Los Alamos NATIONAL LABORATORY EST.1943

LA-UR-21-20936 (Accepted Manuscript)

A scalable algorithm for the optimization of neural network architectures

Li, Ying Wai
Lupo Pasini, Massiniliano
Yin, Junqi
Eisenbach, Markus

Provided by the author(s) and the Los Alamos National Laboratory (2021-04-26).

To be published in: Parallel Computing

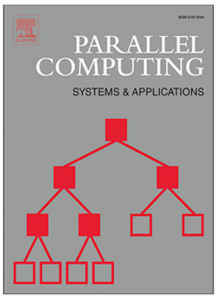
DOI to publisher's version: 10.1016/j.parco.2021.102788

Permalink to record: http://permalink.lanl.gov/object/view?what=info:lanl-repo/lareport/LA-UR-21-20936

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

RESEARCH LIBRARY

Journal Pre-proof



A scalable algorithm for the optimization of neural network architectures

Massimiliano Lupo Pasini, Junqi Yin, Ying Wai Li, Markus Eisenbach

PII: S0167-8191(21)00043-0
DOI: https://doi.org/10.1016/j.parco.2021.102788
Reference: PARCO 102788

To appear in: *Parallel Computing*

Received date : 12 November 2020
Revised date : 19 April 2021
Accepted date : 20 April 2021

Please cite this article as: M.L. Pasini, J. Yin, Y.W. Li et al., A scalable algorithm for the optimization of neural network architectures, *Parallel Computing* (2021), doi: https://doi.org/10.1016/j.parco.2021.102788.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2021 Published by Elsevier B.V.

Journal Pre-proof

A scalable algorithm for the optimization of neural network architectures

Massimiliano Lupo Pasini[a,*], Junqi Yin[b], Ying Wai Li[c], Markus Eisenbach[b]

[a]*Oak Ridge National Laboratory, Computational Sciences and Engineering Division, 1 Bethel Valley Road, Oak Ridge, TN, USA, 37831*
[b]*Oak Ridge National Laboratory, National Center for Computational Sciences, 1 Bethel Valley Road, Oak Ridge, TN, USA, 37831*
[c]*Los Alamos National Laboratory, Computer, Computational, and Statistical Sciences Division, Los Alamos, NM, 87545, USA*

Abstract

We propose a new scalable method to optimize the architecture of an artificial neural network. The proposed algorithm, called Greedy Search for Neural Network Architecture, aims to determine a neural network with minimal number of layers that is at least as performant as neural networks of the same structure identified by other hyperparameter search algorithms in terms of accuracy and computational cost. Numerical results performed on benchmark datasets show that, for these datasets, our method outperforms state-of-the-art hyperparameter optimization algorithms in terms of attainable predictive performance by the selected neural network architecture, and time-to-solution for the hyperparameter optimization to complete.

Keywords: deep learning, hyperparameter optimization, neural network architecture, random search, greedy constructive algorithms, adaptive algorithms

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

Introduction

Deep neural networks (NN) are nonlinear models used to approximate unknown functions based on observational data [1, 2, 3, 4]. Their broad applicability is derived from their complex structure, which allows these techniques to reconstruct complex relations between quantities selected as inputs and outputs of the model [5]. From a mathematical perspective, a NN is a directed acyclic graph where the nodes (also called neurons) are organized in layers. The type of connectivity between different layers is essential for the NN to model complex dynamics between inputs and outputs. The structure or architecture of the graph is mainly summarized by the number of layers in the graph, the number of nodes at each layer and the connectivity between nodes of adjacent layers.

The performance of a NN is very sensitive to the choice of the architecture for multiple reasons. Firstly, the architecture strongly impacts the prediction computed by a NN. Indeed, NN's with different structures may produce different outputs for the same input. On the one hand, structures that are too simple may not be articulate enough to reproduce complex relations. This may result in underfitting the data with high bias and low variance in the predictions. On the other hand, architectures that are too complex may cause numerical artifacts such as overfitting, leading to predictions with low bias and high variance. Secondly, the topology of a NN affects the computational complexity of the model, because an increase in layers and nodes leads to an increase in floating point operations to train the model and to make predictions. Therefore, identifying an appropriate architecture is an important step that can heavily impact the computational complexity to train a deep learning (DL) model and the final attainable predictive power of the DL model itself. However, the parameter space of NN architectures is too large for an exhaustive search. *In fact, the number of architectures grows exponentially with the number of layers, the number of neurons per layer and the connections between layers*.

Several approaches have been proposed in the literature for hyperparameter optimization (HPO) [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] with the goal to identify a NN architecture that outperforms the others in terms of accuracy. Sequential Model-Based Optimization (SMBO) algorithms [7] are a category of HPO algorithm. Examples of SMBO algorithms are Bayesian Optimization (BO) [15, 17] and its less expensive variant Tree-Parzen estimator (TPE), which rely on information available from previously trained models to guide the choice of models to build and train in following steps. The use of past information generally benefits the reduction of the number of neural networks to train in the next iterations, and provides an assessment of uncertainty by incorporating the effect of data scarcity. The

*Corresponding author
Email address: `lupopasinim@ornl.gov` (Massimiliano Lupo Pasini)

Journal Pre-proof

efficacy of the results obtained with BO is highly sensitive to the choice of the prior distribution on the hyperparameter space as well as the acquisition function to select new points to evaluate in the hyperparameter space. Another class of HPO methods is represented by genetic algorithms [18, 19, 20, 21, 22, 23] and evolutionary algorithms (EA) [24, 25], which evolve the topology of a NN by alternatively adding or dropping nodes and connections based on results attained by previous NN models. Incremental, adaptive approaches [26] and pruning algorithms [27, 28] or random dropout [29] can also be computationally convenient because they tend to minimize the number of NN models built and trained. All the SMBO, EA and incremental approaches described above adopt theoretical expedients [30, 31] to reduce the uncertainty of the hyperparameter estimate, but this comes at the price of not being scalable.

Several scalable algorithms for hyperparameter search have been proposed in the literature. Grid Search (GS), or parameter sweep, searches exhaustively through a specified subset of hyperparameters. The subset of hyperparameters and the bounds in the search space are specified manually. Moreover, the search for continuous hyperparameters requires a manually prescribed discretization policy. Although this technique is straightforwardly parallelizable, it becomes more and more prohibitive in terms of computational time and resources when the number of hyperparameters increases. Random Search (RS) [32] differs from GS mainly in that it explores hyperparameters stochastically instead of exhaustively. RS is likely to outperform GS in terms of time-to-solution [32, 33], especially when only a small number of hyperparameters affects the final predictive power of DL model. The independence of the hyperparameter settings used by GS and RS make these approaches appealing in terms of parallelization and obtainable scalability. However, both GS and RS require expensive computations to perform the hyperparameter search.

We present a scalable method to determine, within a given computational budget, the NN with minimal number of layers that performs at least as well, in terms of accuracy and time-to-solution, as NN models of the same structure identified by other hyperparameter search algorithms. The computational budget is an important aspect of the NN training for two important reasons: the available computational power and the period of time when the computational power is available. The former imposes obvious intrinsic limitations, the latter becomes important when critical decisions have to be made in a timely and accurate manner. We refer to our method as *Greedy Search for NN Architecture (GSNNA)*. Although our algorithm increments the number of hidden layers adaptively, it differs from other incremental, adaptive algorithms proposed in the literature [34, 35, 36, 37] in that our algorithm performs a *stratified* (sliced) RS restricted to one hidden layer at each iteration. This stratified RS is the most important difference between GSNNA and previous methods. The selection of the NN models is driven by the validation score, which is used as a metric to quantify the predictive performance of the DL models. Starting with the first layer, a random search is performed in parallel on various instantiations of the DL model, to determine the optimal number of neurons on each layer and the hyperparameters of the associated DL model. Random search would identify the hyperparameters for each of the instantiations, and the performance of the DL model would determine the best number of neurons and retain the hyperparameters associated with best performing model. The same sliced RS procedure is applied to the next layers. The recycling of information from previously evaluated models guarantees a fine level of *exploitation*, and the stratified RS performed at each iteration still guarantees a thorough (albeit not exhaustive) *exploration* of the objective function landscape in the hyperparameter space to prevent stagnations at local minima. *By performing a stratified RS at each iteration, our new approach retains a high level of parallelization, because the NN models can be trained concurrently at each step.*

In this work we focus on two widely used NN architectures: multi-layer perceptrons (MLP) and convolutional NN models (CNN). The performance of the HPO algorithms is evaluated using five standard datasets, each of them is associated with its specifically tailored DL model. The validation of the method will be done by comparing the efficiency of the DL model on the determined NN architecture with the efficiency of the same type of NN identified by other algorithms.

The paper is organized in five sections. Section 1 introduces the DL background. Section 2 explains our novel optimization algorithm for the architecture of NN models. Section 3 describes the computational environment where the numerical experiments are performed, the benchmark datasets, the specifics of the implementations for the each HPO algorithm considered, and the parameter setting for each HPO algorithm. Section 4 presents numerical experiments where we compare the performance of our HPO algorithm with Bayesian Optimization and Tree-Parzen Estimator. Section 5 summarizes the results presented and describes future directions to possibly pursue.

1. Deep learning background

Given an unknown function f that relates inputs x and outputs y as follows

$$y = f(x), \tag{1}$$

a *deep feedforward network*, also called *feedforward neural network* or *multilayer perceptron* (MLP) [5, 10], is a predictive statistical model that approximates the function f by composing together many different functions such that

$$\hat{f}(\mathbf{x}) = f_{L+1}(\cdots f_{\ell+1}(f_\ell(f_{\ell-1}(\ldots f_0(\mathbf{x}))))), \tag{2}$$

where $\hat{f} : \mathbb{R}^p \to \mathbb{R}^b$, and $f_\ell : \mathbb{R}^{p_\ell} \to \mathbb{R}^{p_{\ell+1}}$ for $\ell = 0, \ldots, L+1$. The goal is to identify the proper number L so that the composition in Equation (2) resembles the unknown function f in (1). The composition in Equation (2) is modeled via a directed acyclic graph describing how the functions are composed together. The number L that quantifies the complexity of the composition is equal to the number of hidden layers in the NN. We refer to the input layer as the layer with index $\ell = 0$. The indexing for hidden layers of the deep NN models starts with $\ell = 1$. In this section we consider a NN with a total of L hidden layers. The symbol p_ℓ is used to denote the number of neurons at the ℓth hidden layer. Therefore, p_0 coincides with

the dimensionality of the input, that is $p_0 = p$. The very last layer with index $L+1$ represents the output layer, meaning that $p_{L+1} = b$ coincides with the dimensionality of the output. We refer to $\mathbf{w} \in \mathbb{R}^{N_{tot}}$ as the total number of regression coefficients. Following this notation, the function f_0 corresponds to the first layer of the NN, f_1 is the second layer (first hidden layer) up to f_{L+1} that represents the last layer (output layer). In other words, deep feedforward networks are nonlinear regression models and the non-linearity is given by the composition in Equation (2) to describe the relation between predictors $\mathbf{x}$ and targets $\mathbf{y}$. This approach can be reinterpreted as searching for a mapping that minimizes the discrepancy between values $\hat{\mathbf{y}}$ predicted by the model and given observations $\mathbf{y}$.

Given a dataset with m data points, the process of predicting the outputs for given inputs via an MLP can thus be formulated as

$$\hat{\mathbf{y}} = F(\mathbf{x}, \mathbf{w}), \tag{3}$$

where the operator $F : \mathbb{R}^{p_0} \times \mathbb{R}^{N_{tot}} \to \mathbb{R}^b$ is

$$\begin{aligned} F(\mathbf{x}, \mathbf{w}) = \varphi_{L+1}\Big(\sum_{k_L} w_{k_{L+1}k_L} \varphi_L \Big(\sum_{k_{L-1}} w_{k_L k_{L-1}} \varphi_{L-1}\Big(\dots \\ \dots \varphi_1 \Big(\sum_{i=1} w_{k_1 i} x_i \Big) \Big) \Big) \Big), \end{aligned} \tag{4}$$

where φ_ℓ ($\ell = 1, \dots, L+1$) are activation functions used to generate non-linearity in the predictive model. Using the matrix notation for the weights connecting adjacent layers as

$$W_{\ell,\ell-1} \in \mathbb{R}^{p_\ell \times p_{\ell-1}}, \tag{5}$$

we can rewrite (4) as

$$F(\mathbf{x}, \mathbf{w}) = \varphi_{L+1}\Big(W_{L+1,L} \Big(\varphi_L \Big(\dots \Big(\varphi_1 \Big(W_{1,0} \mathbf{x} \Big) \Big) \Big) \Big) \Big). \tag{6}$$

The composition of the activation functions φ_ℓ with the tensor products using matrices $W_{\ell+1,\ell}$ at the ℓth layer corresponds to the f_ℓ used in Equation (2). The notation in (6) highlights that N_{tot} is the total number of regression weights used by the NN. This value must account for all the entries in $W_{\ell,\ell-1}$'s matrices, that is

$$N_{tot} = \sum_{\ell=1}^{L+1} p_\ell p_{\ell-1}. \tag{7}$$

If the target values are continuous quantities, the very last layer φ_{L+1} is usually chosen to be linear, i.e., the identity function. If the target values are categorical, then φ_{L+1} is usually set to be the logit function. If the number of hidden layers is set to $L = 0$ and φ_1 is set to be the identity function, then the statistical model becomes a classical linear regression model. If the number of hidden layers is set to $L = 0$ and φ_1 is set to be the logit function, then the statistical model becomes a logistic regression model.

In order to exploit local correlations in the data, convolutional kernels can be composed with the activation functions φ_i. Convolution is a powerful mathematical tool that models local interactions between data points. As such, convolution uses the same set of regression coefficients to model local interactions across the entire data instead of using several sets of regression coefficients, one specific for each neighbourhood as a standard MLP architecture would require. The use of the convolution thus significantly reduces the dimensionality of the coefficients needed in DL models to reconstruct local features in regularly structured data. Well known examples of data that respect this geometrical properties are images. NN models that exploit the data locality for the feature extraction are called *Convolutional Neural Networks* (CNN) [38, 39, 40] and they are characterized by a sparse connectivity or sparse weights that stems from the sparse interaction between data. In essence CNN are the nonlinear generalization of kernel regression and they inherit from the linear case the advantages of replacing dense matrix multiplication with sparse matrix multiplications. This benefits the computation by reducing the number of FLOPS required to perform matrix multiplications, and reduces the memory requirement to store the regression weights.

2. Adaptive selection of the number of hidden layers

The goal of our novel HPO algorithm is to determine, within a given computational budget, the NN with minimal number of layers that performs at least as well on training datasets, in terms of accuracy and time-to-solution, as NN models of the same structure identified by other hyperparameter search algorithms. The HPO is performed over a set of hyperparameters which differs according to the type of NN architecture considered. For MLP models, the HPO is performed over the number of hidden layers, the number of neurons per layer, the type of nonlinear activation function at each hidden layer and the batch size used to train the model with a first-order optimization algorithm. For CNN models, the number of neurons is replaced by the number of channels. In addition, the convolutional kernel, the dropout rate and the pooling are optimized as well. In order for the HPO procedure to be applied, the region of the hyperspace explored must be bounded to guarantee that the exploration is restrained within a computational budget for the number of layers and the other NN hyperparameters.

The result of the procedure is dataset dependent, in that it aims to identify a customized neural network architecture that well predicts the input-output relation for the dataset at hand. The dataset is split into a training set, a validation set and a test set. The training portion is used to train the instantiated NN models. The performance of the DL models over the validation set is used to associate the model with a score, which is used to compare the performance of the NN instantiated. The test set is used to quantify the predictive performance of the finally selected NN model by computing the test score. We refer to Section 4.1 for details about the metrics used to measure the performance of a NN.

The pseudo-code that describes GSNNA is presented below, in Algorithm 1. The method starts by performing RS over NN models with one hidden layer and it selects the NN that attains the best predictive performance over the validation portion of the dataset. The random search identifies the hyperparameters for each of the instantiations and the performance of the deep learning model determines the best number of neurons and the

hyperparameters associated with best performing model on the respective datasets to retain. The procedure continues by freezing the number of neurons and the hyperparameters in the previous hidden layers every time a new hidden layer is added, and the sliced RS is performed only on the hyperparameters of the last hidden layer in the architecture. This iterative procedure proceeds until either the validation score reaches a prescribed threshold or the maximum number of hidden layers is reached. An illustration that explains how GSNNA proceeds is shown in Figure 1. The number of neurons needed may vary from layer to layer in order for a NN architecture to attain a desired accuracy. It is thus possible that the NN may have to alternatively expand and contract across the hidden layers to properly model the nonlinear relations between input and output data. GSNNA allows this, as the number of neurons at each selected through a stratified RS may vary for each hidden layer.

Algorithm 1: Greedy Search for Neural Network Architecture (GSNNA)

Input:

- L = maximum number of hidden layers
- N_{max_nodes} = maximum number of nodes (neurons) per layer
- $score_{\text{threshold}}$ = threshold on the final performance prescribed
- *model_eval_iter* = number of model evaluations per iteration

Output: best_model
Set number of hidden layers $\ell = 1$;
Set best_model as linear regression (for regression problems) or logistic regression (for classification problems);
Compute *score*;
while $score < score_{threshold}$ & $\ell \leq L$ **do**
  Build *model_eval_iter* NN models with ℓ hidden layers each;
  Set number of nodes and activation functions for first $(\ell - 1)$ hidden layers as in best_model;
  Perform random search for number of nodes in the last hidden layer and for the remaining hyper-parameters;
  Select best_model as the NN with best performance;
  Retrieve best_model and store info about number of nodes and activation functions per layer;
  $\ell = \ell + 1$;
end
return *best_model*

The stratified RS is the most important difference between GSNNA and previous methods. The main contrast of GSNNA with respect to previous methods is the greedy approach adopted in increasing the number of hidden layers. As more hidden layers are added to the NN architecture, the predictive power of the model increases, but with this also the computational cost for training. Previous methods treat the number of hidden layers as any other hyperparameter, and the methods sometimes construct expensive neural networks at intermediate steps, and these NN are later discarded in favor of smaller ones. By performing a greedy approach on the number of hidden layers, GSNNA avoids this type of extreme situations where very expensive NN are trained and discarded through intermediate steps, and this favors a lower computational cost per iteration. The validation of the method will be shown by comparing the efficiency of the DL model on the determined NN architecture with the efficiency of the same type of DL model identified by other algorithms.

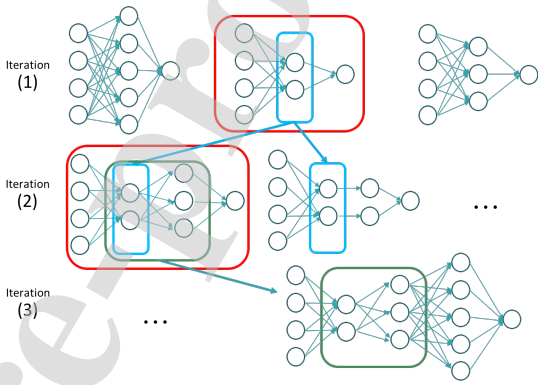


Figure 1: Illustration of the Greedy Search for Neural Network Architecture (GSNNA). The illustration explains how the architecture of the NN is enriched at each iteration. The NN models built at iteration (1) have only one hidden layer and the number of neurons inside the hidden layers is chosen via RS. Every NN is trained and the predictive performance over the validation set is measured. The NN with the best validation score is selected (circled in red). If the attained accuracy meets the requirements prescribed by the user, the algorithms stops and returns the selected NN. Otherwise, the hyperparameters of the first hidden layer are transferred to iteration (2). The NN models built at iteration (2) have the same number of neurons in the first hidden layers as the best NN from iteration (1), whereas the number of neurons at the second hidden layer is chosen with another stratified RS. The NN models are trained and the validation scores from each NN are collected. The NN with the best predictive performance is chosen (circled in red). If the performance meets the requirements, the algorithms stops and returns the selected NN. Otherwise, the information about the numbers of neurons in the first and second hidden layers are transferred to iteration (3), so that another stratified RS takes place on the number of neurons inside the third hidden layer.

2.1. Reduction of dimensionality in the hyperparameter search

Transferring information from smaller to bigger NN models across successive iterations and restricting the RS to the hyperparameters associated only with the last hidden layers reduces the dimension of the hyperparameter space to explore. In this section we compare the dimensionality (number of elements in a set) of the hyperparameter space explored by a standard HPO algorithm (e.g. GS, RS, SMBO, EA) with the dimensionality of the hyperparameter space explored by GSNNA.

Denote the maximum number of neuron per layer with N_{max_nodes} and the maximum number of hidden layers with L. The number of hidden layers and the number of neurons per layer are hyperparameters that affect the structure of the NN

Journal Pre-proof

models, whereas all the other hyperparameters affect the training of the DL model. Because GSNNA differ from state-of-the-art HPO algorithms by the way the number of hidden layers are optimized, this is the only factor that determines a change in dimensionality of the hyperparameter space. The stratified RS in GSNNA allows us to avoid the *curse of dimensionality*, because the number of NN architectures to span at each iterations decreases from $N^L_{max\_nodes}$ (as it is for a standard HPO algorithm) to N_{max_nodes}. The reduced dimensionality of the hyperparameter space leads also to a reduction of the uncertainty over the estimated attainable predictive performance. This is shown in the numerical experiments in Section 4, where the accuracy attained by the NN models selected from multiple runs of GSNNA has narrower confidence intervals than the ones obtained with BO and TPE, indicating that the estimates obtained with GSNNA are more reliable.

2.2. Computational complexity of GSNNA

Let us refer to C as the number of independent model evaluations performed in one iteration of an HPO algorithm, and L the number of HPO iterations performed. The computational complexity of one iteration of GSNNA is $O(C)$ (and hence $O(CL)$ for the whole algorithm), because the algorithm compares the predictive performance of C models and selects the best one to proceed to the next iteration. To put this value in perspective, we remind the reader that the computational complexity of one iteration of BO is cubic both in the number of independent model evaluations and in the number of iterations performed, that is $O((CL)^3)$, and the computational complexity of one iteration of TPE is cubic only in the number of independent model evaluations, that is $O(C^3)$. In terms of computational complexity, GSNNA thus provides a significant improvement with respect to BO and TPE, because the computational complexity per iteration is constant with respect to the iteration count, and the computational complexity of one iteration of HPO is reduced from cubic to linear. This benefit makes GSNNA appealing for scaling purposes with large values of independent model evaluations C. We also remind the reader that the independent model evaluations in each iteration can be performed concurrently, as we did in the numerical experiments described in Section 4 of this work.

3. Algorithm implementation

In this section we describe the computational environment where the numerical experiments were performed, the specifics of the implementations for each of the HPO algorithms considered, the benchmark datasets used and the parameter setting for each HPO algorithm.

3.1. Hardware description

The numerical experiments were performed using Summit [41], a supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory. Summit has a hybrid architecture; each node contains two IBM POWER9 CPUs and six NVIDIA Volta V100 GPUs all connected together with NVIDIA's high-speed NVLink. Each node has over half a terabyte of coherent memory (high bandwidth memory + DDR4) addressable by all CPUs and GPUs plus 800 GB of non-volatile RAM that can be used as a burst buffer or as extended memory. To provide a high rate of I/O throughput, the nodes are connected in a non-blocking fat-tree using a dual-rail Mellanox EDR InfiniBand interconnect.

Name of dataset	Nb. attributes	Nb. data points
Eggbox	2	4,000
Graduate admission	7	400
Computer hardware	9	209
Phishing websites	29	11,055
CIFAR-10	-	60,000

Table 1: Description of the datasets.

3.2. Dataset description

The datasets used are standard benchmark datasets in machine learning, open source and accessible to everyone, and guarantee reproducibility of the results presented. The datasets used for the numerical experiments of this section are summarized in Table 1. The dataset `Eggbox` is artificially constructed by evaluating the function $f(x, y) = [2 + \cos(x/2) * \cos(y/2)]^5$ across 4,000 points in the domain square $[0, 2\pi]^2$ and it is used as a regression problem. The `Graduate admission` dataset [42] is a regression problem that relates the chances of a student's admission to GRE score, TOEFL score, university rating, and other performance metrics. The `Computer hardware` dataset [43, 44] is a regression problem that describes the relative CPU performance data in terms of its cycle time, memory size, and other hardware properties. The `Phishing website` dataset [44] is a classification problem that describes the properties of different websites and classifies them as authentic or fake. The `CIFAR-10` dataset [45] requires solving a classification problem to classify object images into ten categories. The numerical experiments presented in this section are split between the use of MLP and CNN models. The choice of one type of architecture over the other is dictated by the structure of the dataset used to train the NN models. MLP models are used on the `Eggbox`, `Graduate admission`, `Computer hardware`, and `Phishing website` datasets, whereas CNN models are used for the `CIFAR-10` dataset.

3.3. Training, validation, and test data

The datasets are split in three components: the training set, the validation set, and the test set. The training set is used to train every instantiated DL model, the validation set is used to select the best performing model at each iteration and the test set is used at the end to measure the predictive power of the NN selected by each HPO algorithm. For the datasets `Eggbox`, `Graduate admission`, `Computer hardware`, and `Phishing website`, the test set is 10% of the original dataset, the remaining portion is partitioned into training and validation in the percentage of 90% and 10% respectively. For classification prob-

Journal Pre-proof

lems, a stratified splitting is performed to ensure that the proportion between classes is preserved across training, validation, and test sets. The partitioning between training/validation set and test set for the `CIFAR-10` dataset is performed as suggested by the online sources where the datasets can be downloaded 6.

The optimizer used to train the model is the Adam method [46] with an initial learning rate of 0.001. We highlight that the number of epochs to train a neural network is different from the number of iterations performed by the hyperparameter optimization algorithm. In fact, the number of epochs is related to the computation needed to perform every single model evaluation. For all HPO methods (GSNNA, TPE, BO), the maximum number of epochs used to train the neural networks is set to be equal to n, i.e. the number of samples in the training set for each dataset. The actual number of epochs does not necessarily have to be equal to the number n of points in the dataset. If the training is achieved before n, an early stopping is in place to finish the training. If the number of epochs reaches n, that means that the neural network still benefits from the training. Of course, if n is too large, which happens for very large datasets, this may impose unwanted burden on the execution time. But that can be mitigated by trying to find a balance between optimal training and training time.

The cost to train a neural network depends on both the size of the dataset, and on the size of the neural network itself. The larger the neural network and the datasets, the longer it takes to train the neural network. The longer time to train a larger neural network on a larger dataset would translate into an increased computational time to perform every single model evaluation, and this would impact the total time to solution for all the HPO algorithms used.

We also want to point out that the size of a neural network should correlate with the complexity of the relation between inputs and outputs. Having a larger dataset does not necessarily imply needing a larger neural network. For example, one may have infinitely many points aligned on a straight line. The dataset is large, but the complexity required for the predictive model to capture the trend is still very low.

3.4. Setting of the hyperparameter space

The hypercube that delimits the hyperparameter search is defined so as to restrict the hyperparameter search within an affordable computational budget. Due to the computational budget constraint, we limit the maximum number of layers L to 5. The number of neurons (or channels) per layer spans from 1 to the highest integer smaller than $\sqrt{n}$, where n is the number of sample points. The choice of $\sqrt{n}$ as the upper bound of the number of neurons per layer is a common practice adopted in DL to avoid overfitting. The set of activation functions is made of the sigmoid function (denoted as `sigmoid` in the Tables), the hyperbolic tangent (`tanh`), the rectified linear unit function (`relu`) and the exponential linear unit function (`elu`). The kernel size for CNN architecture spans between 2 and 5. The discrete range for the batch size spans from 10 to the closest integer to $\frac{n}{10}$. Also choosing $\frac{n}{10}$ as maximum size of data batches is a reasonable recommendation adopted by DL practitioners to cap the computational cost of each training iteration. The range of search for each hyperparameter is fixed in every HPO algorithm used for the study. Tables 2 and 3 contain a description of the hyperparameters optimized for MLP and CNN architectures with the ranges spanned for each hyperparameter during the optimization.

Hyperparameter	**Search range**
Number of hidden layers	{1,2,3,4,5}
Number of neurons per layer	$[1, \sqrt{n}]$
nonlinear activation function	{`relu`, `sigmoid`, `tanh`, `elu` }
batch size	$[10, \frac{n}{10}]$

Table 2: Hyperparameters optimized for MLP architectures. The value n refers to the size of the dataset.

Hyperparameter	**Search range**
Number of hidden layers	{1,2,3,4,5}
Number of channels per layer	$[1, \sqrt{n}]$
Dropout rate	[0,1]
Pooling	{1,2}
nonlinear activation function	{`relu`, `sigmoid`, `tanh`, `elu` }
batch size	$[10, \frac{N}{10}]$

Table 3: Hyperparameters optimized for CNN architectures. The value n refers to the size of the dataset.

3.5. Setting of the hyperparameter search algorithms

The code to perform GSNNA is implemented in `python 3.5`, and the NN models are built using `Keras.io` [47] which calls `Tensorflow 2.0` backend. The training of the NN models is performed using the GPUs on Summit by calling `cudadnn 9.0` for tensor algebra operations. We compare the GSNNA described in this paper with the TPE and BO. The version of GSNNA that we implemented performs concurrent model evaluations for the RS at each step with a distributed memory parallelization paradigm that uses `mpi4py` [48]. The version of TPE and BO used are provided by the `Ray Tune` library [49] through the routines named `HyperOptSearch` and `BayesOptSearch` respectively. The version of Ray Tune used is 0.3.1. As to `BayesOptSearch`, the utility function is set to `utility_kwargs="kind": 'ucb',"kappa": 2.5,"xi": 0.0`. For both `HyperOptSearch` and `BayesOptSearch`, the model selection and evaluations are scheduled using the asynchronous version of HyperBand [50] called `AsyncHyperBandScheduler`. The time attribute for the scheduler is the training iteration and the reward attribute is the validation score of the NN. The validation score is also used as the stopping criterion of the HPO algorithm. Additional parameters for RayTune's TPE and BO not mentioned here have been left to default value. Our proposed method, GSNNA, is at its first implementation, whereas the RayTune library used to perform HPO with TPE and BO has underwent multiple stages of implementation optimization. Therefore, our comparison between GSNNA, TPE, and BO does not advantage GSNNA over the other HPO algorithms in terms of implementation.

Journal Pre-proof

4. Numerical results

In this section we present numerical experiments for the five benchmark datasets described above, and we focus on the best suited type of neural network structure for each one of the selected datasets. Our numerical experiments compare the performance of GSNNA against BO and TPE in terms of final attainable accuracy of the selected NN architecture and time-to-solution to complete the hyperparameter search.

Numerical tests described in this section focus on weak scaling, meaning that the performance of HPO algorithms is monitored for increased numbers of concurrent model evaluations, with each concurrent model evaluation mapped to a separate MPI process and a separate GPU to train, and the predictive performance of the model is assessed. Strong scaling tests are not included in the discussion for the following reasons. For applications such as the ones considered in this paper, strong scaling requires fixing the number of concurrent model evaluations and progressively increase the computational resources made available for each model evaluation. In our methodology, there is a one-to-one mapping between concurrent model evaluations and GPUs. When the total number of GPUs is less than the concurrent models, the strong scaling boils down to the scaling of the job scheduler, which is outside the scope of this work. When the total number of GPUs is more than the concurrent models, this would translate to using multiple GPUs to perform a single model evaluation instead of using one GPUs as currently done in the work. In the deep learning community, this approach is known as model parallelization. Model parallelization would accelerate the model evaluations and it would equally apply to all the three methods TPE, BO, and GSNNA. However, model parallelization would not accelerate the execution of the hyperparameter optimization algorithms themselves. Therefore, the comparison of TPE, BO, and GSNNA would not differ from the ones presented in this paper in relative terms. Moreover, the small size of the neural networks and the small size of the benchmark datasets used in this work does not justify model parallelization; strong scaling would only bring marginal benefits on the acceleration of model evaluations.

4.1. Comparison for predictive performance and computational time

The first set of numerical experiments compares the predictive power of the GSNNA with TPE and BO. The metric used to quantify the predictive performance of a NN for regression problems is the R^2 score defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{m}(y_i - \bar{y}_i)^2} \tag{8}$$

where y_i are the observations for m data points in the test set, $\hat{y}_i$ are the predictions obtained with the DL model over the test set and $\bar{y}_i$ is the sample mean of the data points over the test set. The metric used to quantify the predictive performance of a NN used for classification problems is the $F1$ score defined as

$$F1 = 2\frac{PPV \cdot TPR}{PPV + TPR}, \tag{9}$$

where $PPV = \frac{\text{true positives}}{\text{true positives+false negatives}}$ is the *precision* or *positive predicted value* and $TPR = \frac{\text{true positives}}{\text{positives}}$ is the *sensitivity*, *recall*, *hit rate*, or *true positive rate*.

For the datasets that require the use of MLP architectures, the number of concurrent model evaluations per iteration is set to 10, 25, 50, 75, and 100 for all the three HPO algorithms. For the `CIFAR-10` dataset that requires the use of CNN architectures, the number of concurrent model evaluations per iteration is set to 150, 300, 450, and 600 to cope with a larger number of hyperparameters to tune. The maximum number of iterations is set to 5 for all the three HPO algorithms and the stopping criterion imposes a threshold on the R^2 score and $F1$ score equal to 0.99.

To guarantee a fair comparison between the different HPO algorithms, the implementations of the three HPO algorithms make use of the same number of concurrent model evaluations, and each implementation of the HPO algorithms maps every concurrent model evaluation to a separate GPU. However, the complexity of (and thus the cost to train) each model per iteration varies according to the specific architectures that the HPO algorithms select at each iteration. Since different HPO algorithms select different architectures to construct and evaluate, this can lead to different computational times. Because Summit has six GPUs per compute node, the total number of Summit nodes used in a numerical experiment is equal to the least integer greater than or equal to the concurrent model evaluations divided by 6.

Figures 2, 3, 4, and 5 correspond to the test cases with MLP models. In these figures, the figures on top show the scores obtained on the test set of the selected MLP model, and the figures at the bottom show the time-to-solution in wall clock seconds. The performance is reported for each hyperparameter search algorithm, averaging over 10 runs with 95% confidence intervals both for the mean value of the predictive performance and for the mean value of the time-to-solution.

The experiments with the `Eggbox` dataset exhibit better results for GSNNA with respect to TPE and BO in terms of predictive power achieved by the selected NN. Moreover, we notice that the confidence band for GSNNA narrows as the number of concurrent evaluations increases. This happens because the inference on the attainable predictive performance becomes more accurate with a higher number of random samples for the stratified RS. A different trend is shown for the confidence band of TPE and BO. In this case, the confidence band does not become narrower by increasing the number of concurrent model evaluations. This highlights the benefit of using a stratified RS in GSNNA: the uncertainty of the random optimization is bounded by reducing the dimensionality of the search space.

In terms of scalability, we notice that GSNNA has a flat weak scaling curve, whereas BO and TPE significantly increase the computational time-to-solution with an increased number of concurrent model evaluations. Although BO and TPE finish in less time than GSNNA for 10 and 50 model evaluations, the final attained accuracy is significantly lower than the one obtained with GSNNA. This indicates that GSNNA better explores the hyperparameter space.

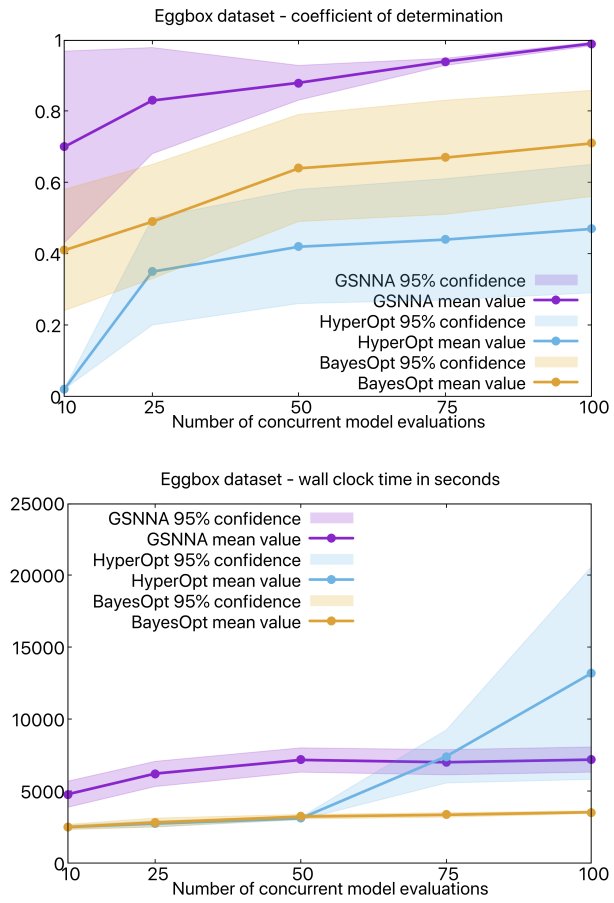


Figure 2: `Eggbox` dataset. Comparison between Greedy search, HyperOptSearch and BayesOptSearch for test cases with MLP architectures. The graph at the top shows the performance obtained by the model selected by the hyperparameter search on the test set. The graph at the bottom shows a comparison of the computational times.

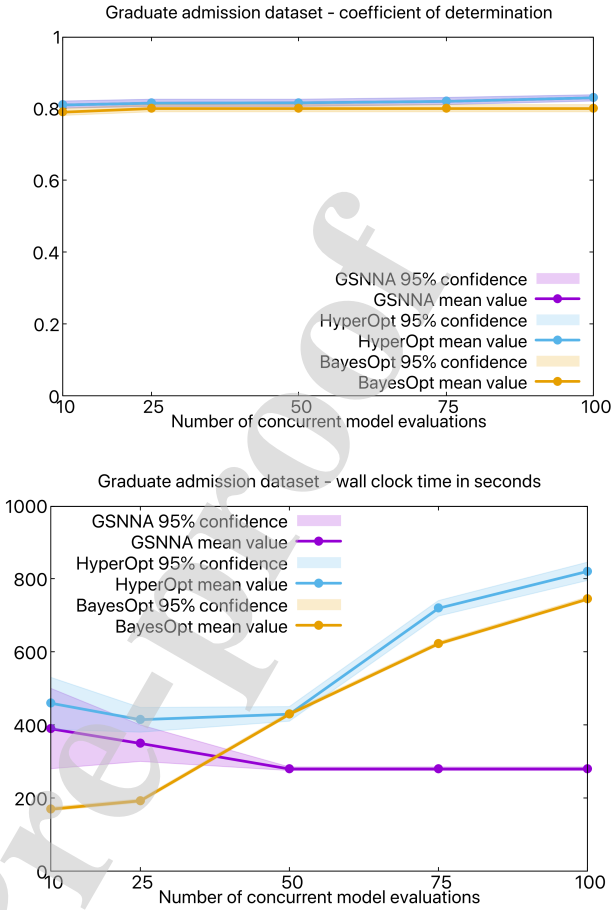


Figure 3: `Graduate admission` dataset. Comparison between Greedy search, HyperOptSearch and BayesOptSearch for test cases with MLP architectures. The graph at the top shows the performance obtained by the model selected by the hyperparameter search on the test set. The graph at the bottom shows a comparison of the computational times.

Similar results in terms of final attainable accuracy and scalability have been obtained for `Graduate admission`, `Computer hardware` and `Phishing websites` datasets. Although different values for the tuning parameter of `BayesOptSearch` have been tested on the datasets considered in this paper, we noticed that the performance of `BayesOptSearch` on these datasets did not significantly change. We also noticed that for the `graduate admission` dataset and the `phishing` dataset, some HPO algorithms reduce the total time of the search for an increased number of concurrent model evaluations, and this goes against an intuitive reasoning. To better understand this phenomenon, we note that the number of concurrent models impacts the computational time in two ways: a higher number of concurrent model evaluations makes it likely to identify a network that attains a desired accuracy faster, but it also needs more time to coordinate the model evaluations between each other. Whether one of these two factors prevails over the other can results in either a reduction or an increase in the total computational time.

The results for the `CIFAR-10` dataset using CNN in Figure 6 show that GSNNA outperforms both TPE and BO algorithms in terms of best attainable predictive performance and computational time. The F1-score is more appropriate than the accuracy (percentage of data points correctly classified) to measure the predictive performance of neural networks for classification purposes in case of class imbalance [51]. However, the accuracy is still the mostly used metric to report the predictive performance of a model on some benchmark dataset such as `CIFAR-10`. In order to facilitate the comparison with other results published in the literature, we also report the accuracy for `CIFAR-10`.

Comparing the architecture selected by GSNNA with state-of-the-art architectures customized for `CIFAR-10` [52], we see that the predictive performance of our architecture has a test error of about 9%, whereas customized architectures currently provide error below 0.1%. In view of this gap between the performance we obtained on `CIFAR-10` with respect to other results published in the literature, we emphasize that the goal of our research is to build an automatic selection of hyperparameters that is as agnostic as possible about the specifics of the dataset at hand. This makes the hyperparameter search more challenging, and the attainable accuracy is generally lower than

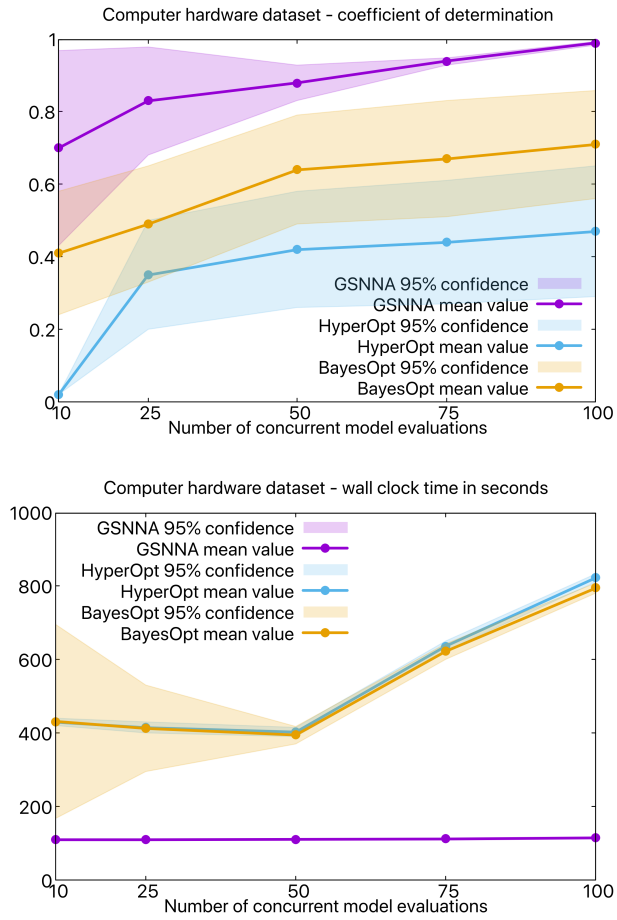


Figure 4: `Computer hardware` dataset. Comparison between Greedy search, HyperOptSearch and BayesOptSearch for test cases with MLP architectures. The graph at the top shows the performance obtained by the model selected by the hyperparameter search on the test set. The graph at the bottom shows a comparison of the computational times.

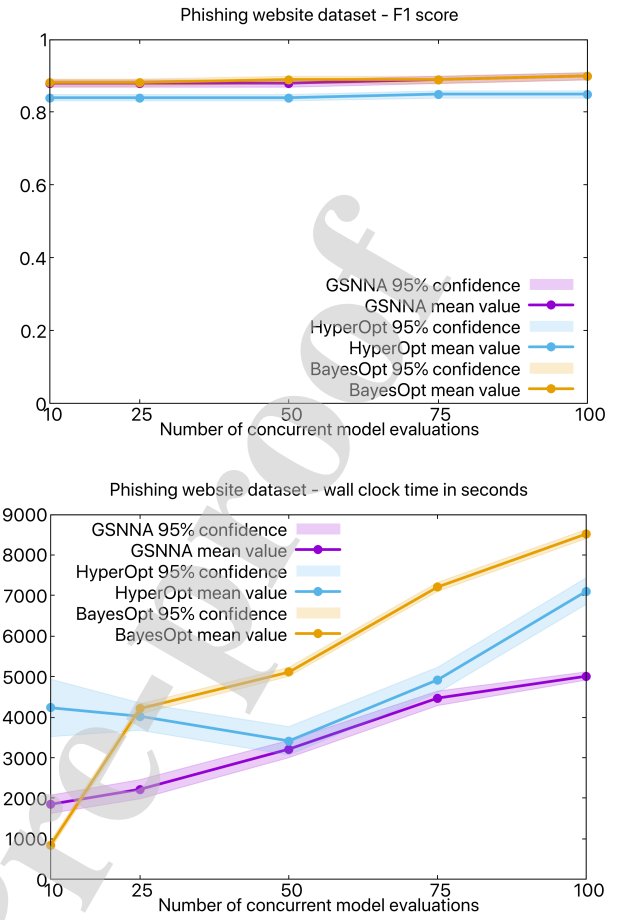


Figure 5: `Phishing` dataset. Comparison between Greedy search, HyperOptSearch and BayesOptSearch for test cases with MLP architectures. The graph at the top shows the performance obtained by the model selected by the hyperparameter search on the test set. The graph at the bottom shows a comparison of the computational times.

the one obtained with customized approaches. Recent results obtained by other researchers [53] show a test error around 12% when a Bayesian approach is used to optimize the architecture of a neural network for the `CIFAR-10` dataset, and this is in line with the results we present here.

4.2. Sensitivity of GSNNA with respect to the number of concurrent model evaluations

In Figure 7 we show the performance obtained with GSNNA on the `Eggbox` dataset and the `Computer hardware` dataset as a function of the number of hidden layers for different numbers of concurrent model evaluations (10, 50, and 100). For both experiments it is clear that the use of a small number of concurrent model evaluations leads to significant fluctuations in the score, as the stratified RS does not explore enough architectures for a fixed number of hidden layers. A progressive increase in the concurrent model evaluations leads to a better inference. This happens because an exhaustive exploration of the stratified hyperparameter space reduces the uncertainty in the attainable best performance of the model. Moreover, a sufficient exploration of the stratified hyperparameter space enables us to highlight the dependence between the maximum attainable performance of the NN and the total number of hidden layers. Indeed, the examples displayed in Figure 7 confirm that nonlinear input-output relations can benefit from a higher number of hidden layers.

In Figure 8 we present a similar analysis using CNN for the `CIFAR-10` dataset. In this case, the number of concurrent model evaluations considered is 150, 300, and 600. The scalability tests for the `CIFAR-10` dataset use a higher number of concurrent model evaluations with respect to the previous datasets because there are more architectural hyperparameters to tune in CNN than in MLP models, as described also by a comparison between Tables 2 and 3. Different from the previous numerical examples, increasing the number of concurrent model evaluations does not benefit the identification of a better performing architecture for the `CIFAR-10` dataset, but a progressive increase of the number of hidden layers still leads to a progressive gain in attainable accuracy.

Journal Pre-proof

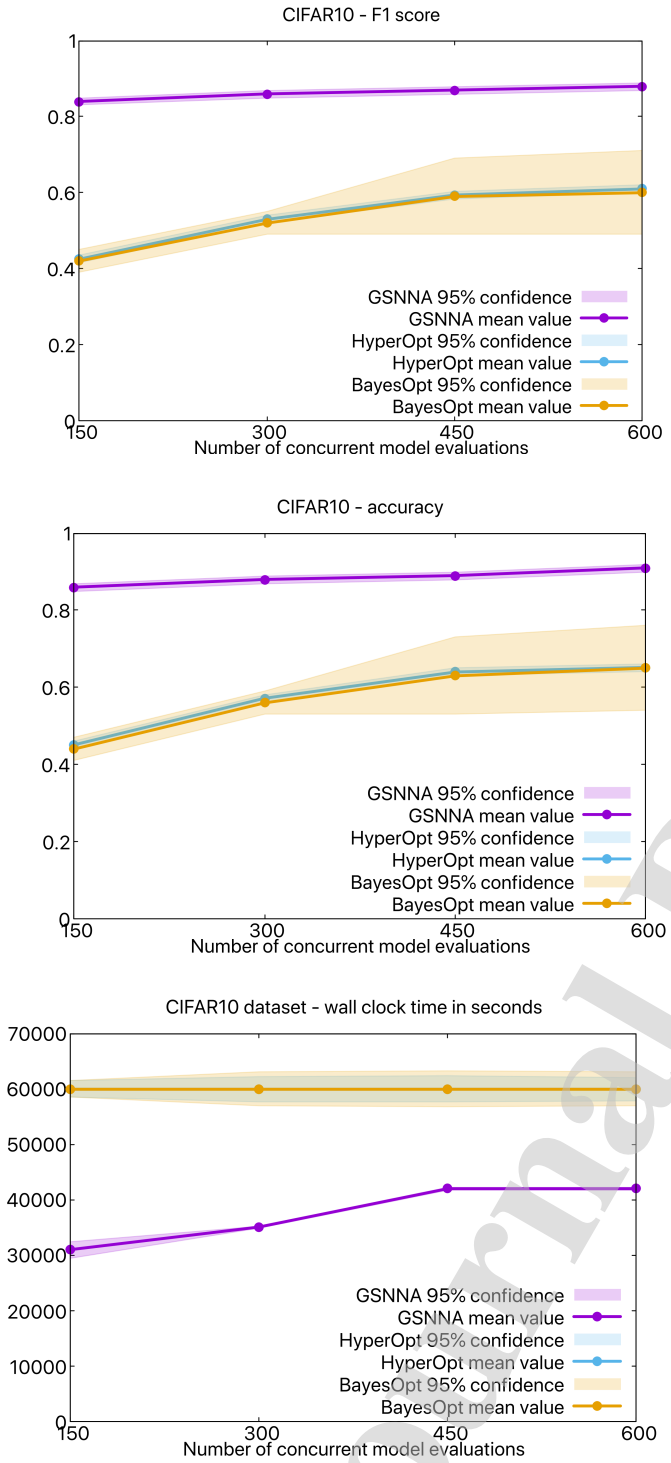


Figure 6: Comparison between GSNNA, HyperOptSearch and BayesOptSearch for test cases with CNN architectures. The comparison is performed for the CIFAR10 dataset. The graph on top shows the performance obtained by the model selected by the hyperparameter search on the test set in terms of F1 score. The graph in the center shows the performance obtained by the model selected by the hyperparameter search on the test set in terms of accuracy. The graph at the bottom shows the computational time.

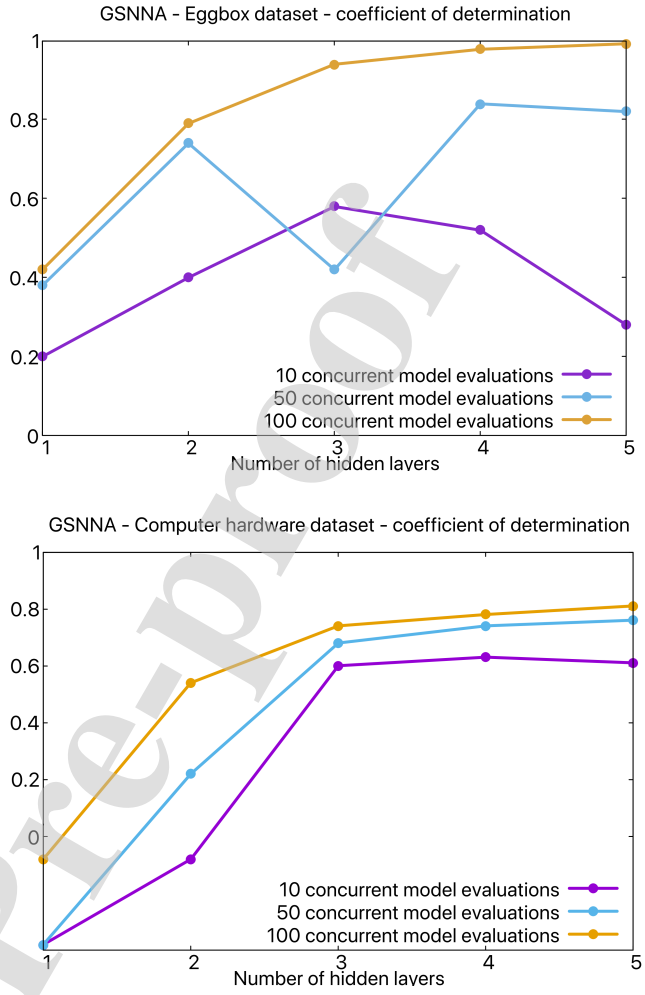


Figure 7: Greedy Search for Neural Network Architecture (GSNNA). Coefficient of determination expressed in terms of the number of hidden layers for `Eggbox`, `Computer hardware` datasets using 10, 50, and 100 concurrent model evaluations. Results are shown for a single run.

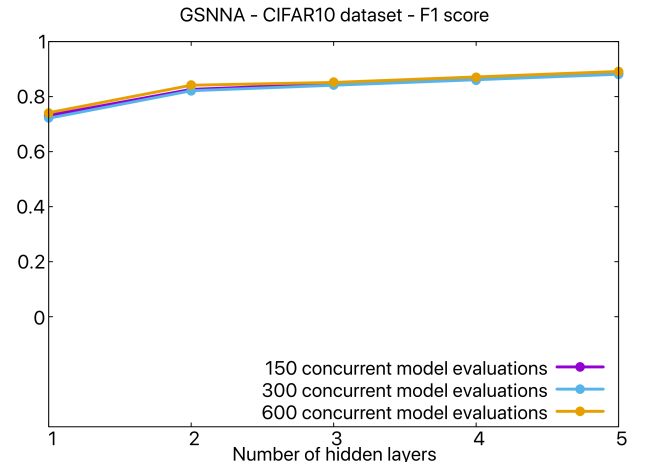


Figure 8: Greedy Search for Neural Network Architecture (GSNNA). Coefficient of determination expressed in terms of the number of hidden layers for `CIFAR10` dataset using 150, 300, and 600 concurrent model evaluations. Results are shown for a single run.

Journal Pre-proof

5. Concluding remarks and future developments

GSNNA aims to determine in a scalable fashion, and within a given computational budget, the NN with minimal number of layers that performs at least as well as NN models of the same structure identified by other hyperparameter search algorithms. The algorithm adopts a greedy technique on the number of hidden layers, which can benefit the reduction of computational time and cost to perform the hyperparameter search. This makes the algorithm not only appealing, but sometimes strongly compelling when computational and memory resources are limited, or when DL driven decisions have to be performed in a timely manner. The recycling of hidden layer configurations disregards an exponential number of architectures in the hyperparameter space. However, having a smaller search space makes the optimization a much more tractable problem with a significant reduction in computational complexity. Moreover, our numerical results show that this does not compromise the final attainable accuracy of the model selected by the optimization procedure.

CIFAR-10 is the largest tested dataset, with 60000 images at 32x32 resolution. ImageNet or the Open Images Dataset have more than a million images and are commonly evaluated at 256x256. At the same efficiency, this could take 1000x more time, and CIFAR-10 already takes about 8 hours. This is a limitation to the applicability of the method. However, the proposed research aims at improving scalability of hyperparameters search algorithms with a *constrained computational budget*. Therefore, while the method is illustrated on modest-size datasets and neural networks, it has promise for implementations on larger datasets and correspondingly larger neural networks under the same computational budget constraints.

For future developments we aim to extend the study to different types of architectures other than multilayer perceptrons and CNN, such as residual neural networks (ResNet), recurrent neural networks (RNN) and long short-term memory neural networks (LSTM). We will also use GSNNA for specific problems by selecting customized attributes other than the score for the HPO, and we will conduct an uncertainty quantification analysis to estimate the sensitivity of the inference on the hyperparameters with respect to the dimension of the hyperparameter space and the number of concurrent model evaluations.

Acknowledgements

Massimiliano Lupo Pasini thanks Dr. Vladimir Protopopescu for his valuable feedback in the preparation of this manuscript and three anonymous reviewers for their very useful comments and suggestions.

This work is supported in part by the Office of Science of the US Department of Energy (DOE) and by the LDRD Program of Oak Ridge National Laboratory. This work used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Y. W. Li was partly supported by the LDRD Program of Los Alamos National Laboratory (LANL) under project number 20190005DR. LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). This document number is LA-UR-21-20936.

References

[1] M. L. Minsky. Some universal elements for finite automata. *In C. E Shannon & J. McCarthy (Eds.), Automata studies, Princeton: Princeton University Press*, pages 117–128, 1956.

[2] J. von Neumann. The general and logical theory of automata. *In L. A. Jeffress (Ed.), Cerebral mechanisms in behavior, New York, Wiley*, pages 1–41, 1951.

[3] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 1958.

[4] F. Rosenblatt. The perceptron: a theory of statistical separability in cognitive systems. *Buffalo: Cornell Aeronautical Laboratory, Inc. Report Number VG-1196-G-1*, 1958.

[5] S. Haykin. *Neural Networks And Learning Machines, Third Edition*. Pearson Education Ltd, 2009.

[6] B. Baker, O. Gupta, N. Nahik, and R. Raskar. Designing neural network architectures using performance prediction. *2018 International Conference on Learning Representations, Workshop Track*, 2018.

[7] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. *Proceeding NIPS'11 Proceedings of the 24th International Conference on Neural Information Processing Systems*, pages 2546–2554, 2011.

[8] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[9] S. Fahlman and C. Lebiere. The cascade-correlation learning architecture. *Advances in neural information processing system*, pages 524–532, 1990.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, Cambridge, Massachusetts and London, England, 2016.

[11] W. Grathwohl, E. Creager, S. K. S. Ghasemipour, and R. Zemel. Gradient-based optimization of neural network architecture. *ICLR 2018 Workshop Track*, 2018.

[12] T. K. Gupta and K. Raza. Optimizing deep neural network architecture: a tabu search based approach. *Neural Processing Letters*, 51:2855–2870, 2020.

[13] C. Liu, B. Zoph, J. Shlen, W. Hua, L. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[14] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 7816–7827. Curran Associates, Inc., 2018.

[15] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. *Proceeding NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems*, 2:2951–2959, 2012.

[16] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.

[17] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams. Scalable bayesian optimization using deep neural networks. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2171–2180, Lille, France, 07–09 Jul 2015. PMLR.

[18] M. Ettaouil, M. Lazaar, and Y. Ghanou. Architecture optimization model for the multilayer perceptron and clustering. *Journal of Theoretical and Applied Information Technology*, 10(1):64–72, 2013.

[19] T. K. Gupta and K. Raza. Optimization of ann architecture: A review on nature-inspired techniques. *Machine learning in Bio-signal and Diagnostic Imaging*, pages 159–182, 2018.

[20] J. Holland. Genetic algorithms,for the science. *Scientific American Edition*, 179:44–50, 1992.

[21] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems Journal*, 4:461–476, 1990.

[22] P. Koehn. *Combining Genetic Algorithms and Neural Networks: The Encoding Problem, Master of Science Thesis*. University of Knoxville, Tennessee, USA, 1991.

[23] J. T. Tsai, J. H. Chou, and T. K. Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Trans. Neural Networks*, 17(1):69–80, 2006.

[24] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, MLHPC '15, pages 1–5, New York, NY, USA, November 2015. Association for Computing Machinery.

[25] Oak Ridge National Laboratory. Multi-node evolutionary neural networks for deep learning (MENNDL). `https://www.ornl.gov/division/csmd/projects/multi-node-evolutionary-neural-networks-deep-learning-menndl`.

[26] N. K. Treadgold and T. D. Gedeon. Exploring constructive cascade networks. *IEEE Transactions on Neural Networks*, 10(6):1335–1350, 1999.

[27] S. W. Stepniewski and A. J. Keane. Pruning backpropagation networks using modern stochastic optimization techniques. *Neural Computing and Applications*, 5(2):76–98, 1997.

[28] Compressing and regularizing deep neural networks. `https://www.oreilly.com/ideas/compressing-and-regularizing-deep-neural-networks`.

[29] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[30] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*, pages 3460–3468, 2016.

[31] T. Hinz, N. Navarro-Guerrero, S. Magg, and S. Wermter. Speeding up the Hyperparameter Optimization of Deep Convolutional Neural Networks. *International Journal of Computational Intelligence and Applications*, 17(02):1850008, June 2018.

[32] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[33] P. Liashchynskyi and P. Liashchynskyi. Grid search, random search, genetic algorithm: a big comparison for NAS. *arXiv:1912.06059v1*, 2019.

[34] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. Adanet: adaptive structural learning of artificial neural networks. *Proceedings of the 34th International Conference on Machine Learning, PMLR*, 70:874–883, 2017.

[35] T. Y. Kwok and D. Y. Yeung. Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Transactions on Neural Networks*, 8(3):630–645, 1997.

[36] D. Liu, T. S. Chang, and Y. Zhang. A constructive algorithm for feedforward neural networks with incremental training. *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 49(12):1876–1879, 2002.

[37] J. .H. Friedman. Multivariate adaptative regression splines. *Annals of Statistics*, 9(1):1–67, 1991.

[38] K. Fukushima. Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[39] A. Krizhevsky, S. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25(2), 2012.

[40] Y. Le Cunn, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceeding of the IEEE*, 86(11), 1998.

[41] Summit - Oak Ridge National Laboratory's 200 petaflop supercomputer. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`.

[42] Kaggle: Your home for data science. `https://www.kaggle.com`.

[43] D. W. Aha, D. F. Kibler, and M. K. Albert. Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5:51–57, 1989.

[44] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/index.php`.

[45] The cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[46] D. P. Kingma and J. L. Ba. Adam: a method for stochastic optimization. *Conference Paper at International Conference on Learning Representations*, 2015.

[47] Keras: The Python Deep Learning library. `https://keras.io`.

[48] MPI for Python. `https://mpi4py.readthedocs.io/en/stable/`.

[49] Ray Tune: Hyperparameter Optimization Framework. `https://ray.readthedocs.io/en/ray-0.3.1/tune.html`.

[50] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: bandit-based configuration evaluation for hyperparameter optimization. *ICLR Conference proceedings*, 2017.

[51] A. Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 2020.

[52] Image classification on cifar-10. `https://paperswithcode.com/sota/image-classification-on-cifar-10`.

[53] J. Wu, X. Y. Chen, H. Zhan, L. D. Xiong, H. Lei, and S. H. Deng. Hyperparameter optimization for machine learning models based on bayesian optimization. *Journal of Electronic Science and Technology*, 17(1):26–40, 2019.

Journal Pre-proof

Journal Pre-proof

Highlights

- Neural networks are computationally expensive to train
- Identifying a well performing neural network with minimal structural complexity helps the training converge faster
- We propose a novel scalable algorithm for the optimization of neural network architectures within a constrained computational budget
- Our novel approach aims to minimize the number of hidden layers in a neural network architecture
- Numerical results performed on supercomputer Summit show that our approach better scales than state-of-the-art algorithms with comparable computational cost
- When our approach has similar time-to-solution than state-of-the-art algorithms, our algorithm identifies a neural network with better predictive performance
- When the neural network identified by our approach has similar predictive performance than the one identified by state-of-the-art hyperparameter optimization algorithms, our algorithm better scales
- The presence of GPUs is fully exploited in our implementation to perform tensor algebra operations for the training of the neural network

Journal Pre-proof

Journal Pre-proof

Conflict of interest listed by authors:

- Massimiliano Lupo Pasini: NONE
- Junqi Yin: NONE
- Ying Wai Li: NONE
- Markus Eisenbach: NONE

Journal Pre-proof