HyperNOMAD: Hyperparameter Optimization of Deep Neural Networks Using Mesh Adaptive Direct Search

DOUNIA LAKHMIRI, SÉBASTIEN LE DIGABEL, and CHRISTOPHE TRIBES, GERAD and Polytechnique Montréal, Canada

The performance of deep neural networks is highly sensitive to the choice of the hyperparameters that define the structure of the network and the learning process. When facing a new application, tuning a deep neural network is a tedious and time-consuming process that is often described as a "dark art." This explains the necessity of automating the calibration of these hyperparameters. Derivative-free optimization is a field that develops methods designed to optimize time-consuming functions without relying on derivatives. This work introduces the HyperNOMAD package, an extension of the NOMAD software that applies the MADS algorithm [7] to simultaneously tune the hyperparameters responsible for both the architecture and the learning process of a deep neural network (DNN). This generic approach allows for an important flexibility in the exploration of the search space by taking advantage of categorical variables. HyperNOMAD is tested on the MNIST, Fashion-MNIST, and CIFAR-10 datasets and achieves results comparable to the current state of the art.

CCS Concepts: • **Mathematics of computing** → *Solvers*; *Mixed discrete-continuous optimization*;

Additional Key Words and Phrases: Deep neural networks, neural architecture search, hyperparameter optimization, blackbox optimization, derivative-free optimization, mesh adaptive direct search, categorical variables.

ACM Reference format:
Dounia Lakhmiri, Sébastien Le Digabel, and Christophe Tribes. 2021. HyperNOMAD: Hyperparameter Optimization of Deep Neural Networks Using Mesh Adaptive Direct Search. *ACM Trans. Math. Softw.* 47, 3, Article 27 (June 2021), 27 pages.
https://doi.org/10.1145/3450975

1 INTRODUCTION

Neural networks are mathematical structures used to solve supervised or unsupervised problems involving images, sounds, and speech, to name a few. In recent years, neural networks gained in popularity due to the emergence of large-size databases and the development of the computational power of contemporary machines, through the use of GPUs in particular. These favorable conditions have allowed neural networks to learn complex structures and achieve a level of precision that can surpass human performance across multiple instances such as robotics [41], medical diagnostics [43], and more.

This work is in part supported by the NSERC Alliance grant 544900-19 in collaboration with Huawei-Canada.
Authors' address: D. Lakhmiri, S. Le Digabel, and C. Tribes, GERAD and Polytechnique Montréal, C.P. 6079, Succ. Centre-ville, Montréal, Qc, H3C 3A7, Canada; emails: {Dounia.Lakhmiri, Sebastien.Le.Digabel, Christophe.Tribes}@gerad.ca.

0098-3500/2021/06-ART27 $15.00
https://doi.org/10.1145/3450975

However, the performance of a neural network is strongly linked to its structure and to the values of the parameters of the optimization algorithm used to minimize the error between the predictions of the network and the data during its training. The choices of the neural network hyperparameters can greatly affect its ability to learn from the training data and to generalize with new data. The algorithmic hyperparameters of the optimizer must be chosen a priori and cannot be modified during optimization. Hence, to obtain a neural network, it is necessary to fix several hyperparameters of various types: real, integer, and categorical. A variable is categorical when it describes a class, or category, without a relation of order between these categories. The search for an optimal configuration is a very slow process that, along with the training, takes up the majority of the time when developing a network for a new application. It is a relatively new problem that is often solved randomly or empirically.

Derivative-free optimization (DFO) [8, 21] is the field that aims to solve optimization problems where derivatives are unavailable, although they might exist. This is the case, for example, when the objective and/or constraint functions are non-differentiable, noisy, or expensive to evaluate. In addition, the evaluation in some points may fail, especially if the values of the objective and/or constraints are the outputs of a simulation or an experience. **Blackbox optimization (BBO)** is a subfield of DFO where the derivatives do not exist and the problem is modeled as a blackbox. This term refers to the fact that the computing process behind the output values is unknown. The general DFO problem is described as

$$\min_{x \in \Omega} f(x),$$

where f is the objective function to minimize over the domain Ω.

There are two main classes of DFO methods: model-based and direct search methods. The first uses the value of the objective and/or the constraints at some already evaluated points to build a model able to guide the optimization by relying on the predictions of the model. For example, this class includes methods based on trust regions [21, Chapter 10] or interpolation models [52]. This differentiates them from direct search methods [31] that adopt a more straightforward strategy to optimize the blackbox. At each iteration, direct search methods generate a set of trial points that are compared to the "best solution" available. For example, the GPS algorithm [59] defines a mesh on the search space and determines the next point to evaluate by choosing a search direction. DFO algorithms usually include a proof of convergence that ensures a good-quality solution under certain hypotheses on the objective function. BBO algorithms extend beyond this scope by including heuristics such as evolutionary algorithms, sampling methods, and so on.

In [5, 10], the authors explain how a **hyperparameter optimization (HPO)** problem can be seen as a blackbox optimization problem. Indeed, the HPO problem is equivalent to a blackbox that takes the hyperparameters of a given algorithm and returns some measure of performance defined in advance such as the time to solution, the value of the best point found, or the number of solved problems. In the case of neural networks, the blackbox can return the accuracy on the test dataset as a measure of performance. With this formulation, DFO techniques can be applied to solve the original HPO problem.

This work presents HyperNOMAD, a package that applies MADS, a direct search method behind the NOMAD software, to tune the hyperparameters that affect the architecture and the learning process of a deep neural network. Figure 1 illustrates the workflow when solving HPO problems with HyperNOMAD. For a given set of hyperparameters, the construction of the network, the network training, validation, and testing are all wrapped as a single blackbox evaluation. One specificity of HyperNOMAD is its ability to explore a large search space by exploiting categorical variables with the implementation of a neighborhood structure specifically designed for the HPO problem considered here.

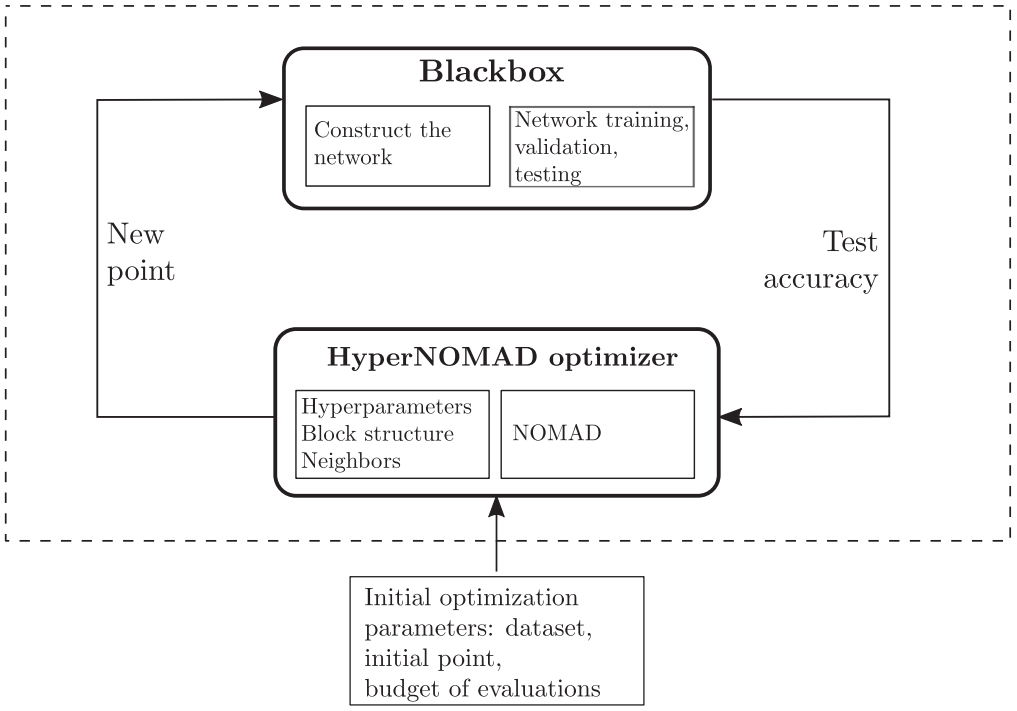


Fig. 1. The HyperNOMAD workflow.

The article is structured as follows. Section 2 presents and discusses some of the main approaches used to solve the HPO problem of neural networks. In Section 3, the experimental setup is explicitly defined, and the instances used to test the proposed approach are presented. Section 4 introduces the HyperNOMAD package and gives an overview of MADS, the algorithm selected to carry out the optimization task including the handling of categorical variables. Computational results are provided and discussed in Section 5. Finally, Appendix A describes the basic usage of HyperNOMAD.

2 LITERATURE REVIEW

Tuning the hyperparameters of a deep neural network is a critical and time-consuming process that was mainly done manually relying on the knowledge of the experts. However, the rising popularity of deep neural networks and their usage for diverse applications called for the automation of this process in order to adapt to each problem.

The hyperparameters that define a deep neural network can be separated into two categories: the ones that define the architecture of the network and the ones that affect the optimization process of the training phase. Tuning the hyperparameters of the first category alone has led to a separate field of research called **Neural Architecture Search (NAS)** [25] that allowed achievement of state-of-the-art performance [53, 65] on some benchmark problems, although at a massive computational cost of 800 GPUs for a few weeks. Typically, one would perform an NAS first and then start tuning the other hyperparameters with the optimized architecture. However, Zela et al. [64] argue that this separation is not optimal since the two aspects are not entirely independent from one another. Therefore, the proposed research considers both aspects at once.

One of the first scientific approaches used to tackle the HPO problem of neural networks is grid search. This method consists of discretizing the hypercube defined by the range of each

hyperparameter and then evaluating each point on the grid. This technique is still used today and is implemented in several HPO libraries such as scikit-learn and Spearmint [50, 56]. It has the advantage of being easy to understand, implement, and parallelize. However, it becomes very expensive when training large networks, which is the case of deep neural networks, or when one seeks to optimize several hyperparameters at once. In addition, grid search does not detect the impact of each hyperparameter on the overall performance of the network.

To avoid the drawbacks of the grid search, an alternative is to use random search [16]. Indeed, a random exploration of the space allows one to evaluate more different values for each of the hyperparameters. This has the advantage of increasing the chances of finding a better configuration but also to highlight the importance of some hyperparameters compared to the others. In addition, the random search makes it possible to highlight these properties with fewer evaluations than an exhaustive grid search. More recently, the Hyperband algorithm [42] was introduced, which is a variant of a random search that uses an early stopping criterion to detect a non-promising point early on in order to save computational resources and time, thus achieving an important speedup compared to other methods. However, despite its advantages over grid search, a random approach is limited because it is not adaptive and it does not exploit the performance scores of each configuration to direct the search. This can also waste resources that could have been better exploited by another optimization approach.

Genetic algorithms are evolutionary heuristics that are also used for the HPO problem. Inspired by biology, a genetic algorithm generates an initial population, i.e., a set of configurations. Then, it combines the best parents to create a new generation of children. It also introduces random mutations to ensure a certain diversity in the population. These heuristics are therefore adaptive, thus exploring the space more wisely even if some randomness remains in the process. These algorithms are often used to optimize hyperparameters [26, 57, 63]. In [45], a method based on particle swarm optimization is able to provide networks with higher performance than those defined by experts in less time than what would have required a grid search or a completely random search. Another approach using the evolutionary algorithm CMA-ES [46] was proposed with satisfactory results.

Other approaches based on machine learning can be found in the literature. For example, the HPO problem can be seen as reinforcement learning [12, 65, 66] where the main difference between each method relies on how the agents are defined and dealt with. In [55], a neural network is able to design other neural networks by learning to explore the possible configurations. There, the HPO of neural networks is seen as a multi-objective problem where one seeks to improve the performance of the network while minimizing the computing power required. This approach, although successful, solves a different problem from the one considered in the context of this study. Also, [18] uses a network of long-term memory neurons to learn the parameters of another multi-layer network that is tested on a binary classification problem.

DFO is naturally adapted to the HPO problem since it aims at solving problems typically given in the form of blackboxes that can be computationally costly to evaluate, with non-existent or unexploitable derivatives. In [10], the authors propose a general way of modeling hyperparameter optimization problems as a blackbox optimization problem. This formulation is used in [44] to optimize 11 hyperparameters (3 real and 8 integer) of the BARON solver. This study compared the solutions found by 27 DFO algorithms on a total of 126 problems. The results show that the DFO methods have reduced the average solution time, sometimes by more than 50%. Another formulation inspired by robust optimization is used in [51], in addition to that of [10], to optimize the hyperparameters of the BFO algorithm [51]. BBO methods are also at the heart of Google Vizier [29], which is a tool that can be used for the HPO problem of machine learning algorithms, and especially for deep neural networks.

Table 1. Selection of Open Source Libraries for the Hyperparameter Optimization Problem

	Optimization Method					Type of Variables		
Package	Grid Search	Random Search	Bayesian Optimization	Model-Based DFO	Direct-Search DFO	Real	Int.	Cat.
scikit-learn [50]	✓	✓	-	-	-	✓	✓	✓
hyperopt [17]	-	✓	✓	-	-	✓	✓	✓
Spearmint [56]	✓	✓	✓	-	-	✓	✓	✓
SMAC [32]	-	-	-	✓	-	✓	✓	✓
MOE [62]	-	-	✓	-	-	✓	-	-
RBFOpt [23]	-	-	-	✓	-	✓	✓	-
DeepHyper [13]	-	✓	-	✓	-	✓	✓	✓
Oríon [20]	-	✓	✓	-	-	✓	✓	✓
Google Vizier [29]	✓	✓	✓	✓	-	✓	✓	✓
HyperNOMAD	-	-	-	-	✓	✓	✓	✓

Bayesian optimization (BO) can be seen as a subclass of DFO methods and, as such, can be used to solve the HPO problem. The BO methods use information collected during previous assessments to diagnose the search space and predict which areas to explore first. Among them, **Gaussian processes (GPs)** are models that seek to explain the collected observations that supposedly come from a stochastic function. GPs are a generalization of multi-variate Gaussian distributions, defined by a mean and a covariance function. GPs are popular models for optimizing the hyperparameters of neural networks [56, 60]. However, the disadvantage of GPs is that they do not fit well to categorical features, and their performance depends on the choice of the kernel function that defines them. **Tree-structured Parzen Estimator (TPE)** is also a Bayesian method that can be used as a model instead of a GP. After a certain number of evaluations, this method separates the evaluated points into two sections: a portion (<25%) of the points with the best performances and the remaining. This method seeks to find a distribution of the best observations to determine the next candidates. TPEs are also used for the HPO of neural networks [15], even if it has the disadvantage of ignoring the interactions between the hyperparameters.

Other model-based DFO methods were also applied to the HPO problem. In [23], the authors applied radial basis functions to model the blackbox as previously defined. This article presents the results obtained on the MNIST dataset [40], then on a problem of interactions between drugs. These tests show that this model provides comparable or better results than popular configurations. In [28], a trust-region DFO algorithm is applied to optimize the hyperparameters of an SVM model. Here again, this approach obtained a more efficient model than those defined by the experts or by a Bayesian algorithm.

The positive results of these methods suggest that the DFO approach is well suited to solve the HPO problem of deep neural networks. This motivated the idea of using the MADS algorithm [7], which is implemented in the NOMAD package [37], especially since it can handle integer and categorical variables [2, 9]. Using the MADS algorithm for hyperparameter tuning has been validated in [47], where an SVM model is calibrated using MADS combined with the Nelder-Mead and VNS search strategies [4, 11].

A non-exhaustive list of open source libraries for HPO is given in Table 1 along with the optimization algorithms implemented in each library and the types of variables handled.

3 EXPERIMENTAL SETUP

This section first defines the generic blackbox approach used for modeling the HPO problem. This is done by listing the different hyperparameters considered to construct, train, and validate a **deep**

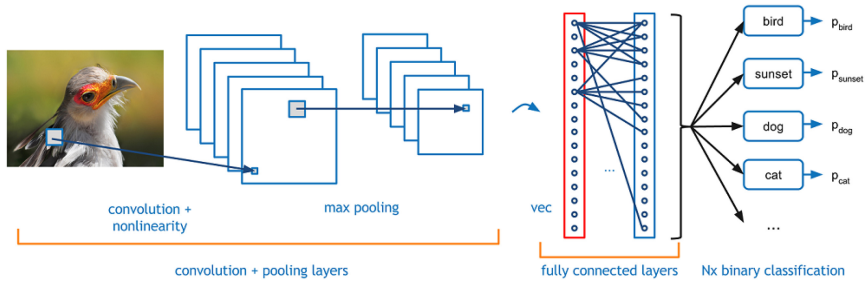


Fig. 2. Example of a convolutional neural network. Image taken from [22].

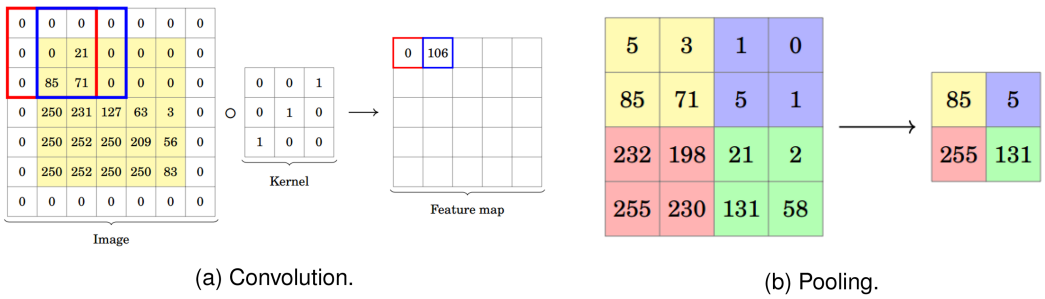


Fig. 3. Illustration of a convolution operation in (a) and a pooling operation in (b). Images taken from [49].

neural network (DNN) in order to obtain its test accuracy. The second part of the section gives an overview of the datasets provided with HyperNOMAD.

3.1 Hyperparameters of the Framework

A variety of hyperparameters must be chosen to tune a DNN for a given application. These hyperparameters affect different aspects of the network: the architecture, the optimization process, and the handling of the data. The following section lists the hyperparameters considered in this study along with their respective types and scopes.

3.1.1 The Network Architecture. A **convolutional neural network (CNN)** is a deep neural network consisting of a succession of convolutional layers followed by fully connected layers as illustrated in Figure 2.

To define a new CNN, one must first decide on the number of convolutional layers. These layers can be seen as matrices in a two-dimensional convolution. The size of the first convolutional layer is determined by the size of the images the network is fed. The size of the remaining layers is computed by taking into account the different operations applied from layer to layer. These operations can be divided into two categories: a convolution or a pooling. Figure 3(a) represents the steps of a convolution operation. The initial image is a 5×5 matrix whose borders are padded with pixels of zeros. A *padding* is an artificial border filled with zeros that is added to a convolutional matrix. It serves to increase the size of the output of the convolution or to give more weights to the values of the original border on the output. The convolution consists of choosing a kernel, or filter, that is passed over the image in order to compute the coefficients of the feature map. The *kernel* can be seen as a small matrix hovered over the image to extract important features that form the feature map where each coefficient is equal to the sum of the products between the coefficients of the image and the ones of the kernel situated in the same position. For example, in Figure 3(a), the

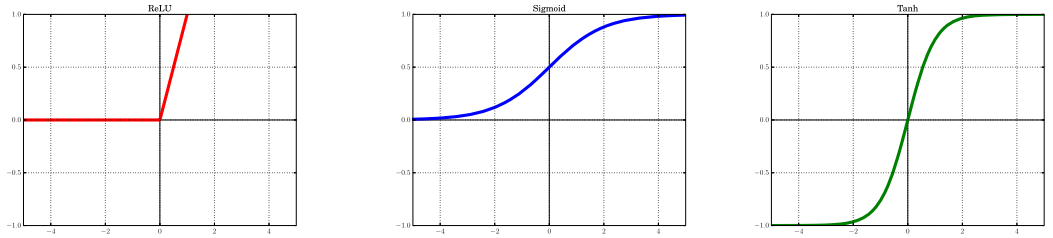


Fig. 4. Examples of activation functions.

coefficient (2,1) of the feature map is obtained by the following operation: $(0 \times 0) + (0 \times 0) + (0 \times 1) + (0 \times 0) + (21 \times 1) + (0 \times 0) + (85 \times 1) + (71 \times 0) + (0 \times 0) = 106$. In general, a convolution can be determined with few factors such as the number of feature maps—or output channels—generated; the size of the kernel, i.e., the size of the kernel matrix, which in turn will affect the size of the feature map; the *stride*, which corresponds to the step by which the kernel is moved over the image; and the previously defined padding. In Figure 3(a), the image is padded with one layer of zeros.

When the feature map is obtained, one can decide to apply a *pooling* operation, which can be seen as a special kernel that keeps the biggest value in each zone of the image. Figure 3(b) illustrates a 2×2 pooling that results in an output of half the size of the feature map. After each convolutional layer, a pooling operation can be applied with the pooling size ranging from 1 (no pooling) to 5.

Each fully connected layer that follows a convolutional layer is determined by the number of neurons it contains. The neurons of a layer are connected to all of the ones in the next layer through weighted arcs. Let $x_1, x_2, \ldots, x_{n_l}$ be the values of the neurons of the layer l and a_j be the value of the neuron j in the layer $l+1$; then $a_j = \phi(\sum_{i=1}^{s_l} w_{ij} x_i)$, where $w_{1j}, w_{2j}, \ldots, w_{n_l j}$ are the weights of the arcs from the neurons of the layer l to the jth neuron of the following layer and ϕ is an activation function used to introduce a non-linearity in the outputs. Figure 4 presents some examples of activation functions. Additionally, the neurons of the fully connected layers are attributed a *dropout rate*, which is the probability that a neuron is deactivated from the layer. The dropout rate acts as a regularization technique that counters the tendency to overfit to the training data and therefore allows the network to better generalize. For the sake of reducing the dimension of the HPO problem, a single dropout rate is applied to all layers.

Table 2 summarizes the hyperparameters responsible for defining the structure of the network. Hyperparameters 2 to 6 must be defined for each convolutional layer and hyperparameter 8 must also be defined for each fully connected layer. Therefore, if n_1 is the number of convolutional layers and n_2 the number of fully connected layers, the total number of hyperparameters responsible for defining the structure of the neural network is $5n_1 + n_2 + 4$. Since this number, and therefore the dimension of the problem, can change, the number of convolutional layers n_1 and fully connected layers n_2 are treated as categorical variables instead of integers. NOMAD can generically handle all types of categorical variables that can affect both the variable type and the problem dimension during an optimization but requires that the user implement the logic of these changes. HyperNOMAD exploits the implementation structure of NOMAD but automates and simplifies this task when handling hyperparameters of a neural network. Further explanations are presented in Section 4.

3.1.2 The Optimizer. For a given network architecture, the training phase is conducted to minimize the error between the predictions of the network and the correct values of the labels assigned

Table 2. Hyperparameters That Define the Architecture of a Neural Network

#	Hyperparameter	Type	Scope
1	Number of convolutional layers (n_1)	Categorical	$\{0, 1, \ldots, 20\}$
2	Number of output channels	Integer	$\{0, 1, \ldots, 50\}$
3	Kernel size	Integer	$\{0, 1, \ldots, 10\}$
4	Stride	Integer	$\{1, 2, 3\}$
5	Padding	Integer	$\{0, 1, 2\}$
6	Pooling size	Integer	$\{1, 2, \ldots, 5\}$
7	Number of full layers (n_2)	Categorical	$\{0, 1, \ldots, 30\}$
8	Size of the full layer	Integer	$\{0, 1, \ldots, 500\}$
9	Dropout rate	Real	$[0; 1]$
10	Activation function	Categorical/Integer	$\{$ReLU (1), Sigmoid (2), Tanh (3)$\}$

to the validation data. Let Θ be a multi-dimensional matrix that stores the weights of the arcs that link each layer l of the network with the next layer $l + 1$, and let

$$J(\Theta) = Err(pred, truth) + \lambda ||\Theta||^2 \tag{1}$$

be the loss function where the second term corresponds to a regularization based on the *weight decay* $\lambda \in [0; 1]$. This aims at reducing the over-fitting during the training. The optimizer must then solve $\min_\Theta J(\Theta)$. Before starting the training phase, the optimizer that carries out this task must be selected along with its specific algorithmic hyperparameters. A stochastic gradient approach is more suitable in this case because of the high dimension of the matrix Θ, which is usually in the millions. At each iteration, the weights of the network are updated by following a stochastic direction with a particular step size, which is called a learning rate in the machine learning context. Similarly to any gradient descent method, the learning rate must be chosen and updated to avoid oscillations or divergence. Substantial research and tricks of the trade are developed to solve this problem [14, 19, 39].

In its original version, the **stochastic gradient descent (SGD)** has one fixed learning rate during the training for all the weights. This issue can somewhat be handled through a scheduler in PyTorch where the user decides to change the learning rate after a certain number of epochs. In HyperNOMAD, the learning rate is updated by the scheduler tool in PyTorch using a cosine annealing strategy. SGD also suffers from oscillating "descent" directions, which tends to slow down the convergence in the training. This problem can be corrected by adding momentum and dampening when producing a new descent direction, which are used as the coefficients in a weighted sum between the last direction taken and the stochastic gradient computed in the current iteration. With the right coefficients, this sum ensures fewer oscillations and a quicker convergence.

Other optimizers have been developed mainly to add an adaptive learning rate that is adjusted to each weight since they are not all equally important regarding the final prediction of the network. The Adagrad optimizer [24], based on SGD, adjusts the learning rate to each weight based on the matrix G_k defined at the iteration k as the sum of the outer products of the gradients computed at each iteration until k, $G_k = \sum_{i=1}^{k} \nabla J(\Theta_i)(\nabla J(\Theta_i))^\top$. The diagonal of this matrix accumulates the gradients of each weight so that the step size update is inversely proportionate to the accumulated gradient for a specific weight. PyTorch also allows for decaying the initial learning rate by multiplying it by the *learning rate decay coefficient* τ.

Table 3. Choices of the Optimizer and the Corresponding Hyperparameters

Optimizer	Update Rule	Hyperparameter	Type	Scope
SGD	$d_{k+1} = \mu d_k + (1-\delta)\nabla J(\Theta_{k+1})$	η: Initial learning rate	Real	$[0; 1]$
	$\Theta_{k+1} = \Theta_k - \eta d_{k+1}$	μ: Momentum	Real	$[0; 1]$
		δ: Dampening	Real	$[0; 1]$
		λ: Weight decay (1)	Real	$[0; 1]$
Adagrad	$G_{k+1} = G_k + \nabla J(\Theta_{k+1})(\nabla J(\Theta_{k+1}))^\top$	η: Initial learning rate	Real	$[0; 1]$
	$\Theta_{k+1} = \Theta_k - \tau \frac{\eta}{\sqrt{\epsilon I + diag(G_{k+1})}} \nabla J(\Theta_k)$	τ: Learning rate decay	Real	$[0; 1]$
		ϵ: Smoothing constant	Real	$[0; 1]$
		λ: Weight decay (1)	Real	$[0; 1]$
RMSProp	$d_{k+1} = \mu d_k + (1-\mu)\nabla J(\Theta_{k+1})$	η: Initial learning rate	Real	$[0; 1]$
	$G_{k+1} = \gamma G_k + (1-\gamma)\nabla J(\Theta_{k+1})(\nabla J(\Theta_{k+1}))^\top$	μ: Momentum	Real	$[0; 1]$
	$\Theta_{k+1} = \Theta_k - \frac{\eta}{\sqrt{\epsilon I + diag(G_{k+1})}} d_{k+1}$	γ: Forgetting factor	Real	$[0; 1]$
		λ: Weight decay (1)	Real	$[0; 1]$
Adam	$d_{k+1} = \beta_1 d_k + (1-\beta_1)\nabla J(\Theta_k)$	η: Initial learning rate	Real	$[0; 1]$
	$G_{k+1} = \beta_2 G_k + (1-\beta_2)\nabla J(\Theta_{k+1})(\nabla J(\Theta_{k+1}))^\top$	β_1: Factor on the running average of the gradient	Real	$[0; 1]$
	$\hat{d}_{k+1} = \frac{d_k}{1-\beta_1}$; $\hat{G}_{k+1} = \frac{G_k}{1-\beta_2}$	β_2: Factor on the running average of the squares of the gradient	Real	$[0; 1]$
	$\Theta_{k+1} = \Theta_k - \frac{\eta}{\sqrt{\epsilon I + diag(\hat{G}_{k+1})}} \hat{d}_{k+1}$			
		λ: Weight decay (1)	Real	$[0; 1]$

The adaptive rule of Adagrad reduces the learning rate of the frequently updated weights rather aggressively. RMSProp [58] corrects this issue by dividing the learning rate by a running average of the squares of the gradient $G_{k+1} = \gamma G_k + (1-\gamma)\nabla J(\Theta_{k+1})(\nabla J(\Theta_{k+1}))^\top$, where γ is the *forgetting factor,* also referred to as the *smoothing constant* in PyTorch. The Adam optimizer [34] expands RMSProp further by taking two factors into account. The first one, β_1, is on the running average on the gradient and the second, β_2 is on the running average on the squares of the gradient as shown in Table 3.

Table 3 presents the list of selectable optimizers considered in the blackbox along with their corresponding hyperparameters. There is one categorical hyperparameter that determines which optimizer is chosen, plus four real hyperparameters related to each. Therefore, the training regime is defined with five hyperparameters in total.

3.1.3 The Training Phase. Before training a network, the data must be separated into three groups; each one is responsible for the training, the validation, and the testing of the network. During the training phase, the network is fed with the training data, performs a forward pass, computes the prediction error, and does a back-propagation in order to update the weights using the optimizer. The way the network is fed is also of great importance. One can choose to input the training data one by one, all at once, or by sending subsets or mini-batches of the data. The batch size is an integer hyperparameter that varies in $\{1, 2, \ldots, n_{\text{train}}\}$, where n_{train} is the size of the training data.

When the network has been fed all of the training data, it is said to have performed an epoch. Usually, the training data has to be passed more than once in order to obtain good weights and a good testing accuracy. Therefore, the number of epochs must be chosen as well. This

Table 4. Datasets Embedded in HyperNOMAD

Dataset	Training Data	Validation Data	Testing Data	Number of Classes
MNIST	40,000	10,000	10,000	10
Fashion-MNIST	40,000	10,000	10,000	10
EMNIST	40,000	10,000	10,000	10
KMNIST	40,000	10,000	10,000	10
CIFAR-10	40,000	10,000	10,000	10
CIFAR-100	40,000	10,000	10,000	100
STL-10	4,000	1,000	8,000	10

hyperparameter is dealt with as follows. The validation accuracy is evaluated after each epoch and the weights of the network responsible for the best validation accuracy are stored. This process is repeated as long as the number of epochs is lower than a certain maximum number of epochs and as long as an early stopping condition has not been satisfied. These early stopping criteria depend on the evolution of the training and validation of the network. When the validation accuracy stagnates or when it stays lower than 20% after 50 epochs, the training can be interrupted in order to save time and computational resources. Once the training is done, the test accuracy is evaluated using the weights that gave the best validation accuracy.

Finally, the blackbox optimization problem is obtained following the model in [10], where the blackbox takes the hyperparameters as inputs and builds the network in order to train, validate, and test it on a given dataset before returning the test error as a measure of performance. In this context, the blackbox takes $5n_1 + n_2 + 10$ mixed variable inputs, where n_1 is the number of convolutional layers and n_2 the number of fully connected layers of the network, and returns the value of the accuracy on the test dataset. This blackbox problem is solved using the NOMAD software [37] described in Section 4.1.

3.2 Datasets

The HyperNOMAD package comes with a selection of datasets all meant for classification problems. Table 4 lists the datasets embedded so far through PyTorch [48], an open source library used to model and manipulate deep neural networks. HyperNOMAD also allows the usage of a personal dataset; see the online documentation at https://github.com/bbopt/HyperNOMAD. When loading a dataset from Table 4, HyperNOMAD applies a normalization and a random horizontal flip to regulate and augment the data.

The rest of the section describes the datasets used for benchmarking HyperNOMAD. First, a validation is done using both MNIST [40] and Fashion-MNIST [61], and once positive results are obtained, the second and more complex dataset, CIFAR-10 [36], is considered.

3.2.1 MNIST. MNIST [40] is a dataset containing 60,000 images of hand-written digits that is usually divided into three categories: 40,000 for training, 10,000 for validation, and the remaining 10,000 for testing. The set is used for developing a convolutional neural network capable of recognizing the digits in each image and assigning it to the correct class. The relative simplicity of this task does not require complex neural networks to obtain a good accuracy. Therefore, this dataset is usually considered as a first validation of a concept and not a sufficient proof of the quality of a method among the machine learning community.

3.2.2 Fashion-MNIST. Fashion-MNIST [61] is a dataset containing 60,000 images of clothing items that belong in 10 different categories. The split into training, validation, and test sets is

similar to MNIST, meaning that 40,000 images are allocating for training, 10,000 for validation, and 10,000 for testing.

3.2.3 CIFAR-10. The second set of tests are performed with CIFAR-10 [36]. This dataset contains 60,000 colored images of objects that belong to 10 different and independent categories. The data is once again divided into three sets: 40,000 for training, 10,000 for validation, and 10,000 for testing.

For these tests, the blackbox within HyperNOMAD is used to construct the convolutional neural network corresponding to the values of the hyperparameters described in Section 3.1. This network is trained, validated, and tested for each dataset according to the mode of operation of HyperNOMAD detailed in Section 4.

4 HYPERNOMAD

The HyperNOMAD package is available on GitHub.[1] It contains a series of Python modules that act as a blackbox, which takes a set of hyperparameters described in Section 3 as inputs and constructs the corresponding network that is trained and tested before returning the test accuracy as the output. This blackbox uses the PyTorch package [48] for its simplicity. HyperNOMAD also contains an interface that runs the optimization of the blackbox using the NOMAD software [37] described in the rest of this section. The basic usage of HyperNOMAD is described in Appendix A.

4.1 Overview of NOMAD

The NOMAD software [37] is a C++ implementation of the MADS algorithm [7, 9], which is a direct search method that generates, at each iteration k, a set of points on the $mesh M^k = \{x + \text{diag}(\boldsymbol{\delta}^k)z : x \in V^k, z \in \mathbb{Z}^n\}$, where V^k contains the points that were previously evaluated (including the current iterate x^k) and $\boldsymbol{\delta}^k \in \mathbb{R}^n$ is the *mesh size vector*.

Each iteration of MADS is divided into two steps: the *search* and the *poll*. The *search* phase is optional and can contain different strategies to explore a wider space in order to generate a finite number of possible mesh candidates. This step can be based on surrogate functions, Latin hypercube sampling, and so forth [4, 11]. The *poll*, on the other hand, is strictly defined since the convergence theory of MADS relies entirely on this phase. In the poll step, the algorithm generates directions around the current iterate x^k to search for candidates locally in a region centered around x^k and of radius, in each dimension, of $\Delta^k \in \mathbb{R}^n$, which is called the *poll size vector*. The set of candidates in this step defines the *poll set* P_k.

If MADS finds a better point than the incumbent, then the iteration is declared a success, and the mesh and poll sizes are increased. However, if the iteration fails, then both parameters are reduced so that $\boldsymbol{\delta}^k \leq \Delta^k$ is maintained. This relation ensures that the set of search directions becomes dense in the unit sphere asymptotically. In addition, NOMAD can handle categorical variables by adding a step in the basic MADS algorithm. A variable is categorical when it can take a finite number of nominal or numerical values that express a qualitative property that assign the variable to a class (or category). The algorithm relies on an ad hoc neighborhood structure, provided in practice by the user as a list of neighbors for any given point. The poll step of MADS is augmented with the so-called *extended poll* that links the current iterate x^k with the independent search spaces where the neighbors can be found. The first neighbor that improves the objective function is chosen and the optimization carries on in the corresponding search space. For more detail on how MADS handles categorical variables, the reader is referred to the following list of articles [1–3, 6, 35]. The MADS algorithm is summarized in Algorithm 1.

[1] https://github.com/bbopt/HyperNOMAD.

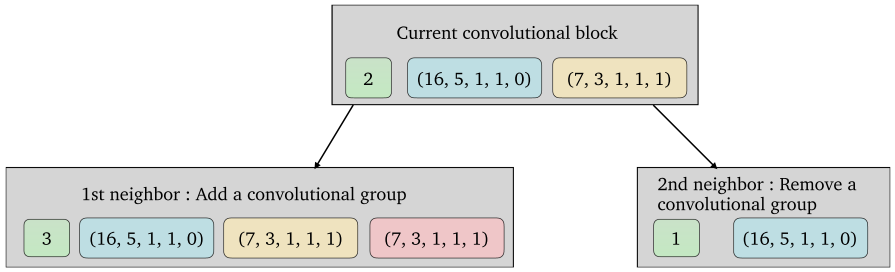


Fig. 5. Example of a convolution block (top) composed of a header (green) containing the categorical variable that describes the number of convolutional layers, followed by two groups of variables, each one describing one convolutional layer. Its first neighbor is obtained by adding a convolutional layer at the right (bottom left) and the second neighbor is obtained by deleting the rightmost convolutional layer (bottom right).

ALGORITHM 1: Mesh adaptive direct search (MADS).

$k = 0, \boldsymbol{\delta}^0, x^0$

[1] Search (optional)

Construct a set of mesh points and evaluate them

If there is a success, go to **[4]**

[2] Poll

Evaluate the points in the poll set P_k

If there is a success, go to **[4]**

[3] Extended poll

Build and evaluate the neighbors of the current incumbent defined specifically for an application

If there is no success, perform an extended poll descent for each neighbor close enough to success

[4] Updates

Update $\boldsymbol{\delta}^k, x^k, M^k, V^k$ depending on the success of the previous phases

If no stopping condition is satisfied: $k \leftarrow k + 1$ and go to **[1]**

4.2 Hyperparameters in HyperNOMAD

The selected neighborhood structure in HyperNOMAD relies on blocks of categorical variables with their associated variables. The following subsections describe this structure.

4.2.1 Blocks of Hyperparameters. HyperNOMAD splits the **hyperparameters (HPs)** defined in Section 3.1 into different blocks: one for the convolution layers, the fully connected layers, and the optimizer and one for each of the other HPs. A *block* is an implemented structure that stores a list of values, each one starting with a header and followed by the associated variables, when applicable, that are gathered into groups. For example, consider a CNN with two convolutional layers, each one defined with the number of output channels, the kernel size, the stride, the padding, and whether a pooling is applied or not as stated in Table 2. Then consider the values $(16, 5, 1, 1, 0)$ and $(7, 3, 1, 1, 1)$. Each set of values corresponds to a group of variables that describes one convolutional layer and both groups are part of the convolution block. The header of the convolution block is the categorical variable that represents the number of convolutional layers (n_1) that the CNN contains as shown in Figure 5 (top). The convolution block is followed by the fully connected block. The header of this block also corresponds to the categorical variable that describes the number of fully

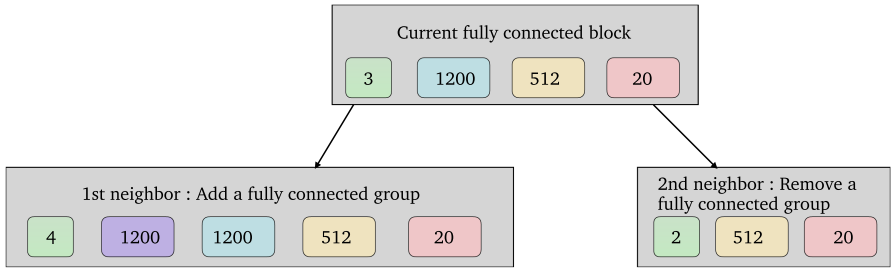


Fig. 6. Example of a fully connected block (top) composed of a header (green) containing the categorical variable that describes the number of fully connected layers, followed by three groups of variables, each one describing the size of one fully connected layer. Its first neighbor is obtained by adding a fully connected layer at the left (bottom left) and the second neighbor is obtained by deleting the leftmost fully connected layer (bottom right).

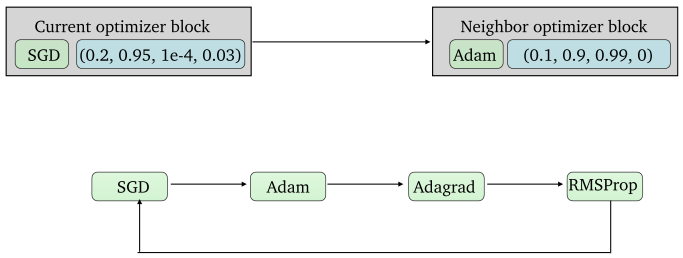


Fig. 7. Example of an optimizer block and its neighbor (top) obtained by selecting the next optimizer by following the order defined in the bottom and initializing the associated variables to their default values.

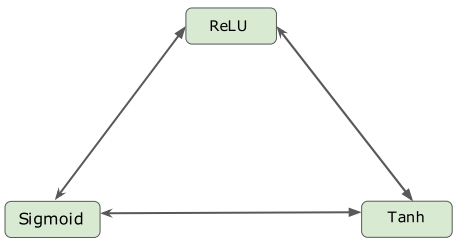


Fig. 8. Illustration of how HyperNOMAD changes the categorical variable corresponding to the activation function.

connected layers. Here, each layer is defined with the number of neurons it contains. Therefore, if n_2 is the value in the header, then there are n_2 groups of a single variable as illustrated in Figure 6 (top). The optimizer block always possesses a fixed size since there is always five HPs that describe the optimizer: The choice of the algorithm and four related HPs as summarized in Table 3. The header of this block is the categorical variable corresponding to the choice of the optimizer and the four associated variables are gathered into one group as shown in Figure 7. The other HPs are put as the headers of their individual block with no associated variables.

4.2.2 Neighborhood Structure. The extended poll of MADS with categorical variables constructs one or more neighbor points from any given point and evaluates them. There are up to three categorical variables that are exploited using an ad hoc generation of neighbor points. A neighborhood structure considering coupled effect between the categorical variables may find

promising search spaces, but it certainly increases the resources needed to carry out the optimization. To limit the number of neighbor points, the neighborhood structure considered here changes one block of variables per neighbor while the other blocks are fixed at the current iterate values. Therefore, neighbors of the convolutional, fully connected optimizer and activation function blocks must first be defined. The neighbor structure of the convolution block is obtained by adding and subtracting a group of associated variables at the end of the block. These operations can only be performed if the resulting size is within the bounds for the variable n_1. When adding a group of associated variables, the values of the associated variables are copied from the rightmost group. Adding or subtracting a group to the convolution block is illustrated in Figure 5.

The neighbor structure of the fully connected block is obtained by adding and subtracting one associated variable at the beginning of the block. These operations can only be performed if the resulting size is within the bounds for the variable n_2. When adding a group, the value of the associated variable is copied and inserted from the leftmost value (see example in Figure 6). The structure of the network varies when adding or subtracting a convolutional or a full layer, and so does the dimension of the HPO problem. Varying the remaining categorical variables has no such effect.

The categorical variable controlling the choice of optimizer has four possible values: SGD, Adam, Adagrad, or RMSprop. The choice of optimizer does not change the dimension of the optimization problem, but it affects the interpretation of the four associated variables related to the optimizer as illustrated in Table 3. A single neighbor point is obtained by cycling through optimizers as shown in Figure 7. For each possible optimizer, there are four associated variables controlling the algorithm with different interpretations. When the optimizer is changed, these variables are reset to their default values. Figure 7 also illustrates the neighbor of a specific optimizer block.

In some cases, a variable controlling a category can be well handled as an integer variable by ordering the categories with a predefined order. This is the case for the variable selecting among the three possible activation functions (see Section 3.1 and Table 2): ReLU, Sigmoid, and Tanh, with corresponding values between 1 and 3. In HyperNOMAD, this variable is treated as a periodic integer—without bounds—instead of a categorical one since the choice of the activation function does not affect the structure of the problem as for the number of convolutional or fully connected layers, nor does it affect the role of other variables as the choice of the optimizer. The choice of the activation function is therefore considered a periodic integer; thus, no explicit neighborhood structure is required.

With the neighborhood structure defined for each of the previous blocks, and knowing that each neighbor of any given point is built so that only one block is changed at a time while the remaining blocks keep the same values as the current iterate, it follows that each point has five neighbors: two obtained from the convolutional block, two from the fully connected block, and one from the optimizer block.

5 COMPUTATIONAL RESULTS

This section summarizes the results obtained by HyperNOMAD and compares them to other methods when applied to the MNIST [40], Fashion-MNIST [61], and CIFAR-10 [36] datasets. For each series of tests, all the hyperparameters discussed in Section 3 are allowed to vary. However, the user of the framework can fix some hyperparameters and choose to focus on others as described in Appendix A. All the following tests give the same budget of 200 blackbox evaluations for every optimization algorithm. Each configuration is allowed to train for a maximum of 100 epochs. The validation error is evaluated to save the best weights that are re-loaded to the network at the end of the training to evaluate the test error. The following tests were run using various Nvidia GPUs (GTX 1070, TITAN Xp, and Tesla P100).

Table 5. Hyperparameters Considered for the Tests on the MNIST Dataset with the Simplified Caffe Blackbox

#	**Hyperparameter**	**Type**	**Scope**
1	Number of convolutional layers	Categorical	{0, 1, 2}
2	Number of output channels	Integer	{1, 2, . . . , 50}
3	Number of full layers	Categorical	{0, 1, 2}
4	Size of the full layer	Integer	{1, 2, . . . , 50}
5	Learning rate	Real	[0; 1]
6	Momentum	Real	[0; 1]
7	Weight decay	Real	[0; 1]
8	Learning decay	Real	[0; 1]

Table 6. Results on MNIST with the Simplified Caffe Blackbox

Algorithm	Average Accuracy on the Validation Set	Average Accuracy on the Test Set
RS	94.02	89.07
SMAC	95.48	97.54
RBFOpt	95.66	97.93
NOMAD	**96.81**	**97.98**

Every section compares HyperNOMAD with **random search (RS)** and a Bayesian method. The hyperopt library [17] is the one used in this comparison since it contains RS in addition to TPE, a Bayesian method that relies on Parzen trees [15]. Note that no comparison with the neural architecture search methods in the AutoML toolbox [27] is included in this study as they operate on a different premise than the one considered here. AutoML exploits previous knowledge on a specific set of applications by building CNNs as a combination of well-defined cells of convolutional networks that were found at an unclear cost. HyperNOMAD, however, starts purposefully without any prior knowledge on the dataset and designs a network from scratch. While this approach can be disadvantageous on some well-studied datasets, it allows HyperNOMAD to stay generic enough to be applicable on any new task. The performances of AutoML are therefore outside the scope of this work and no comparison is carried out between HyperNOMAD and AutoML.

5.1 MNIST

The first tests are performed on the same blackbox provided by the authors of [23], which considers the MNIST dataset with the Caffe library [33]. The NOMAD software is directly used instead of HyperNOMAD, in order to compare the different methods of Table 6. The blackbox takes a simplified set of hyperparameters as described in Table 5, constructs a convolutional neural network that is trained on the MNIST dataset [40], and finally returns the validation accuracy as a performance measure.

The results are obtained by choosing five random seeds and executing the optimization five times for each seed. Table 6 presents the results obtained by RS, RBFOpt, and SMAC, which are taken from [23], and NOMAD. These results show that using NOMAD surpasses all of the other methods in terms of both validation and test accuracy.

Then, HyperNOMAD is tested with PyTorch and its embedded MNIST dataset. The blackbox used for this comparison is the one embedded in HyperNOMAD, which allows for greater flexibility than the previous Caffe blackbox since it takes into account all of the hyperparameters described in Section 3.1.

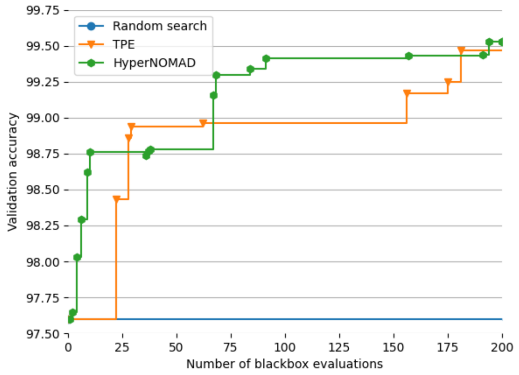


Fig. 9. Comparison between HyperNOMAD, TPE, and RS when launched from the default starting point of HyperNOMAD on the MNIST dataset by representing the successes of each algorithm when a budget of 200 blackbox evaluations is given starting from the default configuration of HyperNOMAD.

The optimization is initialized from the same point corresponding to the default values for the hyperparameters in HyperNOMAD. This initial point has one convolutional layer and two fully connected layers, thus amounting to 17 hyperparameters. The networks are then allowed a budget of 200 epochs. When evaluated, the default configuration obtains a test accuracy of 97.6%. Figure 9 shows the evolution of HyperNOMAD versus the RS and the TPE, both taken from the hyperopt library. After 200 blackbox evaluations, HyperNOMAD finds the best configuration with a final test accuracy of 99.53%. The best solution found by TPE obtains a test accuracy of 99.47% and RS fails to improve on the initial point. Figure 9 shows the evolution of the validation accuracy of the top configuration found by each optimization algorithm.

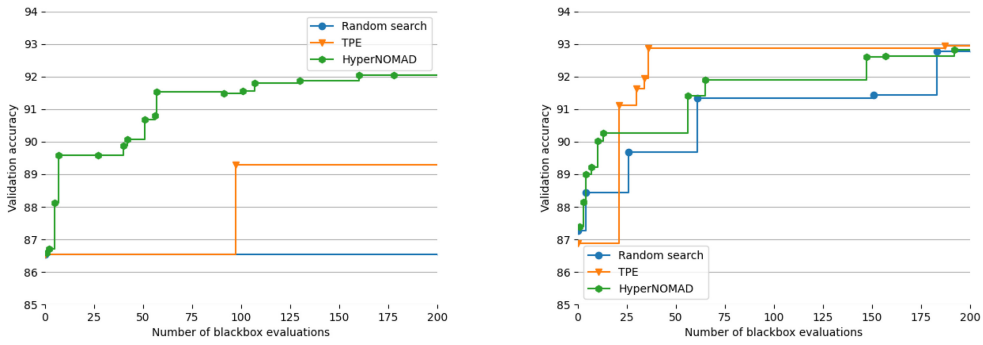
5.2 Fashion-MNIST

For this dataset, HyperNOMAD is again compared against TPE and RS with a budget of 200 blackbox evaluations, meaning that 200 different configurations can be tested.

The first comparison starts from the default initial point of HyperNOMAD, which corresponds to 17 hyperparameters and obtains a test accuracy of 87.28% on Fashion-MNIST. The two categorical variables defining the number of convolutional and fully connected layers can change their values during the same optimization, meaning that the total number of hyperparameters to optimize is dynamic. In this setting, RS does not improve on the initial configuration, TPE obtains one that scores 89.28% and HyperNOMAD ends with 92.05%, as is shown in Figure 10(a).

These results can be explained by the fact that both RS and TPE tend to sample a significant amount of infeasible configurations. A configuration is infeasible when the dimension of the inputs becomes null as it passes through the convolutional layers or when the kernel size is bigger than the image on which it is applied. This situation indicates the presence of hidden constraints [38] that the NOMAD software is able to handle well, contrary to both RS and TPE, which use most of their blackbox evaluation budget on infeasible solutions and are not able to significantly improve on the initial point. HyperNOMAD is more conservative and modifies few values from one configuration to another, which partially explains its ability to surpass the two competitors in this setting. Figure 11 illustrates this behavior by showcasing the number of layers in the configurations evaluated by each optimization algorithm.

To decrease the chances of sampling an infeasible configuration, a second setting is tested where the number of convolutional layers is fixed to 1 and the number of fully connected layers is fixed to 2 during the entire optimization process, which means that all 200 configurations have one



(a) Successes of each method when the number of layers is allowed to change during the optimization.

(b) Successes of each method when the number of layers is fixed during the optimization.

Fig. 10. Comparison between HyperNOMAD, TPE, and RS on the Fashion-MNIST dataset by representing the successes of each algorithm when a budget of 200 blackbox evaluations is given starting from the default configuration of HyperNOMAD.

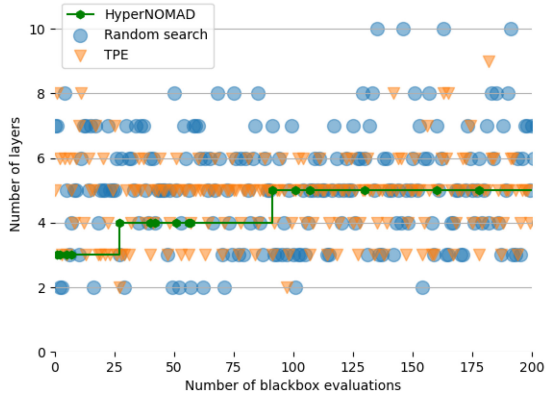


Fig. 11. Number of layers in the configurations sampled by each algorithm during the HPO process on the Fashion-MNIST dataset, with a budget of 200 blackbox evaluations.

convolutional layer and two fully connected ones. Figure 10(b) illustrates the evolution of each optimization method: RS now improves on the initial point by finding a solution with a validation score of 92.77%, HyperNOMAD ends up with a RS score of 92.83%, and TPE finds the best solution with a score of 92.95%.

5.3 CIFAR-10

Similarly to the previous tests, HyperNOMAD is compared to TPE and RS. These tests are initialized with different points, the first being the default values of the hyperparameters in HyperNOMAD with 17 hyperparameters and the second being a network with the VGG-13 architecture. The VGG networks [54] are very deep convolutional neural networks with small kernels. Figure 12 illustrates the architecture of the VGG-16 network.

Figure 13 compares HyperNOMAD, TPE, and RS starting from the default settings of HyperNOMAD, which achieves a test accuracy of 48%. This optimization does not fix any variable, especially the number of convolutional and fully connected layers. RS could not improve on the initial

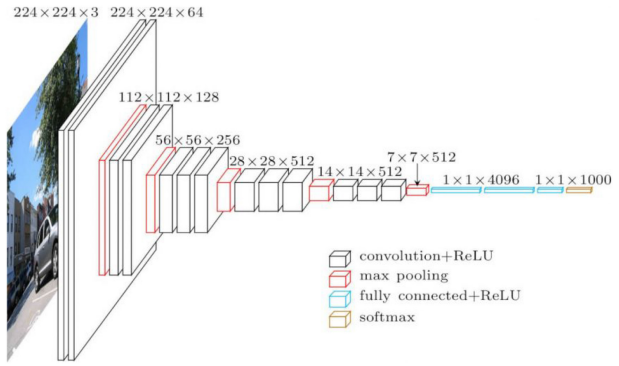
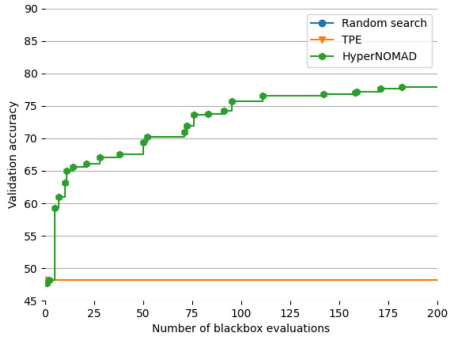
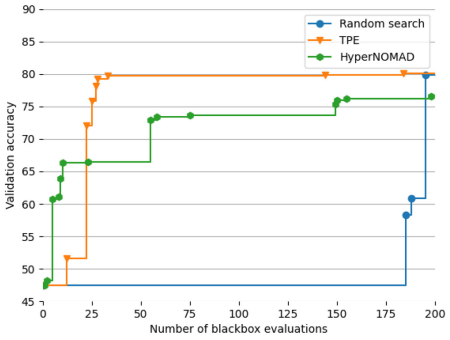


Fig. 12. Architecture of the VGG-16 network. Image taken from [30].



(a) Successes of each method when the number of layers is allowed to change during the optimization. In this case neither TPE nor RS managed to improve on the score of the initial configuration.

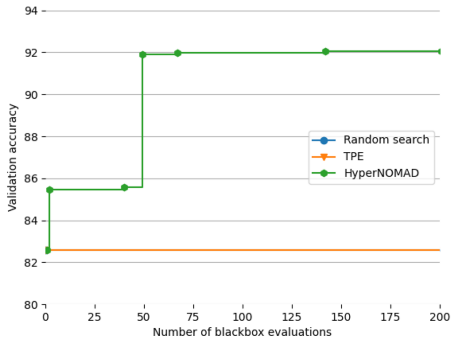


(b) Successes of each method when the number of layers is fixed during the optimization.

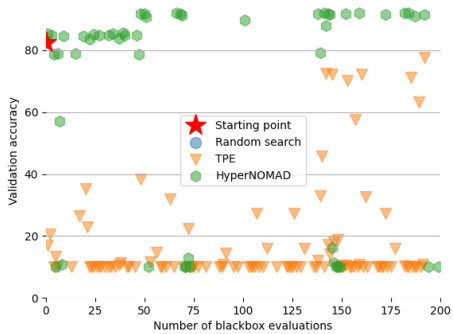
Fig. 13. Comparison between HyperNOMAD, TPE, and RS on the CIFAR-10 dataset by representing the successes of each algorithm when a budget of 200 blackbox evaluations is given starting from the default configuration of HyperNOMAD.

point and neither did TPE. HyperNOMAD finds a solution that achieves 77.94%. Once again, RS and TPE spend their blackbox evaluation budget sampling infeasible configurations. Then, a second run is executed with the number of convolutional and fully connected layers fixed to their initial values of 1 and 2, respectively. Since the starting point is the same as in the previous test, the initial test score is 48% and this time HyperNOMAD improves the least with a final score of 76.57%, RS achieves 79.85%, and TPE finds the best solution with 80.13%. These results on CIFAR-10 highlight the similar behavior with Fashion-MNIST, where both RS and TPE struggle when the dimension of the optimization problem is dynamic but become competitive with HyperNOMAD when the number of layers is fixed.

Figure 14(a) shows the results of a second test performed using an initial point with a VGG architecture with 13 convolutional layers and no fully connected one, which corresponds to 75 hyperparameters, that achieves a test accuracy of 82.57%. The best configuration found by HyperNOMAD achieves a final test accuracy of 92.05%, whereas TPE and RS do not improve on the given initial point for different reasons. RS only samples infeasible architectures, which is expected



(a) Successes of each method when the number of layers is is allowed to change during the optimization. In this case neither TPE nor RS managed to improve on the score of the initial configuration.



(b) Scores of all the feasible architectures sampled by each method during the optimization. RS only sampled infeasible configurations in this case.

Fig. 14. Comparison between HyperNOMAD, TPE, and RS on the CIFAR-10 dataset by representing the successes of each algorithm when a budget of 200 blackbox evaluations is given starting from a VGG-like configuration.

when considering the high dimension of the optimization problem that increases the chances of sampling infeasible configurations. TPE, on the other hand, manages to sample feasible networks but with scores lower than the initial point. This behavior is illustrated in Figure 14(b), where every feasible score obtained by each HPO method is recorded. The best point found by HyperNOMAD is then fully retrained, this time with a budget of 1,000 epochs, and the final test accuracy increases to 93.11%.

6 DISCUSSION

This work introduces HyperNOMAD, a generic framework package for hyperparameter optimization of DNNs using the NOMAD software [37]. The key aspect of this framework is its ability to optimize both the architecture and the optimization phase of a deep neural network simultaneously, and to explore different search spaces during a single execution by taking advantage of categorical variables. The framework obtains good results for the MNIST, Fashion-MNIST, and CIFAR-10 datasets and finds better solutions than TPE and RS. Future work aims at considering a different search space, especially concerning the architecture of the network; considering techniques of data augmentation as additional hyperparameters of the blackbox; adding more flexibility in the way the learning rate is updated; and expanding the framework to other types of problems than classification and providing interfaces compatible with other tools such as Tensorflow or Caffe2.

APPENDICES

A USING HYPERNOMAD

HyperNOMAD is a C++ and Python package dedicated to the hyperparameter optimization of deep neural networks. The package contains a blackbox specifically designed for this problem and provides a link with the NOMAD software [37] used for the optimization. The blackbox takes as inputs the hyperparameters discussed in Section 3.1 and builds a corresponding deep neural network in order to train, validate, and test it on a specific dataset before returning the test accuracy as a measure of performance. NOMAD is then used to minimize this error. The following appendix provides an overview of how to use the HyperNOMAD package.

Prerequisites

HyperNOMAD relies on:

- A compiled version of the NOMAD software available at https://www.gerad.ca/nomad/ for the optimization
- The PyTorch library available at https://pytorch.org/ for modeling the neural network within the blackbox
- A version of Python superior to 3.6
- A version of gcc superior to 3.8

Additionally, HyperNOMAD has the following Python requirements:

- Numpy
- Matplotlib

Installation of HyperNOMAD

HyperNOMAD is available at https://github.com/bbopt/HyperNOMAD. The user must produce the executable `hypernomad.exe` using the provided makefile as follows:

```
> make
    building HYPERNOMAD ...

        To be able to run the example
        the HYPERNOMAD_HOME environment variable
        must be set to the HyperNOMAD home directory
```

When the compilation is successful, a message appears asking to set the `HYPERNOMAD_HOME` environment variable, which can be done by adding a line in the `.profile` or `.bashrc` files:

```
export HYPERNOMAD_HOME=hypernomad_directory
```

The user can check that the installation is successful by trying to run the command:

```
> $HYPERNOMAD_HOME/bin/hypernomad.exe -i

-------------------------------------------------
  HYPERNOMAD - version 1.0
-------------------------------------------------
  Using Nomad version 3.9.0 - www.gerad.ca/nomad
-------------------------------------------------

Run           : hypernomad.exe parameters_file
Info          : hypernomad.exe -i
Help          : hypernomad.exe -h
Version       : hypernomad.exe -v
Usage         : hypernomad.exe -u
Neighbors     : hypernomad.exe -n parameters_file
```

Using HyperNOMAD

The next phase is to create a parameter file that contains the necessary information to specify the classification problem, the search space, and the initial starting point. HyperNOMAD allows

for a good flexibility of tuning a convolutional network by considering multiple aspects of a network at once such as the architecture, the dropout rate, the choice of the optimizer and the hyperparameters related to the optimization aspect (learning rate, weight decay, momentum, etc.), the batch size, and so forth. The user can choose to optimize all these aspects or select a few and fix the others to certain values. The user can also change the default range of each hyperparameter. This information is passed through the parameter file by using a specific syntax where "LB" represents the lower bound and "UB" the upper bound.

```
KEYWORD  INITIAL_VALUE  LB  UB  FIXED/VAR
```

While the hyperparameters have default values in HyperNOMAD, the dataset must be explicitly provided by the user in a separate file in order to specify the considered optimization problem. The following section explains how to specify the necessary parameter file before running an optimization.

Choosing a dataset. The library can be used on different datasets whether they are already incorporated in HyperNOMAD, such as the ones listed in Table 4, or are provided by the user. In the latter case, please refer to the user guide at https://hypernomad.readthedocs.io/en/latest/ for details on how to link a personal dataset to the library. The rest of the section describes how to run an optimization on a dataset provided with HyperNOMAD.

Because of the nature of the applications considered by HyperNOMAD, the computing time can become constraining, especially during the training phase of each configuration, which is why "TOYMNIST" is created as a subset of MNIST containing 300 training images, 100 for the validation, and another 100 for testing. It is added to the package for experimenting with HyperNOMAD without having to wait several hours for each blackbox evaluation.

Specifying the search space. In order to specify the problem to optimize and its parameters, the user must provide a parameter file that contains all the necessary information to run an optimization. As shown below, the parameter file consists of a list of keywords, each corresponding to a hyperparameter, and the values that the user wishes to attribute them. Some of these key words are mandatory such as the dataset, in order to specify the problem, and the number of blackbox evaluations. Other keywords are optional and have default values if they do not appear on the parameter file. Table 7 summarizes all the possible keywords with their default values and ranges. The user can change the lower and upper bounds of a hyperparameter and decide to maintain a hyperparameter at a fixed value during the entire optimization.

Below is a first example of a parameter file that corresponds to the one provided in `$HYPERNOMAD_HOME/examples/mnist_first_example.txt`. First, the MNIST dataset is chosen and HyperNOMAD is allowed to try a maximum of 100 configurations. Then, the number of convolutional layers is fixed throughout the optimization to five. The two "-" appearing after the "5" mean that the default lower and upper bounds are maintained. The kernels, number of fully connected layers, and activation function are respectively initialized to 3, 6, and 2 (Sigmoid), respectively, and the dropout rate is initialized to 0.6 with a new lower bound of 0.3 and upper bound of 0.8 instead of the default range of [0; 1]. Finally, all the remaining hyperparameters from Table 7 that are not explicitly mentioned in this file are fixed to their default values.

Table 7. Keywords for the HyperNOMAD Parameters File

Name	Description	Default Value	Scope
DATASET	Name of the dataset used for the optimization	No default	A dataset from Table 4 or CUSTOM for a custom dataset
NUMBER_OF_CLASSES	Number of classes of the classification problem	No default. Use only if DATASET = CUSTOM	$\{1, 2, \ldots, \infty\}$
MAX_BB_EVAL	Maximum number of calls to the blackbox	No default	$\{1, 2, \ldots, \infty\}$
NUM_CON_LAYERS	Number of convolutional layers	2	$\{0, 1, \ldots, 100\}$
OUTPUT_CHANNELS	Number of output channels for each convolutional layer	6	$\{1, 2, \ldots, 100\}$
KERNELS	Size of the kernel applied to each convolutional layer	5	$\{1, 2, \ldots, 20\}$
STRIDES	Step of the kernel for each convolutional layer	1	$\{1, 2, 3\}$
PADDINGS	Size of the padding for each convolutional layer	0	$\{0, 1, 2\}$
POOLING_SIZE	Size of pooling after each convolutional layer	1	$\{1, 2, 3, 4, 5\}$
NUM_FC_LAYERS	Number of fully connected layers	2	$\{0, 1, \ldots, 500\}$
SIZE_FC_LAYER	Size of each fully connected layer	128	$\{1, 2, \ldots, 1000\}$
BATCH_SIZE	Size of batch for the mini-batch gradient	128	$\{1, 2, \ldots, 400\}$
OPTIMIZER_CHOICE	Optimizer to use from Table 3	3	$\{1, 2, 3, 4\}$
OPT_PARAM_1	Learning rate	0.1	$[0; 1]$
OPT_PARAM_2	Second hyperparameter related to the optimizer	0.9	$[0; 1]$
OPT_PARAM_3	Third hyperparameter related to the optimizer	0.005	$[0; 1]$
OPT_PARAM_4	Fourth hyperparameter related to the optimizer	0	$[0; 1]$
DROPOUT_RATE	Probability that a node will be dropped out	0.5	[0;0.95]
ACTIVATION_FUNCTION	Choice of the activation function from ReLU (1), Sigmoid (2), and Tanh (2)	1	$\{1, 2, 3\}$
REMAINING_HPS	Allows to fix or to vary all the hyperparameters not explicitly mentioned in the parameter file	VAR	{FIXED, VAR}

```
# Mandatory information
DATASET                 MNIST
MAX_BB_EVAL             100

# Optional information
NUM_CON_LAYERS          5  -  -  FIXED # The initial value is fixed
                                        # lower and upper bounds have
                                        # no influence when parameter
                                        # is fixed.

KERNELS                 3 # Only the initial value is set (not fixed)
                          # the lower bound and upper bound
                          # have default values.

NUM_FC_LAYERS           6
ACTIVATION_FUNCTION     2
DROPOUT_RATE            0.6  0.3 0.8  # The lower and upper bounds
                                      # are set to values that are not
                                      # the default ones
REMAINING_HPS           FIXED
```

Below is a second example of a parameter file where the user is only interested in optimizing the fully connected block of a CNN on the MNIST dataset. All the remaining aspects of the network are fixed to their default values throughout the execution of HyperNOMAD. The optimization starts from a point with 10 fully connected layers of the same size of 500 neurons. This parameter file is provided with the package in `$HYPERNOMAD_HOME/examples/mnist_fc_optim.txt`.

```
# Mandatory information
DATASET                 MNIST
MAX_BB_EVAL             150

# Optional information
NUM_FC_LAYERS           10  # Initial value is set to 10
                            # the lower and upper bounds are
                            # the default ones

SIZE_FC_LAYER           500  -  2000 # Initial value is set to 500
                                     # the lower bound is the default one
                                     # the upper bound in now 2000

REMAINING_HPS           FIXED
```

Finally, below is a minimal parameter file where only the mandatory information is specified. The execution of HyperNOMAD starts from the default starting point and all the hyperparameters of Table 7 can be changed. The last line of this file can actually be removed without changing the behavior of HyperNOMAD since the default value for `REMAINING_HPS` is set to `VAR`. Executing HyperNOMAD with this file should return the same values obtained in Figure 9. This file is provided in `$HYPERNOMAD_HOME/examples/cifar10_default.txt`.

```
# Mandatory information
DATASET                 CIFAR10
MAX_BB_EVAL             100

REMAINING_HPS           VAR
```

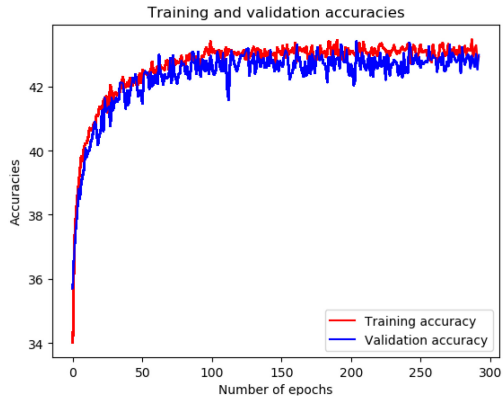


Fig. 15. Example of a window that appears during one evaluation of the blackbox in HyperNOMAD. This figure shows in real time the training and validation accuracies of the current evaluated set of hyperparameters, at each epoch.

Running an execution. The user can run the previous example by executing the following command from the `examples` directory:

```
> $HYPERNOMAD_HOME/bin/hypernomad.exe ./mnist_fc_optim.txt
```

During the optimization, a window appears to plot the training and validation accuracies of the network corresponding to the current point at each epoch as shown in Figure 15. When the optimization is done, HyperNOMAD produces the two files `history.txt` and `stats.txt`. The first one contains each evaluated point and the corresponding testing accuracy, and the second one contains the list of successful points.

ACKNOWLEDGMENTS

The authors would like to thank the Nvidia GPU Grant Program for donating the TITAN Xp GPU used in this research and Dr. Giacomo Nannicini for providing the initial blackbox used for the preliminary testings of HyperNOMAD.

REFERENCES

[1] M. A. Abramson. 2004. Mixed variable optimization of a load-bearing thermal insulation system using a filter pattern search algorithm. *Optimization and Engineering* 5, 2 (2004), 157–177. DOI: https://doi.org/10.1023/B:OPTE.0000033373.79886.54

[2] M. A. Abramson, C. Audet, J. W. Chrissis, and J. G. Walston. 2009. Mesh adaptive direct search algorithms for mixed variable optimization. *Optimization Letters* 3, 1 (2009), 35–47. DOI: https://doi.org/10.1007/s11590-008-0089-2

[3] M. A. Abramson, C. Audet, and J. E. Dennis Jr. 2007. Filter pattern search algorithms for mixed variable constrained optimization problems. *Pacific Journal of Optimization* 3, 3 (2007), 477–500. http://www.ybook.co.jp/online/pjoe/vol3/pjov3n3p477.html.

[4] C. Audet, V. Béchard, and S. Le Digabel. 2008. Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search. *Journal of Global Optimization* 41, 2 (2008), 299–318. DOI: https://doi.org/10.1007/s10898-007-9234-1

[5] C. Audet, C.-K. Dang, and D. Orban. 2014. Optimization of algorithms with OPAL. *Mathematical Programming Computation* 6, 3 (2014), 233–254. DOI: https://doi.org/10.1007/s12532-014-0067-x

[6] C. Audet and J. E. Dennis Jr.2001. Pattern search algorithms for mixed variable programming. *SIAM Journal on Optimization* 11, 3 (2001), 573–594. http://link.aip.org/link/?SJE/11/573/1

[7] C. Audet and J. E. Dennis Jr.2006. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization* 17, 1 (2006), 188–217. DOI: https://doi.org/10.1137/040603371

[8] C. Audet and W. Hare. 2017. *Derivative-Free and Blackbox Optimization*. Springer International Publishing, Cham, Switzerland. DOI: https://doi.org/10.1007/978-3-319-68913-5

[9] C. Audet, S. Le Digabel, and C. Tribes. 2019. The mesh adaptive direct search algorithm for granular and discrete variables. *SIAM Journal on Optimization* 29, 2 (2019), 1164–1189. DOI: https://doi.org/10.1137/18M1175872

[10] C. Audet and D. Orban. 2006. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization* 17, 3 (2006), 642–664. http://dx.doi.org/doi:10.1137/040620886

[11] C. Audet and C. Tribes. 2018. Mesh-based Nelder-Mead algorithm for inequality constrained optimization. *Computational Optimization and Applications* 71, 2 (2018), 331–352. DOI: https://doi.org/10.1007/s10589-018-0016-0

[12] B. Baker, O. Gupta, N. Naik, and R. Raskar. 2016. *Designing Neural Network Architectures Using Reinforcement Learning*. Technical Report. arXiv. http://arxiv.org/abs/1611.02167

[13] P. Balaprakash, M. Salim, T. Uram, V. Vishwanath, and S. Wild. 2018. DeepHyper: Asynchronous hyperparameter search for deep neural networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC'18)*. IEEE, Bengaluru, India, 42–51. DOI: https://doi.org/10.1109/HiPC.2018.00014

[14] Y. Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*. Springer, Berlin, 437–478.

[15] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, 2546–2554.

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

[17] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML'13)*, Vol. 28. JMLR.org, Atlanta, GA, I–115–I–123. http://dl.acm.org/citation.cfm?id=3042817.3042832

[18] T. Bosc. 2016. *Learning to Learn Neural Networks*. Technical Report. arXiv. http://arxiv.org/abs/1610.06072

[19] L. Bottou. 2012. *Stochastic Gradient Descent Tricks*. Lecture Notes in Computer Science (LNCS), Vol. 7700. Springer, Berlin, 430–445. https://www.microsoft.com/en-us/research/publication/stochastic-gradient-tricks/

[20] X. Bouthillier, C. Tsirigotis, F. Corneau-Tremblay, P. Delaunay, R. Askari, D. Suhubdy, M. Noukhovitch, D. Serdyuk, A. Bergeron, P. Henderson, P. Lamblin, M. Bronzi, and C. Beckham. 2019. Oríon - Asynchronous Distributed Hyperparameter Optimization. Retrieved September 19, 2020, from *https://github.com/Epistimio/orion*. DOI: https://doi.org/10.5281/zenodo.3478592

[21] A. R Conn, K. Scheinberg, and L. N. Vicente. 2009. Global convergence of general derivative-free trust-region algorithms to first and second order critical points. *SIAM Journal on Optimization* 20, 1 (2009), 387–415. DOI: https://doi.org/10.1137/060673424

[22] A. Deshpande. 2019. A Beginner's Guide to Understanding Convolutional Neural Networks. https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks. https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks

[23] G. Diaz, A. Fokoue, G. Nannicini, and H. Samulowitz. 2017. An effective algorithm for hyperparameter optimization of neural networks. *IBM Journal of Research and Development* 61, 4 (2017), 9:1–9:11. DOI: https://doi.org/10.1147/JRD.2017.2709578

[24] J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12 (2011), 2121–2159.

[25] T. Elsken, J. H. Metzen, and F. Hutter. 2018. *Neural Architecture Search: A Survey*. Technical Report. arXiv. http://arxiv.org/abs/1808.05377

[26] T. Elsken, J. H. Metzen, and F. Hutter. 2019. *Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution*. Technical Report. International Conference on Learning Representations, New Orleans, LA. https://openreview.net/forum?id=ByME42AqK7

[27] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., Montreal, Canada, 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[28] H. Ghanbari and K. Scheinberg. 2017. *Black-Box Optimization in Machine Learning with Trust Region Based Derivative Free Algorithm*. Technical Report. arXiv. http://arxiv.org/abs/1703.06925

[29] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. 2017. Google Vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Association for Computing Machinery, New York, NY, 1487–1495.

[30] M. Hassan. 2019. VGG16: Convolutional Network for Classification and Detection. https://neurohive.io/en/popular-networks/vgg16/.

[31] R. Hooke and T. A. Jeeves. 1961. "Direct search" solution of numerical and statistical problems. *Journal of the Association for Computing Machinery* 8, 2 (1961), 212–229. DOI: https://doi.org/10.1145/321062.321069

[32] F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, Berlin, 507–523.

[33] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, Association for Computing Machinery, New York, NY, 675–678.

[34] D. P. Kingma and L. B. Jimmy. 2015. *Adam: A Method for Stochastic Optimization*. Technical Report. arXiv. https://arxiv.org/abs/1412.6980

[35] M. Kokkolaras, C. Audet, and J. E. Dennis Jr.2001. Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. *Optimization and Engineering* 2, 1 (2001), 5–29. DOI: https://doi.org/10.1023/A:1011860702585

[36] A. Krizhevsky and G. Hinton. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. Citeseer.

[37] S. Le Digabel. 2011. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software* 37, 4 (2011), 44:1–44:15. DOI: https://doi.org/10.1145/1916461.1916468

[38] S. Le Digabel and S. M. Wild. 2015. *A Taxonomy of Constraints in Simulation-Based Optimization*. Technical Report G-2015-57. Les cahiers du GERAD. http://www.optimization-online.org/DB_HTML/2015/05/4931.html

[39] Y. A. LeCun, L. Bottou, G. B. Orr, and K. R. Müller. 2012. *Efficient BackProp*. Springer, Berlin,9–48. DOI: https://doi.org/10.1007/978-3-642-35289-8_3

[40] Y. LeCun and C. Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/

[41] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. 2018. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *International Journal of Robotics Research* 37, 4–5 (2018), 421–436. DOI: https://doi.org/10.1177/0278364917710318

[42] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18 (2018), 1–52.

[43] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, G. Van Bram, and C. L. Sánchez. 2017. A survey on deep learning in medical image analysis. *Medical Image Analysis* 42 (2017), 60–88. DOI: https://doi.org/10.1016/j.media.2017.07.005

[44] J. Liu, N. Ploskas, and N. V. Sahinidis. 2018. Tuning BARON using derivative-free optimization algorithms. *Journal of Global Optimization* 74, 4 (2018), 611–637. DOI: https://doi.org/10.1007/s10898-018-0640-3

[45] P. R. Lorenzo, J. Nalepa, M. Kawulok, L. S. Ramos, and J. R. Pastor. 2017. Particle swarm optimization for hyper-parameter selection in deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Association for Computing Machinery, New York, NY, 481–488.

[46] I. Loshchilov and F. Hutter. 2016. *CMA-ES for Hyperparameter Optimization of Deep Neural Networks*. Technical Report. arXiv. http://arxiv.org/abs/1604.07269

[47] A. R. Mello, J. de Matos, M. R. Stemmer, A. de Souza Britto Jr, and A. L. Koerich. 2019. *A Novel Orthogonal Direction Mesh Adaptive Direct Search Approach for SVM Hyperparameter Tuning*. Technical Report. arXiv. http://arxiv.org/abs/1904.11649

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., New York, NY, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[49] V. Pavlovsky. 2019. Introduction to Convolutional Neural Networks. https://www.vaetas.cz/posts/intro-convolutional-neural-networks.

[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[51] M. Porcelli and Ph.L. Toint. 2017. BFO, A trainable derivative-free brute force optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables. *ACM Transactions on Mathematical Software* 44, 1 (2017), 6:1–6:25. DOI: https://doi.org/10.1145/3085592

[52] M. J. D. Powell. 2009. *The BOBYQA Algorithm for Bound Constrained Optimization without Derivatives*. Technical Report DAMTP 2009/NA06. Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, England. http://www.damtp.cam.ac.uk/user/na/NA_papers/NA2009_06.pdf.

[53] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. 2018. *Regularized Evolution for Image Classifier Architecture Search*. Technical Report. arXiv. http://arxiv.org/abs/1802.01548

[54] K. Simonyan and A. Zisserman. 2014. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Technical Report. arXiv. http://arxiv.org/abs/1409.1556

[55] S. C. Smithson, G. Yang, W. J. Gross, and B. H. Meyer. 2016. Neural networks designing neural networks: Multi-objective hyper-parameter optimization. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16)*. IEEE, Association for Computing Machinery, New York, NY, 1–8.

[56] J. Snoek, H. Larochelle, and R. Prescott Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS'12) 25*. Curran Associates Inc., Red Hook, NY, 2960–2968. https://dash.harvard.edu/handle/1/11708816

[57] M. Suganuma, S. Shirakawa, and T. Nagao. 2017. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, International Joint Conferences on Artificial Intelligence Organization, 497–504.

[58] T. Tieleman and G. Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*. 26–31 pages. https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf.

[59] V. Torczon. 1997. On the convergence of pattern search algorithms. *SIAM Journal on Optimization* 7, 1 (1997), 1–25. DOI: https://doi.org/10.1137/S1052623493250780

[60] M. Wistuba, N. Schilling, and L. Schmidt-Thieme. 2018. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning* 107, 1 (2018), 43–78. DOI: https://doi.org/10.1007/s10994-017-5684-y

[61] H. Xiao, K. Rasul, and R. Vollgraf. 2017. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. arXiv:cs.LG/cs.LG/1708.07747

[62] Yelp. 2014. Metric Optimization Engine. https://github.com/Yelp/MOE.

[63] S. R. Young, D. C. Rose, T. P. Karnowski, S. H. Lim, and R. M. Patton. 2015. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, Association for Computing Machinery, New York, NY, 1–5.

[64] A. Zela, A. Klein, S. Falkner, and F. Hutter. 2018. *Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search*. Technical Report. arXiv. http://arxiv.org/abs/1807.06906

[65] B. Zoph and Q. V. Le. 2016. *Neural Architecture Search with Reinforcement Learning*. Technical Report. arXiv. http://arxiv.org/abs/1611.01578

[66] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Salt Lake City, UT, 8697–8710.

Received June 2019; revised February 2021; accepted February 2021