POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Optimisation des hyperparamètres des réseaux de neurones profonds

DOUNIA LAKHMIRI
Département de mathématiques et de génie industriel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques

Avril 2021

© Dounia Lakhmiri, 2021.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Optimisation des hyperparamètres des réseaux de neurones profonds

présentée par **Dounia LAKHMIRI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Luc ADJENGUE, président
Sébastien LE DIGABEL, membre et directeur de recherche
Daniel ALOISE, membre
Giacomo NANNICINI, membre externe

DÉDICACE

À mes parents,
mes piliers.

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche Sébastien Le Digabel pour son soutien tout au long de cette aventure, pour sa disponibilité en tout temps, et les précieux conseils prodigués. C'était un réel plaisir d'évoluer sous son encadrement.

Un grand merci ensuite à mes coauteurs, Chritophe Tribes pour toutes les heures passées à développer notre librairie, et Ryan Shahrouz Alimo pour m'avoir accueilli au sein de JPL pour poursuivre une partie de mes recherches. Je remercie également Pierre Girard et Edoh Logo qui ont toujours su répondre à mes questions. Des remerciements aussi pour Luc Adjengue, Daniel Aloise et Giacomo Nannici d'avoir accepté de faire partie de mon jury de thèse.

Je ne remercierai jamais assez mes parents d'avoir cru en moi et de m'avoir soutenu sans failles pendant ces longues années, Maha pour m'avoir épaulé pour aller de l'avant, et Félix pour ses encouragements, son attitude positive et sa générosité.

J'aimerai adresser mes remerciements à tous mes amis, à commencer par ceux du GERAD. À Christian et Joseph autant pour nos fous rires et pour nos longues conversations mathématiques ; à Solène, Ludovic, Alexis, Vinicius et Pierre-Yves pour les sorties, barbecues et randonnées de ces dernières années. À Mohammed, Luna et Imad entre autres pour nos précieuses balades durant cette année spéciale de confinement. À Ismail, Taha, Hamza, Luisa, Sofiane et Michela pour leurs encouragements et toutes les aventures partagées dans les quatre coins du monde.

Finalement, j'aimerai remercier tous mes professeurs de mathématiques qui m'ont transmis l'amour de cette matière et qui ont su m'inspirer à aller de l'avant, un grand merci donc à Mme Rezok, Mme Kibya, Prof. Ouadane et Prof. Altibelli.

RÉSUMÉ

Nous assistons depuis quelques années à l'essor de l'apprentissage automatique qui a été propulsé grâce à la croissance constante de la puissance de calcul des ordinateurs et la collecte d'ensembles de données massifs. Les réseaux de neurones en particulier ont tendance à devenir plus profonds, plus complexes et donc nécessitent un effort en amont pour créer des réseaux spécifiquement conçus pour une tâche et un ensemble de données particuliers. Trouver la bonne architecture et la bonne stratégie d'apprentissage est un problème équivalent à une optimisation d'hyperparamètres (HPO) où la fonction objectif est donnée sous forme de boite noire. Cette tâche d'optimisation est complexe et coûteuse mais elle n'en demeure pas moins nécessaire tant le gain en précision peut s'avérer important.

Cette thèse propose trois travaux qui visent à fournir une solution au problème énoncé en adaptant l'algorithme de recherche directe MADS. Cette approche d'optimisation sans dérivées est naturellement adaptée pour un tel objectif où chaque évaluation est coûteuse, bruitée et où les dérivées ne sont pas disponibles.

Le premier thème décortique le problème HPO pour les réseaux de convolution qui sont spécialisés dans la classification d'images. On y liste les hyperparamètres à considérer et l'espace de recherche qui en résulte, puis exploite tout particulièrement les variables de catégorie pour définir une judicieuse exploration de l'espace. L'approche proposée s'est révélée fructueuse et compétitive contre les méthodes couramment employées dans la pratique et a donné lieu à la création de la librairie HyperNOMAD.

Le second projet a permis de valider notre approche sur une application réelle où on cherche à trouver un réseau de neurones de type auto-encodeur, capable de détecter la présence de transmissions incomplètes parmi celles reçues de l'astromobile *Curiosity*. On propose la variante Δ-MADS qui combine la phase de sonde de MADS à une recherche faisant appel à un autre algorithme d'optimisation sans dérivées Δ-DOGS. Pour ce problème particulier, Δ-MADS a permis de trouver un meilleur réseau que ses deux composantes prises individuellement et que d'autres approches courantes telles que l'optimisation bayésienne ou la recherche aléatoire.

Finalement, la troisième contribution vise à intégrer des fonctions substituts spécifiques dans HyperNOMAD pour réduire son temps d'exécution et sa consommation de res-

sources. Les substituts ajoutés ont en effet accéléré la résolution du problème HPO par un facteur de 4, de diminuer les ressources de calculs employées par un facteur de 3, sans pour autant réduire la qualité des solutions proposées par HyperNOMAD.

ABSTRACT

In recent years, we have seen the rise of machine learning, which has been propelled by the constant growth in computing power and the collection of massive datasets. Neural networks, in particular, tend to become deeper, more complex, and require an upstream effort to create graphs designed explicitly for a specific task and dataset. Finding the right architecture and the right learning strategy is equivalent to a hyperparameter optimization problem (HPO), where the objective function is given in the form of a black-box. This optimization task is complex and costly, but it remains necessary as the gain in precision can be significant.

This thesis includes three contributions that aim at providing a solution to the stated problem by adapting the MADS direct search algorithm. This derivative-free optimization approach is naturally adapted for such an objective where each evaluation is expensive, noisy, and where the derivatives are not available.

The first theme dissects the HPO problem for convolutional networks, which are specialized in image classification tasks, to list the considered hyperparameters and the resulting search space. The presented method exploits categorical variables to define an informed exploration of the space. The proposed approach proved successful and competitive against the methods commonly used in practice and is made available in the form of the HyperNOMAD library.

The second project allowed us to validate our approach on a real-life application where the goal is to tune an autoencoder to detect the presence of incomplete transmissions among those received from the rover *Curiosity*. We propose the variant Δ-MADS that combines the poll phase of MADS with the DFO algorithm Δ-DOGS, plugged into the search step. For this particular problem, Δ-MADS could find a better network than its two components taken individually and than other common approaches such as Bayesian optimization or random search.

Finally, the third contribution integrates specific surrogate functions in HyperNOMAD to reduce its execution time and resource consumption. The enhanced version proved up to 4 times faster and 3 times less consuming without harming the quality of the final solutions.

TABLE DES MATIÈRES

DÉDICACE iv

REMERCIEMENTS v

RÉSUMÉ vi

ABSTRACT viii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xii

LISTE DES FIGURES xiii

LISTE DES SIGLES ET ABRÉVIATIONS xvi

LISTE DES ANNEXES xvii

CHAPITRE 1 INTRODUCTION 1

CHAPITRE 2 REVUE DE LITTÉRATURE 5
- 2.1 Hyperparamètres d'un réseau de neurones 5
  - 2.1.1 L'architecture du réseau 5
  - 2.1.2 L'entraînement du réseau 6
- 2.2 Optimiser les hyperparamètres 9
  - 2.2.1 La recherche sur grille 9
  - 2.2.2 La recherche aléatoire 10
  - 2.2.3 Les algorithmes génétiques 10
  - 2.2.4 L'apprentissage par renforcement 12
  - 2.2.5 L'optimisation bayésienne 13
  - 2.2.6 Autres approches basées sur des modèles 15
- 2.3 L'algorithme MADS 16

CHAPITRE 3 ORGANISATION DE LA THÈSE 20

CHAPITRE 4 ARTICLE I : HYPERNOMAD : HYPERPARAMETER OPTIMIZATION OF DEEP NEURAL NETWORKS USING MESH ADAPTIVE DIRECT SEARCH . . . 23
4.1 Introduction . . . 23
4.2 Literature review . . . 26
4.3 Experimental setup . . . 30
4.3.1 Hyperparameters of the framework . . . 30
4.3.2 Datasets . . . 36
4.4 HyperNOMAD . . . 38
4.4.1 Overview of NOMAD . . . 38
4.4.2 Hyperparameters in HyperNOMAD . . . 39
4.5 Computational results . . . 43
4.5.1 MNIST . . . 44
4.5.2 Fashion-MNIST . . . 45
4.5.3 CIFAR-10 . . . 47
4.6 Discussion . . . 49

CHAPITRE 5 ARTICLE II : TUNING A VARIATIONAL AUTOENCODER FOR DATA ACCOUNTABILITY PROBLEM IN THE MARS SCIENCE LABORATORY GROUND DATA SYSTEM . . . 51
5.1 Abstract . . . 51
5.2 Introduction . . . 51
5.3 Related work . . . 53
5.4 Anomaly Detection with Variational Autoencoders . . . 54
5.4.1 Overview of variational autoencoders . . . 54
5.4.2 Hyperparameters of a VAE . . . 56
5.5 The Δ- MADS method . . . 57
5.6 Numerical results for the MSL data accountability . . . 60
5.7 Conclusion . . . 64

CHAPITRE 6 ARTICLE III : USE OF STATIC SURROGATES IN HYPERPARAMETER OPTIMIZATION . . . 65
6.1 Introduction . . . 65
6.2 Related work . . . 67
6.3 The HyperNOMAD package . . . 69

6.3.1 The blackbox 69
6.3.2 The optimizer 70
6.3.3 The mesh adaptive direct search (MADS) algorithm 71
6.4 Proposed approach 72
6.4.1 Early stopping 73
6.4.2 Ranking 77
6.5 Testing 80
6.6 Discussion 80

CHAPITRE 7 DISCUSSION GÉNÉRALE 83
7.1 Synthèse des travaux 83
7.2 Limitations de la solution proposée 84

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS 86

Bibliographie 88

ANNEXES 100
A.1 Using HyperNOMAD 100

LISTE DES TABLEAUX

Table 4.1 - Selection of open source libraries for the hyperparameter optimization problem. 29
Table 4.2 - Hyperparameters that define the architecture of a neural network. 33
Table 4.3 - Choices of the optimizer and the corresponding hyperparameters. 35
Table 4.4 - Datasets embedded in HyperNOMAD. 37
Table 4.5 - Hyperparameters considered for the tests on the MNIST dataset with the simplified Caffe blackbox. 44
Table 4.6 - Results on MNIST with the simplified Caffe blackbox. 45
Table 5.1 - Hyperparameters related to the training of the VAE. 57
Table 5.2 - List of the hyperparameters of the VAE considered for the tuning problem. 58
Table 5.3 - Performance of the unsupervised learning methods : KMEAN, Gaussian mixtures, autoencoder and variational autoencoder without hyperparameter optimization on the missing data detection problem compared against the current GDS labeler. 62
Table 5.4 - Comparison between the scores of the best VAE found by each hyperparameter optimization method with the advantageous initialization (top) and the disadvantageous one (bottom). 63
Table 6.1 - Comparison between four early stopping strategies implemented in HyperNOMAD in terms of top-1 validation accuracy, overall wall clock time and total number of epochs. The variants are launched on MNIST from two starting configurations, are allowed a total of 200 configuration trials and each network can train during a maximum of 200 epochs. 75
Table 6.2 - List of the static surrogates tested for ranking the pool candidates. Their evaluation cost in terms of epochs and portion of datasets is compared to a full blackbox evaluation (BBE). 78
Table A.1 - Keywords for the HyperNOMAD parameters file. 106

LISTE DES FIGURES

Figure 1.1 - Exemple d'un réseau de neurones de type perceptron multicouches. . 2
Figure 2.1 - Exemple d'une architecture par motifs formée par une succession de blocs. 7
Figure 2.2 - Exemple de combinaisons testées par la recherche sur grille dans un espace de recherche bidimensionnel. 9
Figure 2.3 - Exemple de combinaisons testées par la recherche aléatoire dans un espace de recherche bidimensionnel. 10
Figure 2.4 - Les étapes génériques d'un algorithme évolutionnaire sont une succession de sélection des parents, croisement pour produire la population enfants, introduction d'une mutation puis une mise à jour de la population. 11
Figure 2.5 - Étapes génériques d'un apprentissage par renforcement. L'agent décide d'une action A_k et reçoit un état S_k et une récompense R_k suite à l'évaluation du candidat proposé. 12
Figure 2.6 - Exemple d'une optimisation d'une seule variable par processus gaussien avec une fonction d'acquisition. 14
Figure 2.7 - Exemple des distributions modélisées par les arbres estimateurs de Parzen. 15
Figure 2.8 - Exemple d'échecs successifs dans MADS. La taille du treillis Δ_k^m est plus agressivement rétrécie que celle de la sonde Δ_k^p. La figure est adaptée à partir de [79]. 18
Figure 4.1 - The HyperNOMAD workflow. 26
Figure 4.2 - Example of a convolutional neural network. Image taken from [41]. . 30
Figure 4.3 - Illustration of a convolution operation in (a) and a pooling operation in (b). Images taken from [96]. 31
Figure 4.4 - Examples of activation functions. 32
Figure 4.5 - Example of a convolution block (top) composed of a header (green) containing the categorical variable that describe 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). 41

Figure 4.6 - Example of a fully connected block (top) composed of a header (green) containing the categorical variable that describe 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). 42
Figure 4.7 - 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 42
Figure 4.8 - Illustration of how HyperNOMAD changes the categorical variable corresponding to the activation function.. 43
Figure 4.9 - 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. 46
Figure 4.10 - 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.. 47
Figure 4.11 - 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.. 48
Figure 4.12 - Architecture of the VGG-16 network. Image taken from [61]. 48
Figure 4.13 - 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. 49
Figure 4.14 - 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. 50
Figure 5.1 - Structure of an autoencoder on the left and of a variational autoencoder on the right. 55

Figure 5.2 - Comparison between 5 hyperparameter optimization algorithms : random search, tree Parzen estimator, HyperNOMAD, Δ-DOGS and Δ-MADS, through their convergence curves on two different initializations. 63
Figure 6.1 - The HyperNOMAD workflow is represented by the communication of its two components. The optimizer suggests new candidates based on the NOMAD software and the blackbox trains the corresponding neural network to return its validation or test accuracy as the objective function value. Image adapted from [78]. 70
Figure 6.2 - Two early stopping criteria : a scheduler that detects a plateau and reduces the learning rate until it hits the lower limits (left) and the comparison with a baseline network that stops any evaluation outside the envelope defined by the relative errors compared to the baseline scores(right). 75
Figure 6.3 - Comparison between the resource consumption of HyperNOMAD and one variant that adds an early stopping strategy, in terms of number of epochs used to train each visited candidate during a single HPO. The HPO starts from the default point p_1 on the MNIST dataset. 76
Figure 6.4 - Comparison between the original HyperNOMAD [77, 78] and four variants that implement a strategy to sort the new candidates in order to evaluate the most promising first. Strategies R_1 and R_2 train on a small epoch budget while strategies R_3 and R_4 train on a subset of the data. The optimization is launched on the MNIST dataset from the default point p_1. 79
Figure 6.5 - Comparison between the original HyperNOMAD [77, 78] and four variants that implement a strategy to sort the new candidates in order to evaluate the most promising first. Strategies R_1 and R_2 train on a small epoch budget while strategies R_3 and R_4 train on a subset of the data. The optimization is launched on the MNIST dataset from the default point p_2. 79
Figure 6.6 - Comparison between HyperNOMAD with and without surrogates on the CIFAR-10 dataset, starting from the default settings p_1. 81
Figure 6.7 - Comparison between HyperNOMAD with and without surrogates on the CIFAR-10 dataset, starting from the default settings p_3. 82
Figure A.1 - 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. 105

LISTE DES SIGLES ET ABRÉVIATIONS

BBE	Blackbox evaluation
BBO	Blackbox optimization
BO	Bayesian optimization
DFO	Derivative-free optimization
DNN	Deep neural network
EA	Evolutionary algorithm
GP	Gaussian process
GS	Grid search
HP	Hyperparameter
HPO	Hyperparameters optimization
MADS	Mesh adaptve direct search
RL	Reinforcement learning
RS	Random search
SGD	Stochastic gradient descent
TPE	Tree Parzen estimator

LISTE DES ANNEXES

Annexe A HyperNOMAD : User guide 100

CHAPITRE 1 INTRODUCTION

Durant les dernières années, plusieurs domaines de l'ingénierie et des technologies se sont transformés grâce au développement et à l'application d'outils issus de l'apprentissage automatique [67], et plus largement de l'intelligence artificielle. Ce domaine a permis d'automatiser une multitude de tâches pour assister les experts [52, 83, 86] en apportant des prédictions de plus en plus fiables, parfois capables de justifier l'automatisation de certaines opérations.

L'apprentissage automatique propose en effet un vaste éventail de méthodes capables d'extraire des informations à partir de bases de données préalablement fournies, afin de formuler des prédictions relatives à de futures observations. Parmi ces méthodes, les réseaux de neurones sont une classe particulièrement populaire de part leur capacité à apprendre de données complexes telles que des images, des bandes sons ou des textes, pour produire des prédictions qui rivalisent, ou même surpassent, celles des experts [52, 91]. De plus, la versatilité de ces réseaux leur permet de traiter des problèmes variés tels que la régression, la classification, la détection d'anomalies ou encore la génération de données.

Les réseaux de neurones sont des graphes de calcul que l'on peut lire par couche telle que l'illustre la figure 1.1. Chaque couche comporte un certain nombre de neurones qui reçoivent une information en entrée, lui appliquent une transformation non linéaire puis transmettent le résultat aux neurones de la couche suivante [1]. Le passage d'une couche à l'autre se fait par le biais d'arcs pondérés, dont les poids sont initialisés aléatoirement et mis à jour au fur et à mesure de l'entraînement du réseau. La couche de sortie permet finalement de synthétiser les informations reçues pour produire une prédiction relative aux données lues. Ainsi, les couches d'entrée et de sortie sont spécifiques aux données et au type de prédictions attendues, contrairement aux couches intermédiaires dites «cachées» qui doivent être fixées par l'utilisateur. Cette étape qui consiste à déterminer la structure du réseau est nommée «la recherche d'architecture».

L'entraînement du réseau se fait ensuite de manière itérative où l'ensemble des données d'entraînement est lu à chaque itération pour que les prédictions générées soient com-

1. Certains réseaux, tels que les réseaux récurrents, peuvent transmettre des informations aux couches précédentes. Ces connexions ne font pas partie du cadre de cette thèse et seules les transmissions vers les couches futures sont considérées par la suite.

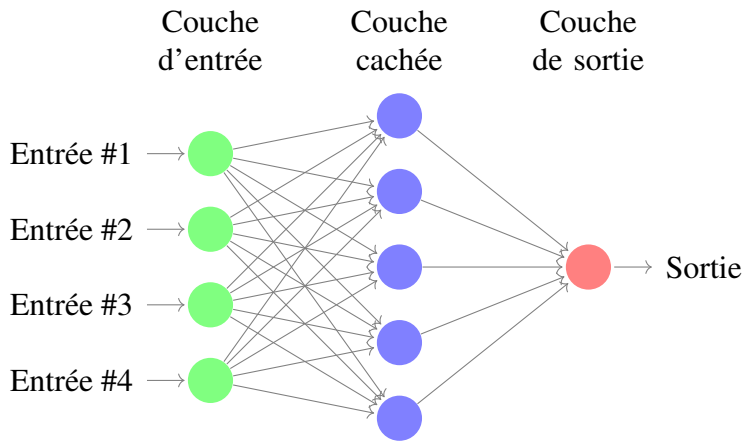


Figure 1.1 Exemple d'un réseau de neurones de type perceptron multicouches.

parées aux valeurs attendues. La différence entre ces deux valeurs permet de définir une fonction d'erreur qui doit être minimisée afin de mettre à jour les poids des arcs du réseau. Cette optimisation est réalisée par à un algorithme adapté qui doit également être choisi et réglé par l'utilisateur.

Ainsi, chaque nouveau réseau de neurones peut être créé en fixant un certain nombre de variables qui définissent les différents aspects du réseau tels que son architecture et son régime d'entraînement. Ces variables sont nommées des hyper-paramètres (HPs) puisque leurs valeurs sont déterminées avant le début de l'apprentissage du réseau et elles restent inchangées tout le long du processus, contrairement au paramètres du réseaux, ou poids, qui sont mis à jour au fur et à mesure que son entraînement s'effectue.

La qualité des prédictions du réseau étant fortement liée au choix des HPs, il est nécessaire de développer les outils appropriés afin d'optimiser ces valeurs tout en prenant compte des caractéristiques du problème d'optimisation ainsi défini. En effet, si on note ϕ le vecteur d'HPs à optimiser, Ω l'ensemble des valeurs possibles pour ϕ, et f la mesure de performance du réseau qui permet de déterminer la qualité de ses prédictions telle que le score de précision sur des données hors celles utilisées lors de l'entraînement, on exprime le problème d'optimisation sous la forme :

$$\min_{\phi \in \Omega} f(\phi, \Theta^*) \quad \text{avec } \Theta^* \in \operatorname*{argmin}_{\Theta} J_\phi(\Theta) \tag{1.1}$$

où Θ^* représente les poids optimaux du réseau, obtenus en minimisant la fonction d'er-

reur J_ϕ. Chaque nouvelle configuration ϕ est équivalente à un nouveau réseau de neurones qui nécessite un entraînement complet afin de mesurer la qualité de ses prédictions. La fonction objectif f est donc coûteuse à évaluer puisqu'un entraînement peut durer plusieurs heures et implique une importante capacité de calcul. De plus, f n'a pas de formulation analytique connue et il est donc impossible de savoir si elle est dérivable ou si des dérivées supposées sont exploitables. Il se peut aussi que certaines configurations ne puissent pas être évaluées et la stochasticité de l'initialisation des poids et celle liée à l'entraînement font que $f(\phi) \neq f(\phi)$ si on évalue le même point à deux reprises. Toutes ces caractéristiques écartent donc une approche d'optimisation usuelle qui repose fortement sur les dérivées et pointent plutôt vers les méthodes d'optimisation sans dérivées.

L'optimisation sans dérivées (DFO) [14, 38] est une branche de méthodes qui n'exploite pas les informations contenues dans les dérivées, soit parce qu'elles n'existent pas ou qu'elles sont trop coûteuses à évaluer. Dans ce cadre, les problèmes d'optimisation sont souvent modélisés sous forme de boîte noire puisqu'on ignore le processus qui associe chaque valeur d'entrée à une valeur de l'objectif, ce qui est le cas lorsque f est le résultat d'une simulation numérique.

On parle d'ailleurs également d'optimisation de boîtes noires (BBO). Ce terme englobe les approches sans dérivées telles que des méthodes heuristiques, algorithmes génétiques ou l'optimisation Bayésienne, qui ne disposent pas de preuves de convergence contrairement aux méthodes dites DFO. On distingue deux classes de méthodes de DFO : les méthodes directes et celles qui se basent sur des modèles. Ces dernières exploitent les informations collectées durant l'évaluation de certains points pour construire des modèles [60, 100] capables de guider l'optimisation. Tandis que la première classe de méthodes se base uniquement sur les valeurs des fonctions et explore l'espace de recherche en suivant des directions de recherches sur un maillage [13, 53, 117] ou sur un simplexe [94].

Cette thèse a pour objectif d'adapter des outils de recherche directe pour résoudre le problème d'optimisation des hyperparamètres (HPO) des réseaux de neurones tel que formulé au problème (1.1). La thèse, rédigée par articles, comprend trois travaux présentés dans les chapitres 4, 5, et 6 dans l'ordre chronologique de leur soumission. Le premier article présente la librairie HyperNOMAD qui adapte l'algorithme *Mesh adaptive direct search* (MADS) pour la HPO des réseaux de convolution. Le second projet, effectué dans le cadre d'un stage de recherche au *Jet Propulsion Laboratory*, a permis

le développement de la variante algorithmique Δ-MADS pour optimiser les HPs d'un réseau autoencodeur variationnel capable de détecter les anomalies parmi les données transmises par l'astromobile *Curiosity*. Finalement, le troisième projet vise à améliorer la librairie HyperNOMAD en exploitant certaines fonctions substituts pour accélérer le processus de HPO sans nuire à la qualité des solutions proposées.

Cette thèse est structurée de la manière suivante : au chapitre 2 nous présentons une revue de littérature où d'abord les diverses méthodes pour la HPO des réseaux de neurones sont présentées puis, l'algorithme MADS est détaillé. Les trois chapitres qui suivent contiennent les trois articles mentionnés ci dessus et sont suivis d'une discussion des résultats puis d'une conclusion.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre offre une synthèse des méthodes utilisées pour résoudre le problème d'optimisation considéré qui s'écrit sous la forme

$$\min_{\phi \in \Omega} f(\phi, \Theta^*) \quad \text{avec } \Theta^* \in \operatorname*{argmin}_{\Theta} J_\phi(\Theta) \tag{2.1}$$

où f est représentée sous forme de boîte noire qui prend en entrée un vecteur d'HPs ϕ appartenant au domaine Ω, entraîne le réseau de neurones équivalent pour obtenir les poids optimaux notés Θ^* en minimisant la fonction d'erreur J_ϕ, et renvoie son score de validation comme valeur de l'objectif.

Afin de se familiariser avec le problème (2.1), on s'intéresse tout d'abord au vecteur ϕ pour comprendre les deux aspects du réseau qu'il permet de définir, à savoir l'architecture et l'entraînement. Par la suite, on présente une analyse des méthodes d'optimisation communément utilisées pour résoudre le problème (2.1), suivi d'une description détaillée de l'algorithme MADS [13] sur lequel se basent les travaux présentés dans les chapitres qui suivent.

2.1 Hyperparamètres d'un réseau de neurones

Un hyperparamètre désigne une variable dont la valeur est fixée au moment de l'initialisation d'un algorithme et qui demeure inchangée tout le long de son exécution. Dans notre contexte, les hyperparamètres servent à déterminer la structure d'un réseau de neurones et son entraînement. Ces deux notions sont donc détaillées dans la suite de cette section.

2.1.1 L'architecture du réseau

L'architecture est le terme employé dans le domaine de l'apprentissage automatique pour désigner la structure du réseau : son nombre de couches, le type de chacune, le nombre de neurones, d'arcs entre chaque neurone, etc. Les réseaux de neurones profonds sont des graphes complexes qui contiennent souvent plusieurs millions d'arcs, et donc de «paramètres», connectant les différentes couches et neurones.

Un réseau se lit comme une succession de couches connectées de telle sorte que l'information lue par la couche d'entrée arrive à atteindre la couche de sortie afin de produire une prédiction de régression ou de classification. Aussi, la manière la plus intuitive de définir une architecture est de choisir un nombre de couches, puis de déterminer les caractéristiques de chacune. L'espace de recherche qui découle de cette définition permet une incomparable flexibilité puisque, théoriquement, toutes les architectures peuvent être générées de la sorte. Néanmoins, la dimension du problème à optimiser devient beaucoup trop importante lorsqu'on veut définir des architectures de plus en plus profondes comme le veut la tendance actuelle. En effet, les réseaux sont passés en quelques années de structures telle que *AlexNet* [76] qui est composée de 8 couches, à *VGG* [110] dont des variantes ont une vingtaine de couches, pour arriver aux réseaux *ResNet* [62] qui peuvent se composer de plus de 1000 couches.

Ce constat oblige donc de repenser l'espace de recherche des architectures, quitte à sacrifier quelques degrés de liberté. Les articles [129, 132] introduisent l'idée de regrouper quelques couches pour former des structures de blocs, ou cellules, afin de réduire la dimension du problème de recherche d'architecture (NAS) tout en produisant des réseaux profonds et suffisamment sophistiqués. Les blocs peuvent être reliés séquentiellement pour former une architecture par motif telle qu'illustrée par la figure 2.1, ou par des connexions plus variées telles que celles retrouvées dans les réseaux *DenseNet* [68] où chaque bloc est relié à tous ceux qui le suivent.

Les travaux présentés dans cette thèse, particulièrement ceux des chapitres 4 et 6, se placent dans un espace où les réseaux sont formés à la chaîne et où chaque couche est précisément définie. Le chapitre 5 se penche spécialement sur les réseaux auto-encodeurs dont la structure particulière est exploitée pour définir un espace de recherche d'architecture bidimensionnel.

2.1.2 L'entraînement du réseau

On se place dans le cadre de l'apprentissage supervisé où l'on dispose d'un ensemble de N données étiquetées notées $(x_i, y_i)_{1 \leq i \leq N}$. Le vecteur x_i représente une donnée d'entrée fournie au réseau et y_i représente la valeur que celui-ci devrait prédire. Notons que dans la pratique, les données sont subdivisées en données d'entraînement, de validation et parfois de tests. Les données d'entraînement forment la plus grande portion des trois et servent à guider le réseau lorsqu'il apprend à ajuster ses prédictions, tandis que les

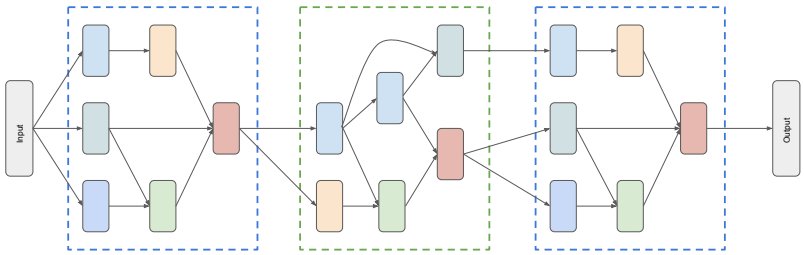


Figure 2.1 Exemple d'une architecture par motifs formée par une succession de blocs.

données de validation sont utilisées pour estimer la qualité de généralisation du réseau. Le score de validation est en effet plus fiable puisque les réseaux peuvent sur-apprendre des données d'entraînement ce qui nuit à leur précision lorsqu'ils sont faces à de nouvelles données. Dans le cadre des projets qui suivent, la boîte noire renvoie les scores sur les données de validation. Les données de test quant à elles peuvent être utilisées à la fin de l'optimisation des HPs pour évaluer la solution finale retenue.

De manière informelle, l'entraînement désigne le processus itératif durant lequel les poids du réseau Θ sont ajustés pour que les prédictions se rapprochent le plus des valeurs des étiquettes des données d'entraînement. À chaque itération, ces dernières sont lues, les prédictions équivalentes sont produites puis comparées aux valeurs des étiquettes. La différence entre les prédictions et les valeurs attendues permet de définir une fonction de perte $J(\Theta)$, à laquelle on ajoute un terme de régularisation pour éviter un sur-apprentissage. L'entraînement revient donc à minimiser la fonction de perte

$$J(\Theta) = Err(h(x), y) + \lambda||\Theta||^2 \tag{2.2}$$

où $h(x)$ désigne les valeurs prédites par le réseaux en lisant les données x, y représente les étiquettes attachées aux données x. La fonction Err est une fonction d'erreur qui dépend de la nature de la tâche effectuée. Dans le contexte de classification aux chapitres 4

et 6, on emploie l'entropie croisée

$$J(\Theta) = -\frac{1}{N}\sum_{n=1}^{N} y_i log(h(x_i)), \tag{2.3}$$

tandis que dans le chapitre 5 où l'entraînement vise une tâche de régression, on emploie l'erreur quadratique moyenne

$$J(\Theta) = \mathbb{E}[h(x) - y]. \tag{2.4}$$

Ces fonctions d'erreurs sont fournies par la libraire Pytorch. Le terme λ est un facteur de dégradation des poids, considéré comme un HP à optimiser, qui sert à freiner le sur-apprentissage. Notons que la pénalité $\lambda||\Theta||^2$ est choisie puisqu'elle fait partie intégrante des fonctions d'erreurs de Pytorch. Il est donc préférable de garder ce terme sans y ajouter une pénalité supplémentaire qui risquerait de ralentir le processus d'entraînement.

Le problème d'optimisation énoncé en (2.2) est de très grande dimension puisque les réseaux contemporains possèdent des millions, voir des milliards de paramètres. Aussi, lorsqu'on cherche à déterminer les HPs d'entraînement, cela revient à sélectionner, et à régler, un algorithme capable de résoudre un problème de si grande taille.

Par exemple, l'algorithme de descente stochastique (SGD) est une des techniques couramment utilisées pour minimiser la fonction de perte. À chaque itération, SGD propose une direction de «descente» [1] qui est suivie le long d'un pas α, dit *taux d'apprentissage*, dont la taille est également un HP à considérer. Plusieurs astuces ont été développées pour aider SGD à minimiser la fonction $J(\Theta)$ telles que l'ajout des techniques de *moments* qui servent à diminuer les oscillations des directions produites par SGD, ou des stratégies de mise à jour du taux d'apprentissage [23, 31, 82].

Par la suite, les HPs d'apprentissage désignent le choix de l'algorithme d'optimisation suivi d'un ensemble d'HPs qui lui sont propres. L'espace de recherche de ces HPs sera spécifié dans chacun des chapitres 4, 5, 6.

1. Toutes les directions générées par SGD n'impliquent pas une diminution de la fonction de perte.

2.2 Optimiser les hyperparamètres

Dans la littérature, le problème (2.1) est souvent traité en deux étapes séparées. La première consiste à chercher l'architecture du réseau avec un protocole d'entraînement fixe, puis l'entraînement est modifié par une optimisation des HPs équivalents. Cette section présente une liste non exhaustive des méthodes employées pour optimiser les HPs des DNNs, que ce soit ceux reliés à la recherche d'architecture ou au régime d'entraînement.

2.2.1 La recherche sur grille

La recherche sur grille (GS) est une des premières heuristiques employées pour résoudre le problème (2.1). Elle consiste simplement à discrétiser l'hypercube défini par les contraintes de boîtes sur chaque HP, puis d'évaluer successivement toutes les combinaisons ainsi créées comme l'illustre la figure 2.2.

La GS a l'avantage d'être facile à implémenter et à paralléliser, elle est encore disponible dans plusieurs librairies dédiées à la HPO des DNNs telles que NNI [93], Orìon [32], et elle est couramment utilisée dans les travaux contemporains tels que [115]. Néanmoins, GS devient trop coûteuses lorsque le nombre d'HPs à optimiser est important, ce qui est souvent le cas dans le cadre de l'apprentissage profond, et nécessite des ressources calculatoires et énergétiques considérables. De plus, cette méthode n'est pas adaptive et ne permet pas de détecter la sensibilité de l'objectif f par rapport à des HPs particuliers.

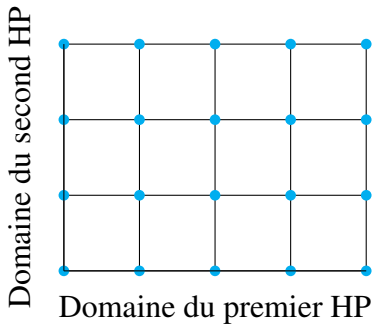


Figure 2.2 Exemple de combinaisons testées par la recherche sur grille dans un espace de recherche bidimensionnel.

2.2.2 La recherche aléatoire

L'heuristique de recherche aléatoire (RS) [25] permet de palier certains inconvénients de la GS. Un échantillonnage aléatoire est capable d'évaluer des combinaisons plus variées que celle déterminées par la grille de GS, ce qui a pour effet de mettre en valeur une potentielle sensibilité de l'objectif par rapport à certains HPs avec bien moins d'évaluations que la GS. Une stratégie courante de la RS commence l'optimisation par un échantillonage grossier qui s'affine au fur et à mesure que des paramètres importants sont mis en évidence. La figure 2.3 illustre un tel comportement.

La RS est également disponible dans plusieurs librairies dédiées au problème de la HPO des DNNs, telles que NNI [93] ou HyperOPT [26].

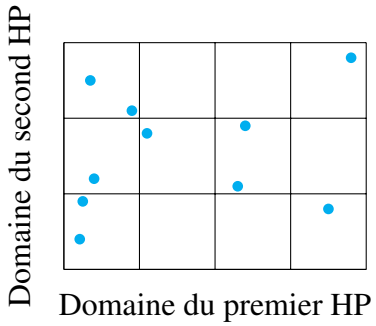


Figure 2.3 Exemple de combinaisons testées par la recherche aléatoire dans un espace de recherche bidimensionnel.

Les deux méthodes exposées jusqu'à présent peuvent être vues comme un point de départ dans la conception d'outils pour optimiser les HPs des réseaux de neurones profonds. Ce sont deux algorithmes peu sophistiqués, surtout à cause de leur manque d'adaptivité qui empêche la recherche d'exploiter les scores des points évalués pour guider l'optimisation, mais elles demeurent pertinentes et populaires dans le domaine de la HPO des DNNs.

2.2.3 Les algorithmes génétiques

Les algorithmes génétiques, ou évolutionnaires (EA), font également partie de la classe des heuristiques de boites noires. Ce sont des techniques itératives qui engendrent et entretiennent *une population*, c'est à dire un ensemble de configurations d'HPs, de telles sorte à améliorer la qualité de la population au fur et à mesure. De manière générale, une

itération EA passe par quatre étapes : la sélection, le croisement, la mutation puis une mise à jour. Ce processus est illustré à la figure 2.4. La sélection détermine les meilleurs éléments de la population, dans ce contexte il s'agit des configurations de réseaux de neurones ayant les meilleures scores de validation. Ce sous ensemble forme une population de «parents» qui seront croisés pour produire une population d'«enfants». Un croisement sert à combiner certains éléments de chaque parent pour produire un enfant. La mutation est un processus souvent aléatoire qui introduit une perturbation dans les structures engendrées par les croisements. Cela a pour but d'éviter une convergence prématurée et d'introduire de nouvelles variantes qui n'étaient pas présentes dans la population initiale. La nouvelle génération d'enfants est ensuite évaluée avant d'enclencher la mise à jour qui va éliminer les moins bonnes configurations de la population pour introduire les meilleurs enfants à leurs places.

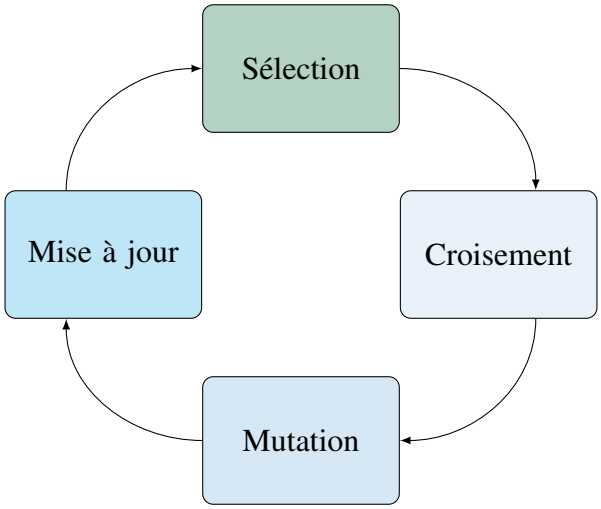


Figure 2.4 Les étapes génériques d'un algorithme évolutionnaire sont une succession de sélection des parents, croisement pour produire la population enfants, introduction d'une mutation puis une mise à jour de la population.

Les EAs sont autant utilisés pour la recherche d'architecture que pour l'optimisation des HPs d'entraînement [29, 50, 54, 87, 105] de part la qualité des solutions trouvées par cette approche. Néanmoins, le coût de manipuler une population de réseaux de neurones profonds est important et nécessite une grande capacité de calcul.

2.2.4 L'apprentissage par renforcement

Dans le cadre de l'apprentissage par renforcement (RL), la génération de nouveau candidat se fait par le biais d'un réseau de neurones, souvent de type récurrent (RNN) ou LSTM [64]. Avec la terminologie liée au RL, on parle d'*environnement* pour désigner l'espace de recherche et d'*agent* pour le réseau RNN qui décide de produire les configurations à évaluer. À chaque itération k, l'agent entreprend l'*action* A_k pour évaluer une nouvelle configuration à partir de l'environnement. Le score obtenu par le nouveau candidat sert à définir l'*état* S_k et la *récompense* R_k dont se sert l'agent pour adapter sa prochaine action A_{k+1}. La figure 2.5 montre le fonctionnement générique d'une méthode RL.

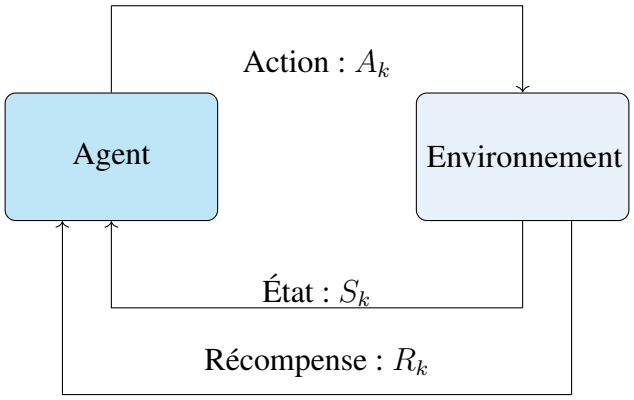


Figure 2.5 Étapes génériques d'un apprentissage par renforcement. L'agent décide d'une action A_k et reçoit un état S_k et une récompense R_k suite à l'évaluation du candidat proposé.

Les premières adaptations de l'approche RL proposée par [19, 130] ont permis d'améliorer l'état de l'art et de proposer des architectures complexes et performantes, mais à un coût très élevé en terme de ressources de calcul. La solution proposée par [130] a été obtenue après 4 semaines de calcul sur 800 GPUs, et [19] a nécessité 10 jours sur 10 GPUs. Suite à cela, des méthodes moins coûteuses ont été développées avec des agents et environnements plus adaptés afin de produire de meilleures solutions à moindre coût. Notamment, la méthode ENAS [98] exploite le partage de paramètres pour produire des solutions en quelques heures sur un seul GPU. Notons que cette approche est également une heuristique qui ne donne donc aucune garantie de convergence.

2.2.5 L'optimisation bayésienne

L'optimisation bayésienne (BO) regroupe des méthodes d'optimisation de boîtes noires qui construisent des modèles, ou substituts, afin d'identifier les zones les plus intéressantes où les meilleurs candidats peuvent se situer. Les modèles utilisés sont dynamiques, c'est à dire qu'ils sont mis à jour au fur et à mesure de la collecte de nouvelles informations, et permettent donc de guider l'optimisation. Dans le cadre de la BO, les observations sont supposées provenir de variables aléatoires, disposant donc d'une distribution, que les méthodes de BO cherchent à modéliser. Cette section présente deux méthodes bayésiennes, à commencer par les processus gaussiens (GP) suivis par les arbres d'estimation de Parzen (TPE).

Processus gaussiens

Les GPs [103] sont une généralisation des distributions gaussiennes mutlivariées que l'on définit par une fonction de moyenne μ et une fonction de covariance σ. Après chaque évaluation, le GP met à jour les estimations de la moyenne et de la covariance des observations pour produire un modèle plus fidèle et donc plus fiable pour identifier judicieusement les prochains candidats à évaluer. Les modèles construits par les GPs sont complexes à optimiser directement dans la pratique, il est souvent préférable de se baser sur des fonctions d'acquisition telle que la probabilité d'amélioration (PI) ou l'amélioration espérée (EI) [34]. La figure 2.6 illustre ce mécanisme qui combine à la fois les estimations du GP avec la fonction d'acquisition EI. À chaque étape, la fonction EI est optimisée pour déterminer le candidat qui a le plus de potentiel à améliorer l'objectif. Une fois ce candidat évalué, le GP et la fonction EI sont mis à jour et le processus recommence.

Les GP sont souvent utilisés pour la HPO puisqu'ils permettent de trouver des configurations supérieures à celles proposées par des experts dans plusieurs cas [112, 122]. Néanmoins, leur efficacité diminue lorsque la dimension du problème à optimiser devient grande, ils ne sont pas adaptés à tout type de variables et leur performance est aussi fortement liée à des réglages préalables tels que le choix de la fonction noyau à employer. Certains travaux ont établi que sous certaines conditions sur l'objectif et sur le noyau utilisé, les GPs associées aux EI qui garantissent la génération d'un ensemble dense de candidats à évaluer sous certaines conditions [120].

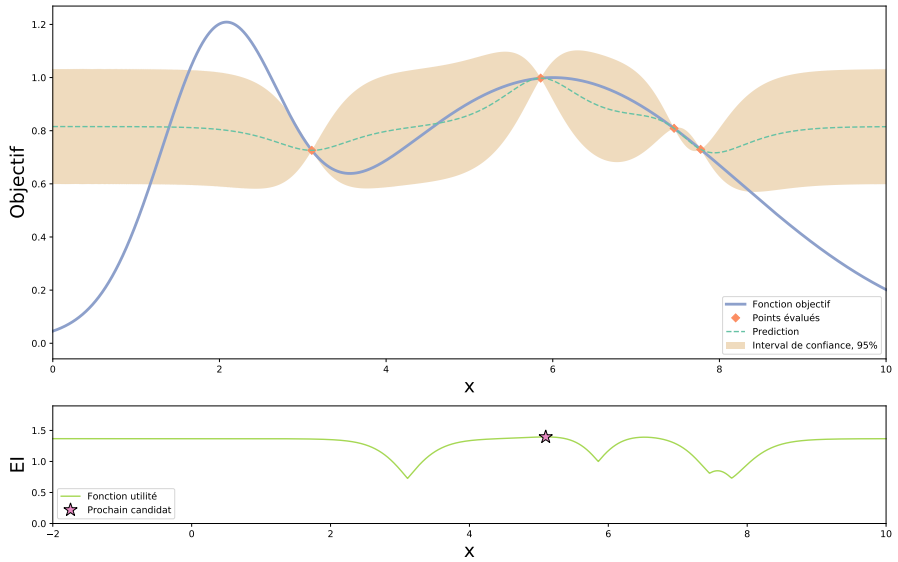


Figure 2.6 Exemple d'une optimisation d'une seule variable par processus gaussien avec une fonction d'acquisition.

Les arbres estimateurs de Parzen

Les arbres estimateurs de Parzen (TPE) [24] adoptent une démarche miroir à celle des GP. Ici, les évaluations faites au cours de l'optimisation sont utilisées pour définir deux classes de points qui sont supposés provenir de deux distributions séparées. La première classe contient une proportion (25%) des meilleurs candidats observés jusqu'à présent, et la deuxième englobe toutes les autres configurations. L'algorithme TPE cherche à modéliser les distributions de ces deux classes pour choisir les prochains points à évaluer à partir de la distribution du meilleur groupe. Ce mécanisme est illustré par la figure 2.7.

La méthode TPE est une heuristique incluse dans plusieurs librairies d'optimisation des HPs des réseaux de neurones [26, 32, 93, 125] car elle est mieux adaptée aux problèmes à plus grande dimension que les GPs, et aux variables mixtes. Néanmoins, les tests comparatifs effectués lors de cette thèse, principalement dans le chapitre 4, ont mis en évidence des limitations de TPE d'une part vis à vis de la dimension du problème de HPO, et d'autre part son étroite dépendance à la qualité du premier échantillon de candidats que TPE génère aléatoirement.

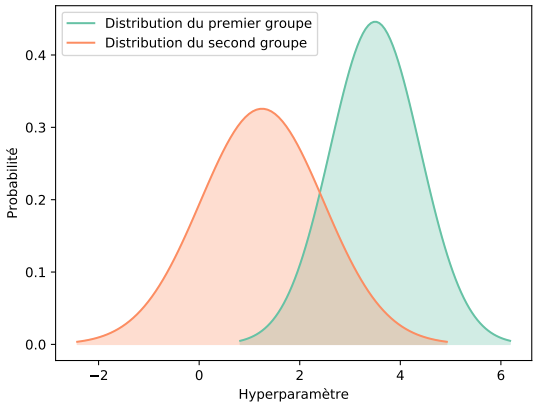


Figure 2.7 Exemple des distributions modélisées par les arbres estimateurs de Parzen.

2.2.6 Autres approches basées sur des modèles

L'idée de construire des modèles dynamiques comme substituts à une coûteuse boîte noire n'est pas propre à l'optimisation bayésienne. Les méthodes basées sur les modèles forment une des grandes classes de l'optimisation sans dérivées, et sont aussi souvent exploitées dans le cadre de la problématique qui nous intéresse.

Dans l'article [43], les modèles à base radiale (RBF) sont exploités pour définir le meilleur réseau de neurones pour résoudre un problème de classification d'images, puis celui de la prédiction d'interactions entre les différentes composantes de médicaments. L'article [57] exploite également des modèles quadratiques pour définir les régions de confiances de l'algorithme DFO afin d'optimiser les HPs de modèles de type *support vector machine* (SVM).

Il existe également une famille de méthodes DFO basées sur les modèles d'interpolation créés à partir de la triangulation de Delaunay. L'algorithme Δ-DOGS [27, 28] en fait partie. Lors de chaque itération k, la triangulation permet de définir un ensemble de simplexes $\{\Delta_k^i\}_i$ possédant chacun un cercle circonscrit de centre z_k^i et de rayon r_k^i. Sur chaque simplexe, Δ-DOGS définit la fonction d'incertitude locale

$$e_k^i(x) = (r_k^i)^2 - ||x - z_k^i||^2$$

ainsi que la fonction de recherche locale

$$s_k^i(x) = p^k(x) - Ke_k^i(x).$$

La fonction p^k désigne le modèle d'interpolation global à l'étape k, tandis que la constante K fixe un compromis entre une exploration globale et une intensification locale. Chaque fonction de recherche locale est minimisée, les minimums locaux sont ensuite comparés pour n'évaluer que le plus prometteur parmi eux. S'ensuit une phase de mise à jour du modèle d'interpolation et de la triangulation. Δ-DOGS est au cœur du chapitre 5 et les détails de son exécution y seront exposés.

Les méthodes exposées jusqu'à présent sont toutes des techniques d'optimisation de boîtes noires aux stratégies variées. Certaines se basent uniquement sur les scores des observations comme la GS, la RS ou EA, tandis que d'autres construisent et adaptent des modèles à mesure que l'optimisation avance, telle que la BO ou RL.

Les travaux de cette thèse reposent sur l'algorithme MADS qui applique une approche directe tout en laissant la possibilité d'y ajouter une recherche basée sur des modèles. MADS peut donc bénéficier des avantages des deux classes de méthodes et offre en plus des garanties de convergence en fonction des propriétés du problème d'optimisation étudié.

2.3 L'algorithme MADS

L'algorithme *Mesh Adaptative Direct Search* (MADS) [13] est une méthode DFO de recherche directe. Cette classe d'algorithmes explore l'espace de recherche à travers des motifs tels des simplexes [94] ou des treillis en se basant uniquement sur les scores des évaluations de la fonction objectif pour aiguiller l'exploration de l'espace.

De manière générale, MADS permet de résoudre itérativement des problèmes formulés sous la forme

$$\min_x \{f(x) : x \in \Omega\} \tag{2.5}$$

en parcourant le long d'un treillis M_k défini à chaque itération k par

$$M_k = \{x + \Delta_k^m Dz, z \in \mathbb{Z}^{n_D}, x \in C\}.$$

où x est un point appartenant à l'ensemble des points visités C, $D \in \mathbb{R}^{n \times n_D}$ est une matrice dont les n_D colonnes forment un ensemble générateur positif, et $\Delta_k^m \geq 0$ désigne la taille du treillis.

Les itérations de MADS se font en deux étapes. La phase de *search* est facultative, n'est pas rigoureusement définie et offre une certaine flexibilité pour y introduire une stratégie d'exploration, basée sur des modèles par exemple, à condition de générer un nombre fini de points projetés sur le treillis pour préserver les propriétés de convergence de MADS. Ces dernières se basent uniquement sur la seconde étape, *la sonde*.

Notons x_k la meilleure configuration visitée jusqu'à l'itération k. L'étape de sonde génère un ensemble de nouveaux candidats autour de x_k, le long de directions de recherche qui forment une base positive, et qui se trouvent à une distance de sonde Δ_k^p qui vérifie $\Delta_k^m \leq \Delta_k^p$. Cet ensemble de points de sonde s'exprime sous la forme

$$P_k = \{x_k + \Delta_k^m d \mid d \in D_k\} \text{ avec } ||\Delta_k^m d|| \approx \Delta_k^p$$

où les vecteurs d de l'ensemble D_k forment une base positive. Par défaut, MADS emploie une stratégie d'évaluation *opportuniste*, ce qui veut dire que l'étape de sonde est interrompue dès qu'une amélioration par rapport à x_k est trouvée, et ce même si tous les points de P_k n'ont pas encore été évalués.

Si un meilleur point que x_k est trouvé, on considère l'itération réussie et on réitère, sinon on dit que l'itération est un échec et Δ_k^m et Δ_k^p doivent être affinés sachant que Δ_k^m est réduit plus rapidement que Δ_k^p. Cette mise à jour est illustrée dans la figure 2.8.

L'algorithme MADS fait partie des méthodes DFO, aussi elle dispose d'une hiérarchie de garanties de convergences en fonction des propriétés de la fonction objectif considérée. Dans un cas général, on peut garantir les propriétés suivantes

Theorem 2.3.1 *Soient $\{\Delta_k^m\}$ et $\{\Delta_k^p\}$ les séquences de tailles du treillis et de sonde générées par l'exécution de MADS pour optimiser une fonction $f : \mathbb{R}^n \to \mathbb{R}$. Les séquences vérifient que*

$$\liminf_{k\infty} \Delta_k^m = \liminf_{k\infty} \Delta_k^p = 0$$

Theorem 2.3.2 *Les directions de sondes générées par MADS une fois normalisées forment asymptotiquement un ensemble dense dans la sphère unité.*

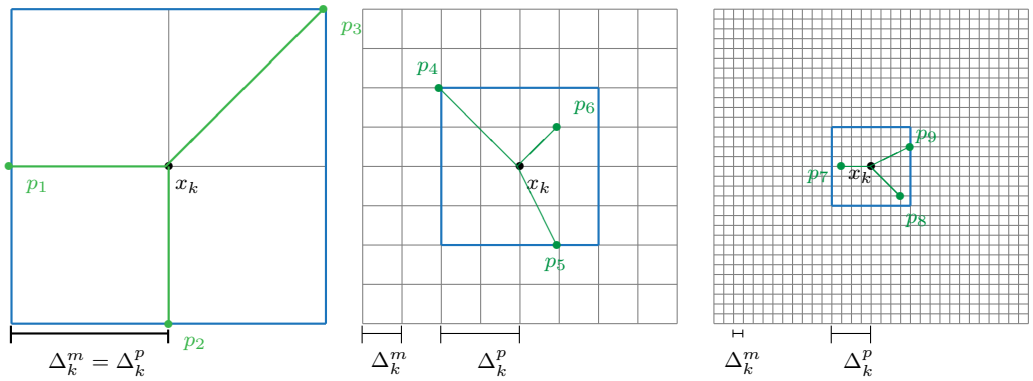


Figure 2.8 Exemple d'échecs successifs dans MADS. La taille du treillis Δ_k^m est plus agressivement rétrécie que celle de la sonde Δ_k^p. La figure est adaptée à partir de [79].

Theorem 2.3.3 *Soit f une fonction Lipschitz autour d'un point raffiné $\hat{x} \in \Omega$ et d une direction raffinante appartenant au cône hyper-tangent $T_\Omega^H(\hat{x})$, alors*

$$f^\circ(\hat{x}, d) \geq 0,$$

où f° est la dérivée directionnelle généralisée.

L'algorithme 1 donne une description de haut niveau des principales étapes de l'exécution de l'algorithme MADS avec une mise en valeur des possibilités d'intégration de fonctions substituts. Les substituts sont dit «dynamiques» lorsqu'ils se mettent à jour au cours de l'exécution pour s'adapter aux informations collectées au fur et à mesure des évaluations, contrairement aux modèles «statiques» dont l'expression ne change pas durant l'optimisation.

Certains modèles, souvent dynamiques, peuvent servir de stratégies exploratoires globales de l'espace de recherche dans l'étape de recherche [10,18] pour permettre à MADS d'atteindre des zones éloignées du centre de la sonde x_k. D'autres substituts, plus souvent statiques, sont employés pour ordonner une liste de candidats à évaluer afin de placer les plus prometteurs en premier. Ce type d'ordonnancement est surtout utile lorsqu'il est combiné à la stratégie d'évaluation opportuniste mentionnée plus haut et peut accélérer le processus d'optimisation. Toujours dans cet esprit d'accélération, on peut introduire des substituts dont le but est de détecter qu'une évaluation en cours ne promet

pas de résultat intéressant pour déclencher un arrêt prématuré de l'évaluation. Ces idées sont développées dans le chapitre 6.

Algorithm 1: MADS

[1] **Initialisation**

$k \leftarrow 0$

Initialiser $\Delta_0^p \geq \Delta_0^m > 0$

Décider des fonctions substituts à utiliser (optionnel)

Définir un point de départ x_0

[2] **Recherche (optionnelle)**

Utiliser une stratégie pour trouver un ensemble de points

$\mathbf{S} = \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_l}\}$.

Projeter les points de S sur M_k pour obtenir $X = \{x_1, x_2, \ldots, x_l\}$

Évaluer les points de X

Si c'est un *succès*, aller à l'étape **[4]**

[3] **Sonde**

Définir l'ensemble P_k

Ordonner les points de P_k ***selon les prédictions d'un substitut***

Évaluer les points de P_k tant qu'aucune amélioration n'est trouvée.

Si une amélioration est trouvée, c'est un *succès*, sinon c'est un *échec*

Aller à l'étape **[4]**

[4] **Mise à jour**

Mettre à jour $\Delta_k^p, \Delta_k^m, x_k, M_k, C$ selon si l'itération est un *succès* ou un *échec*

Si aucune condition d'arrêt n'est satisfaite, aller à l'étape **[2]**

CHAPITRE 3 ORGANISATION DE LA THÈSE

L'objectif principal de cette recherche est de proposer une adaptation de l'algorithme MADS pour le problème de l'optimisation des hyperparamètres des réseaux de neurones profonds. Les caractéristiques de ce problème ainsi que celles de l'algorithme MADS présentées au chapitre précédent, portent à croire qu'une telle approche pourrait se faire une place parmi l'éventail de méthodes couramment utilisées dans la pratique.

Cette thèse est rédigée par articles. Les trois travaux qui la constituent sont présentés dans l'ordre chronologique de leurs soumissions dans les chapitres 4, 5 et 6.

Le chapitre 4 contient le premier article qui a été accepté pour publication dans la revue *ACM Transactions on Mathematical Software*. On y présente la librairie à code source ouvert HyperNOMAD qui permet de définir des architectures et des régimes d'entraînement adaptés à des bases de données spécifiques formées d'images. L'article commence par établir un espace de recherche approprié avant de spécifier la stratégie de son exploration. Cet espace de recherche est défini à partir de la modélisation proposée du problème HPO qui permet de générer des architectures séquentielles où chaque couche est précisément définie. Ainsi, le modèle est intuitif puisqu'il découle naturellement de la lecture du réseau et octroie un grand degré de liberté au vue la sensibilité de la performance à chacun des HPs retenus. Il existe néanmoins d'autres modélisations possibles pour l'architecture telle que celle basée sur une combinaison de blocs pré-établis discutée dans la section 2.1.1.

HyperNOMAD explore cet espace en exploitant la présence de variables de catégories parmi les hyperparamètres pour définir une structure de voisinage afin d'avancer judicieusement durant le processus d'optimisation. Une analyse empirique a permis de choisir le voisinage proposé où le nombre de voisins retenus réalise un bon compromis entre l'ajout d'évaluations de la boîte noire et l'efficacité de l'exploration de l'espace. Le choix desdits voisins s'est fait de sorte à minimiser le risque de générer des architectures non réalisables.

L'article conclut par une série de tests comparatifs entre l'approche proposée, la méthode TPE et la recherche aléatoire sur les bases de données MNIST et CIFAR-10. Le choix de ces deux méthodes se justifie d'abord par la présence de variables entières et de catégories, ce qui élimine certains algorithmes basés sur les modèles. De plus, on écarte

les méthodes qui nécessitent un grand nombre d'évaluations de la boîte noire avant de commencer à produire des solutions de qualité. Il s'agit notamment des algorithmes génétiques ou de l'apprentissage par renforcement. Finalement, on se compare également à la recherche aléatoire que toute nouvelle méthode se doit de surpasser.

L'article présenté au chapitre 5 est le résultat d'un travail effectué lors d'un stage MITACS au laboratoire *NASA-Jet propulsion laboratory*. L'article est soumis au journal *Expert Systems with Applications*. On s'y intéresse à une problématique concrète où le but est de trouver un outil de classification non supervisée capable de précisément détecter la présence d'anomalies, sous forme de transmissions incomplètes, parmi un ensemble de données reçues de l'astromobile *Curiosity* depuis la planète Mars. La détection de transmissions incomplètes doit être à la fois précise et rapide pour demander une retransmission si possible. Cette recherche commence donc par explorer les différents outils de classification non supervisée afin de choisir la plus appropriée pour la base de donnée fournie. En effet, la séparation entre les données complètes et incomplètes n'est pas une tâche évidente puisqu'elles sont générées par le même astromobile, sont relayées par les mêmes satellites et autres systèmes, et sont donc structurellement similaires. Suite à de nombreux tests, les réseaux auto-encodeurs variationnels ont été retenus comme étant les seuls à obtenir des résultats suffisamment prometteurs.

Ainsi, notre problème initial est de nouveau équivalent à une optimisation d'HPs d'un réseau de neurones particulier que l'on propose de résoudre en adaptant l'algorithme MADS. De plus, il est important d'agrémenter cette approche par une méthode de recherche globale qui aide à explorer des zones plus éloignées de l'espace de recherche, ce qui a pour effet de détecter des solutions de qualité plus rapidement et d'échapper au minimas locaux. Cet article développe donc la variante Δ-MADS qui combine l'étape de sonde de MADS à l'étape de recherche globale de l'algorithme Δ-DOGS [28]. Les tests numériques démontrent que Δ-MADS est plus rapide et plus fiable que ses deux composantes prises individuellement, et surtout cet algorithme apporte une nette amélioration à cette détection d'anomalies et est maintenant incorporé dans le système de traitement des données reçues par *Curiosity* et *Perseverance*.

Le chapitre 6 propose une amélioration à la librairie HyperNOMAD dans le but d'accélérer le processus d'optimisation et de réduire la quantité de ressources nécessaires pour cette tâche. L'article en question, qui a été soumis à la revue *SN Operations Research Forum*, intègre deux modèles substituts statiques à HyperNOMAD. Le premier substitut sert à enclencher des arrêts prématurés soit lorsque les scores de la configuration

en cours d'entraînement stagnent ou lorsqu'ils sont jugés trop loin de la meilleure solution visitée lors de l'exécution de HyperNOMAD. Plusieurs tests menés sur la base de données MNIST ont permis de définir une marge d'erreur par rapport à la meilleure configuration afin de minimiser les ressources utilisées sans pour autant arrêter prématurément des configurations ayant le potentiel de réduire cet écart moyennant plus de temps d'entraînement.

Le second modèle permet d'ordonner les candidats de l'étape de sonde de MADS dans l'ordre décroissant de leurs potentielles précisions afin de mieux exploiter la stratégie d'évaluation opportuniste. Le substitut de classement se doit d'être peu coûteux en évaluations mais suffisamment fiable pour ré-ordonner l'évaluation des nouveaux les candidats. Ici encore, les tests sont d'abord menés sur MNIST pour tester l'effet de quelques substituts de classement sur la convergence de HyperNOMAD ainsi que sur le temps d'exécution du processus de HPO.

Les tests numériques démontrent l'intérêt de tels ajouts puisque la version avec substituts arrive à obtenir des solutions comparables à celle trouvées par HyperNOMAD avec une économie moyenne de 30% en ressources de calculs en un temps parfois 4 fois plus rapide.

CHAPITRE 4 ARTICLE I : HYPERNOMAD : HYPERPARAMETER OPTIMIZATION OF DEEP NEURAL NETWORKS USING MESH ADAPTIVE DIRECT SEARCH

Statut : accepté
Journal : ACM - *Tansactions on Mathematical Software*
Coauteurs : Sébastien Le Digabel et Christophe Tribes.

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 [13] to simultaneously tune the hyperparameters responsible for both the architecture and the learning process of a deep neural network (DNN). his 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.

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

4.1 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 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 [83], medical diagnostics [86], 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) [14, 38] 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 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 :

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

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 [38, Chapter 10] or interpolation models [100]. This differentiates them from direct search methods [65] 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 [117] 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 [11, 17], 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 4.1 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.

The manuscript is structured as follows. Section 4.2 presents and discusses some of the main approaches used to solve the HPO problem of neural networks. In Section 4.3, the experimental setup is explicitly defined, and the instances used to test the proposed approach are presented. Section 4.4 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 4.5. Finally, Appendix A.1 describes the basic usage of HyperNOMAD.

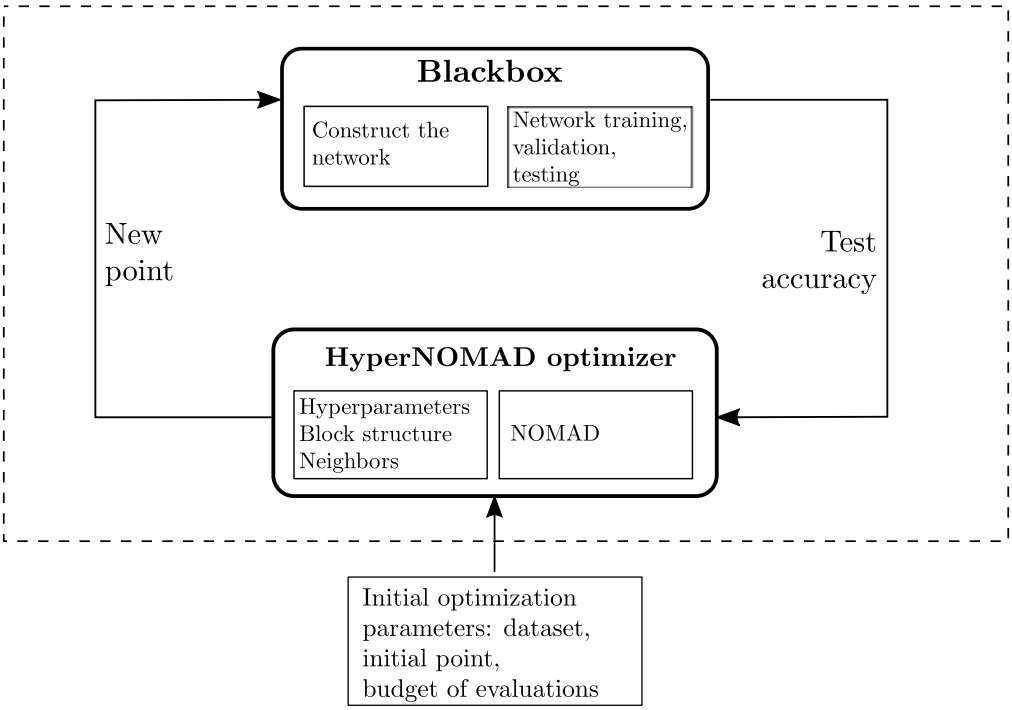


Figure 4.1 The HyperNOMAD workflow.

4.2 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) [49] that allowed to achieve state of the art performance [104, 130] 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. [128] 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 [97, 112]. 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 [25]. 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 [85] 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 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 [50, 113, 126]. In [89], 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 [90] 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 [19, 130, 131] where the main difference between each method relies on how the agents are defined and dealt with. In [111], 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 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, [30] 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.

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 unexploitable derivatives. In [17], the authors propose a general way of modeling hyperparameter optimization problems as a blackbox optimization problem. This formulation is used in [88] 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 [99], in addition to that of [17], to optimize the hyperparameters of the BFO algorithm [99]. BBO methods are also at the heart of Google Vizier [58] 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 information 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 [112, 122]. However, the disadvantage of GPs is that they do not fit well to 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 [24], 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 [42], the authors applied radial basis functions to model the blackbox as previously defined. This article presents the results obtained on the MNIST dataset [81], then on a problem of interactions between drugs. These tests show that this model provides comparable or better results than popular configurations. In [56], 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.

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 [13], which is implemented in the NOMAD package [79], especially since it can handle integer and categorical variables [2, 16]. Using the MADS algorithm for hyperparameter tuning has been validated in [92] where a SVM model is calibrated using MADS combined with the Nelder-Mead and VNS search strategies [10, 18].

A non exhaustive list of open source libraries for HPO is given in Table 4.1 along with the optimization algorithms implemented in each library and the types of variables handled.

Table 4.1 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 [97]	✓	✓	-	-	-	✓	✓	✓
hyperopt [26]	-	✓	✓	-	-	✓	✓	✓
Spearmint [112]	✓	✓	✓	-	-	✓	✓	✓
SMAC [66]	-	-	-	✓	-	✓	✓	✓
MOE [125]	-	-	✓	-	-	✓	-	-
RBFOpt [42]	-	-	-	✓	-	✓	✓	-
DeepHyper [22]	-	✓	-	✓	-	✓	✓	✓
Oríon [32]	-	✓	✓	-	-	✓	✓	✓
Google Vizier [58]	✓	✓	✓	✓	-	✓	✓	✓
HyperNOMAD	-	-	-	-	✓	✓	✓	✓

4.3 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 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.

4.3.1 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.

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 4.2.

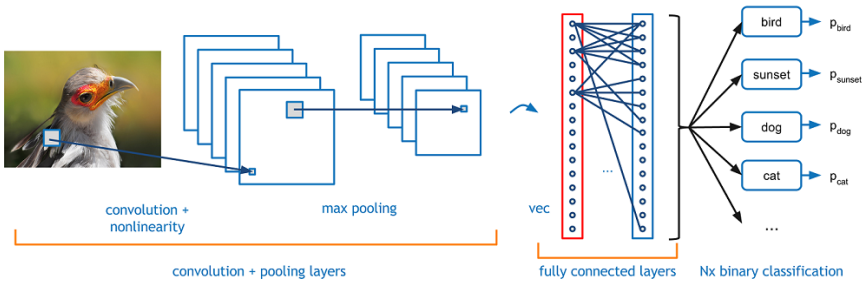
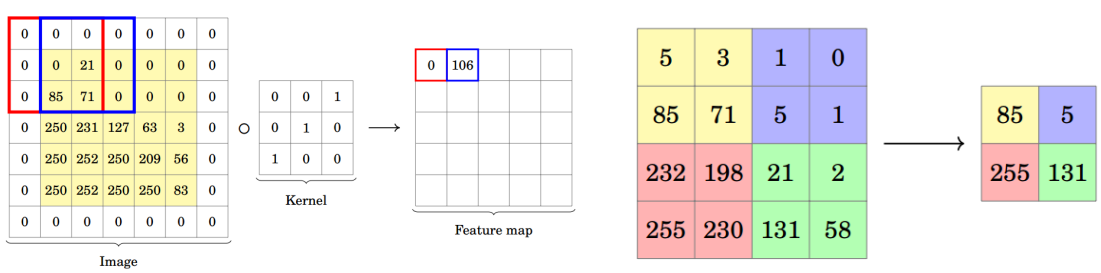


Figure 4.2 Example of a convolutional neural network. Image taken from [41].

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 4.3a 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 4.3a, the 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 4.3a, 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 4.3b 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.



(a) Convolution.

(b) Pooling.

Figure 4.3 Illustration of a convolution operation in (a) and a pooling operation in (b). Images taken from [96].

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 4.4 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.

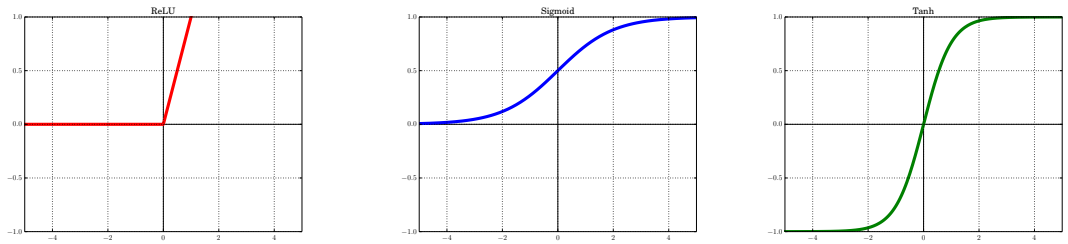


Figure 4.4 Examples of activation functions.

Table 4.2 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 $5n_1 + n_2 + 4$. 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 variables type and the problem dimension during an optimization but requires that the user implements the logic of these changes. HyperNOMAD exploit the implementation structure of NOMAD but automates and simplifies this task when handling hyperparameters of a neural network. Further explanations are presented in Section 4.4.

Table 4.2 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)$\}$

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 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

$$J(\Theta) = Err(pred, truth) + \lambda ||\Theta||^2 \tag{4.1}$$

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 [23, 31, 82].

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 changes 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 [48], 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.

The adaptive rule of Adagrad reduces the learning rate of the frequently updated weights rather aggressively. RMSProp [116] 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 [71] expands RMSProp further by taking two factors into account. The first one, β_1, 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 4.3.

Table 4.3 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.

Table 4.3 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 (4.1)	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 (4.1)	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 (4.1)	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]
	$\hat{d}_{k+1} = \frac{d_k}{1-\beta_1}; \quad \hat{G}_{k+1} = \frac{G_k}{1-\beta_2}$	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 (4.1)	Real	[0;1]

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 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 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 then 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 [17] where the blackbox takes the hyperparameters as inputs, 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 [79] described in Section 4.4.1.

4.3.2 Datasets

The HyperNOMAD package comes with a selection of datasets all meant for classification problems. Table 4.4 lists the datasets embedded so far through PyTorch [95], an open source library used to model and manipulate deep neural networks. Hyper-

NOMAD also allows the usage of a personal dataset; see the online documentation at `https://github.com/bbopt/HyperNOMAD`. When loading a dataset from Table 4.4, HyperNOMAD applies a normalization and a random horizontal flip to regulate and augment the data.

Table 4.4 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

The rest of the section describes the datasets used for benchmarking HyperNOMAD. First, a validation is done using both MNIST [81] and Fashion-MNIST [123] and once positive results are obtained, the second and more complex dataset, CIFAR-10 [75], is considered.

MNIST

MNIST [81] 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.

Fashion-MNIST

Fashion-MNIST [123] 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.

CIFAR-10

The second set of tests are performed with CIFAR-10 [75]. This dataset 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 these tests, the blackbox within HyperNOMAD is used to construct the convolutional neural network corresponding to the values of the hyperparameters described in Section 4.3.1. This network is trained, validated and tested on for each dataset according to the mode of operation of HyperNOMAD detailed in Section 4.4.

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 4.3 as inputs, constructs the corresponding network that is trained and tested before returning the test accuracy as the output. This blackbox uses the PyTorch package [95] for its simplicity. HyperNOMAD also contains an interface that runs the optimization of the blackbox using the NOMAD software [79] described in the rest of this section. The basic usage of HyperNOMAD is described in Appendix A.1.

4.4.1 Overview of NOMAD

The NOMAD software [79] is a C++ implementation of the MADS algorithm [13, 16] which is a direct search method that generates, at each iteration k, a set of points on the *mesh* $M^k = \left\{x + \mathrm{diag}(\boldsymbol{\delta}^k)z : x \in V^k, z \in \mathbb{Z}^n\right\}$ 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

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

generate a finite number of possible mesh candidates. This step can be based on surrogate functions, Latin hypercube sampling, etc. [10, 18] 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 $\boldsymbol{\Delta}^k \in \mathbb{R}^n$, 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 $\boldsymbol{\delta}^k \leq \boldsymbol{\Delta}^k$ is maintained. This relation insures 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, 12, 74]. The MADS algorithm is summarized in Algorithm 2.

4.4.2 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.

Blocks of hyperparameters

HyperNOMAD splits the hyperparameters (HPs) defined in Section 4.3.1 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

Algorithm 2: 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]**

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 4.2. 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 4.5 (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 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 4.6 (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 4.3. 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 4.7. The other HPs are put as the headers of their individual block with no associated variables.

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 4.5.

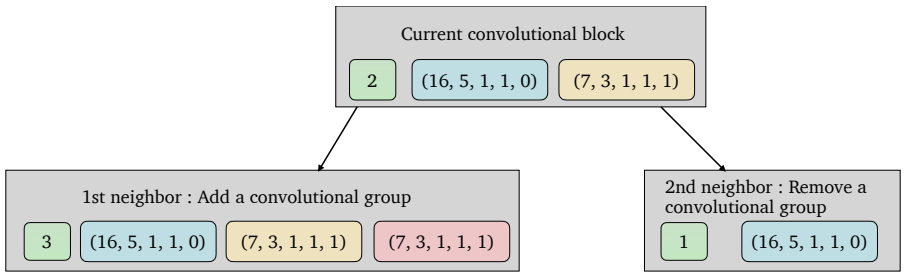


Figure 4.5 Example of a convolution block (top) composed of a header (green) containing the categorical variable that describe the number of convolutional layers, followed by two groups of variables, each one describing one convolutional layer. Its first neighbor is obtained by adding a convolutional layer at the right (bottom left) and the second neighbor is obtained by deleting the rightmost convolutional layer (bottom right).

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 4.6). 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

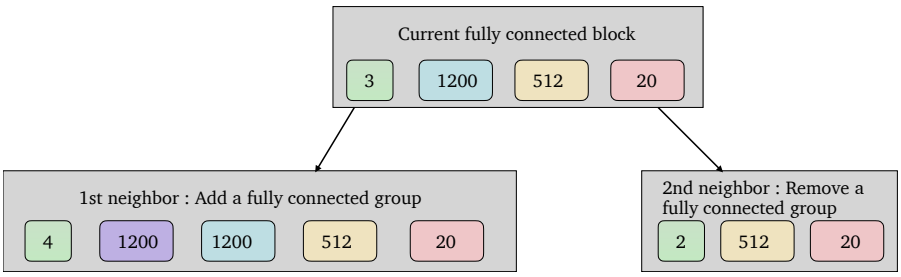


Figure 4.6 Example of a fully connected block (top) composed of a header (green) containing the categorical variable that describe the number of fully connected layers, followed by three groups of variables, each one describing the size of one fully connected layer. Its first neighbor is obtained by 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).

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 4.3. A single neighbor point is obtained by cycling through optimizers as shown in Figure 4.7. 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 4.7 also illustrates the neighbor of a specific optimizer block.

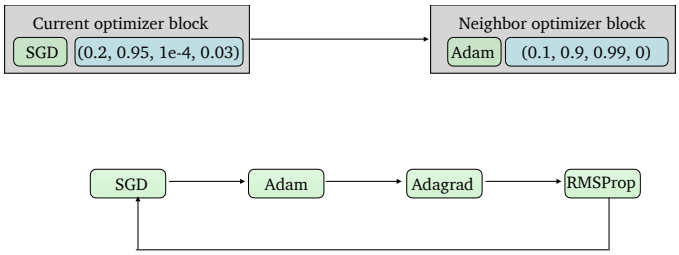


Figure 4.7 Example of an optimizer block and its neighbor (top) obtained by selecting the next optimizer by following the order defined in the bottom, and initializing the associated variables to their default values

.

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 se-

lecting among the three possible activation functions (see Section 4.3.1 and Table 4.2) : 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.

4.5 Computational results

This section summarizes the results obtained by HyperNOMAD and compares them to other methods when applied to the MNIST [81], Fashion-MNIST [123] and CIFAR-10 [75] datasets. For each series of tests, all the hyperparameters discussed in Section 4.3 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.1. 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

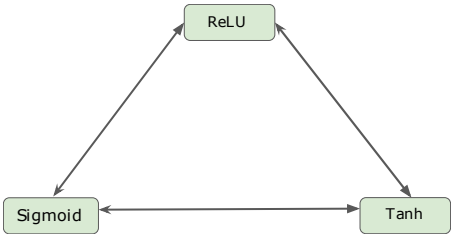


Figure 4.8 Illustration of how HyperNOMAD changes the categorical variable corresponding to the activation function.

GPUs (GTX 1070, TITAN Xp, and Tesla P100).

Every section compares HyperNOMAD with random search (RS) and a Bayesian method. The hyperopt library [26] is the one used in this comparison since it contains RS in addition to TPE, a Bayesian method that relies on Parzen trees [24]. Note that no comparison with the neural architecture search methods in the AutoML toolbox [55] is included in this study as they operate on a different premise than the one considered here. AutoML exploits a 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.

4.5.1 MNIST

The first tests are performed on the same blackbox provided by the authors of [42] which considers the MNIST dataset with the Caffe library [69]. The NOMAD software is directly used instead of HyperNOMAD, in order to compare the different methods of Table 4.6. The blackbox takes a simplified set of hyperparameters as described in Table 4.5, constructs a convolutional neural network that is trained on the MNIST dataset [81], and finally returns the validation accuracy as a performance measure.

Table 4.5 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	$\{1, 2, \ldots, 50\}$
3	Number of full layers	Categorical	$\{0, 1, 2\}$
4	Size of the full layer	Integer	$\{1, 2, \ldots, 50\}$
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 4.6 presents the results obtained by RS, RBFOpt, and SMAC, that are taken from [42], and NOMAD. These results show that using NOMAD surpasses all of the other methods in terms of both validation and test accuracy.

Table 4.6 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**

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 a greater flexibility than the previous Caffe blackbox since it takes into account all of the hyperparameters described in Section 4.3.1.

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 4.9 shows the evolution of HyperNOMAD versus the random search (RS) and the tree Parzen estimator (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 4.9 shows the evolution of the validation accuracy of the top configuration found by each optimization algorithm.

4.5.2 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-

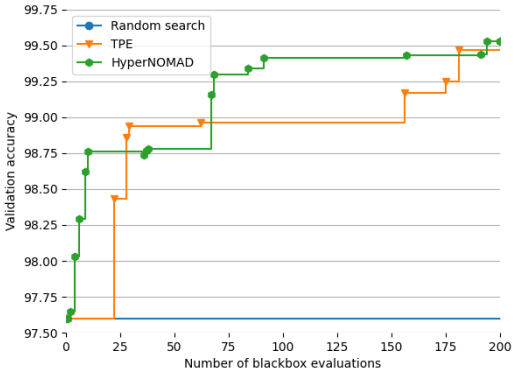


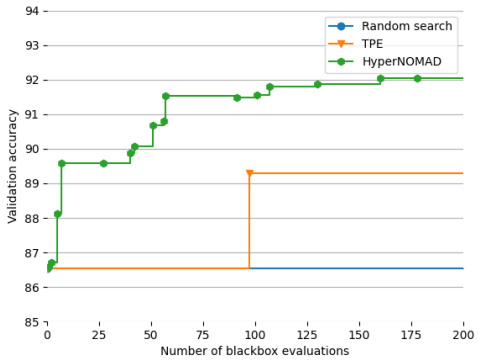
Figure 4.9 Comparison between HyperNOMAD, TPE and RS when launched from the default starting point of HyperNOMAD, on the MNIST dataset by representing the successes of each algorithm when a budget of 200 blackbox evaluations is given starting from the default configuration of HyperNOMAD.

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 4.10a.

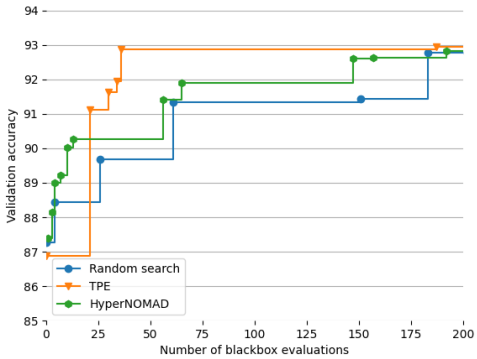
These results can be explained by the fact that both RS and TPE tend to sample a significant amount of infeasible configurations. A configuration is infeasible when the dimension of the inputs becomes null as it passes through the convolutional layers or when the kernel size is bigger than the image on which it is applied. This situation indicates the presence of hidden constraints [80] which 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 4.11 illustrates this behaviour 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 convolutional layer and two fully connected ones. Fi-

gure 4.10b 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%.



(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.

Figure 4.10 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.

4.5.3 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 [110] are very deep convolutional neural networks with small kernels. Figure 4.12 illustrates the architecture of the VGG-16 network.

Figure 4.13 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 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

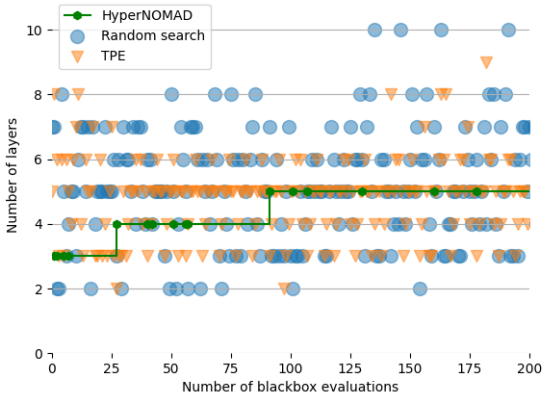


Figure 4.11 Number of layers in the configurations sampled by each algorithm during the HPO process on the Fashion-MNIST dataset, with a budget of 200 blackbox evaluations.

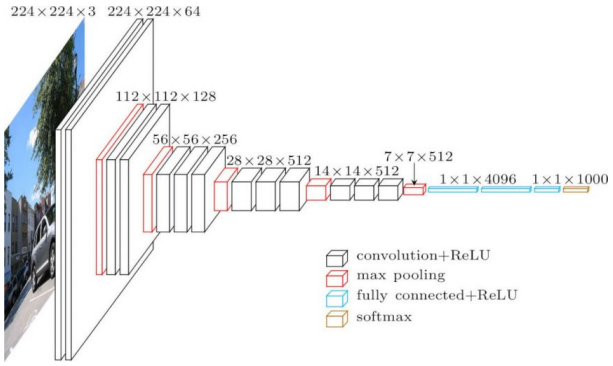
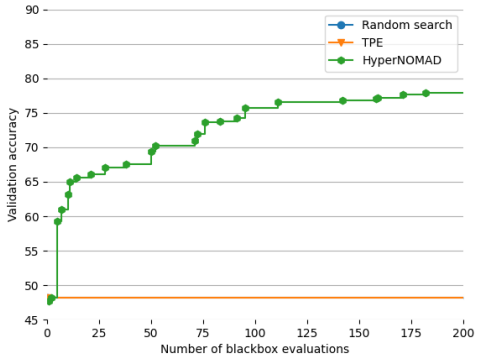


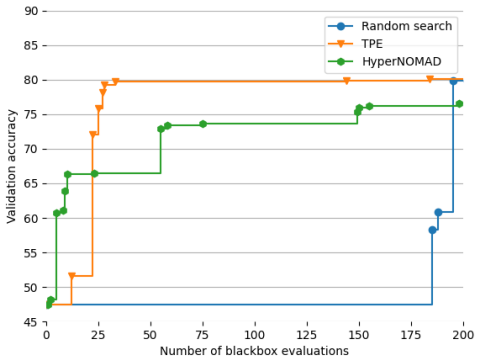
Figure 4.12 Architecture of the VGG-16 network. Image taken from [61].

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 behaviour 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 4.14a 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



(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.



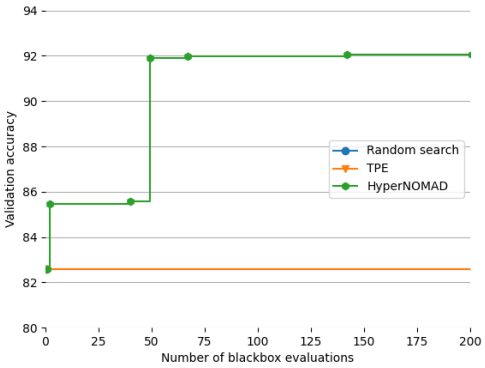
(b) Successes of each method when the number of layers is fixed during the optimization.

Figure 4.13 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.

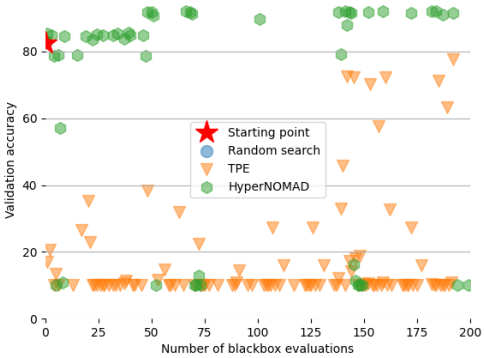
configuration found by HyperNOMAD achieves a final test accuracy of 92.05% whereas TPE and RS do not improve on the given initial point for different reasons. RS only samples infeasible architectures, which is expected when considering the high dimension of the optimization problem that increases the 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 4.14b 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%.

4.6 Discussion

This work introduces HyperNOMAD, a generic framework package for hyperparameter optimization of DNNs using the NOMAD software [79]. The key aspects 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 re-



(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.

Figure 4.14 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.

sults 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, 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.

CHAPITRE 5 ARTICLE II : TUNING A VARIATIONAL AUTOENCODER FOR DATA ACCOUNTABILITY PROBLEM IN THE MARS SCIENCE LABORATORY GROUND DATA SYSTEM

Statut : soumis pour révisions
Journal : *Expert systems with applications*
Coauteurs : Ryan Shahrouz Alimo et Sébastien Le Digabel

5.1 Abstract

The Mars Curiosity rover is frequently sending back engineering and science data that goes through a pipeline of systems before reaching its final destination at the mission operations center making it prone to volume loss and data corruption. A ground data system analysis (GDSA) team is charged with the monitoring of this flow of information and the detection of anomalies in that data in order to request a re-transmission when necessary. This work presents Δ-MADS, a derivative-free optimization method applied for tuning the architecture and hyperparameters of a variational autoencoder trained to detect the data with missing patches in order to assist the GDSA team in their mission.

5.2 Introduction

In the NASA Mars Science Laboratory (MSL), a ground data system analysis (GDSA) team is tasked with the analysis of telemetry data sent by the Mars Curiosity rover that travels through a pipeline of satellites and receptors. During its journey back to Earth, this data can be subjected to corruption and volume loss that needs to be detected efficiently in order to ask for a re-transmission when necessary. This problem is akin to an anomaly detection task where one must learn from unlabelled data to differentiate between a normal behaviour and outliers in order to identify the anomalous data. This task is so far handled manually by human experts and needs to be automated to speed up the treatment process and possibly increase the detection accuracy.

In the recent years, deep learning algorithms have shown their efficiency in solving several challenging regression and classification problems [21, 40, 70] thanks in parts to the

ever growing performances of deep neural networks. These computational graphs can learn from complex, real world datasets and make predictions with an accuracy that can sometimes surpass human experts. This study focuses on a particular type of autoencoders (AE) which are deep neural networks (DNNs) used for data compression, matrix factorization, anomaly detection, etc. AEs are unsupervised or semi-supervised learning algorithms that start by compressing the input data to map it into a lower dimension latent space before decompressing it back to recreate the original input. The learning phase aims at recreating the input data as closely as possible by minimizing the reconstruction error between the original data and its reconstruction as represented in Figure 5.1a. The usual framework for anomaly detection with AEs is to collect the reconstruction errors for all points of the dataset and find a threshold value that will determine which errors are considered outliers. AEs have proven themselves to be efficient tools for unsupervised anomaly detection [45, 102] with the caveat that the architecture and hyperparameters of such neural networks must be adequately chosen to get a competitive performance for real life applications.

As for any neural network, the choices for the hyperparameters that define the architecture as well as the training phase have a great impact on the overall precision of the network and its ability to generalize. This tedious and consuming process, in terms of time and computational power, can be modeled as a blackbox optimization problem. Blackbox optimization is a subfield of derivation-free optimization (DFO) [14,39], a discipline that considers optimization problems without relying on derivatives since they may not exist or are too complex to compute. DFO also covers the case where a function evaluation is the result of an expensive computation or a simulation, seen as blackboxes, that can fail at some points. In this case, the blackbox is defined so that its input is a particular configuration and the output is the test error of the corresponding network after it is trained on the data set. This work presents a new approach named Δ-MADS obtained by merging two DFO schemes, HyperNOMAD [78] and Δ-DOGS [28] in order to exploit the strong suits of each. The resulting algorithm is applied on the previously described tuning problem.

The remaining of the paper is organized as follows : Section 5.3 gives an overview of anomaly detection techniques and of the main approaches used to solve the HPO problem of deep neural networks. Section 5.4 presents Variational Autoencoders, their architecture, how they are trained for anomaly detection problems and the hyperparameters focused on in this paper. Section 5.5 describes the hybrid DFO algorithm which is

tested on this particular problem and benchmarked against other optimization schemes in Section 5.6. Finally, Section 5.7 synthesizes the results in a short conclusion.

5.3 Related work

Anomaly detection is an expanding field of research with many real life applications such as credit card fraud detection [35], finding network intrusions in the context of cyber security [119], industrial damage detection [124], etc. These problems usually amount to a classification task on imbalanced data, meaning that the outliers represent more often than not a small fraction of the overall data set. Also, depending on the data set, this problem can be a supervised, a semi-supervised or an unsupervised task. This work focuses on the two latter cases since the labels of the training data are the only ones available. Some popular clustering methods such as Gaussian mixtures, K-Means, DBSCAN, etc. can be applied on unsupervised anomaly detection problems [5, 51, 84] but they often fall short when dealing with high dimensional data with complex structures contrary to deep learning algorithms. In a semi-supervised or unsupervised context, generative neural networks such as AEs can be adapted for anomaly detection problems [45,102,108] by training them on normal data so that they learn to reproduce or generate good behaviors with a small reconstruction error. During this training, each data point is first reduced to a lower dimension representation that is expanded to its original size afterwards. The implicit hypothesis is that outliers should be different enough from normal data so that a trained AE will get a higher reconstruction error on outliers than on a data that behaves like the normal training points. However, some real life applications do not satisfy this hypothesis such as the one considered in this study. Indeed, all the data received by the MSL goes through the same steps and systems and has therefore the same underlying structure which represents a challenge for AEs that struggle with separating the normal behavior from the anomalies.

Variational autoencoders (VAEs) [44] are generative, probabilistic graphical models that share a similar architecture with regular AEs as shown in Figure 5.1b with the main difference residing in the latent representation of the data. Instead of mapping each input point to a deterministic lower dimensional vector, a VAE maps it with a region in the latent space by learning the parameters of a probability distribution that approximates the posterior. As shown in [8], VAEs can surpass normal AEs in anomaly detection problem such as finding the outliers on the MNIST data set. Further details on VAEs and

their training are presented in Section 5.4.

As any DNN, VAEs are extremely sensitive to their structure, or architecture, and to the values of the hyperparameters related to the optimization process that happens during the training phase. Many different approaches were explored to automate the search for optimal hyperparameters starting with the grid search which evaluates all the possible combinations of hyperparameters from a constrained search space. This method is clearly expensive and does not scale well with the dimension of the problem. The random search [25] has been shown to be more effective than grid search but is still highly expensive and lacks adaptiveness to the problem. Bayesian methods offer a more sophisticated alternative by either constructing a model over the objective function f in the case of a Gaussian processes [121] or random forests [33], or over the distribution of the good and bad configurations in the case of tree parzen estimators [24], by using the previously evaluated points. Other approaches were tested such as reinforcement learning [19, 130] which is successfully used to find the appropriate architecture of convolutional neural networks, and more recently the HyperNOMAD [77, 78] software, based on the mesh adaptive direct search (MADS) algorithm [15], was able to yield good results when optimizing both the architecture and the training hyperparameters simultaneously. The main drawback of this software is its lack of global exploration strategy. A hybrid algorithm is proposed in this work that combines the local refinement of HyperNOMAD with the global search of Δ-DOGS [28], another DFO algorithm equipped with a global search model based on Delaunay's triangulation which was shown to explore efficiently the search space on smaller dimension problems and on limited types of variables. This new approach manages to exploit the advantages of each method and is further discussed in Section 5.5.

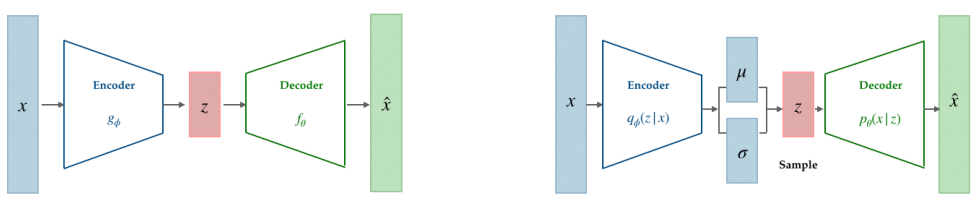
5.4 Anomaly Detection with Variational Autoencoders

The following section provides a high-level description of variational autoencoders (VAEs), their architecture and training before going through the list of hyperparameters considered for the anomaly detection problem.

5.4.1 Overview of variational autoencoders

A VAE, as shown in Figure 5.1b, is a deep neural network made up of three sections : an encoder defined by the weights ϕ, an encoding layer in the middle of the network of size

n_e and a decoder defined by the weights θ. Let $x \in \mathbb{R}^{n_0}$ be an input vector which is first passed to the encoder that reduces its dimension from layer to layer until reaching the middle of the network which has the smallest size : n_e. At this point, the VAE generates two vectors $\mu, \sigma \in \mathbb{R}^{n_e}$ that represent the mean and variance of a normal distribution from which a sample $z \in \mathbb{R}^{n_e}$ is drawn. Therefore, the VAE associates the input vector x, and consequently its class, with a region in the latent space where its lower dimensional representation z is more likely to be. Note that the choice of the normal distribution is used for practical purposes and can be altered if needed. The role of the encoder of a VAE is changed from a deterministic compressing function in the case of a standard AE, to a probabilistic model that learns the distribution of the latent representation z for a given input x noted $q_\phi(z|x)$. The second phase consists of passing the latent vector z to the decoder that expands it from layer to layer until forming a reconstruction $\hat{x} \in \mathbb{R}^{n_0}$ which is compared to the original input x. Once again, the decoder is no longer the deterministic function of a standard AE, but is now a probabilistic model $p_\theta(x|z)$ that learns the distribution of $\hat{x}$, and therefore of x, knowing the input z.



(a) Autoencoder architecture : the input vector x is passed to the encoder g_ϕ that produces a lower dimension representation z which is fed to the decoder f_θ to produce the reconstruction $\hat{x}$.

(b) Variational autoencoder architecture : the input vector x is passed to the encoder $q_\phi(z|x)$ that produces the mean μ and standard deviation σ of a normal distribution from which a sample z is drawn to be fed to the decoder $p_\theta(x|z)$ to produce the reconstruction $\hat{x}$.

Figure 5.1 Structure of an autoencoder on the left and of a variational autoencoder on the right.

Therefore, the latent representation z, and consequently $\hat{x}$ are not deterministic for the same input x which poses a challenge during the training of the VAE, especially when the error is backpropagated through the network to update its weights. The reparametrization

trick [44] was introduced to solve this exact problem : the latent representation is now calculated as $z = \mu + \epsilon\sigma$ with $\epsilon \sim N(0, 1)$ instead of randomly sampling $z \sim N(\mu, \sigma)$ directly. By moving the random sampling to ϵ, the backpropagation, which can not be applied on a stochastic term, can effectively reach the layers μ, σ and the rest of the encoder.

The loss function of the VAE is composed of a term that quantifies the reconstruction error between the input x and the reconstruction $\hat{x}$, plus a regularization term on the latent space to ensure that the encoded distribution is close to a standard normal distribution using the Kulback-Leibler divergence as seen in the following Equation :

$$L = ||\hat{x} - x|| + D_{KL}(q_\phi(z|x)||p(z)) \tag{5.1}$$

where $p(z) \sim N(0, I)$ and D_{KL} is the Kulback-Leibler divergence term between $q_\phi(z|x)$ and $p(z)$. This regularization gives desirable properties to the latent space by ensuring a good distribution of the latent variables which is essential for a generative model [9].

5.4.2 Hyperparameters of a VAE

This study focuses on tuning both the architecture and the learning hyperparameters to obtain an effective VAE for a particular anomaly detection task. The number of variables defining the architecture can be significantly reduced by considering a symmetric VAE with layers of decreasing, respectively increasing, size in the encoder, respectively decoder. The architecture can therefore be described by two integer variables, one representing the number of encoding layers and the second for the dimension of the latent space. The size of the remaining encoding, respectively decoding, layers can be deduced from this two values by imposing a linear decrease, respectively increase. The activation function is used to introduce a nonlinear transformation after each layer of the encoder, respectively the decoder. The dropout rate is added as a regularization technique to avoid the over-fitting issue. As for the training phase, the batch size determines the number of training data points passed to the VAE at the same time which affects both the learning of the network and the speed of the training. Additionally, the choice of the optimizer algorithm along with four hyperparameters are also considered in this tuning problem which are summarized in Table 5.1.

Table 5.1 Hyperparameters related to the training of the VAE.

Optimizer	**Hyperparameter**	**Type**	**Range**
Stochastic Gradient Descent (SGD)	Initial learning rate	Real	[0;1]
	Momentum	Real	[0;1]
	Damping	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]
	Smoothing constant	Real	[0;1]
	Weight decay	Real	[0;1]

where β_1 is the factor for the first moment estimates and β_2 is factor for the second moment estimates.

In practice, VAEs can be used for a semi-supervised anomaly detection problem by applying the following protocol : the network is trained on normal data so that it learns to replicate normal behaviors only which results in bigger reconstruction errors when anomalous data is passed through the VAE. In order to classify each input data, the generated error is compared to a certain threshold value $\alpha \in \mathbb{R}$ which can either be fixed by the user according to some acquired knowledge on the classification task at hand, or has to be also tuned as an additional hyperparameter which is the case for the application considered here.

5.5 The Δ- MADS method

This section describes Δ-MADS, a hybrid algorithm that mixes the local search of HyperNOMAD [77, 78] with the global exploration scheme of Δ-DOGS [28]. Δ-MADS is

Table 5.2 List of the hyperparameters of the VAE considered for the tuning problem.

Hyperparameter	**Type**	**Range**
Number of encoding layers	Integer	$[1, 50]$
Dimension of the latent space	Integer	$[1, n_0[$
Batch size	Integer	$[10, 512]$
Activation function	Categorical	1 : ReLU, 2 : Sigmoid, 3 : Tanh.
Dropout rate	Real	$[0, 1]$
Optimizer choice	Categorical	1 : SGD, 2 : Adam. 3 : Adagrad. 4 : RMSProp.
4 HPs of the optimizer (Table 5.1)	Real	$[0, 1]$
Threshold α	Real	$[0.50, 1]$

designed to solve derivative-free optimization problems formulated as follows :

$$\min_{x \in \Omega} f(x) \tag{5.2}$$

where $\Omega = \{x \in \mathbb{R}^n \quad | \quad a \leq x \leq b \text{ with } a, b \in \mathbb{R}^n\}$. The notation x^N refers to the integer and categorical components of the vector x and x^R the real elements of x. The entire vector x can be reconstructed by combining x^N and x^R which is written as $x = x^N \cup x^R$. In the context of this specific application, tuning a variational autoencoder can be modeled as a derivative-free, and more specifically a blackbox, optimization problem where the objective function f takes a set of hyperparameters, builds the corresponding variational autoencoder, trains, validates and tests its performance before returning the mean $F1$ score, described in Section 5.6, on the test set as a measure of performance.

HyperNOMAD, being based on MADS [15], is an iterative algorithm with two phases. The first one, called the *search*, is an optional and flexible step where a global optimization scheme can be implemented and the second one, called the *poll*, is rigorously established. At each iteration k, the mesh $M_k = \{x + \Delta_k^m Dz, z \in \mathbb{N}^{n_D}, x \in C\}$ is defined where C, called the cache, is the list that stores all of the previously evalua-

ted points, the matrix $D \in \mathbb{R}^{n \times n_D}$ has columns that form a positive spanning set and $\Delta_k^m \in \mathbb{R}^+$ is the mesh size. The *poll* starts around the current point x_k and defines the poll set $P_k = \{x_k + \Delta_k^m d \mid d \in D_k\}$ with $||\Delta_k^m d|| \approx \Delta_k^p$, that contains the candidates evaluated opportunistically, meaning that the *poll* step will end as soon as a better point is found. In that case, a new iteration starts with a larger mesh size and otherwise, the mesh size is reduced. HyperNOMAD is adapted to handle real, integer and categorical variables [2, 16]. The later requires to define an *extended poll* [16] which links the different search spaces related to different values of the categorical variables. The MADS algorithm offers a hierarchy of convergence results depending on the properties of the optimization problem. In [15], the authors prove that MADS converges to a Clarke, respectively contingent KKT, stationary point if f is Lipschitz near the limit point, respectively strictly differentiable at the limit point. The convergence of MADS is derived solely from the *poll* step and is maintained if the *search* generates a finite number of trial points each time it is called which are then projected onto the mesh M_k at iteration k.

Δ-DOGS is a family of iterative derivative-free optimization methods [6, 7, 28] that rely on a surrogate model of the objective function to direct the optimization. The surrogate search function s is computed at each iteration by combining an interpolation function p with an artificially generated uncertainty function e based on Delaunay's triangulation that plays a similar role to the acquisition functions in a Bayesian optimization scheme so that $s(x) = p(x) - Ke(x), x \in \mathbb{R}^n$. The tuning parameter K depends on the target value y^* that the user hopes to reach during the optimization. Δ-DOGS is proven to globally converge in the case of convex optimization problem where the Lipschitz bound of the objective function is bounded [28], however it scales poorly to the dimension of the optimization problem n and is not adapted to handle mixed variable problems.

The Δ-MADS algorithm 3 mixes aspects of the two DFO schemes previously described by implementing the surrogate function of Δ-DOGS into the *search* step of HyperNOMAD. Also, while the *poll* step is optimizing the entire set of hyperparameters listed in Section 5.4, the *search* phase keeps the integer and categorical variables x_k^N fixed and optimizes the sub-problem considering only the continuous variables x_k^R therefore reducing the dimension of the problem and interpolating only on continuous functions. The algorithm starts with an initial point x_0 and a target value y_0 passed onto the search of Δ-DOGS which is given a certain budget of function evaluations, the best solution found in the *search* is passed to the *poll* from which the local refinement starts. The target value y_k is re-evaluated at each iteration depending on whereas it was achieved or not. For

this minimization problem, the target y_k is decreased if y_{k-1} was attained or improved upon, and increased if not. The algorithm alternates this way between the two phases until the function evaluation budget is depleted or a convergence condition is achieved. The convergence properties of this novel approach are inherited from MADS since the *search* step is guaranteed to produce a finite number of mesh candidates.

Algorithm 3: Δ-MADS : Hybrid between HyperNOMAD and Δ-DOGS

initialization : $x_0, y_0, \epsilon \in]0, 1[, k = 0$;
while *not stop* **do**
  Search step : Fix the integer and categorical variables and apply Δ-DOGS on the remaining variables x_k^R with the target y_k ;
  return new x'^R_k, reconstruct the complete vector $x'_k = x_k^N \cup x'^R_k$ and project it on the mesh ;
  Poll step : Apply HyperNOMAD on the new x'_k and return the best feasible solution found x_{k+1} ;
  Updates : Set f_{k+1} the best objective value ;
  if $f_k < y_k$ **then**
    $y_{k+1} = y_k - \epsilon$;
  else
    $y_{k+1} = y_k + \epsilon$;
  end
end

5.6 Numerical results for the MSL data accountability

The Mars Curiosity rover transmits telemetry data to the MSL ground system operations team through a complex pipeline of systems where each transfer leaves the data inevitably susceptible to corruptions. In order to increase the traceability in this process, the ground data system (GDS) records metadata about the transmissions at three locations in the downlink process : the orbiter used to transmit the data, JPL Data Control, and the data's final destination in the MSL GDS. This metadata is analyzed by experts to determine if the original data is successfully transferred, called a complete pass, or not, called an incomplete pass.

This work considers the latest dataset available at this time which includes a total of 9805 passes, 8493 of which are complete passes and the remaining 1312 are incomplete. Each data point is a vector of dimension 43 with real, integer and boolean features such

as the time of transmission or the position of the intermediate orbiter. With this proportion of around 13% anomalies, a 87% classification accuracy can easily be achieved simply by labelling every pass as a complete one which shows the limitation of usual accuracy metrics when dealing with unbalanced datasets. In this case, other metrics such as the precision P, recall R and $F1$ score, shown in Equation 5.3, are more adequate to correctly evaluate the quality of a classifier. For each class, the precision P measures how many of the classifier's predicted labels are correct. The recall R quantifies how many true members of a certain class are correctly identified and the $F1$ score combines both metrics as follows :

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN}, \quad F1 = 2\frac{R \times P}{R + P} \tag{5.3}$$

where TP is the number of true positives, TN the number of true negatives and FN the number of false negatives.

As an initial step toward solving this anomaly detection problem, different unsupervised methods are tested without hyperparameter optimization. The results of these different methods are compiled in Table 5.3 and compared against the GDS labeler, which is the previous algorithm used by the GDSA team to identify incomplete passes. The GDS Labeler yields the best results for correctly identifying complete passes, however the recall of 55% on incomplete passes means that only 55% of the anomalies are correctly detected as such. Plus, the GDS Labeler takes about 5 hours to complete the anomaly detection task.

Both the KMEAN and Gaussian mixture algorithms are taken from the `scikit-learn` [97] library, and are tasked of producing two clusters with all their remaining parameters set to their default values. The VAE, whose hyperparameters are chosen without any prior intuition, gets the best results on correctly detecting incomplete passes compared to the rest of the detection methods which justifies the hyperparameter optimization effort conducted especially considering that its training, validation and testing takes around 50 seconds.

The blackbox that models this problem takes all the hyperparameters listed in Section 5.4 as inputs, constructs the corresponding VAE and splits the data into three sets : training, validation and test with the training set containing complete passes only and the remaining two are a mix of normal behavior and anomalies. After the training and validation of the VAE, the blackbox returns the average of the $F1$ scores on complete and incom-

Tableau 5.3 Performance of the unsupervised learning methods : KMEAN, Gaussian mixtures, autoencoder and variational autoencoder without hyperparameter optimization on the missing data detection problem compared against the current GDS labeler.

	GDS labeler		KMEAN		Gaussian mixture		Untuned AE		Untuned VAE	
	Cpl.	Inc.	Cpl.	Inc.	Cpl.	Inc.	Cpl.	Inc.	Cpl.	Inc.
P	0.94	0.74	0.54	0.12	0.95	0.38	0.55	0.74	0.84	0.75
R	0.97	0.55	0.04	0.77	0.80	0.73	0.63	0.62	0.79	0.80
$F1$	**0.95**	0.63	0.08	0.20	0.87	0.50	0.58	0.67	0.81	**0.77**

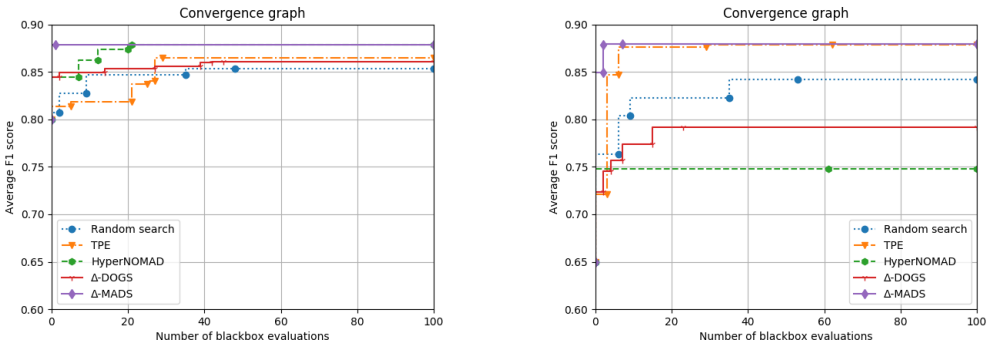
plete passes on the test set as a performance measure. The comparison is conducted with two different starting points, one that gives an initial $F1$ score of 81%, refereed to as a advantageous initialization, and the other gives an initial $F1$ score of 65% which is refereed to as the disadvantageous initialization. The goal here is to observe the behavior of the HPO algorithms in two different settings.

Table 5.4 compiles the scores of the best configuration found by each hyperparameter optimization method : random search (RS), tree parzen estimator (TPE), HyperNOMAD, Δ-DOGS and Δ-MADS presented in Section 5.5, starting from the advantageous, respectively the disadvantageous, initialization. Both the random search and the tree parzen estimator are tested through the Hyperopt library [26]. Each solution is evaluated, in the sense of the blackbox, five times and the mean and standard deviation of each score are reported in Table 5.4. In both cases, the results show that all the hyperparameter optimization schemes were able to improve on the original VAEs, thus proving the necessity of this tuning effort in a real life application. Starting with the advantageous initialization, the scores of the best solutions obtained by each scheme can be split into two sets : the random search, tree Parzen estimator and Δ-DOGS ended with configurations with an $F1$ score of $85 - 86\%$ on complete passes and 84% on incomplete passes and both HyperNOMAD and the Δ-MADS obtained the best solutions with an $F1$ score of 88% on complete passes and 87% on the incomplete ones. The advantage of Δ-MADS is better highlighted through Figure 5.2a that shows how the Δ-MADS algorithm is always ahead of HyperNOMAD for a certain budget of blackbox evaluations. The hybrid algorithm allows then to reduce the computational cost of the HPO problem. The example with the disadvantageous configuration gives more contrasted results : HyperNOMAD gets the worst results on both complete and incomplete passes which highlights its difficulty to

move from a disadvantageous starting point. Δ-DOGS has a similar $F1$ score of 76% on the complete passes and a better performance on the incomplete ones with an $F1$ score of 79% which is believed to be a consequence of the presence of integer and categorical variables in the HPO problem. The best $F1$ score of 87% on the complete passes is obtained by both Δ-MADS and the tree parzen estimator and Δ-MADS achieves the best results of 87% on the incomplete ones. Once again, Δ-MADS does so with significantly less blackbox evaluations than any other HPO scheme as is shown in Figure 5.2b. All the HPO methods require an execution time between $1, 5$ and 2 hours to evaluation 100 configurations with each blackbox evaluation lasting from 35 to 70 seconds.

Table 5.4 Comparison between the scores of the best VAE found by each hyperparameter optimization method with the advantageous initialization (top) and the disadvantageous one (bottom).

	RS		TPE		HyperNOMAD		Δ-DOGS		Δ-MADS	
	Cpl.	Inc.	Cpl.	Inc.	Cpl..	Inc.	Cpl.	Inc.	Cpl.	Inc.
P	$0.93 \pm 3e^{-2}$	$0.78 \pm 1e^{-2}$	$0.93 \pm 3e^{-2}$	$0.78 \pm 1e^{-2}$	$0.97 \pm 2e^{-3}$	$0.79 \pm 1e^{-3}$	$0.93 \pm 4e^{-3}$	$0.77 \pm 3e^{-3}$	$0.97 \pm 2e^{-3}$	$0.79 \pm 1e^{-3}$
R	$0.80 \pm 2e^{-2}$	$0.92 \pm 3e^{-2}$	$0.79 \pm 8e^{-3}$	$0.92 \pm 4e^{-2}$	$0.80 \pm 2e^{-2}$	$0.97 \pm 2e^{-3}$	$0.70 \pm 5e^{-3}$	$0.92 \pm 5e^{-3}$	$0.80 \pm 1e^{-3}$	$0.97 \pm 2e^{-3}$
$F1$	$0.86 \pm 8e^{-3}$	$0.84 \pm 1e^{-2}$	$0.86 \pm 1e^{-2}$	$0.84 \pm 2e^{-2}$	$\mathbf{0.88 \pm 1e^{-3}}$	$\mathbf{0.87 \pm 1e^{-3}}$	$0.85 \pm 3e^{-3}$	$0.84 \pm 3e^{-3}$	$\mathbf{0.88 \pm 2e^{-4}}$	$\mathbf{0.87 \pm 5e^{-4}}$
P	$0.91 \pm 2e^{-2}$	$0.77 \pm 6e^{-3}$	$0.95 \pm 9e^{-3}$	$0.78 \pm 5e^{-3}$	$0.73 \pm 1e^{-1}$	$0.76 \pm 7e^{-2}$	$0.95 \pm 9e^{-3}$	$0.67 \pm 1e^{-3}$	$0.97 \pm 8e^{-3}$	$0.79 \pm 3e^{-3}$
R	$0.80 \pm 6e^{-3}$	$0.90 \pm 3e^{-2}$	$0.80 \pm 8e^{-3}$	$0.95 \pm 1e^{-2}$	$0.84 \pm 1e^{-1}$	$0.54 \pm 2e^{-2}$	$0.64 \pm 5e^{-3}$	$0.95 \pm 8e^{-3}$	$0.79 \pm 8e^{-3}$	$0.97 \pm 4e^{-3}$
$F1$	$0.85 \pm 1e^{-2}$	$0.83 \pm 1e^{-2}$	$\mathbf{0.87 \pm 3e^{-3}}$	$0.86 \pm 3e^{-3}$	$0.76 \pm 3e^{-2}$	$0.60 \pm 1e^{-1}$	$0.76 \pm 1e^{-3}$	$0.79 \pm 2e^{-3}$	$\mathbf{0.87 \pm 8e^{-3}}$	$\mathbf{0.87 \pm 3e^{-3}}$



(a) Convergence graphs for each HPO algorithm on the advantageous initialization.

(b) Convergence graphs for each HPO algorithm on the disadvantageous initialization.

Figure 5.2 Comparison between 5 hyperparameter optimization algorithms : random search, tree Parzen estimator, HyperNOMAD, Δ-DOGS and Δ-MADS, through their convergence curves on two different initializations.

5.7 Conclusion

This work presents Δ-MADS, a hybrid derivative-free optimization algorithm applied to solving the hyperparameter optimization problem of a variational autoencoder capable of adequately detecting anomalous data sent by the Mars Curiosity rover to the Mars science laboratory. The positive results obtained, especially on detecting anomalies, show the importance of such tools to assist the human experts in dealing with this type of unsupervised anomaly detection problems. The numerical results show that Δ-MADS is able to score better than its two components : Δ-DOGS and HyperNOMAD separately plus, it also allows to find better configurations with less computational budget compared to other schemes. Additionally, the algorithm can be used in broader applications since it does not rely on any prior knowledge on the dataset or any other bias.

CHAPITRE 6 ARTICLE III : USE OF STATIC SURROGATES IN HYPERPARAMETER OPTIMIZATION

Statut : soumis
Journal : *SN operations research forum*
Coauteur : Sébastien Le Digabel

Abstract Optimizing the hyperparameters and architecture of a neural network is a long yet necessary phase in the development of any new application. This consuming process can benefit from the elaboration of strategies designed to quickly discard low quality configurations and focus on more promising candidates. This work aims at enhancing HyperNOMAD, a library that adapts a direct search derivative-free optimization algorithm to tune both the architecture and the training of a neural network simultaneously, by targeting two keys steps of its execution and exploiting cheap approximations in the form of static surrogates to trigger the early stopping of the evaluation of a configuration and the ranking of pools of candidates. These additions to HyperNOMAD are shown to improve on its resources consumption without harming the quality of the proposed solutions.

Keywords : Early stopping - Static surrogate - Hyperparameter optimization - Architecture search - Derivative-free optimization - Blackbox optimization.

6.1 Introduction

The efficacy of deep neural networks (DNN) to discover patterns in complex datasets is seen throughout multiple applications where the appropriate variant of a DNN often manages to score higher than human experts or other machine learning algorithms and even to push the current state of the art. A particular class of DNN includes convolutional neural networks (CNN) which are used for image classification, image segmentation, or object detection. They have been gaining in popularity and attention from the scientific community in the recent years as they are at heart of important technological advances such as imitation learning [52] or medical imagery analysis [91] to name a few.

An important challenge when adopting a deep learning approach for a new task is to find the appropriate neural network by deciding on a set of hyperparameters that determine the architecture, i.e, the number of layers, type of blocks and connection, and the training regime. The final score of the network is highly sensitive to the tuning of these hyperparameters and there is no rigorous or intuitive rule that directly provides a range for their optimum values. This hyperparameter optimization (HPO) problem can therefore be framed as a blackbox optimization problem where the objective value is the result of a computation with no analytical formula and no known derivatives. Moreover, this process is time consuming and expensive in computational resources as each configuration trial is equivalent to training a new network long enough to infer its generalization score. In a supervised learning setting, the HPO problem can be expressed as

$$\min_{\Phi} f(\Phi, \theta^*) \quad \text{with } \theta^* \in \operatorname*{argmin}_{\theta} \mathcal{L}_{\Phi,\theta}(X, Y) \tag{6.1}$$

where f is the objective function that corresponds to a measure of performance of the neural network, usually the validation loss, Φ is the mixed integer vector of all the hyperparameters that define the network, X, Y are the validation data and labels, $\mathcal{L}$ is the loss function that the network optimizes during the training by updating the weights θ.

Derivative-free optimization (DFO) [14, 39] provides a class of methods that are well suited to tackle such blackbox HPO problems as they do not need the explicit expression of the objective function and/or the constraints, nor do they rely on the derivatives for their execution. Two classes of DFO algorithms can be defined : Model-based and direct search methods. Model-based algorithms use a static or dynamic surrogate function $\hat{f}$ as an approximation of the true objective f to guide the optimization. Static surrogates, also called simplified physics surrogates, are defined before the start of the optimization and remain unchanged during the execution in contrast to dynamic surrogates, such as Gaussian processes that are updated with each iteration of the DFO method or with each new evaluation of the objective function. Direct search methods base their exploration of the search space solely on the values of the evaluated points and explore said space through a pattern as it is the case for the Mesh Adaptive Direct Search (MADS) [13] or on a simplex in the case of the Nelder-Mead algorithm [94]. The properties of DFO methods explain their popularity in the context of HPO of deep neural networks where they are often included in specialized libraries such as hyperopt [26] or Oríon [32]. Similarly, the HyperNOMAD toolbox [77, 78] is developed as an adaptation of MADS

to simultaneously optimize the architecture and the training phase of a CNN for a given dataset as expressed in (6.1). This open source library was shown to have a competitive performance against other popular approaches such as Bayesian optimization [24] or a random search [25] on the MNIST [81], Fashion-MNIST [123] and CIFAR-10 [75] datasets.

The objective of this work is to develop a set of protocols that speed up the HPO process in HyperNOMAD. The proposed measures are based on the use of two static surrogates that affect different steps of the algorithm execution : first the training log of the best encountered network serves as a baseline of comparison with the currently evaluated point and allows for early stopping decisions if the performance of the current network seems unsatisfactory. Second, another static surrogate is employed to rank the candidates to be evaluated at each iteration of HyperNOMAD which applies an opportunistic evaluation strategy, meaning that an iteration is interrupted as soon as a better score is recorded, therefore disregarding the other pool of candidates. The implementation of these two surrogates is shown to significantly speed up the HPO process thus saving expensive resources. Note that the discussed methods and improvements are not particularly dependant on HyperNOMAD alone but can also be applied or added to another optimization scheme.

The rest of the document is structured as follows : Section 6.2 provides an overview of the literature regarding strategies for early stopping and accelerating the costly HPO problem. Section 6.3 goes through the algorithm behind HyperNOMAD to highlight where the added strategies are implemented. Section 6.4 dives into the details of the proposed speed ups and the management of each surrogate. Finally, Section 6.5 compares the original version of HyperNOMAD with the proposed additions and a synthesis of these results is presented in the discussion.

6.2 Related work

The common problem of fine tuning a neural network to yield the best performance measure for a specific dataset is highly complex and consuming in terms of time and computational resources, so much so that it is often treated in two separate steps : one for finding the neural architecture and one for optimizing the hyperparameters related to the training of the network. Regardless of which phase is considered, the problem can be formulated as an expensive blackbox optimization problem [14, 39] since the evaluation

of the objective function is equivalent to training a new neural network with a specific configuration for a certain amount of time in order to estimate its final accuracy. That necessary amount of time is unknown a priori and has a direct impact on the quantity of resources needed to carry out this task. Moreover, experience shows that not every hyper-parameter configuration can yield satisfactory results and it is best to develop the tools that quickly detect such cases and prematurely stop their training, thus saving valuable resources than can be allocated to more promising candidates.

Early stopping is first introduced as a regularization technique to ovoid the overfitting issue that arises in machine learning. Theoretically, the training of a network, and more generally any machine learning algorithm, should be interrupted as soon as the validation error starts to increase as that behaviour indicates a training saturation and an overfitting to the training data. However, as the authors of [101] point out, this criteria can not be directly applied on a real validation error curve that tend to be irregular This work also provides a set of adapted criteria that prompt early stopping when the increase of the validation error exceeds an experimentally predetermined threshold, when the deterioration of the validation error is greater than the improvement of the training error or when the generalisation error consistently increases over a certain number of *epochs*. An epoch describes one pass on the entire training data. The training usually consists in multiple epochs before reaching convergence. Each of these criteria is shown to provide good a trade-off between the overall training time and the final performance of the network on a collection of problems. Early stopping is also exploited in Hyperband [85] which adapts the random search (RS) [25] with an efficient resource management scheme. This algorithm starts with allocating a fraction of the resources to each new configuration and compares the resulting validation loss, half of the the low performing candidates are stopped and the other half is granted more resources and the process is repeated throughout the execution of Hyperband with only the relative top performers being trained with the full resources. Compared to other HPO algorithms such as Bayesian methods, Hyperband allows for a significant speed up on multiple datasets such as MNIST and CIFAR-10. The same principle of "starting many, stopping early, continuing some" is applied in [46] where the authors obtained the best results when discarding the least performing networks after $20\% - 30\%$ of the total training budget. All aforementioned works rely on the observed validation curves, yet they highlight a different aspect of the early stopping decisions. In [101], stopping the training is only dependent on the scores of the network itself whereas [46, 85] stop networks for their relative scores compared to

a pool of candidates.

The HPO process can also benefit from developing ranking tools able to isolate promising candidates from a pool of new configurations. This idea is well known in the general optimization context where multiple research showed that combining an appropriate ranking tool with an opportunistic evaluation strategy is beneficial in terms of convergence speed [4, 59, 107]. An opportunistic strategy implies stopping the current iteration as soon as an improvement is recorded. For the deep learning context, survey [49] dedicates a section to some of the methods used to estimate the final validation performance, or to a lesser extent, the expected scores in future iterations. These strategies extrapolate the validation or training curves [47] via a regression method by factoring the observed scores and, possibly, the architectural and learning hyperparameters of the network [20, 73]. The usefulness of these estimators in speeding-up hyperparameter optimization or neural architecture search is shown in each work but their main challenge is their need to be trained on a substantial amount of data before having reliable predictions. Another score estimation route uses low fidelity surrogates such as training a network with a lower epoch budget [127], on a subset of the dataset [72, 118] or on lower resolution images [36]. The predicted scores with these methods are expected to be under-estimations of the real validation scores, however they can still be used as a ranking tool to separate the candidate configurations between the top and low performers as long as a good balance in the low fidelity estimate is found so that the under-estimations are not too large.

6.3 The HyperNOMAD package

The open-source library HyperNOMAD [77, 78] is designed as an adaptation of the NOMAD software [79] to optimize the hyperparameters of deep neural networks as formulated in (6.1). This package allows searching for both the architecture and the convolutional network's training regime for a specific dataset. It contains two main components : the blackbox and the optimizer, as illustrated in Figure 6.1 and presented in the following section.

6.3.1 The blackbox

This component regroups the Python and Pytorch [95] modules responsible for creating the equivalent deep neural network to the provided vector of hyperparameters. The

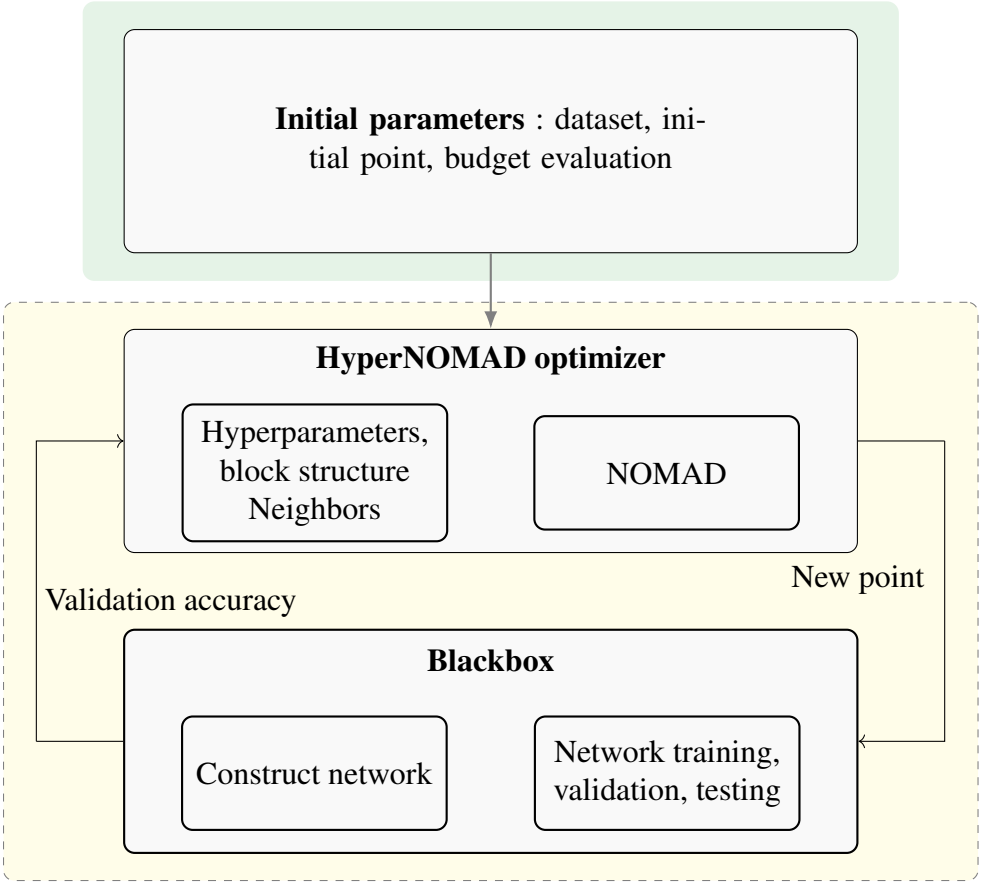


Figure 6.1 The HyperNOMAD workflow is represented by the communication of its two components. The optimizer suggests new candidates based on the NOMAD software and the blackbox trains the corresponding neural network to return its validation or test accuracy as the objective function value. Image adapted from [78].

blackbox module fully trains the resulting network before returning its validation accuracy or its validation loss as a performance measure. The package provides some computer vision datasets such as MNIST [81], Fashion-MNIST [123], CIFAR-10/100 [75] and STL10 [37] but also allows to plug-in a new dataset if needed.

6.3.2 The optimizer

The second module is first responsible for handling the collected categorical, integer, or real HPs. Categorical HPs include non-ordinal variables such as the activation function or the choice of the training optimizer and some ordinal variables such as the number of convolutional or fully connected layers. Then, the module launches the optimization

through the NOMAD software, which is the official implementation of the Mesh Adaptive Direct Search (MADS) [13] algorithm further discussed in Section 6.3.3.

In HyperNOMAD, each convolutional layer is defined by five HPs : the number of output channels, the kernel size, the stride, the padding, and the pooling size ; And each fully connected layer is determined by the number of nodes it contains. Hence, if n_1 indicates the number of convolutional layers and n_2 the number of fully connected layers, the network's architecture is determined by $5n_1 + n_2$ HPs. Considering the remaining HPs such as the optimizer, batch size or dropout rate, the dimension of the complete HPO problem becomes $5n_1 + n_2 + 10$. This dimension varies during the optimization if the values of n_1 and n_2 evolve, explaining why these HPs are considered categorical variables. A change in their values signals a change of search space, which HyperNOMAD handles through the extended polls in Algorithm 4 with a predefined neighborhood structure that states that each CNN has five neighbors. Four of these neighbors are found by adding/subtracting one convolutional/fully connected layer and the fifth neighbor keeps the same architecture and changes the optimizer for the training task.

6.3.3 The mesh adaptive direct search (MADS) algorithm

As previously stated, the optimizer in HyperNOMAD launches the MADS algorithm to explore the HPs search space. At each iteration k, MADS works on a mesh M_k defined by a mesh size Δ_k that is expanded or reduced depending on whether the previous iteration is successful or not. An iteration is successful if an improvement on the incumbent is recorded ; otherwise, it is a failure.

Each iteration is composed of two steps. First, the *search* step is an optional and flexible phase containing any exploration strategy as long as a finite number of trials projected on the mesh M_k are evaluated. Then the *poll* step starts by listing a set of poll points around the incumbent $P_k = \{x_k + \Delta_k d \mid d \in D_k\}$ where D_k is a set of search directions that form a positive basis. The search directions are generated so that the equivalent unit directions become asymptotically dense in the unit sphere. The points $p_k \in P_k$ are evaluated with an *opportunistic* strategy, which means the poll step is interrupted as soon as a configuration scores better than the incumbent x_k, even if other poll points are still available and hence will not be tested. Algorithm 4 summarizes the key steps of the MADS algorithm with an emphasis on where surrogate functions can be integrated.

In the general framework, surrogate functions are optionally plugged into the MADS

execution at different stages. For example, a static or dynamic surrogate can be used to explore a wide range of the space during the search phase to suggest new candidates. The poll step can also benefit from surrogate assistance in few distinct aspects, such as deciding on the evaluations to interrupt or providing a ranking so that MADS evaluates the most promising candidate first. Such rankings coupled with the opportunistic strategy are essential to target better solutions faster [107], especially when time and resource constraints are as severe as in the current context. The present work focuses mainly on improving the poll step by targeting both the resource allocation and the ranking aspects of poll points evaluation. The incorporated improvements are discussed in detail in Section 6.4.

Algorithm 4: MADS with static surrogates for ranking and early stopping.

[0] Initialization
iteration $k = 0$, configuration x_0, mesh size Δ_0 and mesh M_0,
ranking surrogate S_1 and early stopping surrogate S_2

[1] Search of iteration k (optional)
Construct a set of mesh points and evaluate them
If there is a success, go to **[3]**

[2] Poll of iteration k
Define the poll set P_k of new candidates around x_k,
along search directions that define a positive basis.
Sort the poll points with surrogate S_1
Evaluate the points in P_k with possible early stopping (S_2)
If there is a success, interrupt the evaluations and go to **[3]**

[3] Updates of iteration k
Update Δ_k, x_k, M_k depending on the success of the previous phases
If no stopping condition is satisfied : $k \leftarrow k + 1$ and go to **[1]**

6.4 Proposed approach

This section details the proposed enhancements to speed-up the resolution of the HPO problem with HyperNOMAD. This objective is achieved by introducing an early stopping mechanism that quickly interrupts the poor performing candidates, coupled with a ranking strategy that allows to evaluate the most promising poll candidates first. All tests in this section are on the MNIST dataset and start the HPO process from two configurations. First with the default initialization of HyperNOMAD, noted p_1, that corresponds to a network with one convolutional layer and two fully connected layers, which amounts

to 17 hyperparameters. The second initialization, noted p_2, adds a convolutional layer to the default point, which amounts to 22 hyperparameters.

6.4.1 Early stopping

Implementing an adequate early stopping strategy is paramount to the efficient execution of any hyperparameter optimization technique as the complete evaluation of each encountered candidate is merely unreasonable due to the number of wasted resources this direct approach can cause. The challenge is to detect when a network is worth training further and when it needs to be interrupted based on the observed validation curve and compared to other candidates previously trained.

The validation accuracy of a network should gradually improve during the training of a DNN until hitting a plateau or a maximum value after which the accuracy decreases. Such scenarios need to be detected to avoid depleting the remaining training budget on a configuration that can not improve its current validation score any further. However, a plateau does not necessarily mean that the best validation accuracy is reached but rather that the current training HPs, especially the learning rate, need to be updated. To this effect, the PyTorch library [95] provides multiple scheduler variants to manage the learning rate during the training. In particular, the `ReduceLROnPlateau` scheduler can detect plateaus and prompt the desired change in the learning rate. A common strategy is to reduce the learning rate progressively until it reaches a predefined minimum value, which triggers an early stopping. Figure 6.2a illustrates the benefit of such a mechanism as adding the scheduler allows saving half the training budget in that particular case.

Early stopping is also a decision based on the relative score of the current network compared to the candidates previously evaluated. Such comparison targets the networks whose validation scores are too far from the best solution seen thus far, even if that score gradually improves as the training advances. When training a new candidate, its validation accuracy curve is compared against the best scoring network, called the baseline, at specific epochs : $[5, 10, 25, 50, 100, 125, 150]$ with each milestone having an associated error margin : $[0.5, 0.6, 0.7, 0.8, 0.85, 0.9, 0.95]$. After 5 epochs, a new configuration needs to score at least 50% of the baseline score, 60% at 10 epochs, and so on. As such, the error margins define an envelope under the baseline curve, and any network that scores lower than the allowed error margin is interrupted at the next milestone epoch. This early stopping mechanism is illustrated in Figure 6.2b. If a better solution is found,

the corresponding network becomes the new baseline, and the envelope is redefined with its validation accuracy curve. Consequently, as the HPO advances, better baselines are found, and only the high scoring networks are allowed a high training budget and the low performances are quickly detected, and interrupted.

Early stopping effects are first tested on the MNIST dataset by comparing the original version of HyperNOMAD with a variant that implements one early stopping strategy. Every HPO execution starts from the same initial configuration and allows for 200 black-box evaluations (BBE), meaning that 200 different configurations are trained and tested. The comparison aims at observing the effect on the score of the best solution obtained and the quantity of resources needed to carry out the entire optimization task. Resource consumption is measured in terms of overall wall-clock time and total number of epochs used in training every visited configuration. In this phase, every execution is run on one Nvidia P100 GPU, and the batch size is fixed to 256. The tested early stopping strategies are :

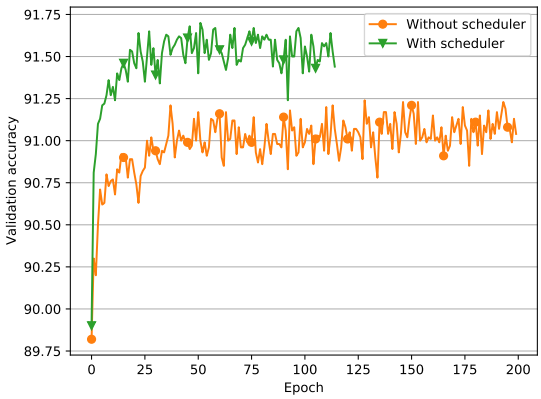
— Default settings of HyperNOMAD : stops if the validation accuracy does not surpass 12% after 25 epochs or when the standard deviation of the validation loss over the last 50 epochs is lower than 10^{-3}.

— Last success : stops the training when the last improvement on the validation accuracy is recorded more than 25 epochs ago.

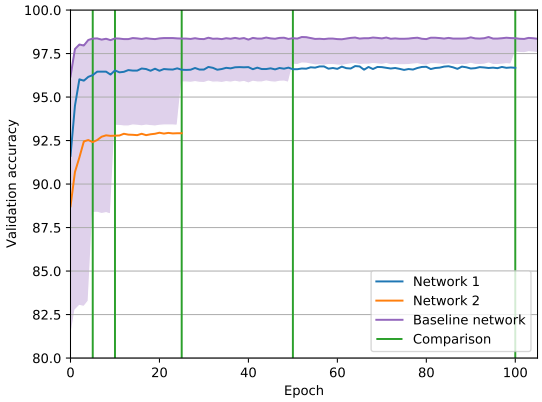
— Scheduler alone : corresponds to the `ReduceLROnPlateau` scheduler that divides by 10 the learning rate if the validation accuracy has not improve for the last 25 epochs until the learning rate becomes lower than 10^{-8}.

— Baseline and scheduler : takes the previous scheduler scheme and adds the relative scores comparison by defining a baseline and the corresponding envelop as previously detailed.

Figures 6.3 displays the number of epochs used in training each of the 200 configurations evaluated during each execution. The original version is the most resource-consuming, with a mean number of epochs per training at 170. All three added early stopping strategies manage to reduce this measure. The mean number of epochs per training drops to 101.4 with the scheduler, 72.5 with the last success criteria, and combining the scheduler with the baseline comparison brings it down to 45.9. Tables 6.1a and 6.1b corroborate this conclusion as the proposed approach is the quickest by a factor of 4 in the second example, plus the score of the best solution does not appear to significantly deteriorate compared to the original version of HyperNOMAD.



(a) Example of early stopping due to the scheduler.



(b) Example of early stopping due to the comparison with the baseline network.

Figure 6.2 Two early stopping criteria : a scheduler that detects a plateau and reduces the learning rate until it hits the lower limits (left) and the comparison with a baseline network that stops any evaluation outside the envelope defined by the relative errors compared to the baseline scores(right).

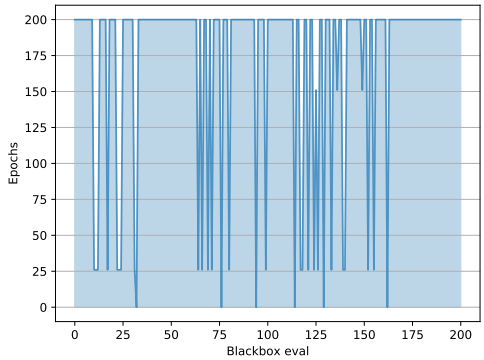
Table 6.1 Comparison between four early stopping strategies implemented in HyperNOMAD in terms of top-1 validation accuracy, overall wall clock time and total number of epochs. The variants are launched on MNIST from two starting configurations, are allowed a total of 200 configuration trials and each network can train during a maximum of 200 epochs.

(a) Scores of HyperNOMAD on MNIST with each stopping criteria starting from the default point p_1.

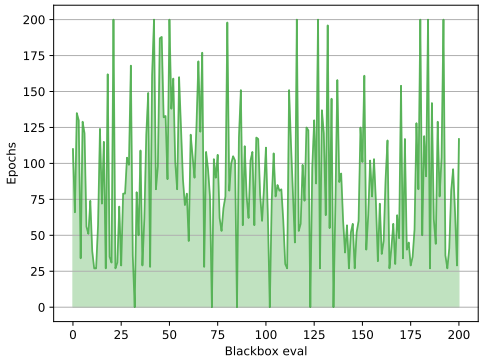
Early stopping strategy	Top-1 val. acc.	Wall-clock time (s)	Total epochs
HyperNOMAD	99.38%	305856	35503
Last success	99.40%	129600	17534
Scheduler	99.29%	170208	21987
Scheduler and baseline	**99.41%**	**126144**	**9681**

(b) Scores of HyperNOMAD on MNIST with each stopping criteria starting from the configuration p_2.

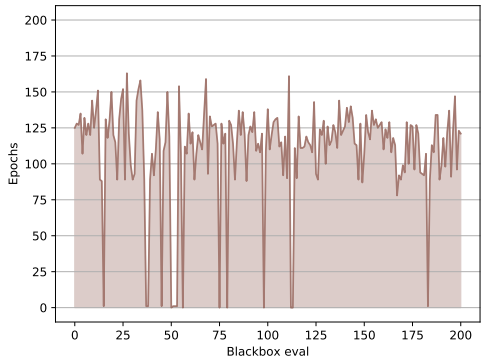
Early stopping strategy	Top-1 val. acc.	Wall-clock time (s)	Total epochs
HyperNOMAD	99.43%	280800	32712
Last success	**99.51%**	280800	11480
Scheduler	99.29%	170208	18580
Scheduler and baseline	99.44%	**64800**	**8655**



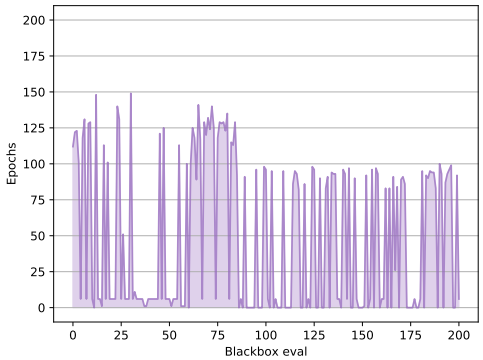
(a) HyperNOMAD : Stop if the standard deviation of the validation loss is lower than $1e^{-3}$.



(b) HyperNOMAD + last success : Stop if the last improvement on the validation score is recorded more than 25 epochs ago.



(c) HyperNOMAD + scheduler : Stop if a plateau is detected with the scheduler.



(d) HyperNOMAD + baseline + scheduler : Stop if a plateau is detected with the scheduler, or if the network scores poorly compared to the baseline.

Figure 6.3 Comparison between the resource consumption of HyperNOMAD and one variant that adds an early stopping strategy, in terms of number of epochs used to train each visited candidate during a single HPO. The HPO starts from the default point p_1 on the MNIST dataset.

6.4.2 Ranking

In addition to early stopping, ranking a new pool of candidates based on the expected performance can accelerate the overall HPO process. In HyperNOMAD, the evaluations are opportunistic, meaning that the poll step at iteration k stops as soon as a point $p_k \in P_k$ scores better than the incumbent x_k. Therefore, the opportunistic strategy prioritizes fast improvements and, when coupled with an adequate ranking tool, has the potential to explore the search space more efficiently and find the best configurations that much quicker.

Static surrogates, or low fidelity estimates, are approximating functions that do not change during the HPO execution, contrary to dynamic surrogates, which are updated to account for the collected information with each iteration. Creating a low fidelity surrogate to estimate the accuracy of a DNN can mean training on a fraction of the full epoch budget or on a subset of the dataset in this particular setting. The ideal static surrogate combines a cheap evaluation with a reliable estimation to correctly rank a pool of candidates. Note that in the context of HyperNOMAD and MADS, a good static surrogate does not need to produce accurate approximations of the true objective but rather preserves the ranking order between the candidates.

The effects of ranking the poll points are observed on the MNIST dataset by comparing four static surrogates added to HyperNOMAD. Two surrogates train the candidates on a low training budget, and two others train on a subset of the dataset. Recall that the proposed early stopping strategy results in a mean epoch budget per blackbox evaluation at around 45. Therefore, a static surrogate must use a lower training budget to qualify as a cheap approximation of the true objective evaluation. Similarly, only a small portion of the dataset can be used for the second set of surrogates. The tested surrogates are summarized in Table 6.2.

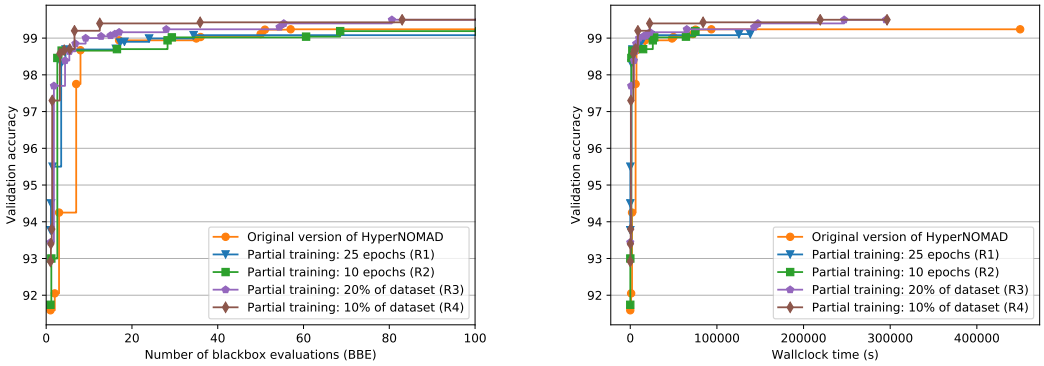
All variants start from the same configurations p_1 and p_2, are allowed 100 blackbox evaluations with a budget of 200 epochs each, and no early stopping criteria are considered at this stage. Evaluating a surrogate must also account in the number of blackbox evaluations as they add to the resource consumption of such HPO optimization. The cost of each surrogate in terms of blackbox evaluation is provided in Table 6.2. Figures 6.4 and 6.5 summarize the results regarding the best validation accuracy per number of blackbox evaluations and overall clock-time. Once again, all executions are run on a single Nvidia P100 GPU.

Table 6.2 List of the static surrogates tested for ranking the pool candidates. Their evaluation cost in terms of epochs and portion of datasets is compared to a full blackbox evaluation (BBE).

Function	Training budget (epochs)	Portion of dataset	Cost ratio to full BBE
Objective function	200	100%	100%
Surrogate R_1	25	100%	12.5%
Surrogate R_2	10	100%	5%
Surrogate R_3	200	20%	20%
Surrogate R_4	200	10%	10%

Figures 6.4a and 6.5a show the effect of adding ranking surrogate on the overall convergence of HyperNOMAD. In Figure 6.4a, all variants are relatively equivalent and surpass HyperNOMAD in the early stages of the execution. This tendency shifts however after around 10 blackbox evaluations when R_3 and R_4 score better quality solutions faster than the other variants. In fact, R_1 and R_2 have limited benefits to HyperNOMAD since the latter appears to have an equivalent performance in this case. This observation is more obvious in Figure 6.5a where adding R_1 or R_2 significantly slow down the execution of HyperNOMAD.

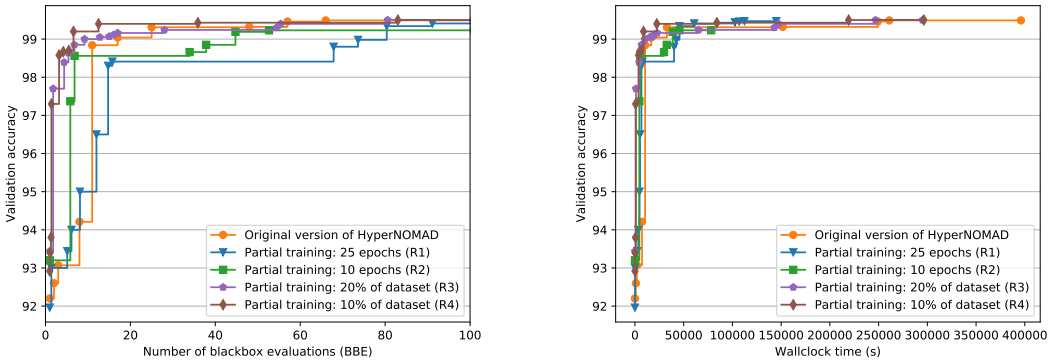
Figures 6.4b and 6.5b compare the convergence of each algorithm in term of overall wall-clock time. The benefit of adding a sorting surrogate is apparent in both executions as HyperNOMAD is the variant that takes the longest to evaluate the 100 blackbox trials. This is a coherent observation with the fact that every one of the 100 evaluations is equivalent to fully training a network on the entire dataset. Also, in both series of tests, surrogates R_1 and R_2 are the fastest and often terminate the 100 blackbox evaluation budget before all the other variants. It appears that training on such low epoch budgets, even on the full dataset, is faster than training on 10% of the dataset during 200 epochs. However, this observation is expected to change when early stopping is integrated again, which stops R_3 and R_4 from using the full 200 epochs in their evaluation. Plus, since R_1 and R_2 harm the best configuration score, both these strategies are discarded, and it is R_4 that is retained.



(a) Convergence of each variant in terms of validation accuracy per number of BBE.

(b) Convergence of each variant in terms of validation accuracy per overall execution time.

Figure 6.4 Comparison between the original HyperNOMAD [77, 78] and four variants that implement a strategy to sort the new candidates in order to evaluate the most promising first. Strategies R_1 and R_2 train on a small epoch budget while strategies R_3 and R_4 train on a subset of the data. The optimization is launched on the MNIST dataset from the default point p_1.



(a) Convergence of each variant in terms of validation accuracy per number of BBE.

(b) Convergence of each variant in terms of validation accuracy per overall execution time.

Figure 6.5 Comparison between the original HyperNOMAD [77, 78] and four variants that implement a strategy to sort the new candidates in order to evaluate the most promising first. Strategies R_1 and R_2 train on a small epoch budget while strategies R_3 and R_4 train on a subset of the data. The optimization is launched on the MNIST dataset from the default point p_2.

6.5 Testing

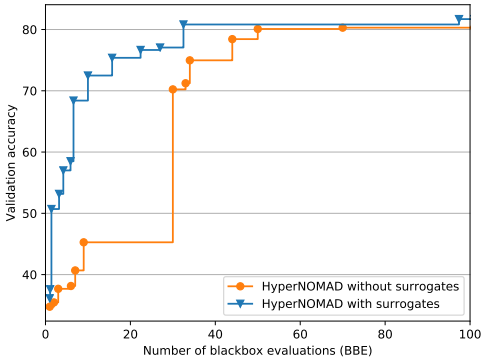
This section incorporates both aspects of Section 6.4 into the HyperNOMAD framework. Early stopping is triggered from a scheduler or by comparison with the baseline network ; and ranking a new pool of candidates is achieved through training each candidate on 10% of the dataset.

The tests are now launched on the CIFAR-10 dataset, are allowed 100 blackbox evaluations with a maximum training budget of 200 epochs per configuration. This time, each execution is launched on two Nvidia P100 GPUs. One series of tests starts from the default configuration of HyperNOMAD noted p_1, and the second starts from a network with a 5 convolutional layers and 1 fully connected layer, noted p_3, which is equivalent to 36 hyperparameters.

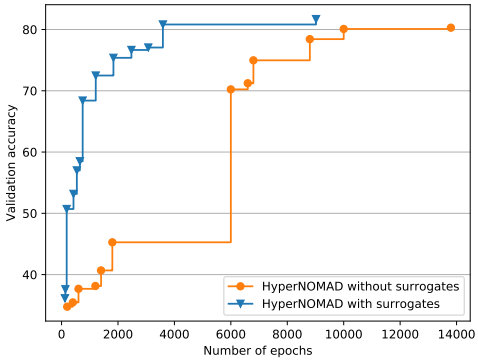
Figure 6.6 summarizes the results from the comparison between the original version of HyperNOMAD and the one with the proposed enhancements when launched from the default settings p_1. The benefits of early stopping and ranking the new candidates are apparent, with the new version scoring better than HyperNOMAD in terms of best validation score per blackbox evaluation budget as seen in Figure 6.6a. Figure 6.6b also shows that adding surrogates saves training resources in terms of total number of epochs to get to better quality solution, and Figure 6.6c corroborates this tendency when comparing the overall wall-clock time to deplete all 100 blackbox evaluations. The second test starts from configuration p_3. Figure 6.7a shows that the version with the surrogates is slightly ahead when comparing the best validation score per blackbox evaluation budget. Figure 6.7b illustrates a gain of 27% in total epoch budget, yet this gain is not translated in terms of overall wall-clock time as shown in Figure 6.7c. The difference is believed to be caused by the communication between the two GPUs used at once.

6.6 Discussion

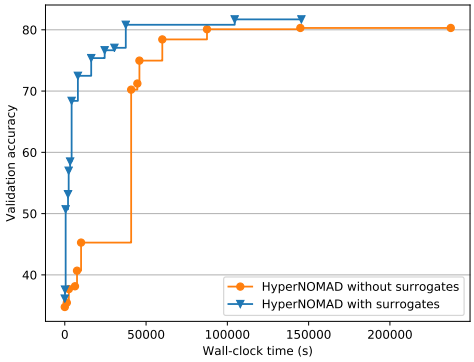
This work proposes a solution to speed up the time and resource-consuming process of optimizing the hyperparameters of deep neural networks, based on incorporating two strategies based on static surrogates into the HyperNOMAD framework. The first defines a new early stopping strategy that quickly and effectively interrupts poorly performing networks. The second allows ranking a pool of new candidates to detect and then evaluate the most promising first. Both techniques are shown individually and collecti-



(a) Convergence in terms of validation accuracy per number of BBE.



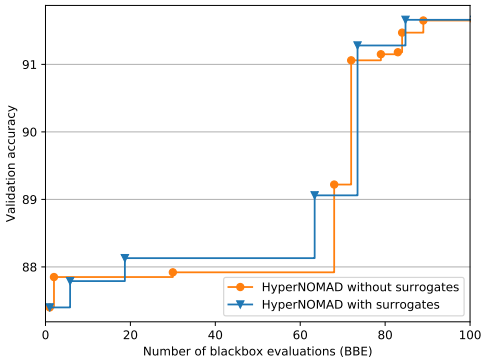
(b) Convergence in terms of validation accuracy per number of epochs.



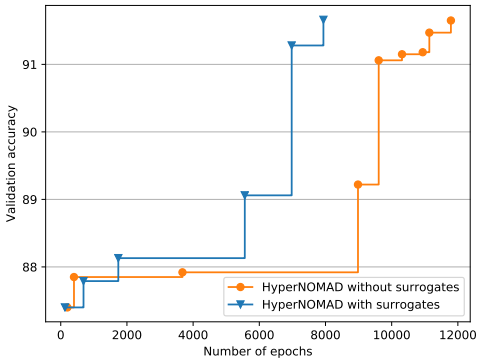
(c) Convergence in terms of validation accuracy per execution time.

Figure 6.6 Comparison between HyperNOMAD with and without surrogates on the CIFAR-10 dataset, starting from the default settings p_1.

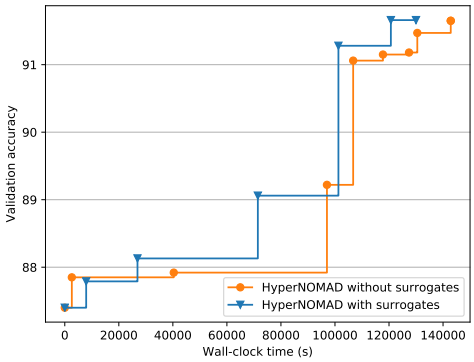
vely to improve on the quality of solution and on the amount of computational resources needed.



(a) Convergence in terms of validation accuracy per number of BBE.



(b) Convergence in terms of validation accuracy per number of epochs.



(c) Convergence in terms of validation accuracy per execution time.

Figure 6.7 Comparison between HyperNOMAD with and without surrogates on the CIFAR-10 dataset, starting from the default settings p_3.

CHAPITRE 7 DISCUSSION GÉNÉRALE

Les travaux présentés dans cette thèse ont mené au développement de nouveaux outils pour la résolution du problème d'optimisation des hyperparamètres de réseaux de neurones profonds en se basant sur l'algorithme MADS, une méthode DFO qui combine une recherche directe à la possible utilisation de modèles.

7.1 Synthèse des travaux

Les contributions de cette thèse sont multiples. Tout d'abord, la librairie HyperNOMAD a été proposée pour créer et entraîner des réseaux de convolutions capables d'effectuer des tâches de classifications d'images. HyperNOMAD apporte une modélisation de boîte noire reliée à un espace de recherche d'architectures où les opérations de chaque couche sont définies dans les détails. L'optimisation des hyperparamètres est menée par une adaptation de l'algorithme MADS qui exploite les variables de catégories pour définir l'étape d'*Extended poll*. Cette dernière permet de basculer entre des espaces d'architectures en ajoutant ou soustrayant une couche à la fois, ou entre des espaces d'optimisation d'entraînement lorsque HyperNOMAD décide d'employer un nouvel optimiseur. Les tests effectués dans le chapitre 4 ont mis en valeur la pertinence de l'approche proposée puisque HyperNOMAD arrive non seulement à trouver de meilleures configurations que la méthode Bayésienne TPE ou que la recherche aléatoire, mais il est aussi capable de gérer des problèmes d'optimisation de plus grande taille surtout lorsque cette dernière est variable. Notons que tous les tests présentés n'utilisent aucun pré-entraînement ni transfert de poids pour simuler le cas le plus générique où le but est de trouver le meilleur réseau possible, sans connaissances préalables et avec un budget de recherche limité. C'est ce qui explique en partie que les scores rapportés soient inférieurs à ceux de l'état de l'art sur CIFAR-10.

La seconde contribution décrite au chapitre 5 est à la fois algorithmique et pratique. On y expose la variante Δ-MADS qui combine l'étape de sonde de MADS à l'étape de recherche de l'algorithme Δ-DOGS. À chaque itération, le vecteur des hyperparamètres est séparé en variables réelles et variables entières ou de catégories. L'étape de recherche qui fait appel à Δ-DOGS n'optimise que les variables réelles puisqu'elles sont mieux adaptées au modèle d'interpolation qui y est employé, tandis que la sonde prend

en compte l'ensemble complet des hyperparamètres. Cette variante est testée sur un problème d'ingénierie où le but est de trouver un réseau de neurones auto-encodeur capable de détecter avec précision les transmissions incomplètes reçues depuis les astromobiles sur la planète Mars. Les résultats rapportés sont supérieurs à l'approche précédemment employée par le *Mars Science Laboratory, ground data system* puisqu'on passe d'un taux de rappel de 55% à 87% au niveau de l'identification des transmissions incomplètes.

Le chapitre 6 apporte une amélioration au niveau des ressources employées par HyperNOMAD lors de son exécution. La motivation de cette recherche est basée sur les travaux menés au chapitre 4 où on a pu observer la quantité de ressources de calculs et le temps, parfois allant jusqu'à 2 semaines, nécessaires à une seule tâche d'optimisation. L'analyse des exécutions précédentes a mis en évidence le fait que toutes les configurations ne se valent pas et ne doivent donc pas bénéficier du même budget d'évaluation, c'est à dire d'entraînement. Aussi, deux stratégies basées sur des modèles ont été intégrées à HyperNOMAD, la première vise à détecter les configurations peu intéressantes pour arrêter prématurément leurs entraînements ; la seconde cherche à exploiter la stratégie d'évaluation opportuniste en proposant un classement des candidats pour que les plus prometteurs soient placés en tête de liste. La combinaison de ces deux substituts a permis un temps d'exécution parfois jusqu'à 4 fois plus rapide et une complexité de calcul, mesurée en nombre d'*epochs*, jusqu'à 3 fois moindre.

7.2 Limitations de la solution proposée

La librairie proposée dans les chapitres 4 et 6 considère un espace de recherche d'architectures bien spécifique qui produit des structures séquentielles, avec des connexions simples où chaque couche est uniquement reliée à la suivante. Le nombre de degrés de libertés autorisés limite rapidement la profondeur des réseaux que l'on peut produire comme le montre l'exemple du réseau VGG à 13 couches qui est modélisé par 75 HPs. Par ailleurs, lorsqu'on examine les meilleures architectures proposées ces dernières années, on observe des structures formées de blocs souvent spécifiquement pensés pour réduire la complexité de l'entraînement, tels que les blocs *Bottleneck* ou *Inverted residual* [106], avec des connexions diverses où une information peut sauter quelques couches, se subdiviser pour subir des transformations parallèles dont les résultantes sont ensuite combinées [114]. Ce sont autant de variantes architecturales que l'espace de recherche actuel ne permet pas de former, du moins pas avec un coût raisonnable.

De même, l'espace de recherche de la stratégie d'entraînement peut être étendu pour inclure d'autres algorithmes d'optimisation, et pour y ajouter les stratégies de mise à jour du taux d'apprentissage durant un même entraînement. En effet, le taux d'apprentissage qui correspond à la longueur de pas à suivre le long d'une direction de «descente» doit être réduit plus on s'approche d'un minimum local, ou, au contraire, doit être agrandi pour échapper à une convergence prématurée. Là encore, ces stratégies ne sont pas génériques et doivent être spécifiquement adaptées au réseau de neurones en question et à la base de données sur laquelle il s'entraîne.

Au delà des aspects structurels et d'apprentissages, une recherche de réseau de neurones inclut également un examen des données utilisées auxquelles on doit souvent appliquer certaines transformations, surtout lorsque la base de données est petite, pour favoriser un entraînement robuste et des performances généralisables. Le traitement des données joue un rôle très important qui peut être également formulé comme un problème d'optimisation d'hyperparamètres où on cherche à définir les transformations à appliquer aux données soit avant le début de l'entraînement, soit au fur à mesure avec des stratégies variables à chaque *epoch* [63, 109].

Par ailleurs, le chapitre 5 met en lumière l'intérêt d'inclure des substituts dans l'étape de recherche de MADS, même s'ils n'optimisent qu'un sous problème en considérant uniquement les variables continues. Cette approche est prometteuse et mérite d'être développée comme partie intégrante de la librairie HyperNOMAD.

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS

Cette thèse apporte une adaptation de l'algorithme MADS pour optimiser les HPs des réseaux de neurones profonds. À notre connaissance, il s'agit ici de la première application d'une méthode adaptive de recherche directe pour cette problématique. Les trois articles exposés répondent chacun à une facette du vaste problème énoncé. La première contribution est donnée sous la forme de la librairie HyperNOMAD, qui est spécialement conçue pour l'optimisation des HPs des réseaux de convolutions. La variante Δ-MADS, développée au cours du second article, confirme l'efficacité des méthodes DFO sur une application réelle. Quant au troisième article, il propose une amélioration de HyperNOMAD basée sur des substituts statiques, afin de réduire sa consommation de ressources, sans pour autant nuire à son efficacité.

Chaque adaptation de MADS s'est avérée concluante au fil des études menées, compétitive vis à vis des méthodologies courantes telles que l'expertise humaine, la recherche aléatoire ou les méthodes bayésiennes, et surtout prometteuse quant aux développements potentiels. Voici, par ailleurs, quelques pistes d'améliorations pour palier aux limitations évoquées au chapitre précédent :

— L'étape de recherche de MADS est un des points forts de cet algorithme. Le chapitre 5 a particulièrement mis en évidence l'intérêt d'ajouter une stratégie de recherche globale pour une exploration efficace de l'espace de recherche et une détection rapide des zones où se trouvent les meilleures configurations. La librairie HyperNOMAD doit donc être agrémentée de stratégies de recherches variées telles que les modèles d'interpolations ou d'autres méthodes d'échantillonnages spécialement pensées pour le problème en question.

— Afin de réduire la dimension du problème HPO et de produire des architectures plus sophistiquées, il est nécessaire d'inclure un second espace de recherche pour les architectures où celles-ci sont définies par blocs. Pour ce faire, on peut commencer par développer une banque de blocs, des stratégies de concaténation et potentiellement adapter HyperNOMAD pour y ajouter une structure de voisinage compatible avec les architectures par blocs.

— La boîte noire de HyperNOMAD peut être développée pour modéliser d'autres

types de réseaux outre les réseaux de convolutions ou les auto-encodeurs, tels que les réseaux LSTM ou les transformeurs qui se spécialisent dans le traitement du langage naturel, ou la gestion des séries temporelles.

— Dans la pratique, les propriétés de la base de données sont tout aussi importantes que l'architecture ou l'entraînement du réseau de neurones. Idéalement, une base de données se doit d'être suffisamment vaste et diversifiée pour qu'un réseau puisse obtenir des prédictions précises et robustes. Aussi, existe-t-il plusieurs stratégies et politiques d'augmentation et de transformations des données que l'on peut appliquer avant ou pendant l'entraînement. Le choix de telles stratégies peut donc être vu comme une HPO supplémentaire que l'on se doit d'intégrer au fonctionnement de HyperNOMAD.

— L'interface actuelle de HyperNOMAD peut être repensée pour la rendre plus conviviale et facile à installer. Puisque le langage Python est assez répandu dans le domaine, on peut proposer un module HyperNOMAD que l'utilisateur pourra installer avec une commande pip, qui se chargera des compilations nécessaires et de la définition des variables d'environnement. L'utilisateur doit être capable de définir son problème d'optimisation, de lancer l'exécution de HyperNOMAD à partir du même script Python, de suivre l'optimisation en recevant des informations continues sur le déroulement de l'opération, puis d'obtenir un rapport final contenant des graphes de convergences, des statistiques sur les configurations visitées ou encore sur l'efficacité des options qu'il aurait sélectionnées.

Bibliographie

[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] Mark A Abramson, Charles Audet, and John E Dennis. Generalized pattern searches with derivative information. *Mathematical Programming*, 100(1) :3–25, 2004.

[5] S. Agrawal and J. Agrawal. Survey on anomaly detection using data mining techniques. *Procedia Computer Science*, 60 :708–713, 2015.

[6] R. Alimo, P. Beyhaghi, and T. Bewley. Delaunay-based derivative-free optimization via global surrogates. part iii : nonconvex constraints. *Journal of Global Optimization*, pages 1–34, 2020.

[7] R. Alimo, D. Cavaglieri, P. Beyhaghi, and T. Bewley. Design of imexrk time integration schemes via delaunay-based derivative-free optimization with nonconvex constraints and grid-based acceleration. *Journal of Global Optimization*, pages 1–25, 2020.

[8] J. An and S. Cho. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2(1), 2015.

[9] A. Asperti. Sparsity in variational autoencoders. *arXiv preprint arXiv :1812.07238*, 2018.

[10] 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.

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

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

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

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

[15] C. Audet and J. E. Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1) :188–217, 2006.

[16] 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.

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

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

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

[20] B. Bowen Baker, G. Otkrist, R. Ramesh, and N. Nikhil. Accelerating neural architecture search using performance prediction. In *NIPS Workshop on Meta-Learning*, 2017.

[21] A. Balakumar and S. Senthil. Machine learning is the future for lung cancer prognosis and prediction. In *Applications of Deep Learning and Big IoT on Personalized Healthcare Services*, pages 176–196. IGI Global, 2020.

[22] 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, Bengaluru, India, 2018. IEEE.

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

[24] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546––2554, Red Hook, NY, USA, 2011. Curran Associates Inc.

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

[26] 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, Atlanta, GA, USA, 2013. JMLR.org.

[27] P. Beyhaghi and T. Bewley. Implementation of cartesian grids to accelerate delaunay-based derivative-free optimization. *Journal of Global Optimization*, 69(4) :927–949, 2017.

[28] P. Beyhaghi, D. Cavaglieri, and T. Bewley. Delaunay-based derivative-free optimization via global surrogates, part i : linear constraints. *Journal of Global Optimization*, 66(3) :331–382, 2016.

[29] Erik Bochinski, Tobias Senst, and Thomas Sikora. Hyper-parameter optimization for convolutional neural network committees based on evolutionary algorithms. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3924–3928. IEEE, 2017.

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

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

[32] X.r 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. Oríon - Asynchronous Distributed Hyperparameter Optimization. `https://github.com/Epistimio/orion`, October 2019. (last accessed on 2020-09-19).

[33] L. Breiman. Random forests. *Machine learning*, 45(1) :5–32, 2001.

[34] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[35] F. Carcillo, Y. A. Le Borgne, O. Caelen, Y. Kessaci, F. Oblé, and G. Bontempi. Combining unsupervised and supervised learning in credit card fraud detection. *Information sciences*, 2019.

[36] P. Chrabaszcz, I. Loshchilov, and F. Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv :1707.08819*, 2017.

[37] A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 215–223, 2011.

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

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

[40] L. Deng and Y. Liu. *Deep learning in natural language processing*. Springer, 2018.

[41] 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.

[42] 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.

[43] Gonzalo I Diaz, Achille Fokoue-Nkoutche, Giacomo Nannicini, and Horst Samulowitz. An effective algorithm for hyperparameter optimization of neural networks. *IBM Journal of Research and Development*, 61(4/5) :9–1, 2017.

[44] P. K. Diederik and M. Welling. Auto-encoding variational bayes, 2013.

[45] G. Dlamini, R. Galieva, and M. Fahim. A lightweight deep autoencoder-based approach for unsupervised anomaly detection. In *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–5. IEEE, 2019.

[46] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith. Fine-tuning pretrained language models : Weight initializations, data orders, and early stopping. *arXiv preprint arXiv :2002.06305*, 2020.

[47] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[48] 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.

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

[50] T. Elsken, J. H. Metzen, and F. Hutter. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. Technical report, International Conference on Learning Representations, New Orleans, USA, 2019.

[51] H. S. Emadi and S. M. Mazinani. A novel anomaly detection algorithm using dbscan and svm in wireless sensor networks. *Wireless Personal Communications*, 98(2) :2025–2035, 2018.

[52] M. Faria, P. Prado, R. M. S. Julia, and T. Lídia Bononi Paiva. Improving fifa player agents decision-making architectures based on convolutional neural networks through evolutionary techniques. In Ricardo Cerri and Ronaldo C. Prati, editors, *Intelligent Systems*, pages 371–386, Cham, 2020. Springer International Publishing.

[53] E. Fermi and N. Metropolis. Numerical solution of a minimum problem. Los Alamos Unclassified Report LA–1492, Los Alamos National Laboratory, Los Alamos, USA, 1952.

[54] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[55] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., Montreal, Canada, 2015.

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

[57] Hiva Ghanbari and Katya Scheinberg. Black-box optimization in machine learning with trust region based derivative free algorithm. *arXiv preprint arXiv :1703.06925*, 2017.

[58] 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, New York, NY, USA, 2017. ACM, Association for Computing Machinery.

[59] Genetha A Gray and Tamara G Kolda. Algorithm 856 : Appspack 4.0 : Asynchronous parallel pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software (TOMS)*, 32(3) :485–507, 2006.

[60] J. h. Ryu and S. Kim. A derivative free trust region method for biobjective optimization. *SIAM Journal on Optimization*, 24(1) :334–362, 2014.

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

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[63] Daniel Ho, Eric Liang, Xi Chen, Ion Stoica, and Pieter Abbeel. Population based augmentation : Efficient learning of augmentation policy schedules. In *International Conference on Machine Learning*, pages 2731–2741. PMLR, 2019.

[64] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8) :1735–1780, 1997.

[65] 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.

[66] 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, Berlin, Heidelberg, 2011. Springer, Springer Berlin Heidelberg.

[67] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning : methods, systems, challenges*. Springer Nature, 2019.

[68] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet : Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv :1404.1869*, 2014.

[69] 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, New York, NY, USA, 2014. ACM, Association for Computing Machinery.

[70] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu. A survey of deep learning-based object detection. *IEEE Access*, 7 :128837–128868, 2019.

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

[72] A. Klein, S. Falkner, S. Bartels, P. Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets, 2017.

[73] A. Klein, S. Falkner, T.S. Jost, and F. Hutter. Learning curve prediction with bayesian neural networks. In *International Conference on Learning Representations*, 2017.

[74] 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.

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

[76] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[77] D. Lakhmiri. HyperNOMAD. `https://github.com/bbopt/HyperNOMAD`, 2019.

[78] D. Lakhmiri, S. Le Digabel, and C. Tribes. HyperNOMAD : Hyperparameter optimization of deep neural networks using mesh adaptive direct search. Technical Report G-2019-46, Les cahiers du GERAD, 2019.

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

[80] S. Le Digabel and S.M. Wild. A Taxonomy of Constraints in Simulation-Based Optimization. Technical Report G-2015-57, Les cahiers du GERAD, 2015.

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

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

[83] 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.

[84] L. Li, R. J. Hansman, R. Palacios, and R. Welsch. Anomaly detection via a gaussian mixture model for flight operation and safety monitoring. *Transportation Research Part C : Emerging Technologies*, 64 :45–57, 2016.

[85] 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.

[86] 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.

[87] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018.

[88] J. Liu, N. Ploskas, and N.V. Sahinidis. Tuning BARON using derivative-free optimization algorithms. *Journal of Global Optimization*, 74(4) :611–637, 2018.

[89] 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*, page 481–488, New York, NY, USA, 2017. ACM, Association for Computing Machinery.

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

[91] G. Marques, D. Agarwal, and I. de la Torre Díez. Automated medical diagnosis of covid-19 through efficientnet convolutional neural network. *Applied Software Computing*, 96 :106691, 2020.

[92] 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.

[93] Microsoft. Neural network intelligence. `https://github.com/microsoft/nni`, 2019.

[94] J.A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4) :308–313, 1965.

[95] 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. PyTorch : An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., NY, USA, 2019.

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

[97] 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.

[98] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient Neural Architecture Search via Parameter Sharing, 2018.

[99] 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.

[100] 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.

[101] L. Prechelt. *Early Stopping — But When ?*, pages 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[102] O. I. Provotar, Y. M. Linder, and M. M. Veres. Unsupervised anomaly detection in time series using lstm-based autoencoders. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, pages 513–517. IEEE, 2019.

[103] N. Quadrianto, K. Kersting, and Z. Xu. *Gaussian Process*, pages 428–439. Springer US, Boston, MA, 2010.

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

[105] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[106] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2 : Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[107] L.A. Sarrazin-Mc Cann. *Opportunisme et ordonnancement en optimisation sans dérivées*. Master's thesis, Polytechnique Montréal, `https://publications.polymtl.ca/3099/`, 2018.

[108] M. Schreyer, T. Sattarov, D. Borth, A. Dengel, and B. Reimer. Detection of anomalies in large scale accounting data using deep autoencoder networks. *arXiv preprint arXiv :1709.05254*, 2017.

[109] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1) :1–48, 2019.

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

[111] 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)*, pages 1 – 8, New York, NY, USA, 2016. IEEE, Association for Computing Machinery.

[112] 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, Red Hook, NY, USA, 2012. Curran Associates Inc.

[113] 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, Melbourne, Australia, 2017. ACM, International Joint Conferences on Artificial Intelligence Organization.

[114] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[115] Mingxing Tan and Quoc Le. Efficientnet : Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[116] 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.

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

[118] I. Trofimov, N. Klyuchnikov, M. Salnikov, A. Filippov, and E. Burnaev. Multi-fidelity neural architecture search with knowledge distillation. *arXiv preprint arXiv :2006.08341*, 2020.

[119] A. M. Vartouni, S. S. Kashi, and M. Teshnehlab. An anomaly detection method to detect web attacks using stacked auto-encoder. In *2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)*, pages 131–134. IEEE, 2018.

[120] Emmanuel Vazquez and Julien Bect. Convergence properties of the expected improvement algorithm with fixed mean and covariance functions. *Journal of Statistical Planning and inference*, 140(11) :3088–3095, 2010.

[121] C. K. Williams and C. E. Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

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

[123] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST : a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017.

[124] W. Yan and L. Yu. On accurate and reliable anomaly detection for gas turbine combustors : A deep learning approach. *arXiv preprint arXiv :1908.09238*, 2019.

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

[126] 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*, pages 1–5, New York, NY, USA, 2015. ACM, Association for Computing Machinery.

[127] A. Zela, A. Klein, S. Falkner, and F. Hutter. Towards automated deep learning : Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv :1807.06906*, 2018.

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

[129] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018.

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

[131] 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, Salt Lake City, Utah, USA, 2018. IEEE.

[132] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc 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.

ANNEXE A HYPERNOMAD : USER GUIDE

A.1 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 [79] used for the optimization. The blackbox takes as inputs the hyperparameters discussed in Section 4.3.1, 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, 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.

```
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 4.4, 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 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 A.1 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) 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 A.1 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
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
                             # 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 A.1 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 4.9. 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 A.1. 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.

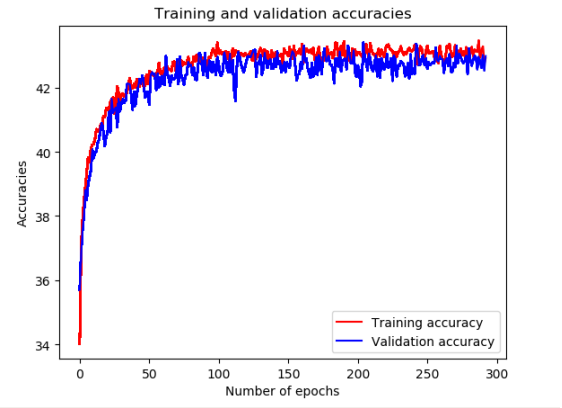


Figure A.1 Example of a window that appears during one evaluation of the blackbox in HyperNOMAD. This figure shows in real time the training and validation accuracies of the current evaluated set of hyperparameters, at each epoch.

Tableau A.1 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 4.4 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 4.3	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}