Hyper-Parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization

Pablo Ribalta Lorenzo
Future Processing
Gliwice, Poland
pribalta@future-processing.com

Jakub Nalepa
Future Processing/Silesian University of Technology
Gliwice, Poland
jakub.nalepa@polsl.pl

Luciano Sanchez Ramos
Universidad de Oviedo
Departamento de Informática
Oviedo, Spain
luciano@uniovi.es

José Ranilla Pastor
Universidad de Oviedo
Departamento de Informática
Oviedo, Spain
ranilla@uniovi.es

ABSTRACT

The need of manual hyper-parameter selection can seriously hamper the model optimization of Deep Neural Networks (DNNs). Conventional automated approaches tackling this problem suffer from poor scalability or fail in certain scenarios. In this paper, we introduce a parallel method that applies Particle Swarm Optimization (PSO) for the hyper-parameter selection in DNNs. To estimate the best hyper-parameters, a population of particles is evolved, with their fitness calculated in parallel. The experimental results demonstrate very desirable scalability properties for different DNNs. We show that the parallel PSO can further optimize existent models designed by experts in an affordable amount of time.

CCS CONCEPTS

•Computing methodologies → Parallel algorithms; Machine learning; *Neural networks; Bio-inspired approaches;*

KEYWORDS

Deep neural networks, hyper-parameter selection, particle swarm optimization, parallel evolutionary algorithm

ACM Reference format:
Pablo Ribalta Lorenzo, Jakub Nalepa, Luciano Sanchez Ramos, and José Ranilla Pastor. 2017. Hyper-Parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3084211

1 INTRODUCTION

Selecting the hyper-parameters of a DNN model can be arguably one of the hardest stages of its design. DNNs have achieved an unprecedented success in many areas, particularly due to the accessibility to very large datasets and GPUs. These allow for training the models that can surpass human performance at many tasks, but once these models become sufficiently complex, finding their best parameters is a difficult problem even for experts in the field.

Most DNN models are driven by a set of hyper-parameters that control many aspects of the algorithm behavior. These hyper-parameters affect the time and memory cost of running the algorithm, the quality of the model resulting from the training process, or its ability to generalize to the unseen data when deployed in the wild. For these reasons, finding the best possible parameters for a given DNN architecture becomes crucial.

There are two main approaches to the hyper-parameter selection: manual or automatic methods. Retrieving the hyper-parameters manually assumes that there exists an understanding of how the hyper-parameters affect the outcome of the model and make it achieve a good generalization. On the other hand, automatic hyper-parameter selection methods greatly reduce the need for this understanding, but they come at the expense of the costlier computation.

Hyper-parameter selection for a DNN can be seen as an optimization problem, where the objective is to find the hyper-parameters that minimize an objective function—typically the validation error. In the simplest formulations, this problem is unconstrained, but limitations (e.g, training time or generalization accuracy) can be imposed. In traditional approaches, whether manual or automatic, it is assumed that although it could be possible to obtain the gradient of the error measure, it is infeasible to compute it in practice.

This paper introduces a parallel variant of PSO for optimizing the hyper-parameters of a DNN model sustained by the findings obtained in our recent work [23]. We exploit task parallelism in order to concurrently evaluate the fitness (defined as the classification accuracy of a trained model over a validation set) of each particle in the swarm in every generation. We attempt to keep the amount of parallelization proportional to the number of independent tasks to be performed, with the goal of simplifying the load balancing and enforcing loose parallelism between the concurrent tasks. Additionally, our experimental study revealed that utilizing the archive of already-analyzed particles during the optimization greatly decreases the execution time of the parallel PSO.

We assume that the PSO hyper-parameters (e.g, the swarm size and the maximum number of generations) remain fixed, keeping the focus on the speedup obtained by parallelization. Our approach

GECCO '17 Companion, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3067695.3084211

currently requires almost no user interaction, and can be considered as a step towards the parameter-less design of DNNs.

1.1 Contributions

An extensive experimental study, involving scenarios with various DNN architectures trained on the well known MNIST dataset and using different hardware setups, showed that:

- Parallel PSO displays very desirable scalability properties while consistently delivering high-quality results.
- Applying parallelism in PSO allows for the efficient exploration of large parts of the entire solution space, without a drastic increase in the computation time.
- Exploiting the archive of the DNN models trained during the execution (for different hyper-parameter combinations) helps significantly decrease the running time of the parallel PSO.
- Parallel PSO significantly improves the existent DNN topologies, boosting the performance of LeNet-4 over the MNIST dataset.

1.2 Paper Structure

In Section 2, we discuss the state of the art on the hyper-parameter selection in DNNs, alongside the parallel evolutionary algorithms. Our parallel PSO is introduced in Section 3. The experimental study is discussed in Section 4. Section 5 concludes the paper.

2 RELATED WORK

In this section, we discuss the state of the art on automatic hyper-parameter selection (Section 2.1), alongside the main approaches and techniques in parallel evolutionary algorithms (Section 2.2), in order to contextualize our work within the literature.

2.1 Automatic Hyper-Parameter Optimization

Searching for good hyper-parameters can be cast as an optimization problem, typically convex, where the hyper-parameters of the model are the decision variables. The cost to be optimized is the validation set error resulting from training using these hyper-parameters. However, in most practical settings, it is infeasible to compute the gradient of a differential error measure on the validation set due to its high computation or memory cost [3, 21]. All approaches for the automatic optimization of the DNN hyper-parameters can be split into *model-free* and *model-based* methods.

2.1.1 *Model-Free Methods.*

Model-free approaches are characterized by not utilizing the knowledge about the solution space extracted during the optimization. This lack of adaptability makes them simple to implement, at the expense of poor results for large hyper-parameter spaces (the information about the regions of the search space that have been already traversed, along with their characteristics are not propagated to the next algorithm iterations).

Grid Search (GS) is commonly used for optimizing DNN hyper-parameters only if their number is very low, and it works by training the underlying DNN architecture using all joint specifications of the hyper-parameter values [4]. Then, the validation set error is retrieved for each DNN, and the best-performing hyper-parameters are obtained. The evident downside of this method is its poor scalability, with its cost growing exponentially for each dimension in the search space.

Random Search (RS) is a trivial to implement alternative to GS, more convenient to use and faster to converge to an acceptable parametrization. Although different improvements can be added to RS, it is typically not *adaptive*, although it can be a part of hybrid approaches that refine its performance drastically [4, 15].

2.1.2 *Model-Based Methods.*

Model-based methods build a model of the validation set error, and then elaborate hyper-parameter values by performing optimization within this model. Most of such model-based methods use a Bayesian regression approach in order to estimate the expected value of the validation set error for each hyper-parameter, alongside the uncertainty around this expectation. Thus, optimization involves a careful balance between exploration and exploitation of the solution space. Contemporary approaches to this include **Spearmint [24]** and **Tree Parzen Estimators [4]**. Currently, Gaussian Processes based Bayesian optimization methods cannot be unambiguously recommended as an established tool for the hyper-parameter optimization. Their major drawback is that the inference time grows cubically in the number of observations, as it necessitates the inversion of a dense covariance matrix.

Other model-based techniques include various Radial Basis Function (RBF) surrogate models [11] and evolutionary approaches, encompassing covariance matrix adaptation evolution strategies **CMA-ES** [20] or **PSO**, which has been proposed in our very recent work [23]. In PSO, a population of candidate solutions (representing tuples of the hyper-parameter values) evolves in time in search of the best possible DNN parameters. The extensive experimental study revealed that PSO is able to not only optimize *simple* DNN architectures (e.g., including one up to three convolutional layers), but also to significantly improve existing models developed by the experienced practitioners.

2.2 Parallel Evolutionary Algorithms

Evolutionary Algorithms (EAs) are inspired by the biological evolutionary processes that occur in nature [2, 9]. The main idea behind these approaches is that in order for a population of individuals to adapt to some environment, it should behave like a natural system. Therefore, survival and reproduction are promoted by the removal of useless or harmful traits, and by rewarding desirable behavior. An EA works by maintaining a population of candidate solutions and evolving it by iteratively applying a set of evolutionary operators: *selection*, *recombination* and *mutation* [25]. The resulting process tends to converge to a globally satisfactory, if not optimal, solution to the problem much like the populations of different species of organisms adapt to their surrounding environment [14]. Due to their wide applicability, they have been exploited to tackle a plethora of various challenging optimization tasks [10, 18].

Parallelism has become a key technology in order to deliver an increase of performance in those problems that include an expensive fitness evaluation, which can be met in principle by adding processors, memory, and interconnection network. There are two main reasons for parallelizing an evolutionary algorithm: one is to achieve time savings by distributing computational effort, and

the second is to benefit from a parallel setting from the algorithmic point of view, in order to retrieve higher-quality individuals [22].

The first type of parallel EAs exploit available machines to process independent populations. This approach to simultaneous work can be very useful, for instance, in running several versions of the same algorithm with different initial conditions. On the other hand, the second approach is directed towards genuinely parallel EAs. There are several possible levels at which an EA can be parallelized: the *population*, *individual* and *fitness* levels.

Individual or population-based parallel approaches usually rely on the spatial structure of the population, isolating some subgroups of individuals and limiting their ability to couple with other elements of the population. The two most important categories in this approach are the *island* and *grid* models. In the island-model parallel EAs, the parallel processes (referred to as the *islands*) often co-operate with each other to effectively guide the search towards the best parts of the solution space. This co-operation is steered by the *co-operation scheme* that defines the co-operation topology, frequency and strategies for handling emigrants and immigrants. However, if the islands do not co-operate during the execution, then this model can be seen as the *batch* processing.

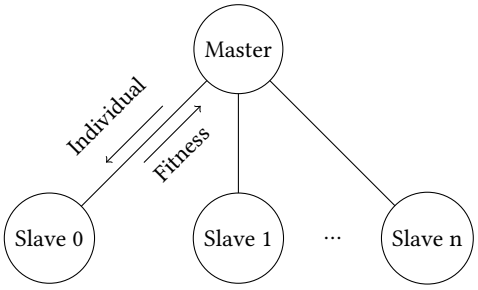

Figure 1: Schematic view of the master-slave model of parallel evolutionary algorithms, in which the slave processes evaluate the fitness values of the received individuals.

Parallelization at the fitness evaluation level does not require any change to the standard EA, since the fitness of an individual is independent from the rest of the population. An obvious approach is to evaluate each individual fitness simultaneously using a different processing unit, with the master process managing the population and handing out individuals to evaluate to a number of slave processes (this model is presented in Figure 1).

3 PARALLEL PSO FOR HYPER-PARAMETER OPTIMIZATION IN DNNS

As discussed in [23], the hyper-parameter selection is an optimization task where the objective is to find a value that minimizes a value of the loss function $\mathcal{L}(\boldsymbol{T}; \mathcal{M})$ for a model $\mathcal{M}$ (elaborated using a training algorithm $\mathcal{A}$) over a training set $\boldsymbol{T}$. The model is parametrized by a set of hyper-parameter values $\boldsymbol{\lambda}$, and it is given as $\mathcal{M} = \mathcal{A}(\boldsymbol{T}; \boldsymbol{\lambda})$ [6]. The objective of the hyper-parameter selection process is to retrieve the parameters $\boldsymbol{\lambda}^*$ that yield a desired model $\mathcal{M}^*$, while minimizing $\mathcal{L}(\boldsymbol{V}; \mathcal{M}^*)$, where $\boldsymbol{V}$ is the validation set [7]:

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{T}; \mathcal{M}) = \arg\min_{\boldsymbol{\lambda}} f(\boldsymbol{\lambda}; \mathcal{A}, \boldsymbol{T}, \boldsymbol{V}, \mathcal{L}). \quad (1)$$

The generalization performance of $\mathcal{M}$ is quantified using the test set Ψ, that was unseen during the optimization. In the following sections, we discuss the parallel PSO for retrieving the desired hyper-parameter values in detail.

3.1 PSO for Hyper-Parameter Optimization

PSO is a population-based stochastic algorithm which is based on social-psychological principles. Unlike other evolutionary algorithms, e.g., genetic or hybrid genetic algorithms, PSO does not exploit the selection operation—all population members (referred to as *particles*) survive from the beginning of a trial until its end. Their interactions result in the iterative improvement of their quality, quantified as the fitness value [12]. The swarm, being a set of particles, moves towards the best-fitted individual in the population.

In our recent work, we introduced PSO for the hyper-parameter selection task [23]. In this algorithm, $f : \mathbb{R}^k \rightarrow \mathbb{R}$ denote the objective function, which maps a tuple of k hyper-parameter values to a real number (the classification accuracy of the trained DNN obtained for $\boldsymbol{V}$)—it is a fitness function in the proposed PSO. The goal is to find a solution $\boldsymbol{\lambda}^*$ for which $f(\boldsymbol{\lambda}^*) \geq f(\boldsymbol{\lambda})$ for all $\boldsymbol{\lambda} \in \chi$, where χ is the set of all possible combinations of hyper-parameter values. In PSO (Figure 2), the initial swarm of particles (created during the *swarm initialization*—see Section 3.2) undergo the *swarm evolution* (Section 3.3), until the termination condition has been reached. PSO can be stopped if (i) the hyper-parameter values that allow to retrieve a model of the desired quality have been obtained, (ii) the maximum execution time has been exceeded, or (iii) the maximum number of swarm generations ($G_{\max}$) have been already processed. Finally, the best particle is returned.

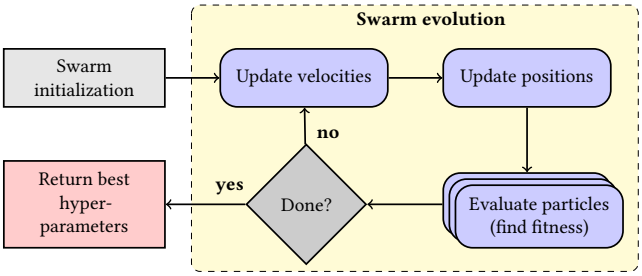

Figure 2: Flowchart of the parallel PSO. The parallel operation (fitness evaluation) is rendered as a stacked step.

3.2 Swarm Initialization

In our PSO, each particle represents a set of hyper-parameters—a numerical vector $\boldsymbol{\lambda}$ of k dimensions randomly initialized in a search space $\mathbb{R}^k$. It can be interpreted as a point in a high-dimensional space (the number of dimensions reflects the number of hyper-parameters of the DNN architecture that is being optimized). Each particle traverses the search space towards the best-fitted particles, hence the swarm moves towards the desired regions of the space.

For each *i-th* particle in the swarm, its initial position $\boldsymbol{\lambda}_i$ is randomly sampled from a uniform distribution $\mathcal{U}(\boldsymbol{b_l}, \boldsymbol{b_u})$, bounded by the hyper-parameter lower and upper limits ($\boldsymbol{b_l}$ and $\boldsymbol{b_u}$). The initial position $\boldsymbol{\lambda}_i$ is the current best known position of this particle

$\boldsymbol{\lambda}_i^*$, and if $f(\boldsymbol{\lambda}_i^*) > f(\boldsymbol{\lambda}^S)$, it is stored as the new best known position in the swarm $\boldsymbol{\lambda}^S$. The velocity $\boldsymbol{v}_i$ of this particle is randomly drawn from a uniform distribution. Similarly, the values are bounded by the upper and lower hyper-parameter limits. Afterwards, the swarm of s particles ($\{\boldsymbol{\lambda}_i, \boldsymbol{v}_i, \boldsymbol{\lambda}_i^* = \boldsymbol{\lambda}_i\}_{i=0,1,\cdots<s}$ tuples) undergoes the evolution.

3.3 Swarm Evolution

In each swarm generation g (where $G_{\max}$ denotes the maximum number of generations), the velocity of all particles is updated (line 4 in Algorithm 1):

$$\boldsymbol{v}_i \leftarrow \omega \boldsymbol{v}_i + \phi_p \boldsymbol{r_p}(\boldsymbol{\lambda}_i^* - \boldsymbol{\lambda}_i) + \phi_g \boldsymbol{r_g}(\boldsymbol{\lambda}^S - \boldsymbol{\lambda}_i), \tag{2}$$

where $\boldsymbol{r_p}, \boldsymbol{r_g}$ are drawn from a uniform distribution $\mathcal{U}(0,1)$ (this stochastic component is applied to the velocity updates to diversify the search), ω is an inertia weight scaling factor, and ϕ_p and ϕ_g are the acceleration coefficients. These factors control the influence of the best particle position ($\boldsymbol{\lambda}_i^*$), alongside the best swarm position ($\boldsymbol{\lambda}^S$) on the velocity. The position $\boldsymbol{\lambda}_i$ is updated, and it becomes $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i + \boldsymbol{v}_i$ (line 5). Afterwards, the best position for each particle is modified, together with the best position in the swarm (lines 8 and 10)—they are updated only if they have been changed. For more details, see [23].

3.4 Parallel Fitness Evaluation

In this paper, we introduce the parallel version of our PSO algorithm for the hyper-parameter optimization in DNNs. Parallelization of PSO allows to simultaneously extract the fitness of a number of particles when hardware resources are available. Calculating the value of the fitness function for each particle in the swarm is an inherently parallelizable operation (line 6 in Algorithm 1), since these evaluations are independent from each other. Hence, the master-slave model introduced in Section 3 has been tailored to PSO. In the parallel PSO, the fitness values of the particles are elaborated by the available slave (GPU)—see Figure 2, in which the parallel fitness evaluation is annotated as a stacked operation.

Procedure 1 PSO evolution with the parallel fitness evaluation.

```
1:  while g ≤ G_max do
2:      for i = 0, 1, ..., s do
3:          r_p, r_g ~ U(0, 1)
4:          Update velocity v_i
5:          λ_i ← λ_i + v_i
6:          Calculate f(λ_i) by the available slave
7:          if f(λ_i) > f(λ_i*) then
8:              λ_i* ← λ_i
9:              if f(λ_i*) > f(λ^S) then
10:                 λ^S ← λ_i*
11:     g ← g + 1
12: return λ^S
```

It is worth mentioning that this is the most computationally intensive part of the PSO optimization because it requires training the DNN model, and then classifying the validation set to quantify its performance. Importantly, in PSO we introduced an *archive*, which contains the models already trained using the hyper-parameter values that have been analyzed during the evolution process. Therefore, if the particle is attracted to the position which was already visited during the evolution, then the fitness is immediately retrieved from the archive—it greatly speeds up the computation. Once the fitness values are found for all the particles in the swarm, the master process controls the evolution as presented in the previous sections.

4 EXPERIMENTAL RESULTS

Our experimental study is divided into two main experiments. Firstly, we verify how the parallel PSO copes with relatively small hyper-parameter search spaces. For this purpose, we use the parallel PSO to optimize our experimental DNN architecture (which is quite shallow) over the MNIST dataset, in the parallel and sequential setups. Afterwards, we investigate the behavior of the parallel PSO for a larger hyper-parameter search space over LeNet-4 on MNIST, comparing the classification accuracy obtained using this architecture with the state-of-the-art performance for MNIST.

4.1 Experimental Setup

Our parallel PSO for the hyper-parameter selection was implemented in Python utilizing the NumPy library. The DNNs were trained using Keras [5] coupled with a TensorFlow [1] backend over CUDA 8.0 and CuDNN 5.1. The experiments were executed on the computational cluster consisting of the GPU nodes, equipped with the Intel Xeon E5-2698 v3 (40M Cache, 2.30 GHz) with 128GB of RAM processors, and NVIDIA Tesla K80 GPU 24GB DDR5. In the **setup A**, we exploit 1 GPU in the node, whereas in the **setup B**, all 6 GPUs available in the node are used.

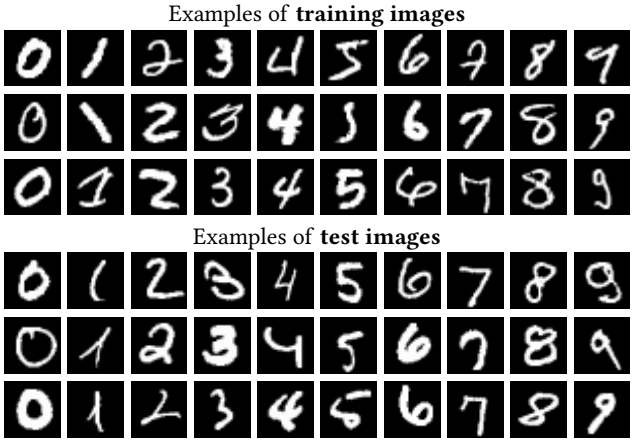

Figure 3: Example training and test MNIST images.

We used a fixed parametrization for the internals of PSO across all experiments, where $\omega = \phi_p = \phi_g = 0.5$ (search parameters). The parallel PSO is terminated if $G_{\max} = 100$ generations have been processed. For all experiments, we exploit 10-fold cross-validation, where $|\boldsymbol{T}| = 9\,|\boldsymbol{V}|$. The training set $\boldsymbol{T}$ is used for the DNN training, while $\boldsymbol{V}$ is exploited to find the fitness during the PSO optimization. The generalization performance is quantified as the accuracy over the unseen test set Ψ. The DNNs are trained for a maximum of 100

epochs using batches of size 128, with the objective of minimizing the categorical cross-entropy (multi-class log loss). We used the ADAM optimizer to steer the learning rate [13].

In this work, we focused on the multi-class classification, and the well-known **MNIST** benchmark dataset to verify the efficacy of the parallel PSO [17]—this set was exploited to assess the classification performance of the DNNs optimized using our parallel approach. MNIST is a dataset of handwritten digits encompassing 70,000 grayscale images (28×28 pixels) divided into 10 classes, with approx. 7,000 images per class. Example images belonging to each class are rendered in Figure 3—it is easy to notice its high intra-class variability (see e.g., training set images representing 1's). There are 60,000 training images in $\boldsymbol{T}$, and 10,000 test images in Ψ (both sets are balanced). The validation sets $\boldsymbol{V}$ (for the 10-fold cross validation) are extracted from $\boldsymbol{T}$, $|\boldsymbol{T}| = 9\,|\boldsymbol{V}|$, and $\boldsymbol{T} \cap \boldsymbol{V} = \emptyset$. Each experiment is run 10× without overlaps between the folds.

4.2 Parallel PSO in Small Search Spaces

In the experiments, we also verify the influence of the archive on the PSO performance and convergence capabilities. This archive caches already investigated hyper-parameter combinations alongside the classification accuracy, in order to avoid training the DNN with the same hyper-parameter values. The swarm size $s = 6$ is utilized across all trials in this experiment, and the underlying DNN which is being optimized is SimpleNet-1 over the MNIST dataset. This experiment was run using both setups A and B.

Table 1: Parameters of the layers in SimpleNet-1.

Layer type	Parameters	Values
Convolutional (C)	Receptive field size ($s_F \times s_F$)	$s_F \geq 2$
	No. of receptive fields (n)	$n \geq 1$
Max Pooling (P)	Stride size (ℓ)	$\ell \geq 2$
	Receptive field size (s_P)	$s_P \geq 2$

SimpleNet-1 is our experimental DNN (one convolution and one max pooling layer, terminated by a 10-output Softmax activation). The hyper-parameters of the layers, alongside their allowed ranges are collected in Table 1. SimpleNet-1 is visualized in Figure 4.

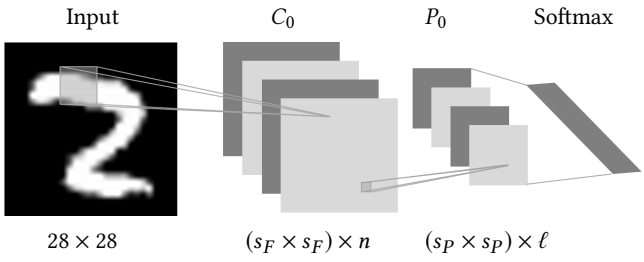

Figure 4: SimpleNet-1 applied to the MNIST image.

In Figure 5, we show that the parallel PSO consistently delivers very similar best results for both setups A and B. This reflects that the parallelization does not compromise the quality of the hyper-parameters. For the average and best results, the spread of the retrieved accuracy values is practically the same, reflecting identical capabilities in the exploration of the search space. These capabilities are driven by the swarm size, which is kept fixed at 6 particles (however, it has been shown that the swarm size does not influence the final accuracy significantly [23]). Finally, the differences in the worst results respond to the randomness of the particle initialization (the termination condition of the parallel PSO was the maximum number of processed generations, hence the swarm did not converge to the same-quality results—it could be addressed by increasing $G_{\max}$).

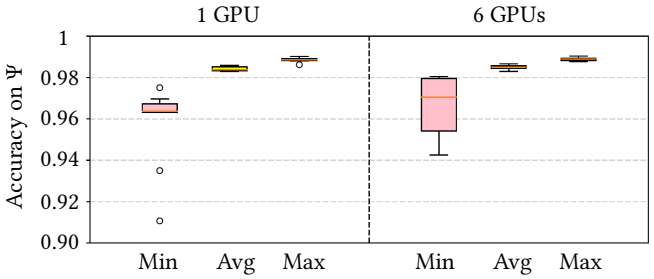

Figure 5: Minimum, average and maximum accuracy (of the best PSO particles across all folds) obtained for Ψ using the optimized SimpleNet-1 for setups A and B.

In order to check if the differences in the average and best accuracy values are statistically important (for both setups), we performed the two-tailed Wilcoxon tests (we verified the null hypothesis saying that *"using both setups leads to the same-quality results"*). This hypothesis could not be rejected (the p-values were $p = 0.1471$ and $p = 0.2113$ for the average and best cases, respectively), therefore both serial (for 1 GPU) and parallel (6 GPUs) retrieve the final DNN parameters of very similar quality.

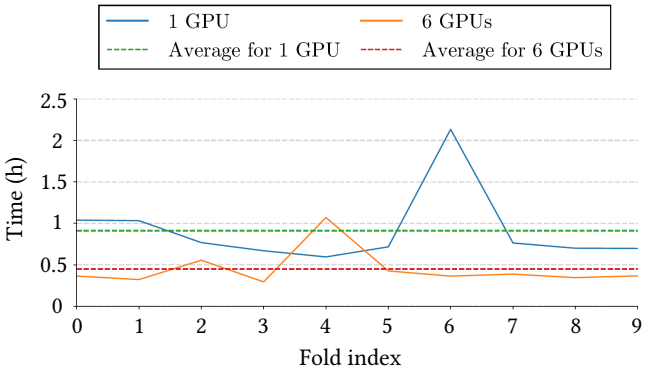

Figure 6: Total execution time per fold for setups A and B, and their corresponding average.

In Figure 6, we present the total execution time across 10 folds in both setups A and B, alongside the average execution time. It may be noticed that the speedup $\mathcal{S}$ of the parallel version is not close to $\mathcal{S} = 6$ (which would indicate the linear scalability), and it remains at $\mathcal{S} \approx 2$. Although it could be interpreted as an indicator of poor scalability of the parallel PSO, this relatively low $\mathcal{S}$ value is the result of applying the archive during the optimization. In small search spaces, PSO—whether parallel or not—can quickly converge towards a final, well-fitted solution. In this case, the archive is constantly being read, hence the execution time of the following generations drastically drops. Therefore, in scenarios with an expensive objective function,

a particle evolution time is strongly influenced by its initialization and the distance it needs to cover towards the final position (i.e., the number of hyper-parameter combinations that are being tested on its path, if they are not cached in the archive).

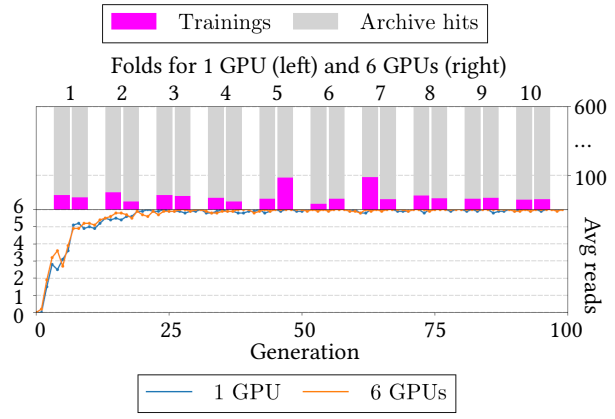

Figure 7: Proportion of combinations trained during the PSO optimization, rendered in light pink (the Y-axis is scaled for visibility) compared with the hyper-parameter values read from the archive (top). Average archive reads per generation across all folds for setups A and B (bottom).

Figure 7 visualizes the influence of the archive on the swarm optimization process. In the upper part of this figure, we observe that at least 80% of the time, PSO is reading the fitness of the hyper-parameter combination (i.e., particle position) from the archive, which is performed almost instantly. This shows that for such a small hyper-parameter space, the evolution converges quickly (in a very small number of generations), and after that the particles lose their velocity and remain static. This removes the need for additional trainings of the DNN. In the bottom part of the figure, we observe in more detail that the optimization converges after around 10 swarm generations (around the 25^{th} generation rarely any additional training takes place). Therefore, the archive is the main element delivering the speedup in this scenario, as the parallelization can only be effectively applied during the first 10 generations with a marginal chance to show observable results.

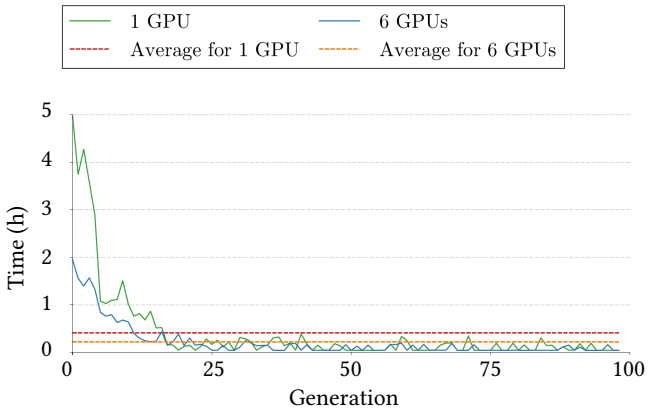

Figure 8: Total and average time per generation for SimpleNet-1 over MNIST on the setups A and B.

In Figure 8, we render the average generation time for both setups, along with the average generation time (the average generation time is roughly 2× smaller for the parallel PSO). As already mentioned, the processing time of a single generation differs across the sequential and parallel PSO for the initial generations (when the size of the archive is still relatively small and the particles traverse the unseen parts of the solution space). Hence, the archive is the pivotal component of PSO which allows to notably decrease its running time for small search spaces.

Although the parallel PSO delivers results of an identical quality with a 2× speedup in scenarios where the hyper-parameter space is small, it becomes critical to use the archive of the previously analyzed hyper-paramater combinations. However, it is still notable how—on such a small proportion of trainings—a parallel implementation of PSO can make a significant difference and deliver an undeniable speedup. This comes to show how much improvement can be gained from the application of parallelism in PSO.

4.3 Parallel PSO in Larger Search Spaces

In this experiment, we optimize an existent DNN architecture using parallel PSO (with the swarm size $s = 6$) for MNIST—we focus on the well-known canonical DNN model (LeNet-4) which has a reported error rate of 1.1% on this dataset [16]. Let FC_i define a fully connected layer with s_{FC} denoting its size. The original LeNet-4 hyper-parameters are given in Table 2, along with the parameter limits for PSO—the original LeNet-4 values have been doubled. The experiment was run exclusively in the setup B.

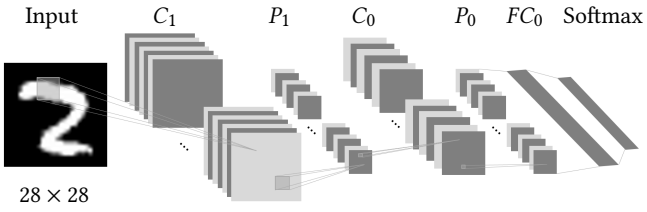

Figure 9: LeNet-4 applied to the MNIST image.

LeNet-4 is a 5-level convolutional neural network designed for the digit classification [16]. It is formed by two blocks of convolution and max pooling layers, finished by a fully connected layer and a Softmax (Figure 9). Its parameters are gathered in Table 2.

Results for DropConnect [8] (currently reported as the best performing DNN over the MNIST dataset with an error rate of 0.21%) are overlayed in Figure 10, in order to contextualize our results, elaborated using the parallel PSO within the state of the art. The conclusion that we can obtain from this experiment is that the parallel PSO can reliably optimize more complex models in larger search spaces, and achieve the results that maximize the potential of the existing architecture (LeNet-4).

It is also worth mentioning that the parallel PSO retrieved the statistically better hyper-parameter values (leading to the better-performing DNN) compared with the sequential PSO in the best case (at $p = 0.0074$, using the two-tailed Wilcoxon tests)—the parallel PSO delivered the average best accuracy of 99.36%, and the accuracy for the best fold was 99.45%, whereas the serial PSO gave

Table 2: Original LeNet-4 hyper-parameter values, lower and upper parameter boundaries for parallel PSO, along with the hyper-parameters obtained using the parallel PSO.

Layer	Orig. LeNet-4 [16]	b_l	b_u	PSO results
C_0	$n = 24$ $s_F = 5$	$n = 1$ $s_F = 2$	$n = 48$ $s_F = 10$	$n = 29$ $s_F = 8$
P_0	$s_P = 2$ $\ell = 2$	$s_P = 2$ $\ell = 2$	$s_P = 4$ $\ell = 4$	$s_P = 3$ $\ell = 2$
C_1	$n = 50$ $s_F = 5$	$n = 1$ $s_F = 2$	$n = 100$ $s_F = 10$	$n = 92$ $s_F = 5$
P_1	$s_P = 2$ $\ell = 2$	$s_P = 2$ $\ell = 2$	$s_P = 4$ $\ell = 4$	$s_P = 2$ $\ell = 1$
FC_0	$s_{FC} = 500$	$s_{FC} = 1$	$s_{FC} = 1000$	$s_{FC} = 258$

the average best accuracy equal to 99.08%, with the best fold of 99.34% (the average differences between the independent runs for each fold were not statistically important for the serial and parallel PSO, since $p = 0.0629$). In Table 2, we gather the hyper-parameter values retrieved using the parallel PSO. It can be observed, that they notably differ from the baseline LeNet-4 hyper-parameters which have been tuned by an experienced practitioner (see e.g., the number of receptive fields for C_1). Hence, our approach can further boost the performance of a DNN architecture which is carefully designed by a human.

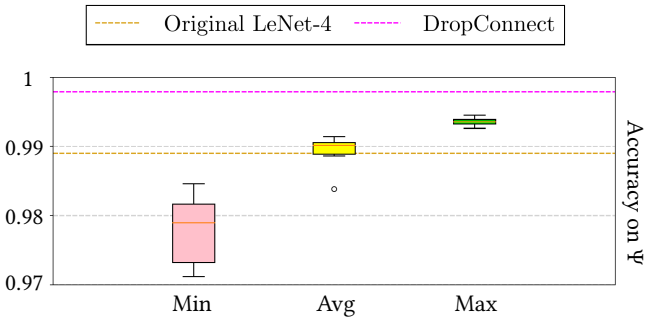

Figure 10: Minimum, average and maximum accuracy (of the best PSO particles across all folds) obtained for Ψ using the optimized LeNet-4 architecture for setup B.

In Figure 11, we observe that for the parallel PSO ($s = 6$) on setup B, the optimization of LeNet-4 over MNIST takes on average less than 3 hours. This, combined with the low discrepancy among the best reported hyper-parameter combinations, makes the parallel PSO a very affordable alternative to optimize models similar to LeNet-4, even with the additionally imposed time constraints.

In this scenario, with a significantly larger search space, the influence of the archive on the PSO evolution time is reduced as shown in Figure 12. The number of the performed DNN trainings often surpasses 50% of the total fitness evaluations across all folds. In the bottom of this figure, we observe that the average number of the archive hits remains steadily growing, as opposed to Figure 7 where it converged quite rapidly. For LeNet-4, it might be an indicator of an insufficient number of generations (the optimization ran for 100 swarm generations), reflecting that some of the particles still had momentum when the algorithm finished, and could potentially explore high-quality regions of the solution space. This might ultimately be a factor influencing the termination condition, as it would be an indicator of when the convergence has been achieved. In these circumstances, we can see the parallelism playing a bigger role in the optimization as opposed to the previous experiment.

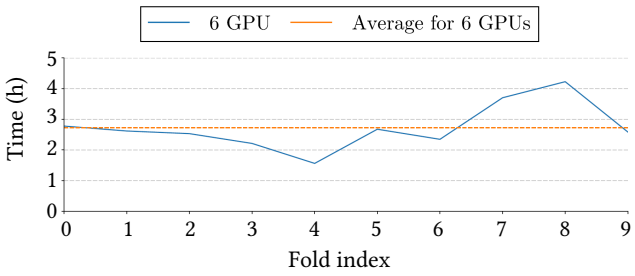

Figure 11: Total execution time per fold for LeNet-4 optimized in setup B, and the average of all executions.

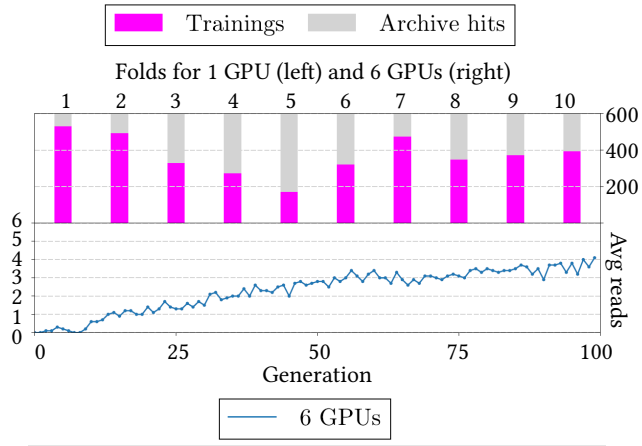

Figure 12: Proportion of combinations trained during the PSO optimization, rendered in light pink compared with the hyper-parameter values read from the archive (top). Average archive reads per generation across all folds for setups A and B (bottom).

Although all trainings to be performed in a generation can be parallelized, the time necessary to process each generation is always bounded to the longest training tasks (i.e., to the largest time of calculating the fitness of a particle in the swarm). As shown in Figure 13, the time per generation remains stable, for as long as at least one DNN training is executed by the slave nodes. In the case of many tasks running in parallel, the time per generation is still determined by the slowest-to-evaluate combination of hyper-parameters, meaning that once the other particles have finished being evaluated, the slave nodes will remain idle waiting for the last one to complete (these slaves would be assigned with other particles for evaluation if the swarm size was larger). In this scenario, our parallel setup matches the number of GPUs available with the swarm size (s = #GPUs = 6), however, this behavior would be mitigated for larger values of s. Therefore, the average time per generation in this experiment matches the average of the longest hyper-parameter combinations to be calculated across the folds.

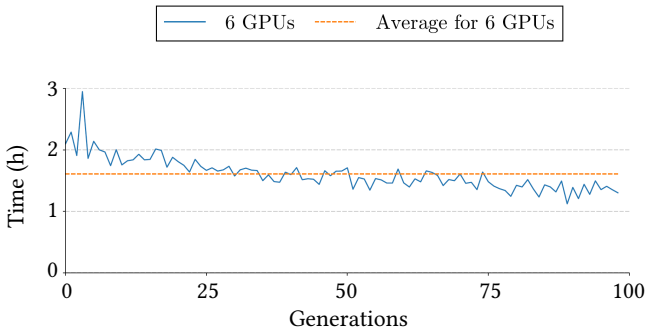

Figure 13: Total and average time per generation for LeNet-4 over MNIST on the setup B.

5 CONCLUSIONS

In this paper, we proposed a parallel version of the PSO algorithm for the hyper-parameter optimization in DNNs. The population of particles (each representing a combination of hyper-parameter values) is evolved in parallel searching for the hyper-parameters that yield the best possible classification performance of the underlying DNN model. Experiments performed on the widely-used multi-class dataset of hand-written digits (MNIST), revealed that PSO greatly benefits from the speedup delivered by parallelization of the fitness calculation. By running this fitness calculation concurrently, there is a big potential for the speedup directly proportional to the number of GPUs for large swarm sizes (where the number of particles in the swarm is larger than the number of available GPUs). The fact that this parallel evaluation is simple to implement is another remarkable property of the parallel PSO. The DNNs architectures optimized using our parallel PSO retrieved the classification performance surpassing the performance of the architecture elaborated using a human practitioner, and was very close to the best-known state-of-the-art classification accuracy.

Most of our effort has focused on comparing the performance of a serial implementation of PSO and its parallel counterpart. The convergence capabilities of our methods are significantly affected by the presence of the archive, which serves as a cache that stores the already visited hyper-parameter positions during the swarm evolution. This archive notably reduces the execution time in small search spaces, but still remains extremely useful in larger ones. A comprehensive set of illustrations provided insights into the capabilities of the parallel PSO, alongside the influence of the archive on the entire optimization process.

There are a number of common modifications to PSO that help improve its exploration and exploitation capabilities [19]. Optimizing those parameters could further improve the PSO performance alongside its ease of use. However, we focused on the analysis of the throughput derived by the introduction of parallel processing, and we ultimately tried to characterize the resulting speedup. Our future work encompasses introducing strategies to overcome the expensive evaluation of the objective function to be able to optimize notably larger DNN models. Also, we work on the adaptation and augmentation schemes which will allow for expanding the initial (quite often very simple) DNN architectures for challenging tasks in the medical imaging field, by dynamically adding new layers of different types.

ACKNOWLEDGMENTS

PRL and JN were supported by the Polish National Centre for Research and Development under the Innomed grant POIR.01.02.00-00-0030/15. LSR and JRP were supported by the MEIC from Spain/FEDER (grants TEC2015-67387-C4-3-R and TIN2014-56967-R) and by the Principality of Asturias grant FC-15-GRUPIN14-073.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Enrique Alba and Marco Tomassini. 2002. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 6, 5 (2002), 443–462.

[3] Yoshua Bengio. 2000. Gradient-Based Optimization of Hyperparameters. *Neural Computation* 12, 8 (2000), 1889–1900.

[4] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13 (2012), 281–305.

[5] François Chollet. 2015. Keras. https://github.com/fchollet/keras. (2015).

[6] Marc Claesen and Bart De Moor. 2015. Hyperparameter Search in Machine Learning. *CoRR* abs/1502.02127 (2015), 1–5.

[7] Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. 2014. Easy Hyperparameter Search Using Optunity. *CoRR* abs/1412.1114 (2014), 1–5.

[8] Sanjoy Dasgupta and David Mcallester. 2013. Regularization of Neural Networks using DropConnect. In *Proc. ICML*, Vol. 28. JMLR Conf. Proc., 1058–1066.

[9] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. 2015. Distributed Evolutionary Algorithms and Their Models. *Applied Soft Computing* 34, C (2015), 286–300.

[10] Raquel Hernández Gómez, Carlos A. Coello Coello, and Enrique Alba Torres. 2016. A Multi-Objective Evolutionary Algorithm Based on Parallel Coordinates. In *Proc. GECCO*. ACM, New York, NY, USA, 565–572.

[11] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine A. Shoemaker. 2016. Hyperparameter Optimization of Deep Neural Networks Using Non-Probabilistic RBF Surrogate Model. *CoRR* abs/1607.08316 (2016), 1–8.

[12] James Kennedy. 2010. Particle Swarm Optimization. In *Encyclopedia of Machine Learning*, Claude Sammut and Geoffrey I. Webb (Eds.). Springer, USA, 760–766.

[13] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014), 1–15.

[14] John R Koza. 1994. Genetic programming II: Automatic discovery of reusable subprograms. *Cambridge, MA, USA* (1994).

[15] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation. In *Proc. ICML*. ACM, USA, 473–480.

[16] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proc. IEEE*. 2278–2324.

[17] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, (2010).

[18] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-Objective Evolutionary Algorithms: A Survey. *Comput. Surveys* 48, 1 (2015), 13:1–13:35.

[19] Jiao Liu, Di Ma, Teng-bo Ma, and Wei Zhang. 2017. Ecosystem particle swarm optimization. *Soft Computing* 21, 7 (2017), 1667–1691.

[20] Ilya Loshchilov and Frank Hutter. 2016. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. *CoRR* abs/1604.07269 (2016), 1–8.

[21] Dougal Maclaurin, David Duvenaud, and Ryan Adams. 2015. Gradient-based Hyperparameter Optimization through Reversible Learning. In *Proc. ICML*, David Blei and Francis Bach (Eds.). JMLR Conf. Proc., 2113–2122.

[22] Jakub Nalepa and Miroslaw B locho. 2015. Co-operation in the Parallel Memetic Algorithm. *International Journal of Parallel Programming* 43, 5 (2015), 812–839.

[23] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. 2017. Particle Swarm Optimization for Hyper-Parameter Selection in Deep Neural Networks. In *Proc. GECCO*. ACM, USA, 1–8. DOI: http://dx.doi.org/10.1145/3071178.3071208

[24] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proc. NIPS*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Assoc., 2951–2959.

[25] Marco Tomassini. 1999. Parallel and distributed evolutionary algorithms: A review. (1999), 1–25. Working paper.