Les Cahiers du GERAD ISSN: 0711–2440

HyperNOMAD: Hyperparameter optimization of deep neural networks using mesh adaptive direct search

D. Lakhmiri,
S. Le Digabel, C. Tribes

G–2019–46

July 2019

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2019
– Bibliothèque et Archives Canada, 2019

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2019
– Library and Archives Canada, 2019

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

HyperNOMAD: Hyperparameter optimization of deep neural networks using mesh adaptive direct search

Dounia Lakhmiri
Sébastien Le Digabel
Christophe Tribes

GERAD & Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada, H3C 3A7

dounia.lakhmiri@gerad.ca
sebastien.le.digabel@gerad.ca
christophe.tribes@gerad.ca

July 2019
Les Cahiers du GERAD
G–2019–46

Abstract: 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), and that allows for an important flexibility in the exploration of the search space by taking advantage of categorical variables. This new approach is tested on the MNIST and CIFAR-10 data sets and achieves results comparable to the current state of the art.

Keywords: Deep neural networks, neural architecture search, hyperparameter optimization, blackbox optimization, derivative-free optimization, mesh adaptive direct search, categorical variables

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

1 Introduction

Neural networks are mathematical structures used to solve supervised classification problems such as images, sounds and speech, to name a few. In the recent years, neural networks gained in popularity and were declined in different versions: deep, convolutional, recurrent, etc. in order to adapt to specific problematics. This popularity is due to the emergence of large size databases and the development of 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 [39], medical diagnostics [41], and more.

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 the derivatives are unavailable, although they might exist. This is the case for example when the objective and/or constraints 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 no 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 follows:

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

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 interpolations models [50]. This differentiates them from direct search methods [30] 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 [57] 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 one. 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 of resolution, 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 data set as a mesure 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.

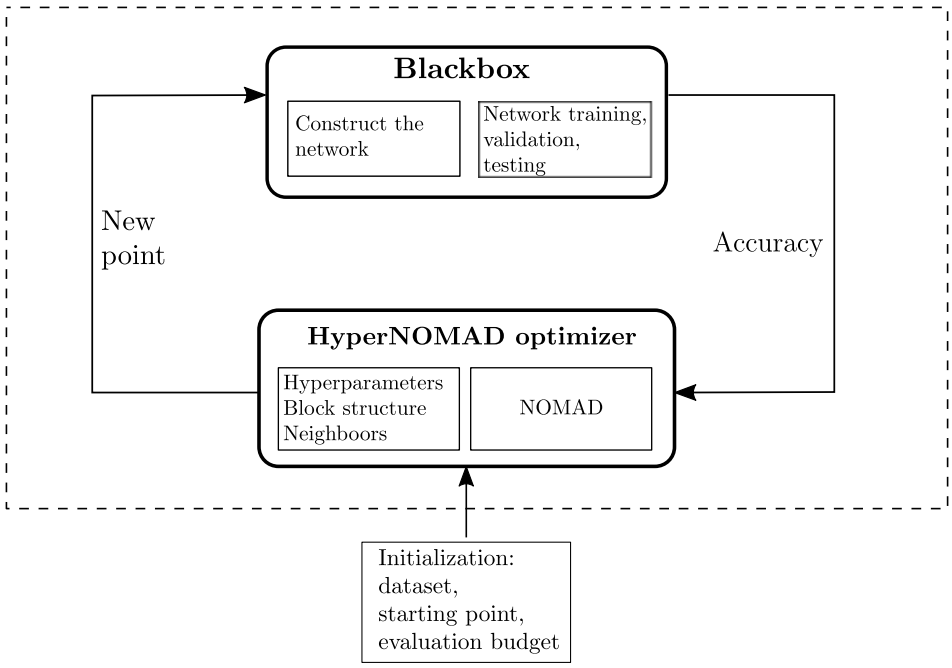


Figure 1: The HyperNOMAD workflow.

The manuscript 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 automatization of this process in order to adapt to each problematic.

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 to achieve state of the art performances [51, 62] on some benchmark problems, although at a massive computational cost of 800 GPUs for a few weeks. Typically, one would perform a NAS first and then start tuning the other hyperparameters with the optimized architecture. However, Zela et al. [61] 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 approach used to tackle the HPO problem of neural networks is the grid search. This method consists of discretizing the hypercube defined by the range of each hyperparameter

and then evaluating each points on the grid. This technique is still used today and is implemented in several HPO libraries such as scikit-learn and Spearmint [48, 54]. 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, the grid search ignores 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 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 [40] was introduced, which is a variant of a random search that uses an early stopping criteria 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 the 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 allowing to explore the space more wisely even if they remain impregnated with a random character. These algorithms are often used to optimize hyperparameters [26, 55, 60]. In [43], 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 [44] 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, 62, 63] where the main difference between each method relies on how the agents are defined and dealt with. In [53], a neural network is able to design other neural networks by learning to explore the possible configurations. Here, the HPO of neural networks is seen as a multiobjective 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 neurones to learn the parameters of another multi-layer network that is tested on a binary classification problem.

Derivative-Free Optimization (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 nonexistent or inoperable derivatives. In [10], the authors propose a general way of modeling hyperparameter optimization problems as a blackbox optimization problem. This formulation is used in [42] 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 resolution time, sometimes by more than 50%. Another formulation inspired by robust optimization is used in [49], in addition to that of [10], to optimize the hyperparameters of the BFO algorithm [49]. BBO methods are also at the heart of Google Vizier [28] which is a tool that can be used for the HPO problem of machine learning algorithms, and especially for deep neural networks.

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 informations collected during previous assessments to diagnose the search space and predict which areas to explore first. Among them, Gaussian processes (GP) 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 [54, 58]. However, the disadvantage of GPs is that they do not fit well to the categorical features, and its performance depends on the choice of the kernel function that defines it. 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 data set [37], then on a problem of interactions between drugs. These tests show that this model provides comparable or better results than popular configurations. In [27], a trust-region DFO algorithm is applied to optimize the hyperparameters of a SVM model. Here again, this approach obtained a more efficient model than those defined by the experts or by a Bayesian algorithm.

Thus, 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 into the NOMAD package [36], especially as it can handle integer and categorical variables [2, 9]. Using the MADS algorithm for hyperparameter tuning has been validated in [45] where a SVM model is calibrated using MADS combined with the Nelder-Mead and VNS search strategies [4, 11].

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

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**	**Direct-search**	**Real**	**Int.**	**Cat.**
scikit-learn [48]	✓	✓	-	-	-	✓	✓	✓
hyperopt [17]	-	✓	✓	-	-	✓	✓	✓
Spearmint [54]	✓	✓	✓	-	-	✓	✓	✓
SMAC [31]	-	-	-	✓	-	✓	✓	✓
MOE [59]	-	-	✓	-	-	✓	-	-
RBFOpt [23]	-	-	-	✓	-	✓	✓	-
DeepHyper [13]	-	✓	-	✓	-	✓	✓	✓
Oríon [20]	-	✓	-	-	-	✓	✓	✓
Google Vizier [28]	✓	✓	✓	✓	-	✓	✓	✓
HyperNOMAD	-	-	-	-	✓	✓	✓	✓

3 Experimental setup

This section first defines the blackbox approach used for modeling the HPO problem. This is done by listing the different hyperparameters considered to construct, train and validate a deep neural network (DNN) in order to obtain its test accuracy. The second part of the section gives an overview of the data sets 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.

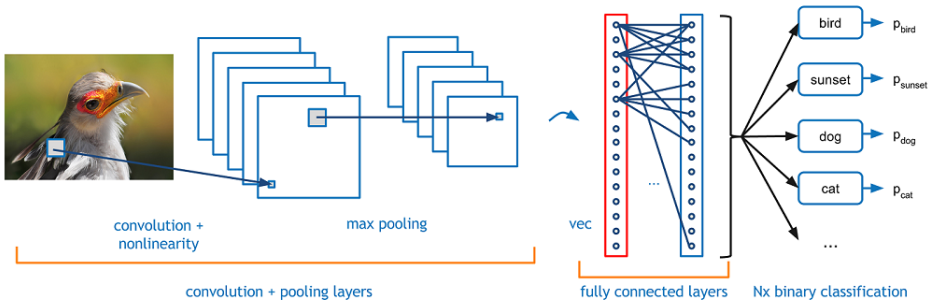


Figure 2: Example of a convolutional neural network. Image taken from [22].

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 3a represents the steps of a convolution operation. The initial image is a 5×5 matrix whose borders are padded with zeros. The convolution consists of choosing a kernel that is passed over the image in order to compute the coefficients of the feature map. 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. In Figure 3a, the coefficient (2,1) of the feature map is obtained by the following operation: $(0\times0)+(0\times0)+(0\times1)+(0\times0)+(21\times1)+(0\times0)+(85\times1)+(71\times0)+(0\times0)=106$. 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 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 padding. In Figure 3a, the image is padded with one layer of zeros.

When the feature map is obtained, one can decide to apply a pooling operation to decrease the size of the output by keeping only the biggest coefficients in a certain area. Figure 3b illustrates a 2×2 pooling that results in an output of half the size of the feature map.

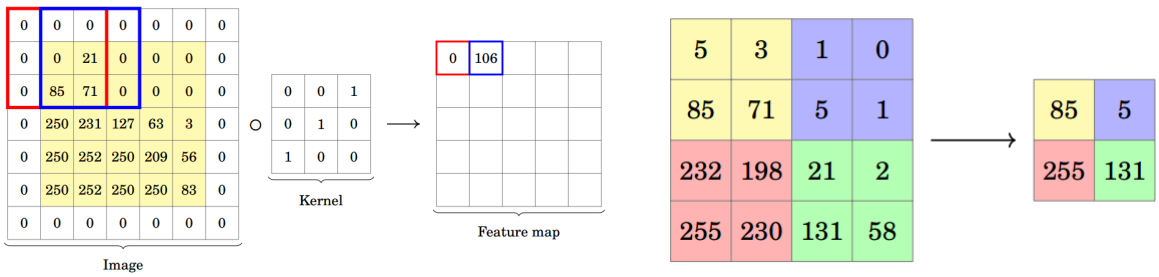


Figure 3: Illustration of a convolution operation in (a) and a pooling operation in (b). Images taken from [47].

Each fully connected layer that follows the convolutional ones is determined by the number of neurones it contains. The neurones 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 neurones of the layer l and a_j be the value of the neurone 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 neurones of the layer l to the $j-th$ neurone 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.

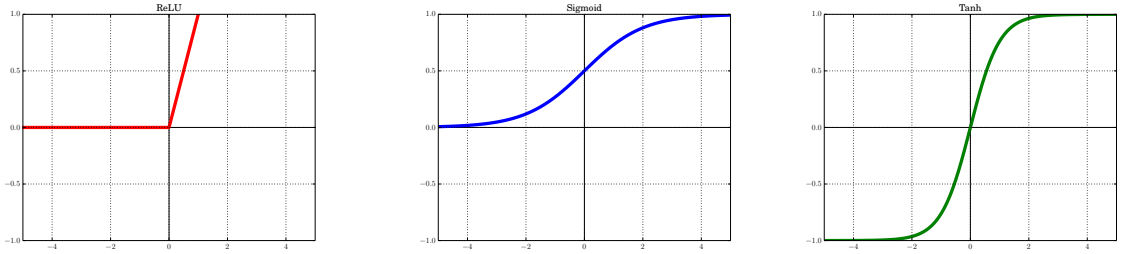


Figure 4: Examples of activation functions.

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 the 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 $5n_1 + n_2 + 4$.

Table 2: Hyperparameters that define the architecture of a neural network.

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

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 attached to the validation data. Let Θ be a multi-dimensional matrix that stores the weights of the arcs that link each layer of the network l with the next one $l+1$, and let $J(\Theta)$ the sum of the errors between the predictions and the labels for all the data. 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 this problem 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 accordingly to avoid oscillations or divergence. Substantial research and tricks of the trade are developed to solve this problematic [14, 19, 38]. The optimizers Adam [33], Adagrad [24] and RMSProp [56] have embedded strategies to adapt the learning rate at each iteration and for each weight. SGD however does not require external management. In HyperNOMAD, the learning rate of SGD is divided by 10 every 100 epochs as long as its value is greater than 10^{-6}.

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 and always four real hyperparameters related to it. This aspect of the network relies on defining five hyperparameters in total.

Table 3: Choices of the optimizer and the corresponding hyperparameters.

Optimizer	Hyperparameter	Type	Scope
Stochastic Gradient Descent (SGD)	Initial learning rate	Real	[0;1]
	Momentum	Real	[0;1]
	Dampening	Real	[0;1]
	Weight decay	Real	[0;1]
Adam	Initial learning rate	Real	[0;1]
	β_1	Real	[0;1]
	β_2	Real	[0;1]
	Weight decay	Real	[0;1]
Adagrad	Initial learning rate	Real	[0;1]
	Learning rate decay	Real	[0;1]
	Initial accumulator	Real	[0;1]
	Weight decay	Real	[0;1]
RMSProp	Initial learning rate	Real	[0;1]
	Momentum	Real	[0;1]
	α	Real	[0;1]
	Weight decay	Real	[0;1]

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 do 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 size of the mini-batches is an integer hyperparameter that varies between $[1, n_{train}]$, 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 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 (usually 500) 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 staggers or when it stays lower than 20% after 50 epochs then the training can be interrupted in order to save time and computational ressources. 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]. This 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 data set. This blackbox problem is solved using the NOMAD software [36] described in Section 4.1.

3.2 Data sets

The HyperNOMAD package comes with a selection of data sets all meant for classification problems. Table 4 lists the data sets embedded so far through PyTorch [46], a relatively complete tool to model and manipulate deep neural networks. HyperNOMAD also allows the usage of a personal data set by following the instructions given in Appendix A. When loading a data set 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 data sets used for benchmarking HyperNOMAD. First, a validation is done using MNIST [37] and once positive results are obtained, the second and more complex data set, CIFAR-10 [35], is considered.

Table 4: Data sets embedded in HyperNOMAD.

Data set	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

3.2.1 MNIST

MNIST [37] is a data set 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 data set 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 CIFAR-10

The second set of tests are performed with CIFAR-10 [35]. This data set contains 60,000 colored images of objects that belong to ten 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 this test, 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 on CIFAR-10 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 programs wrapped into a blackbox responsible for constructing, training and evaluating the test accuracy of a neural network depending on the values of the hyperparameters described in Section 3. This blackbox uses the PyTorch package [46] for its simplicity. HyperNOMAD also contains an interface that runs the optimization of the blackbox using the NOMAD software [36] 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 [36] 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* $M^k = \left\{x + \text{diag}(\boldsymbol{\delta}^k)z : x \in V^k, z \in \mathbb{Z}^n\right\}$ 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 hyper-cube sampling, etc. The *poll*, on the other hand, is strictly defined since the convergence theory of MADS relies solely on this phase. Here, the algorithm generates directions around the current iterate x^k

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

to search for candidates locally in a region centered around x^k and of radius, in each dimension, of $\boldsymbol{\Delta}^k \in \mathbb{R}^n$, which is called the *poll size vector*. The set of candidates in this phase defines the *poll set* P_k.

If MADS finds a better point 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 $\boldsymbol{\delta}^k \leq \boldsymbol{\Delta}^k$ is maintained. This relation insures that the set of search directions becomes dense in the unit sphere asymptotically. The MADS algorithm is summarized in Algorithm 1.

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 **[3]**
[2] Poll
Evaluate the points in the poll set P_k
[3] 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]**

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, 2, 3, 6, 34].

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, 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 constitute 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 connected layers. Here, each layer is defined with the number of neurones 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 variable.

2	16	5	1	1	0	7	3	1	1	1					
3	16	5	1	1	0	7	3	1	1	1	7	3	1	1	1
1	16	5	1	1	0										

Figure 5: Example of a convolution block (top). Its first neighbor is obtained by adding a convolutional layer (middle) and the second neighbor is obtained by subtracting a convolutional layer (bottom).

3	1200	512	20	
4	1200	1200	512	20
2	512	20		

Figure 6: Example of a fully connected block (top). Its first neighbor is obtained by adding a fully connected layer (middle) and the second neighbor is obtained by subtracting a fully connected layer (bottom).

1	0.2	0.95	$1e^{-4}$	0.03
2	0.1	0.9	$5e^{-4}$	0

Figure 7: Example of an optimizer block. Its neighbor is obtained by selecting the next optimizer and by initializing the associated variables to their default values.

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 considers each categorical variable independently. Hence, to create a neighbor point related to a given block, all the remaining values are fixed at the current iterate values. The neighbor structure of the convolution block is obtained by adding and subtracting a group of associated variables at the right 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 most right 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 left 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 most left 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 not 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 looping between the optimizers listed in Table 3 from top to bottom. 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 initial values. 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. The choice of activation function and the remaining variables are not treated as categorical variables and no neighborhood structure is required.

5 Computational results

This section summarizes the results obtained by HyperNOMAD and compares them to other methods when applied to the MNIST [37] and CIFAR-10 [35] data sets. For both 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 are allowed a maximum of 100 blackbox evaluations due to time limitations since one call to the blackbox takes, in average, three to four hours.

5.1 MNIST

The first tests are performed on the same blackbox provided by the authors of [23] which considers the MNIST data set with the Caffe library [32]. 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 data set [37], and finally returns the validation accuracy as a measure of performance.

Table 5: Hyperparameters considered for the tests on the MNIST data set 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]

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 a random sampling (RS), RBFOpt, and SMAC, that are taken from [23], and NOMAD. These results show that using NOMAD surpasses all of the other methods in terms of both the validation and the test accuracies.

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**

The next phase consists to test HyperNOMAD, this time with PyTorch and its embedded MNIST data set, and to compare it with other methods such as a random search and a Bayesian method. The hyperopt library [17] is the one used in this comparison since it contains a random search in addition to TPE, a Bayesian method that relies on Parzen trees [15]. The blackbox used for this comparison is the one embedded in HyperNOMAD which allows for a greater flexibility than the previous Caffe blackbox since it takes into account all of the hyperparameters described in Section 3.1. The optimization is launched from the same point that corresponds to the default values for the hyperparameters in HyperNOMAD. This initial point contains 22 hyperparameters and obtains a test accuracy of 93.36%. Figure 8 shows the evolution of HyperNOMAD versus the two variants of hyperopt (RS and TPE) for this test where, after 100 blackbox evaluations, HyperNOMAD finds the best configuration with a final test accuracy of 99.61%. The best solution found by TPE obtains a test accuracy of 99.17% and the random search fails to improve the initial point.

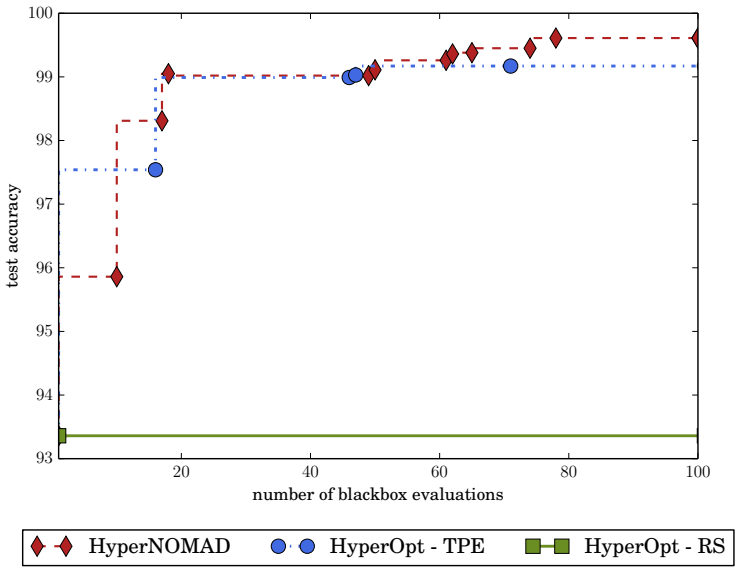


Figure 8: Comparison between HyperNOMAD**, TPE and RS when launched from the default starting point of** HyperNOMAD**, on the MNIST data set.**

5.2 CIFAR-10

Similarly to the previous test, HyperNOMAD is compared to TPE and the random search. These tests are launched using different starting points, the first being the default values of the hyperparameters in HyperNOMAD with 22 hyperparameters and the second being a network with the VGG-13 architecture. The VGG networks [52] are very deep convolutional neural networks with small kernels. Figure 9 illustrates the architecture of the VGG 16 network.

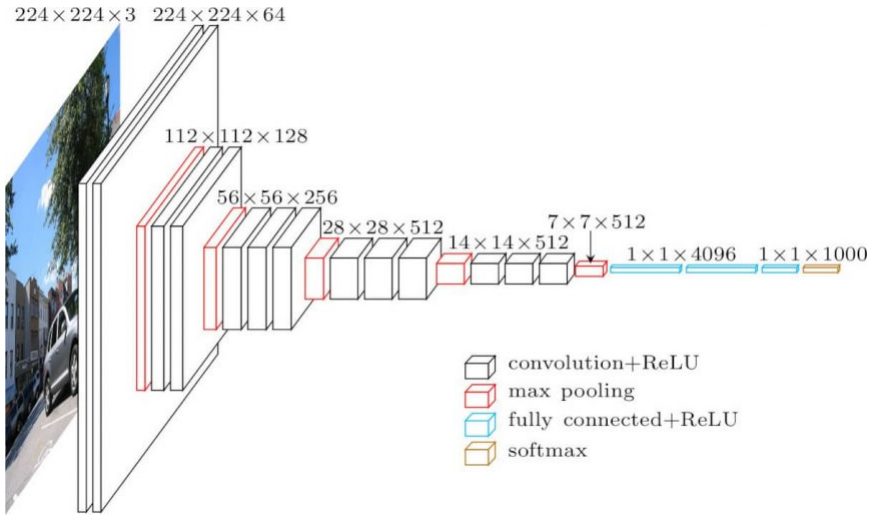
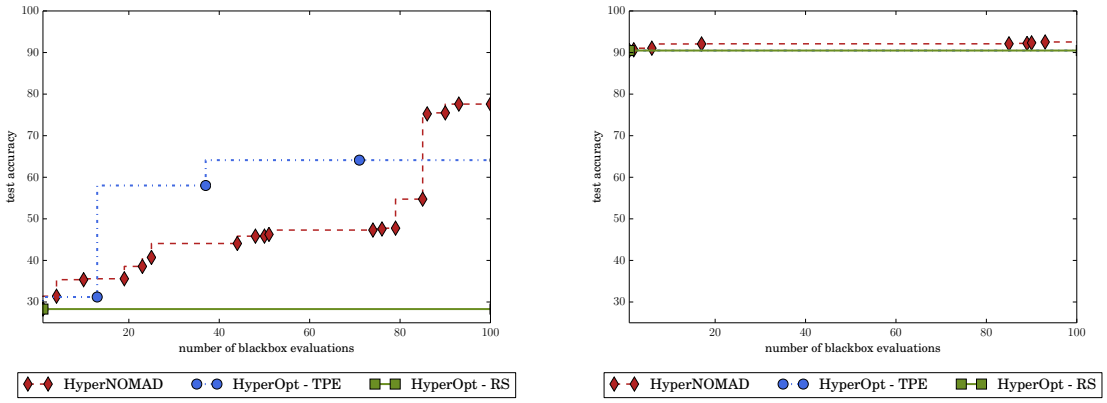


Figure 9: Architecture of the VGG-16 network. Image taken from [29].

Figure 10a compares HyperNOMAD, TPE and the random search starting from the default settings of HyperNOMAD which achieve a test accuracy of 28.3%. Once again, the random search could not bring any improvements to the initial point whereas the best solution of TPE obtains a test accuracy of 64.12% and HyperNOMAD finds a solution that achieves 77.6%.

Figure 10b shows the results of a second test performed using a starting point with a VGG-13 architecture, which corresponds to 62 hyperparameters, that achieves a test accuracy of 90.8%. In this example neither the random search nor TPE are able to improve on the initial point given. Moreover, they spend all their evaluation budget sampling non feasible architectures. An architecture is infeasible when the size of the image passed through the convolutional layers becomes nil. This behavior can be explained by the sampling strategy of both methods since they tend to change multiple hyperparameters at once thus increasing the probability of obtaining a non feasible architecture. HyperNOMAD is much more conservative when choosing a new point to evaluate which is why 50 evaluated points are feasible. The best configuration found by HyperNOMAD achieves a final test accuracy of 92.54%.



(a) Default starting point. **(b) Starting from a VGG architecture.**

Figure 10: Comparison between HyperNOMAD, TPE and RS, on the CIFAR-10 data set.

6 Discussion

This work introduces HyperNOMAD, a framework package for hyperparameter optimization of DNNs using the NOMAD software [36]. The key aspects of this framework is its ability to optimize both the architecture and the optimization phase of a deep neural network simultaneously on the one hand, and to explore different search spaces during a single execution by taking advantage of categorical variables. The framework obtains good results for both the MNIST and CIFAR-10 data sets and finds better solutions than TPE and a random search as is illustrated in Figure 10a. Future work aims at considering different 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 provide 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 problematic and provides a link with the NOMAD software [36] used for the optimization. The blackbox takes as inputs the hyperparameters discussed in Section 3.1, builds a corresponding deep neural network in order to train, validate and test it on a specific data set before returning the test accuracy as a mesure 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.

Installation of HyperNOMAD

HyperNOMAD is available at https://github.com/DouniaLakhmiri/Hyper NOMAD . 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 an environment variable `HYPERNOMAD_HOME` 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
Neighboors    : 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, etc. 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.

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

While the hyperparameters have default values in HyperNOMAD, the data set 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 data set

The library can be used on different data sets 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 in https://hypernomad.readthedocs.io/en/latest/ for details on how to link a personal data set to the library. The rest of the section describes how to run an optimization on a data set 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 informations 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 data set, 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 to 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 data set 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) 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.

```
# 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 a the MNIST data set. 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 neurones. 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 10a. This file is provided in `$HYPERNOMAD_HOME/examples/cifar10_default.txt`.

```
# Mandatory information
DATASET                 CIFAR10
MAX_BB_EVAL             100

REMAINING_HPS           VAR
```

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 11. When the optimization is done, HyperNOMAD produces the two files `history.txt` and `stats.txt`. The first contains each evaluated point and the corresponding testing accuracy, and the second contains the list of successful points.

Table 7: Keywords for the HyperNOMAD parameters file.

Name	Description	Default value	Scope
`DATASET`	Name of the data set used for the optimization	No default	A data set from Table 4 or `CUSTOM` for a custom data set
`NUMBER_OF_CLASSES`	Number of classes of the classification problem	No default. Only use if `DATASET` = `CUSTOM`	$\mathbb{N}$
`MAX_BB_EVAL`	Maximum number of calls to the blackbox	No default	$[1; \infty]$
`NUM_CON_LAYERS`	Number of convolutional layers	`2`	[0;100]
`OUTPUT_CHANNELS`	Number of output channels for each convolutional layer	`6`	[1;100]
`KERNELS`	Size of the kernel applied to each convolutional layer	`5`	[1;20]
`STRIDES`	Step of the kernel for each convolutional layer	`1`	[1;3]
`PADDINGS`	Size of the padding for each convolutional layer	`0`	[0;2]
`DO_POOLS`	Apply a pooling after each convolutional layer	`0`	{0,1}
`NUM_FC_LAYERS`	Number of fully connected layers	`2`	[0;500]
`SIZE_FC_LAYER`	Size of each fully connected layer	`128`	[1;1,000]
`BATCH_SIZE`	Size of batch for the mini-batch gradient	`128`	[1;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, Sigmoid and Tanh	`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`}

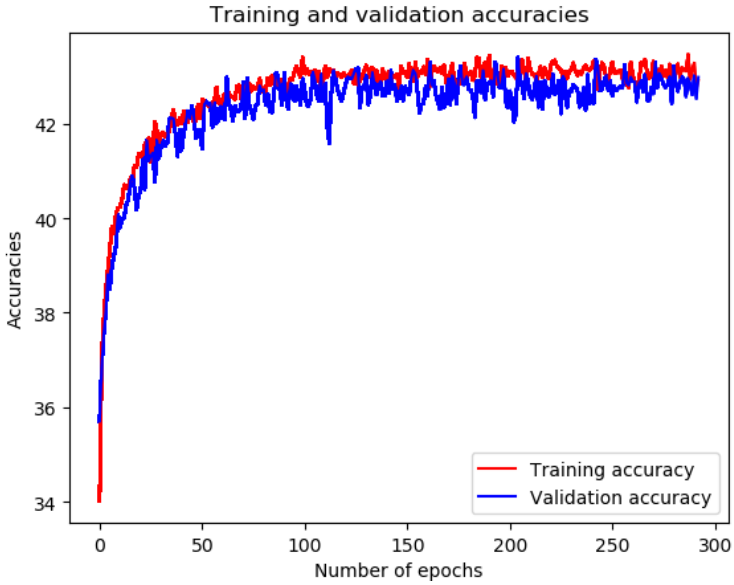


Figure 11: Example of a window that appears during one evaluation of the blackbox in HyperNOMAD. This figure shows in real time the training and validation accuracies of the current evaluated set of hyperparameters, at each epoch.

References

[1] M.A. Abramson. Mixed variable optimization of a Load-Bearing thermal insulation system using a filter pattern search algorithm. Optimization and Engineering, 5(2):157–177, 2004.

[2] M.A. Abramson, C. Audet, J.W. Chrissis, and J.G. Walston. Mesh Adaptive Direct Search Algorithms for Mixed Variable Optimization. Optimization Letters, 3(1):35–47, 2009.

[3] M.A. Abramson, C. Audet, and J.E. Dennis, Jr. Filter pattern search algorithms for mixed variable constrained optimization problems. Pacific Journal of Optimization, 3(3):477–500, 2007.

[4] C. Audet, V. Béchard, and S. Le Digabel. Nonsmooth optimization through Mesh Adaptive Direct Search and Variable Neighborhood Search. Journal of Global Optimization, 41(2):299–318, 2008.

[5] C. Audet, C.-K. Dang, and D. Orban. Optimization of algorithms with OPAL. Mathematical Programming Computation, 6(3):233–254, 2014.

[6] C. Audet and J.E. Dennis, Jr. Pattern search algorithms for mixed variable programming. SIAM Journal on Optimization, 11(3):573–594, 2001.

[7] C. Audet and J.E. Dennis, Jr. Mesh Adaptive Direct Search Algorithms for Constrained Optimization. SIAM Journal on Optimization, 17(1):188–217, 2006.

[8] C. Audet and W. Hare. Derivative-Free and Blackbox Optimization. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, Cham, Switzerland, 2017.

[9] C. Audet, S. Le Digabel, and C. Tribes. The Mesh Adaptive Direct Search Algorithm for Granular and Discrete Variables. SIAM Journal on Optimization, 29(2):1164–1189, 2019.

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

[11] C. Audet and C. Tribes. Mesh-based Nelder-Mead algorithm for inequality constrained optimization. Computational Optimization and Applications, 71(2):331–352, 2018.

[12] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. Technical report, arXiv, 2016.

[13] P. Balaprakash, M. Salim, T. Uram, V. Vishwanath, and S. Wild. DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pages 42–51, 2018.

[14] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In Neural networks: Tricks of the trade, pages 437–478. Springer, 2012.

[15] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In Advances in neural information processing systems, 2011.

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

[17] J. Bergstra, D. Yamins, and D.D. Cox. 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, volume 28 of ICML'13, pages I–115–I–123. JMLR.org, 2013.

[18] T. Bosc. Learning to Learn Neural Networks. Technical report, arXiv, 2016.

[19] L. Bottou. Stochastic Gradient Descent Tricks, volume 7700 of Lecture Notes in Computer Science (LNCS), pages 430–445. Springer, 2012.

[20] X. Bouthillier and C. Tsirigotis. Oríon: Asynchronous Distributed Hyperparameter Optimization. https://github.com/Epistimio/orion, 2019.

[21] A.R. Conn, K. Scheinberg, and L.N. Vicente. Introduction to Derivative-Free Optimization. MOS-SIAM Series on Optimization. SIAM, Philadelphia, 2009.

[22] A. Deshpande. 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, 2019.

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

[24] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159, 2011.

[25] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. Technical report, arXiv, 2018.

[26] T. Elsken, J. H. Metzen, and F. Hutter. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. In ICLR 2019, 2019.

[27] H. Ghanbari and K. Scheinberg. Black-Box Optimization in Machine Learning with Trust Region Based Derivative Free Algorithm. Technical report, arXiv, 2017.

[28] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1487–1495. ACM, 2017.

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

[30] R. Hooke and T.A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. Journal of the Association for Computing Machinery, 8(2):212–229, 1961.

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

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia, pages 675–678. ACM, 2014.

[33] D.P. Kingma and L.B. Jimmy. Adam: A Method for Stochastic Optimization. Technical report, arXiv, 2015.

[34] M. Kokkolaras, C. Audet, and J.E. Dennis, Jr. Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. Optimization and Engineering, 2(1):5–29, 2001.

[35] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[36] S. Le Digabel. Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm. ACM Transactions on Mathematical Software, 37(4):44:1–44:15, 2011.

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

[38] Y.A. LeCun, L. Bottou, G.B. Orr, and K.R. Müller. Efficient BackProp, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

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

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

[41] 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. A survey on deep learning in medical image analysis. Medical image analysis, 42:60–88, 2017.

[42] J. Liu, N. Ploskas, and N.V. Sahinidis. Tuning BARON using derivative-free optimization algorithms. Journal of Global Optimization, 2018.

[43] P.R. Lorenzo, J. Nalepa, M. Kawulok, L.S. Ramos, and J.R. Pastor. Particle swarm optimization for hyper-parameter selection in deep neural networks. In Proceedings of the Genetic and Evolutionary Computation Conference. ACM, 2017.

[44] I. Loshchilov and F. Hutter. CMA-ES for hyperparameter optimization of deep neural networks. Technical report, arXiv, 2016.

[45] A.R. Mello, J. de Matos, M.R. Stemmer, A. de Souza Britto Jr, and A.L. Koerich. A Novel Orthogonal Direction Mesh Adaptive Direct Search Approach for SVM Hyperparameter Tuning. Technical report, arXiv, 2019.

[46] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In NIPS-W, 2017.

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

[48] 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. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[49] M. Porcelli and Ph.L. Toint. 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):6:1–6:25, 2017.

[50] M.J.D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report DAMTP 2009/NA06, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Silver Street, Cambridge CB3 9EW, England, 2009.

[51] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. Technical report, arXiv, 2018.

[52] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. Technical report, arXiv, 2014.

[53] S.C. Smithson, G. Yang, W.J. Gross, and B.H. Meyer. Neural networks designing neural networks: multi-objective hyper-parameter optimization. In 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2016.

[54] J. Snoek, H. Larochelle, and R. Prescott Adams. Practical Bayesian optimization of machine learning algorithms. In Advances in Neural Information Processing Systems (NIPS) 25, pages 2960–2968, 2012.

[55] M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In Proceedings of the Genetic and Evolutionary Computation Conference, pages 497–504. ACM, 2017.

[56] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 2012.

[57] V. Torczon. On the convergence of pattern search algorithms. SIAM Journal on Optimization, 7(1):1–25, 1997.

[58] M. Wistuba, N. Schilling, and L. Schmidt-Thieme. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. Machine Learning, 107(1):43–78, 2018.

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

[60] 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. ACM, 2015.

[61] A. Zela, A. Klein, and S. Falknerand F. Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. Technical report, arXiv, 2018.

[62] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. Technical report, arXiv, 2016.

[63] B. Zoph, V. Vasudevan, J. Shlens, and Q.V. Le. Learning transferable architectures for scalable image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 8697–8710, 2018.