific applications. Since the optimizer is for continuous optimization problems, it has been necessary to incorporate a new encoding scheme which is able to represent neural network architectures in vectors of real numbers. This strategy, which is optimizer-independent, is based on how multi-dimensional matrices can be stored in a single-dimension matrix (vector) in Computer Science. It can represent any architecture using a single dimension or component. Additionally, the use of a class for disabled layer enhances the flexibility of the method.

This proposal has been studied by designing a convolutional neural network for image classification based on the CIFAR-10 dataset. The proposed method has been able to find neural network architectures outperforming those designed by human experts for the referred dataset with the imposed

restrictions. These results support the proposal and let us think of the possibility of replacing the tedious human-based stages of neural network design by automatic optimizers.

This work focuses on the use of heterogeneous high-performance computing platforms to support the network design method. Population-based metaheuristics like TLBO generally need to evaluate numerous solutions, which is computationally demanding, and also need GPUs to run in reasonable times. Hence, the parallelization strategy focuses on the evaluation of candidate solution, i.e., potential neural network architectures. It is decoupled from the optimizer and implemented as an external module known as the oracle, which is responsible for assessing the candidate solutions provided by the optimizer and managing the computational resources dynamically. The oracle considers two consecutive stages: the first one runs on CPU using OpenMP to detect feasible and unfeasible architectures. Feasible ones, which will need further evaluation on GPU, pass to the second stage. At it, the oracle uses a master process and different workers linked to GPUs to dynamically assess the feasible architectures. This strategy obtains a speedup factor of 90 for the first stage using 96 CPU cores, and a speedup factor of 3.83 for the second one using 4 GPUs. Together, the overall speedup achieved is 4.2 out of a maximum of 4.39 with respect to a sequential execution.

In the future, we will implement a mechanism to group disabled layers altogether to have permanently simplified architectures and reduce the search space. The possibility of reducing the training dataset to accelerate the assessment of candidate solutions will be also considered. Finally, different optimization methods will be compared within the proposed framework of continuous neural network encoding and decoupled oracle-based evaluation.

Declarations

Ethical Approval

This article does not contain any studies with human participants or animals performed by any of the authors.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

Conceptualisation, M.L., N.C.C, J.F.S, B.P and P.M.O; Methodology: M.L., N.C.C, J.F.S, B.P and P.M.O; Implementation: M.L. and N.C.C; Experimentation: M.L., N.C.C; Writing - original draft preparation: M.L., N.C.C, J.F.S and P.M.O; Writing - review and editing: J.F.S, B.P and P.M.O. All authors have read and agreed to the published version of the manuscript.

Funding

This work has been funded by the projects R+D+i RTI2018-095993-B-I00 and PID2021-123278OB-I00 from MCI-N/AEI/10.13039/501100011033/ and ERDF funds; by the Andalusian regional government through the project P18-RT-119, by the University of Almería through the project UAL18-TIC-A020, and by 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 funded by the Ministry of Digital Transformation, Industry, Knowledge and Universities of the Andalusian regional government.

Availability of data and materials

The source code of the Oracle is available in Github: https://github.com/marcoslupion/NAS_MultiGPU.

References

[1] Sharma, N., Sharma, R., Jindal, N.: Machine learning and deep learning applications-a vision. Global Transitions Proceedings **2**(1), 24–28 (2021). https://doi.org/10.1016/j.gltp.2021.01.004

[2] Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788 (2016). https://doi.org/10.48550/ARXIV.1506.02640

[3] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016). https://doi.org/10.48550/ARXIV.1512.03385

[4] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Proceedings of the 3^{rd} International Conference on Learning Representations, pp. 1–14 (2015). https://doi.org/10.48550/ARXIV.1409.1556

[5] Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G.G., Tan, K.C.: A survey on evolutionary neural architecture search. IEEE transactions on neural networks and learning systems, 1–21 (2021). https://doi.org/10.1109/TNNLS.2021.3100554

[6] Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. In: Proceedings of the 5^{th} International Conference on Learning Representations, pp. 1–16 (2017). https://doi.org/10.48550/ARXIV.1611.01578

[7] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: International Conference on Machine Learning, pp. 2902–2911 (2017). https://doi.org/10.48550/ARXIV.1703.01041. PMLR

[8] Wang, B., Sun, Y., Xue, B., Zhang, M.: Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In: 2018 IEEE Congress on Evolutionary Computation, pp. 1–8 (2018). https://doi.org/10.1109/CEC.2018.8477735. IEEE

[9] Byla, E., Pang, W.: Deepswarm: Optimising convolutional neural networks using swarm intelligence. In: UK Workshop on Computational Intelligence, pp. 119–130 (2019). https://doi.org/10.1007/978-3-030-29933-0_10. Springer

[10] Rao, R.V., Savsani, V.J., Vakharia, D.P.: Teaching–learning-based optimization: an optimization method for continuous non-linear large scale problems. Information sciences **183**(1), 1–15 (2012). https://doi.org/10.1016/j.ins.2011.08.006

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

[12] Jameel, S.M., Hashmani, M.A., Rehman, M., Budiman, A.: An adaptive deep learning framework for dynamic image classification in the internet of things environment. Sensors **20**(20) (2020). https://doi.org/10.3390/s20205811

[13] Orts, F., Ortega, G., Puertas, A.M., García, I., Garzón, E.M.: On solving the unrelated parallel machine scheduling problem: active microrheology as a case study. The Journal of Supercomputing **76**(11), 8494–8509 (2020). https://doi.org/10.1007/s11227-019-03121-z

[14] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009 Parallel Processing, pp. 863–874. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_80

[15] Luk, C.-K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55 (2009)

[16] McCormick, P., Inman, J., Ahrens, J., Mohd-Yusof, J., Roth, G., Cummins, S.: Scout: a data-parallel programming language for graphics processors. Parallel Computing **33**(10), 648–662 (2007). https://doi.org/10.1016/j.parco.2007.09.001

[17] Martinez, D., Brewer, W., Behm, G., Strelzoff, A., Wilson, A., Wade, D.: Deep learning evolutionary optimization for regression of rotorcraft vibrational spectra. In: 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC), pp. 57–66 (2018). https://doi.org/10.1109/MLHPC.2018.8638645

[18] Patton, R.M., Johnston, J.T., Young, S.R., Schuman, C.D., Potok, T.E., Rose, D.C., Lim, S., Chae, J., Hou, L., Abousamra, S., Samaras, D., Saltz, J.: Exascale deep learning to accelerate cancer research. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 1488–1496 (2019). https://doi.org/10.1109/BigData47090.2019.9006467

[19] Balaprakash, P., Salim, M., Uram, T.D., Vishwanath, V., Wild, S.M.: Deephyper: Asynchronous hyperparameter search for deep neural networks. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pp. 42–51 (2018). https://doi.org/10.1109/HiPC.2018.00014

[20] Salim, M.A., Uram, T.D., Childers, J.T., Balaprakash, P., Vishwanath, V., Papka, M.E.: Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows. arXiv preprint arXiv:1909.08704 (2019). https://doi.org/10.48550/ARXIV.1909.08704

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

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

[23] Torres-Moreno, J.L., Cruz, N.C., Álvarez, J.D., Redondo, J.L., Giménez-Fernandez, A.: An open-source tool for path synthesis of four-bar mechanisms. Mechanism and Machine Theory **169**, 104604 (2022)

[24] Boussaïd, I., Lepagnot, J., Siarry, P.: A survey on optimization metaheuristics. Information sciences **237**, 82–117 (2013)

[25] van Geit, W., De Schutter, E., Achard, P.: Automated neuron model optimization techniques: a review. Biological cybernetics **99**(4), 241–251

(2008)

[26] Cruz, N.C., Álvarez, J.D., Redondo, J.L., Berenguel, M., Ortigosa, P.M.: A two-layered solution for automatic heliostat aiming. Engineering Applications of Artificial Intelligence **72**, 253–266 (2018). https://doi.org/10.1016/j.engappai.2018.04.014

[27] Yeniay, Ö.: Penalty function methods for constrained optimization with genetic algorithms. Mathematical and computational Applications **10**(1), 45–56 (2005). https://doi.org/10.3390/mca10010045

[28] Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)