Hyperparameter Optimization for Large-scale Machine Learning

Thesis · October 2018

DOI: 10.13140/RG.2.2.33876.65927

1 author:

Aitor Palacios Cuesta
Technische Universität Berlin
1 PUBLICATION **1** CITATION

Technical University of Berlin

Faculty of Electrical Engineering and Computer Science
Database Systems and Information Management Group

Master Thesis

Hyperparameter Optimization for Large-scale Machine Learning

Aitor Palacios Cuesta

Matriculation number: 396276

October 2018

Supervised by
Prof. Dr. Volker Markl

Advised by
Behrouz Derakhshan

Abstract

Hyperparameter optimization is a crucial task affecting the final performance of machine learning solutions. This thesis analyzes the properties of different hyperparameter optimization algorithms for machine learning problems with large datasets. In hyperparameter optimization, we repeatedly propose a hyperparameter configuration, train a machine learning model using that configuration and validate the performance of the model. In order to scale in these scenarios, this thesis focuses on the data-parallel approach to evaluate hyperparameter configurations. We utilize Apache Spark, a data-parallel processing system, to implement a selection of hyperparameter optimization algorithms.

Moreover, this thesis proposes a novel hyperparameter optimization method, Bayesian Geometric Halving (BGH). It is designed to benefit from the characteristics of data-parallel systems and the properties of existing hyperparameter optimization algorithms. BGH enhances Bayesian methods by combining them with adaptive evaluation. At the same time, it has a time-efficient execution independently of the number of configurations evaluated.

Various experiments compare the performance of several hyperparameter optimization algorithms. These experiments extend their theoretical comparison in order to understand under which conditions some algorithms perform better than the others. The experiments show that BGH improves the results of both Bayesian and adaptive evaluation methods. Indeed, one of the variants of BGH achieves the best result in every experiment where they are compared.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschlielich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, 19.10.2018

.....

Aitor Palacios Cuesta

Contents

List of Figures

Abbreviations

ALS	**A**lternating **L**east **S**quares
BGH	**B**ayesian **G**eometric **H**alving
GP	**G**aussian **P**rocess
MCMC	**M**arkov **C**hain **M**onte **C**arlo
SH	**S**uccessive **H**alving
SMAC	**S**equential **M**odel-based **A**lgorithm **C**onfiguration
SVM	**S**upport **V**ector **M**achines
TPE	**T**ree-structured **P**arzen **E**stimaton

Chapter 1

Introduction

In machine learning, the objective is to learn a function which maps an input of possibly multiple dimensions (known as features) to a certain output (whether a real number, a class or a structure such as a tree). Given a model, this function is automatically learned based on training data (data for which we know both the input and the desired output), in contrast to traditional programming. However, previous to that learning process, there are several choices that have to be made when solving a given problem using machine learning techniques. The most obvious one may be the choice of the model or learning algorithm to use. Another one is how to preprocess (transform) the available training data and features in order to obtain an optimal result. Finally, even though a machine learning model automatically learns a function by estimating a set of parameters given the training data, the learning process is governed from the beginning by a set of configurable options known as hyperparameters. In this sense, we can consider a certain algorithm (e.g. logistic regression, a popular algorithm to perform classification) as a set of models, and the combination of an algorithm and a specific set of hyperparameters as an actual model instance. The goal of any machine learning pipeline (the concatenation of every preprocessing and model transformations applied to the input data) is to optimize a given metric on certain data of interest for the problem at hand. All these choices can significantly affect the performance of the pipeline, therefore establishing a good configuration is crucial for finding good results.

In practice, a common approach is to manually explore different hyperparameter configurations until a satisfactory result is reached or the available budget is exhausted. Nevertheless, this is approach is not feasible or desirable for the majority of applications. Typically, the search space of these configurations is very large. There are many

preprocessing methods, learning algorithms, and hyperparameters. For each hyperparameter, there can be infinitely many possible values (this is the case for any continuous hyperparameter). Therefore, it seems sensible to explore different configurations in some other way than manually choosing one configuration at a time and checking its result, using certain hints and intuition to sequentially find a configuration which is (close to) the optimal one. Most commonly, the majority of the time spent in this search is spent in exploring values for the hyperparameters of a given learning algorithm, that is, in configuring the last stage of the machine learning pipeline. It is probably the part which more subtly affects the final performance and where it is most difficult to find an optimal configuration in practice. To be consistent with the vocabulary usually employed in the literature, this thesis will refer to the general configuration problem as hyperparameter optimization or tuning, although it is not exclusively restricted to that final part of the pipeline and can be easily extended to the other ones as well. Similarly, this work will refer to "hyperparameters" and to "configurations" interchangeably.

It would make sense to take advantage of computational resources and apply brute force in order to search for the most optimal configuration automatically if the feedback of the performance for a configuration were immediately given. This can be done by covering the search space with a grid of values to try out and finding the best performing one, a technique known as grid search. However, in some cases, training a single machine learning model with one hyperparameter configuration may require several days or weeks [4]. Given the large configuration space as indicated above, an exhaustive search over this space is unfeasible. Moreover, in general, the time spent training an algorithm directly depends on the size of the training data. Nowadays, with the large quantity of data already available and its constant growth, it is becoming more frequent that models are trained on very large datasets, which further increases the training time. Therefore, it is crucial to use techniques which find a configuration with the best possible result in the shortest possible time, making an efficient use of the training process. During the last decade, several techniques have been explored and developed for this purpose, commonly referred to as hyperparameter optimization techniques (though they are also applicable to the configuration search of other parts of the pipeline, not only the learning algorithm). This thesis studies these techniques.

As mentioned above, the data available to solve machine learning problems has constantly been growing in the recent years, and will continue to do so in the following ones. If the data collected to train a learning algorithm is appropriate and representative, larger training data leads to more accurate results (because the information available to make inference is richer). Hence, machine learning practitioners want to

use as much data as possible to train their models. New data-parallel systems (such as Apache Spark [5] or Apache Flink [6]) have been developed to be able to process all the available data in a fast and efficient way by making use of computer clusters, an easily accessible resource nowadays. These systems include implementations of machine learning algorithms and pipelines to train machine learning models and perform predictions with large datasets [7].

It seems obvious that the efficiency in configuration search for those pipelines turns to be even more important with such large datasets. Moreover, as these systems use more computational power in efficient ways, they can use more resources to explore more configurations faster than traditional single computer systems, resulting in a more advantageous scenario for automatic search. However, these systems do not currently include automatic techniques to search for these configurations, apart from grid search [8]. This thesis will focus on analyzing the quality and suitability of different hyperparameter optimization techniques for the large-scale regime and implementing the most promising ones in one of these platforms.

1.1 Problem statement

This thesis studies different hyperparameter optimization techniques with a focus on their suitability for training processes with large datasets. For that, we will explore how they can be applied in a data-parallel system to find better hyperparameters as fast as possible. More concretely, the goals of this study are:

- An analysis of state-of-the-art hyperparameter optimization methods, comparing them and evaluating their characteristics and suitability for large-scale machine learning and data-parallel systems.
- Devising a novel approach in which hyperparameter optimization methods can be modified or combined to increase their efficiency for large datasets.
- An implementation of the most promising methods in one of the most popular data-parallel systems, Apache Spark [5].
- Experimentation and evaluation of results comparing the performance of different methods, including the novel strategy proposed in this thesis, with regards to finding best quality models in the shortest period of time.

1.2 Outline of the thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the hyperparameter optimization problem in depth, including background about the most relevant hyperparameter optimization methods described in the literature. Chapter 3 describes the characteristics of data-parallel systems, with a focus on Apache Spark and its machine learning library. Chapter 4 introduces Bayesian Geometric Halving, a novel hyperparameter optimization method designed for the large-scale scenario. Chapter 5 describes how we implement the most promising hyperparameter optimization algorithms in Apache Spark. In Chapter 6, we compare these algorithms on a range of problems with varying datasets, hyperparameters and algorithms. Finally, Chapter 7 gives final remarks and suggests future work based on this study.

Chapter 2

Hyperparameter optimization

Machine learning algorithms try to find a function which minimizes a certain loss on the outputs of some input-output pairs (usually represented as a validation dataset). These algorithms learn such a function by using another set of input-output pairs, a train dataset, to adjust a set of parameters so a certain loss of the algorithm's predictions on that training dataset is minimized. These parameters are adjusted by using some optimization procedure which takes advantage of the formulation of the model with respect to the parameters (e.g. calculating their gradient with respect to the training examples). However, the behaviour of this learning procedure is controlled by some parameters defined prior to the learning phase, known as hyperparameters. We can illustrate this with the example of linear regression. It is a regression model with the following formulation:

$$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + ...\beta_n * x_n \tag{2.1}$$

Where y is the value to be predicted, x_t corresponds to feature t, and β_t corresponds to coefficient t. These coefficients are the inner parameters of the linear regression model. They are adjusted during the learning phase as to minimize the differences between the model's predicted y and the ones from the training data given the corresponding features. However, there are several choices to be made before starting this learning phase. These coefficients can be optimized using different iterative algorithms, such as gradient descent or l-bfgs. The maximum number of iterations and the convergence tolerance of these optimization algorithms need also to be set. Moreover, regularization can be used during optimization in order to generalize well on unseen data (as we will discuss in the next section). How strong the regularization applied is and which type of regularization it performs is yet another example of hyperparameters that need to be set for the linear

regression model before its training phase begins. Furthermore, β_0 (the intercept term) can be included in the formulation of the model or not.

These hyperparameters can significantly affect the resulting loss on the validation dataset and hence the main objective of machine learning algorithms. Therefore, we would like to have a way to optimize the values of these hyperparameters with respect to the loss we are interested in. Equation 2.2 gives a formal definition of this optimization task:

$$\lambda^* = argmin_{\lambda \in \Lambda}\big(L(X_{validation}, A_\lambda(X_{train}))\big) \tag{2.2}$$

Where λ is a set of hyperparameter values, Λ is the space of possible values of those hyperparameters, L is the loss function which has as inputs $X_{validation}$, the validation dataset, and $A_\lambda(X_{train})$, the function learned by the algorithm A with hyperparameters λ after being trained on X_{train}, the training dataset. The result of this loss function is found by calculating the outputs given by the resulting function on the inputs of $X_{validation}$ and comparing them with the corresponding outputs of $X_{validation}$ (the actual ground truth).

In general, what distinguishes the optimization of these hyperparameters from the usual optimization of a learning algorithm's parameters is that they cannot be optimized in an automatic fashion. This is the case because there is no useful information on how the loss on the training or validation datasets varies with respect to them. That is, there is no gradient of them with respect to the datapoints nor a closed-formed solution, so we cannot apply the same learning strategies that we use with the algorithm's parameters. Due to the aforementioned observations, the optimization of these hyperparameters is commonly treated as a black box optimization problem, in which given a set of inputs (hyperparameters) we observe a certain output (loss on a validation dataset), but we have no way of knowing how it varies in advance for input values other than those we have tried. Nevertheless, later in this chapter we will realize that we can actually include some prior information on how these outputs behave under different conditions, which we can leverage for the optimization procedure. In this chapter, we will explore all the relevant aspects of hyperparameter optimization, including the main algorithms proposed in the literature in Section 2.3.

2.1 Evaluation methods

A common problem when training machine learning algorithms is to avoid overfitting. Overfitting happens when a learning algorithm performs very well on the training dataset, achieving a very small loss on it, while not performing well on the validation dataset (and, as we have seen, minimizing the loss on the validation dataset is the final objective). There are various techniques (known as regularization techniques) to avoid this from happening, for example modifying the loss function in certain ways to make the learning algorithm generalize correctly and therefore perform better on the validation dataset. This is not of our interest in hyperparameter optimization, and these techniques are usually controlled by certain hyperparameters which can be treated as any other hyperparameter. However, this problem has a similar counterpart in hyperparameter optimization. When we want to compare two functions learned by machine learning algorithms (e.g., we want to compare two different learning algorithms or different hyperparameters for the same algorithm), we compare their loss on the validation dataset. This can potentially be a problem when the validation dataset is not completely representative of the (unseen) data on which we are interested to make good predictions. It is possible that we are overfitting the choice of the learning algorithm or its hyperparameters to this validation dataset, while discarding the configuration which would be actually the best for the data of interest.

A technique to prevent this from happening is to compare several functions learned on the validation set, selecting the most promising ones and then discriminating those according to their loss on yet another set of input-output pairs, the test dataset. Since the hyperparameter search procedure has not had any information of the performance on this test dataset, it cannot possibly overfit to it. Note that overfitting to the validation dataset is not our concern in hyperparameter optimization because we have no indication whatsoever of whether certain hyperparameters are overfitting to the validation dataset, and our only objective is to simply minimize the loss of the hyperparameters on this validation dataset. In contrast, certain signs can be observed in the parameters of a machine learning model when it overfits during the learning procedure (e.g. very large weight values in linear and logistic regression), which is the information used by regularization methods to prevent it form happening.

Another technique to solve this problem is to evaluate the loss of interest by means of cross-validation, in particular k-fold cross-validation. Its main idea is to calculate the loss as an average of the losses resulting from using different validation datasets. K-fold cross-validation proceeds by partitioning a given dataset into K parts. Then, at each

step, the loss function is evaluated on a different partition of this dataset, using the remaining $K-1$ partitions as the training dataset for this step. Finally, all the losses are averaged. Equation 2.3 describes this calculation:

$$L_{cv}(X, A_\lambda(X)) = \frac{1}{K}\sum_{k=1}^{K} L(X_k, A_\lambda(X - X_k)) \tag{2.3}$$

Where L_{cv} is the final averaged loss function of cross-validation, K is the number of folds, L is the loss function as defined in Equation 2.2, X_k is the partition k of the dataset X and $X - X_k$ is the dataset X minus the partition X_k.

It is possible to use the cross-validation loss as the loss to be minimized during hyperparameter optimization. Nevertheless, in this thesis we will focus on the minimization of Equation 2.2 with the loss evaluated on a validation dataset. There are three main reasons to justify this decision. First, k-fold cross-validation requires training (a very computationally expensive procedure) and evaluating the learning algorithm k times. This would mean that the time spent in hyperparameter optimization would be, in principle, multiplied by k. Second, the focus of this thesis is in large datasets. The larger the evaluation dataset is, the lower the risk of overfitting the hyperparameters will be, because this dataset will better represent the unseen data of interest. Thus, a good performance in it will translate to a good performance on unseen data. Third, equivalently to what regularization techniques for learning algorithms do, cross-validation prevents overfitting by modifying the shape of the loss function. In standard regularization, the learning algorithm usually employs exactly the same optimization algorithm (e.g. stochastic gradient descent), which still tries to find the absolute minimum of such (modified) loss function. In the same way, our hyperparameter optimization procedure can use the same optimization algorithms which will try to find the absolute minimum of the cross-validation loss function. When we are interested in comparing certain optimization algorithms, whether they optimize the parameters of learning algorithms, or, as in our case, perform hyperparameter optimization; we try to evaluate which algorithm finds the lowest value of the given loss function in a shorter period of time. This means that these algorithms only have a defined objective given a specific loss function: to find the minimum of this particular loss function. Changing it, as with cross-validation or regularization, will only change the shape of the minimization problem, in a similar way as optimizing a different hyperparameter space or using a different dataset. In short, it is equivalent to performing a different experiment to evaluate the performance of the algorithms.

There are works such as Krueger et al. [9] which try to lower the computational time needed for performing cross-validation as compared to a single training-testing iteration by not using every fold in every case, deciding how many and which ones to use according to certain indicators (e.g. statistical tests). Also, as we will see in Section 2.3, some hyperparameter optimization algorithms have a way of reducing the execution time for cross-validation. They do so by posing this optimization as a multi-objective problem where minimizing the loss on each fold is one objective (equally weighting the objectives, hence, minimizes the average loss). Again, we will not focus on cross-validation, but these techniques can be interesting for optimizing hyperparameters in some cases, especially for small datasets.

2.2 Types of hyperparameters and practical considerations

As it is the case with the features of any machine learning problem, there are two main categories of hyperparameters according to their representation: categorical and ordered. Within the ordered hyperparameters, we can distinguish two types: continuous hyperparameters (represented as real numbers) and discrete ones (represented as integers). Ordinal variables which do not directly correspond to integers are also possible, though they can be transformed into them according to their index without a significant loss of information, so we will not pay attention to them.

The type of these variables can have a significant role on the performance of hyperparameter optimization algorithms, as we will see in the Section 2.3. Similarly to standard machine learning algorithms, some algorithms handle every type of variable natively while others require all variables to be transformed into real numbers in order to be executed. These transformations (e.g. categorical numbers to real numbers) have a certain loss of information associated and performing an appropriate transformation in every scenario can turn out to be important. For example, the performance of certain algorithms which do not handle categorical variables can be comparatively worse when many of such variables are present. Therefore, this is a relevant aspect to keep in mind when comparing algorithms theoretically and experimentally.

Another categorization of hyperparameters is based on their conditionality. Some hyperparameters are conditional, which means that they are only relevant ("active") when other hyperparameters take certain values. Others are relevant for any combination of the rest of hyperparameters, which are sometimes referred to as "global" hyperparameters. The clearest example of this is a configuration which includes both the learning

algorithm to use, for example deciding between logistic regression and naive bayes, and the hyperparameters corresponding to each of those algorithms. When one of the learning algorithms is chosen, the values of the hyperparameters from the other algorithm will be irrelevant (inactive). In this case, the global hyperparameter is the one corresponding to the choice of learning algorithm, while all the others are conditional (on that global hyperparameter). Hyperparameters within a given learning algorithm can also be conditonal (e.g., hyperparameters specific to each layer in a neural network with a variable number of layers). Most of the hyperparameter optimization methods output a proposal of the values of all the hyperparameters present to perform evaluation regardless whether they are active or not. Also, many of them need to receive feedback for every hyperparameter, even if they were inactive and have not being used at all for evaluation. Lévesque et al. [10] have examined this aspect and have shown how different strategies of giving feedback for inactive hyperparameters (imputing values) can have significant benefits in the performance of the optimization algorithms.

An important characteristic to keep in mind in practice is that different hyperparameter configurations can lead to different training times. The simplest example is when the number of iterations is one of the hyperparameters optimized for an iterative machine learning algorithm. Logically, a larger value of that hyperparameter will result in a higher training time. Other clear examples are the the depth of a tree in decision trees or the number of decision trees in random forests.

For every hyperparameter a range of possible hyperparameter values has to be defined for performing search. Even if a specific hyperparameter value can be theoretically any real number, it is not possible to search through an infinite unbounded space and therefore establishing bounds is necessary. The tighter the bounds are (while including the best configurations), the easier the search will be. We will see that some optimization methods are more sensitive to the size of the range than others.

Moreover, we might want to transform the range of some hyperparameters in practice, even though they keep the same type after transformation. A specific case is the logarithmic transformation. Some hyperparameters have a multiplicative (or cumulative) effect during the training of machine learning algorithms (e.g. the learning rate in neural networks). It is common practice to transform such hyperparameters from a linear to a logarithmic scale. An example to clarify the reason behind it is to perform uniform sampling over this hyperparameter's range. With a linear scale, approximately

90% of the samples will have the same order of magnitude as the upper bound. In contrast, with a logarithmic scale, each order of magnitude present in between the lower and the upper bound has an equal probability of being sampled. Even if uniform sampling is not actually performed and some other method is used, having a representation in an adequate scale can improve the performance of hyperparameter optimization algorithms. Works like Snoek et al. (2014) [11] try to learn the appropriate transformations automatically within the hyperparameter optimization loop. These type of methods are out of scope in this work and will not be studied further, but they can be an important practical aspect to improve the results in hyperparameter optimization problems.

2.3 Algorithms for hyperparameter optimization

This section describes the most popular hyperparameter optimization algorithms presented in the literature. The algorithms explored in the first three subsections all share a common characteristic: they treat the optimization as a black-box problem, and hence do not intervene in the evaluation process for a given hyperparameter configuration. Although their evaluation strategy is all the same (training the model with a given configuration to termination and evaluating it on an independent dataset), they differ on how they propose new configurations to evaluate. In the subsequent subsections, algorithms which also modify the evaluation strategy depending on the value of hyperparameters are described.

2.3.1 Grid search

Grid search is the most basic approach towards automating hyperparameter optimization. To perform this search, the user defines a set of possible values for each hyperparameter. Then, every possible combination of those values (which form a grid) are evaluated for the given algorithm, and the combination of hyperparameters which achieves the minimum loss on the validation set is returned as the optimal configuration.

Formally, the input of the algorithm is a set of hyperparameters to optimize $(h_1...h_K)$ and for each of them a set V of possible values $(V_1...V_K)$. The combination of those values results in $\prod_{k=1}^{k=K} |V_k|$ number of trials. This makes the number of trials grow exponentially with the number of hyperparameters (a problem sometimes referred to in machine learning as the *curse of dimensionality).*

Despite the relative computational inefficiency of this approach, it has been the most popular hyperparameter optimization method during several years, not only for practical applications but also in the machine learning research community. As an example, a survey of all the papers accepted to NIPS 2014 (one of the biggest conferences in the machine learning community), pointed out that 82 out of 86 papers which employed some kind of automatic hyperparameter search were in fact using grid search [12].

The are some reasons to explain this apart from the adoption time necessary for newly developed hyperparameter optimization algorithms (practically all of them were introduced since 2012). First of all, it is trivial to implement. Moreover, it is straightforward to parallelize, as every trial is independent of the rest and can be evaluated at any point in time. Thus, the simple strategy of evaluating several points from the grid simultaneously is optimal. Finally, it can be relatively reliable for optimizing a low number of hyperparameters (i.e., grids in low dimensional spaces) with few different values, where it can avoid suffering from the curse of dimensionality.

2.3.2 Random search

Another relatively simple approach is to also build a grid of points and execute trials for each of them independently, but to select these points in a random fashion. Instead of defining a set of points for each hyperparameter, the user defines a range from which to search values for these points, typically sampling them from a uniform distribution in that range.

Even though the user has no control over the specific points in the hyperparameter space to try and random search can result in a less evenly spaced set of points, it offers significant advantages over grid search. The main one is noticeable when some dimensions (hyperparameters) are more important than others for the variation of the loss. For a relatively high number of dimensions it is quite unlikely that all of them have the same impact in the metric to be optimized, so this is a common scenario. The reason for this is that grid search explores a relatively small number of different values for a single dimension as compared to the total number of trials. On the other hand, random search will almost certainly try a new value for each hyperparameter in each trial given a sufficiently large search space. Its consequence is that grid search allocates much more resources for the exploration of unimportant dimensions than random search does.

An illustration of the previous observation is shown in Figure 2.1. In this example of two dimensions, one hyperparameter has a significantly bigger influence on the result. The distributions on the sides represent the marginal distribution of the metric evaluated with respect to the hyperparameters. While the evenly spaced grid search only has three different trial values for the important hyperparameter out of 9 total trials, random search results in nine different trial values for it, being able to identify a more promising value for that dimension and therefore for the combined hyperparameter configuration.

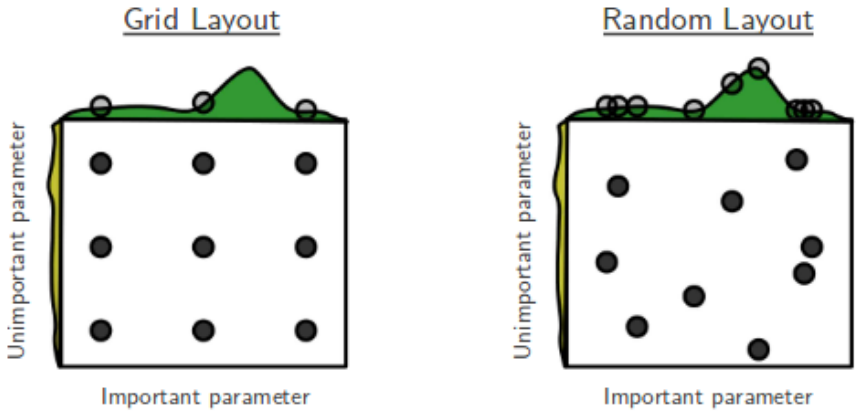


FIGURE 2.1: Example layout of trial points of grid and random search in two dimensions (taken from Bergstra and Bengio [1]).

It would be possible for a user with knowledge of the relative importance of each hyperparameter to define an appropriate grid with a larger number of values to try for that hyperparameter than for the others. However, as Bergstra and Bengio [1] show, even for a fixed algorithm being evaluated on similar datasets, the importance of different hyperparameters in the final result varies considerably, making it difficult to have a reliable estimation beforehand.

Apart from this main advantage, random search offers additional benefits when performing experiments. While grid search requires an exhaustive evaluation of each trial point to have full coverage, random search experiments can be stopped at any point and still form a complete experiment. This has interesting implications regarding fault tolerance in cluster setups. If a computer fails while performing a trial, that trial can simply be discarded or restarted without further communication or rescheduling. Moreover, if the resources vary during the optimization process, more or less trials and be scheduled without having to adjust the trial points as it would be the case in grid search. At the same time, random search is simple to implement and also trivial to parallelize (just by

evaluating several random points simultaneously), two of the main positive aspects of grid search as we have seen in the previous section.

2.3.3 Bayesian optimization methods

More complex algorithms, referred to as Bayesian optimization methods, base their hyperparameter proposals on past information of hyperparameter evaluations, acting in a sequential manner. They are also known as sequential model-based optimization or adaptive configuration methods.

These models try to predict the outcome of a training run by means of a function which is significantly less expensive to evaluate than training the learning algorithm and evaluating its loss on the validation dataset. This function, known as "surrogate function", takes a set of hyperparameters as an input and gives a prediction of the loss of those hyperparameters on the validation dataset as an output. As such, any regression model can potentially be used for this purpose. This function is continuously updated based on past information of the resulting loss after training and evaluating with hyperparameters of previous trials. This makes it possible to find an input configuration with predicted promising results and evaluate the learning algorithm with it, trying to avoid training the algorithm with "bad" hyperparameter values, and subsequently updating the surrogate function. Algorithm 1 describes the strategy followed by these methods.

Algorithm 1 Sequential Model-based Optimization

1: Input: Learning algorithm A, initial model M_0, number of trials T, acquisition function S, loss function L
2: $H = \emptyset$
3: **for** $t = 1...T$ **do**
4: $\quad x = S(M_{t-1})$
5: $\quad$ Evaluate $L(A_x)$ (Expensive step)
6: $\quad H = H \bigcup (x, L(A_x))$
7: $\quad$ Fit a new Model M_t to H
8: **return** H

Intuitively, these methods try to mantain an informed estimate of the possible results of new hyperparameter evaluations based on past information, trying to minimize the number of trials needed to find an optimal (or good enough) hyperparameter configuration. There are various methods following this principle, which differ in the type of surrogate function they use to estimate the quality of unseen hyperparameters and how they suggest a new configuration based on that function (using an acquisition function). These proposals have to balance exploration and exploitation. That is, they have to

make use of the knowledge about areas where hyperparameters are estimated with high confidence to give good results, but at the same time they have to explore areas where the results are unknown.

The most relevant and successful methods presented in literature are described in the following subsections.

2.3.3.1 SMAC

SMAC (Sequential Model-based Algorithm Configuration) [13] was one of the first algorithms of this kind to become widely used, and still is one of the most popular choices for hyperparameter optimization. Initially, it was developed with a focus on finding optimal configurations according to a set of related objectives, instead of only one task. In machine learning problems, the evaluation of a configuration is commonly considered a single-objective problem. Nevertheless, optimizing a set of objectives can have interesting applications. One example is that, in the case of evaluating by performing cross-validation, it can avoid evaluating every fold every time and just use a few of them. It does so by posing each fold evaluation as a separate problem, while still optimizing for every fold. Another example is transfer learning [14]. Establishing the same configuration for different problems will typically result in a downgrade of performance compared to optimizing for those problems individually. However, there might be scenarios where the goal problem might not be represented by any dataset available but may be similar to a few of them, making this technique a reasonable choice. This aspect is however not further explored in this thesis, and to find a comprehensive description of the mechanism involved in the multi-objective optimization of SMAC the reader can refer to Hutter et al. (2011) [13] (named as "intensification mechanism" in that work), including how different problems can be assigned features about their characteristics ("instance features") to improve performance.

The surrogate model employed in SMAC is based on random forests [15] for regression. They have shown good performance in a variety of challenges [16] and are a common choice for both performing classification and regression. Random forests are ensemble methods whose basic idea is to train a certain number of decision trees and combine their individual outputs. The way these trees differ to each other is in the way they select features for each bifurcation of their branches: for each node to be split, only a certain proportion p of randomly chosen features are considered, leading to diverse

trees. Apart from these proportion, other hyperparameters of this surrogate function are the number of trees B to employ and the minimal number of datapoints to be present in a node to be split further, n_{min}. The authors of SMAC set fixed values for these hyperparameters of the surrogate function, setting their values to $B = 10$, $p = 5/6$ and $n_{min} = 10$. The output of the different decision trees for each configuration is combined to give two results: an empirical mean, μ, and an empirical variance, σ^2; calculated as the mean and variance of the single predictions given by each tree.

Once the surrogate model is defined, it is also necessary to establish how new configurations are selected based on the predictions of this model. For that purpose, SMAC uses the expected improvement (abbreviated as EI) criterion [17], as a function $\text{EI}(\theta)$ of a configuration θ. This criterion balances exploitation and exploration by favouring choices with small predicted loss and high predicted variance, respectively. In the original work of Hutter et al. (2011) [13], a version of the expected improvement for log-transformed cost is used, as the original objective was to minimize the logarithm of the running time of certain optimization algorithms not related with machine learning. However, in the hyperparameter optimization area, the original expected improvement criterion is used as a surrogate function with SMAC. Equation 2.4 shows the expected improvement function:

$$EI(\theta) = \sigma(\theta) \cdot (Z \cdot \phi(Z) + \varphi(Z)) \tag{2.4}$$

Where $Z = \frac{f_{min} - \mu(\theta)}{\sigma(\theta)}$, ϕ refers to the cumulative distribution function of the standard Gaussian distribution (i.e. with mean 0 and variance 1), φ is the probability distribution function of that standard Gaussian and f_{min} is the minimum loss present in the training data of the surrogate model (i.e. minimum loss achieved so far by the learning algorithm on the validation dataset).

It is still not clear, though, how to find configurations which have high EI, and this could be seen as an optimization problem by itself. The original approach is to start a local search on the top 10 best performing configurations found so far. This local search works by proposing candidates modifying one single hyperparameter at a time from the original configuration, using a different strategy for integer, categorical and numerical hyperparameters. The exact details of this search are described in Section 4.3 of the original paper [13]. Then, the EI function evaluates other 10.000 randomly selected points and all the configurations evaluated are sorted by EI in a list. Then, they interleave one randomly sampled configuration between each 2 elements in that

list to include configurations unbiased by the current state of the surrogate function as potential proposals. Finally, this results in a list where the top elements can be chosen as new suggestions (in the original multi-objective framework, at least 2 points are chosen at a time, so at least one random point is chosen).

In the hyperparameter optimization setting, random forests present an important advantage as compared to other surrogate functions: they are able to explicitly model categorical variables due to their capability to split branches based on not only numerical ranges but also on specific values (because of individual decision trees being able to do so). As we will see later in this thesis, some surrogate functions need to transform these categories in one way or another into a numerical representation, losing some information on the way. Moreover, they can handle conditional hyperparameters in a natural way, without the need to explicitly specify their structure due to the hierarchical and conditional nature of decision trees. Since some hyperparameter choices will be irrelevant given others as discussed in Section 2.2, random forests' trees will, over time, learn to split their branches based on only relevant active hyperparameters in the current branch. This makes irrelevant hyperparameters lose all their (negative) influence.

However, random forests also present certain drawbacks as surrogate functions. Due to their empirical nature, their variance estimates can be inaccurate, which can lead to suboptimal choices in the acquisition function. Additionally, they require a relatively large amount of training data to start giving sensible predictions. Therefore, the first proposals made by this surrogate function are usually not ideal. It can be a good idea to first generate some configurations with some other strategy (e.g. random search), feed the surrogate function with the results obtained with that strategy and start using it from that point. The introduction of random points in its list of proposals also tries to tackle this problem and the potential of initially failing to explore by "getting trapped" in a certain hyperparameter region.

2.3.3.2 TPE

As explained in the previous section, SMAC's surrogate function explicitly models the posterior probability of the loss $p(l|\theta)$ given a configuration θ. On the contrary, the TPE (Tree-structured Parzen Estimator) algorithm [18] models $p(l)$ and $p(\theta|l)$ to make proposals. Depending on a threshold loss (l^*), it builds two different probability functions for the configurations, described in Equation 2.5:

$$p(\theta|l) = \begin{cases} s(\theta) & \text{if } l < l^* \\ g(\theta) & \text{if } l \geq l^* \end{cases} \tag{2.5}$$

Where l^* is chosen as a γ-quantile of the losses observed so far (originally the authors chose to set γ to 0.15, that is, the 15% quantile). Note that the notation of the first function varies with respect to the original formulation to be consistent with the use of l as the loss in this thesis. In essence, TPE builds a probability density function for parameters which perform "good" (better than the threshold, represented by $s(\theta)$) and another one for hyperparameters which perform "bad" (worse than the threshold, represented by $g(\theta)$). They also employ the expected improvement criterion of Equation 2.4, which however for this model has a different derivation described in Equation 2.6:

$$EI(\theta) \propto \left(\gamma + \frac{g(\theta)}{s(\theta)}(1-\gamma)\right)^{-1} \tag{2.6}$$

This criterion is used to calculate the points with the highest EI. Intuitively, it assigns high values for points with high probability of being "good" according to the $s(\theta)$ distribution and low probability of being "bad" according to the $g(\theta)$ distribution. In this case, the way TPE searches for configurations with the highest EI is by sampling candidates according to the $s(\theta)$ density function (points which probably return a good result according to the model) and evaluating them according to the Equation 2.6.

Both $s(\theta)$ and $g(\theta)$ are modelled in exactly in the same way, but using different hyperparameter values (those corresponding to smaller and greater loss than the threshold specified above, respectively). They place density in the neighbourhoods of the tried configurations which have resulted in a loss within their range. The densitify functions have a tree structure where each node is a 1-dimensional Parzen estimator [19]. Each node corespondends to a single hyperparameter. Depending on the type of the hyperparameter (discrete, or continuous), the node produces different estimations. The conditional dependence between hyperparameters is explicitly expressed through its tree structure.

One clear advantage of this approach is the possibility to explicitly define conditional dependence between hyperparameters. Hyperparameters which are not active in a given configuration never affect the expected improvement function nor the resulting surrogate model after training. This can be very beneficial for highly conditional scenarios such as deep neural networks or whole pipeline optimization, and it is more efficient than

SMAC for this purpose as it does not need to learn the conditional dependencies based on data. Moreover, similar to SMAC, TPE is able to explicitly model both numerical and categorical variables without any need to transform them.

Nevertheless, TPE's capability of explicitly modelling conditionality comes at a (potentially) great cost. TPE assumes independence between hyperparameters which are not in the same branch from the root to the leaves. In other words, the only dependency that TPE is able to express is that of being active or inactive according to other hyperparameters. This incapability of modelling joint probabilities (as opposed to, for example, SMAC) has important implications: in TPE, the value of a proposed hyperparameter does not depend on the value of any other hyperparameter (only on whether it is active or not), which can be an oversimplification. What a good value for one hyperparameter is can be highly correlated (or even determined) by the value of others (e.g. as it is typically the case between the learning rate and minibatch size hyperparameters in neural networks). These correlations cannot be captured by TPE.

2.3.3.3 Gaussian processes

Another set of hyperparameter optimization techniques are based on using Gaussian Processes as a surrogate function. A Gaussian Process (GP) is a regression technique which is a generalization of the Gaussian probability distribution to the function space [20]. At every point in the input space of a function, the prior probability of its output is defined by a a Gaussian probability distribution, with a certain mean and variance estimation of the distribution at that point. In fact,a GP defines a multivariate Gaussian distribution over any finite number of points. The posterior function given a set of observed points is also a GP. The calculation of its posterior mean and variance at any point has a closed form, a fundamental property which makes this model attractive as a surrogate function. How these probability distributions exactly look like depends on a set of configurable hyperparameters or beliefs. For example, what we believe the mean value of the function is, how much noise is present in the observations or how the outputs of different inputs relate to each other. Given certain GP hyperparameters, the posterior distribution of a certain input given a set of observed points will have one form or another. Equation 2.7 and Equation 2.8 show the calculation of the predictive mean and variance functions of the posterior, respectively:

$$\mu(x|D) = K(X,x)^t(K(X,X) + \sigma^2 I)^{-1}(Y - \mu_0) + \mu_0 \tag{2.7}$$

$$\Sigma(x|D) = K(x,x) - K(X,x)^t(K(X,X) + \sigma^2 I)^{-1}K(X,x) + \sigma^2 \tag{2.8}$$

Where D corresponds to the training datapoints (pairs of configurations tried and resulting loss), μ_0 is the prior mean GP hyperparameter, σ is the noise GP hyperparameter, K is the covariance function employed (further described below in this section), X is the set of tried hyperparameters in D, Y is a vector of their corresponding losses, x is the configuration whose mean or variance is to be predicted and I is the identity matrix.

For a more detailed description of Gaussian Processes see Rasmussen and Williams [20]. One relevant aspect is that Gaussian processes scale linearly in the number of input variables and cubically in the number of training datapoints due to the calculation of the inverse of a matrix containing those. Next, I will focus on the Spearmint [21] implementation of a GP-based hyperparameter optimization method described by Snoek et al. (2012) [22], as it is a standard in experiments comparing algorithms in the literature.

The covariance function plays an important role in the shape of the GP by defining the interaction between input points. Intuitively, it defines how observed points are believed to influence the posterior function, i.e., how far away values are influenced by them and in which way. There are many possible covariance functions (also known as kernels) which can be used, but the original Spearmint work proposes to use the ARD Matérn 5/2 kernel, which has since become a popular choice for hyperparameter optimization and which is described in Equation 2.9:

$$K_{M52}(x;x^{'}) = \theta \cdot \left(1 + \sqrt{5r^2(x,x^{'})} + \frac{5}{3}r^2(x,x^{'})\right) \cdot exp\{-\sqrt{5r^2(x,x^{'})}\} \tag{2.9}$$

Where $r^2(x,x^{'}) = \sum_{d=1}^{D} \frac{(x_d - x_d^{'})^2}{\lambda_d^2}$, the square of the usual euclidian distance between two vectors divided by a lengthscale vector λ which is a hyperparameter of this kernel (and therefore a GP hyperparameter). The amplitude θ is the other hyperparameter of this kernel.

As we have observed in equations 2.7, 2.8 and 2.9; the definition of a GP has its own hyperparameters. The lengthscale vector, amplitude, prior mean and noise need to be determined to perform computations. It seems difficult to set reasonable values for all these GP hyperparameters that function well in a variety of scenarios. Nevertheless, thanks to the properties of Gaussian Processes, it is possible to avoid doing so manually in two different ways. First, it is possible to calculate the likelihood of the outputs of observed points depending on different GP hyperparameter values, selecting then the values which maximize this likelihood. Moreover, it is possible to avoid

setting a point estimate (i.e., setting a single value) for these GP hyperparameters altogether by marginalizing the acquisition function over the GP hyperparameters. This means calculating the acquisition function over the space of possible GP hyperparameter values and integrating their result (averaging in the discrete case) to obtain the final acquisition function. In this case, the GP hyperparameters have in turn their own prior functions associated, which control the space of possible GP hyperparameter values and influence their likelihood in order to integrate over them. These priors do need to be set manually, but their influence is significantly smaller as compared to setting manual GP hyperparameter values and this approach is hence more flexible. In order to calculate the acquisition function over GP hyperparameters, Spearmint uses a Monte Carlo technique, slice sampling [23]. This technique samples possible hyperparameter values according to their likelihood given a set of observed points, which are then used to calculate an average of the acquisition function.

The acquisition function used to suggest new points based on the posterior predicted mean and variance of a point is the expected improvement criterion, exactly in the same form as the one used for SMAC and described in Equation 2.4. This function is the most popular one in the area of hyperparameter optimization using GP surrogates. However, this is not the only option and other acquisition functions such as Probability of Improvement [24], GP upper-confidence bound [25] or Thompson Sampling [26] are also used.

The article introducing Spearmint also describes another acquisition function which takes into account the predicted training time corresponding to a certain configuration. The reason for this is to propose points not only according to their predicted loss, but also favouring those which are estimated to train faster (and therefore have faster evaluation). It is shown that this can result in a faster optimization procedure, arriving to similar solutions in less wall-clock time. For that purpose, another independent GP models a function with hyperparameters as input variables and logarithmic training time as output. This output is then combined with the expected improvement from Equation 2.4 to obtain this new acquisition function, referred to as expected improvement per second.

Figure 2.2 shows the behavior of a Gaussian Process surrogate and a corresponding acquisition function (defined as u) on a toy one dimensional problem with two observations. Note that the goal in this case is to maximize the objective function (dashed line, defined as f), as opposed to our usual definition. We can see that the areas with high values for the acquisition function have high predicted mean (bold line) combined with

relatively large variance (blue shade). The next configuration to be proposed would be the one corresponding to the maximum of the acquisition function (red triangle). Once this configuration is evaluated, the model will discover that its value is lower than the point on the right-hand side, so it will not improve the best hyperparameter configuration found so far.

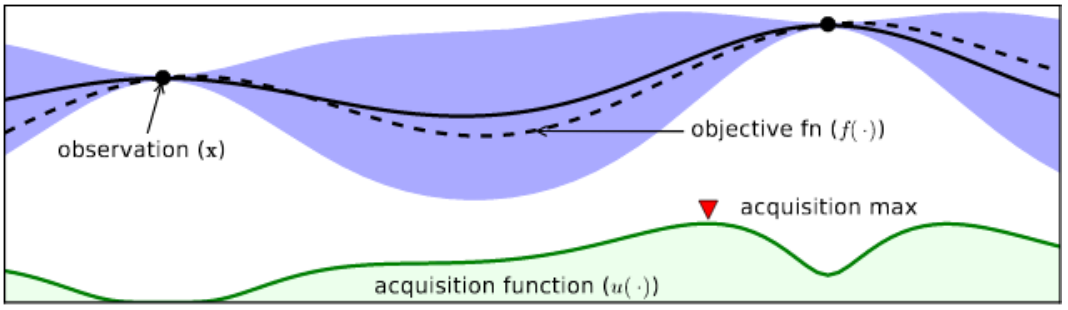


FIGURE 2.2: Gaussian process optimization example (taken from Brochu et al. [2]).

Once the acquisition function is defined, as it was the case with SMAC, it is necessary to find a strategy in order to search for the inputs that maximize it. Although not described in the original work [22], the search method included in the initial Spearmint implementation is based on evaluating this function on a grid of points covering the whole hyperparameter space and selecting a set of points with highest EI. After that, a local search starts in each of these promising points. This search is done by a quasi-Newton optimization algorithm, L-BFGS-B, which makes use of the output of a function and its gradient to find a local minimum [27]. The differentiability of the ARD Matérn 5/2 kernel chosen for the covariance function makes it possible to easily calculate the gradient of the EI, thus allowing the usage of this optimization method.

One of the characteristics that makes Gaussian Processes interesting for hyperparameter optimization is that they do not require a relatively large number of training examples to give sensible predictions. Instead, they make an optimal (given their formulation) estimation of the outputs even at early stages. Moreover, they accurately model not only the predicted mean but also the variance (in contrast to SMAC), which allows for the acquisition function to be more precise. Additionally, they explicitly model joint probabilities of all the hyperparameters (in contrast to TPE), and does so in a continuous way, rather than by splitting the hyperparameter space (like SMAC). Finally, the dependence on their own hyperparameters is smaller than for other methods, thanks to the possibility to either automatically estimate them for a given problem or to marginalize them.

As weak points, GPs cannot directly handle categorical nor conditional hyperparameters. For categorical and integer hyperparameters, transformations need to be done to embed them in a continuous range. Even though there are tricks to minimize the impact of conditional hyperparameters, as explained in Section 2.2, they always introduce some noise in the behavior of Gaussian processes. For these reasons, there can be a downgrade of the performance of this model when several categorical and conditional hyperparameters are present. There is also the possibility to place independent Gaussain Processes over hyperparameters at different layers in the conditional structure, as Bergstra et al. do in their work [18]. However, this means losing the joint probability over variables represented in different GPs (in a similar way to TPE), and therefore the capability to model their correlations.

Another concern is their potential overhead: their cubic complexity in the number of training instances can lead to relatively large times to suggest the next candidate configuration, as we will see further in Chapter 6. The large training times of the machine learning models being optimized should dominate this overhead, more so in the large-scale regime. Yet, at some point (e.g. after 10000 training trials), it will have a significant impact on the total optimization time. It is not generally common to perform so many trials during a hyperparameter optimization process due to the long training times, but it is possible that this is desirable for some applications. The optimization process can be restarted at any point in time by keeping the history of hyperparameters tried and their corresponding loss, making this situation more likely to happen. Snoek et al. (2015) propose a way to circumvent this problem, using a neural network as a surrogate function which has similar properties to the ones of GPs while scaling linearly in the number of observations [28].

2.3.4 Parallelizing Bayesian optimization methods

We have seen in the previous section how adaptive configuration methods rely on the results of previously evaluated configurations. Thus, they are inherently sequential, which makes it difficult to run them in parallel. Recently, the introduction of multi-core processors and cloud computing infrastructures has facilitated the parallel execution of workloads. Therefore, the possibility of evaluating simultaneously a certain number of configurations can provide a great advantage in terms of wall-clock time. For this reason, strategies adapting the methods of Section 2.3.3 have been designed in order to propose more than one configuration at a time before receiving further feedback.

SMAC can be parallelized in several ways, and Hutter et al. (2012) explore how to do it for the original setting of a multi-objective black-box function [29]. A simple way to propose N configurations is to simply take the top-N elements from the list constructed by SMAC (as explained in Section 2.3.3.1). Then, each of these configurations can be evaluated in parallel. The configurations with high EI resulting from this strategy may lead to excessively low exploration due to being concentrated in a similar region of the hyperparameter space. However, every other point is randomly sampled, which compensates for this fact. A more sophisticated way to do so, employed by Hutter et al. (2012) [29], is to use a version of the upper-confident bound as acquisition function, as described in Equation 2.10 :

$$UCB(\theta) = -\mu(\theta) + k \cdot \sigma(\theta) \tag{2.10}$$

Where k is a fixed value to be determined. Using this equation, the idea is to randomly sample different values of k, resulting in different acquisition functions, and search for points which maximize each of these functions. Then, the maximizing points are combined and evaluated simultaneously. This combination of points found with different criteria leads to more diversity than the simple strategy, which can be interesting to increase the exploration of these parallel evaluations.

TPE also has a straightforward way of suggesting multiple configurations to be run in parallel. Bergstra et al. propose to simply rely on the stochasticity of its random draws to optimize the acquisition function in order to find diverse points with high EI [18]. Since these draws are expected to differ from run to run, we can take the top-N configurations with highest EI out of all the points sampled and evaluate them (similarly to the simple SMAC strategy described above). Originally, they claim to run one acquisition function search per configuration proposed. Nevertheless, draws across iterations are being sampled from exactly the same distribution. Hence, this approach can be substituted by sampling N times the number of points sampled in each of those runs and taking the top-N suggestions as previously described.

Regarding Gaussian Processes, there is a wider variety of options which have been studied. Simply selecting the top-N points with the highest EI as a result of the EI optimization search is a suboptimal option. This strategy leads to excessive exploitation, reinforced by the continuous nature of GPs. Several techniques for suggesting simultaneous points in a more principled way are proposed in the literature [30–32]. At first, it could seem possible to calculate the acquisition function over more than one point

making use of the multivariate Gaussian distribution that GPs define. This calculation, however, becomes intensive for a large number of points. Searching for multiple configurations that jointly maximize this function results in an optimization problem of a very large dimensionality. Additionally, this optimization is possibly derivative-free if this acquisition function is estimated by Monte-Carlo integration to tackle its complexity [33]. There is a possible alternative to suggest configurations to be evaluated simultaneously in a sequential manner. This is done by first proposing one configratuion and then estimating its associated loss in some way without actually evaluating it, temporarily giving this estimation as feedback to the model. Depending on how the estimation of the loss is calculated, different methods using this strategy exist.

The most obvious one is to use the mean loss predicted by the conditional distribution of the GP for that point and temporarily include that result to propose the next configuration. This method, however, is not promising because it leads to excessive exploitation. It focuses too much on the model's current predictions and repeatedly reinforces the regions that it considered promising during the first proposal, as Ginsbourger et al. (2008) demonstrate [30]. Another option is to use the "constant liar" approach. It is based on always assigning the same constant value to the proposed configurations as the estimation of the loss. This constant value can be chosen in several ways. It has been suggested to use the maximum, mean and minimum loss achieved so far, which results in higher to lower exploration, in respective order [30]. A newer mixed technique following this strategy has been studied. It is based on sampling one set of points using the constant liar approach with the maximum loss value, another set with the minimum loss value, and proposing the set of points which has the highest EI under the current GP [32]. Finally, the GP defines a Gaussian distribution on the loss of each of these points, so it is possible to draw loss samples from this distribution. Based on these samples, we can integrate the acquisition function over the space of possible outcomes of each pending hyperparameter, averaging over possible losses of pending configurations according to the distribution. This integration is done in an equivalent way to the integration over the GP hyperparameters described in Section 2.3.3.3. This last technique is the one implemented in Spearmint and described by Snoek et al. (2012) [22].

One important thing to keep in mind is that this parallelization does not come without any drawbacks. All the previous techniques include in its proposals the configuration which would be proposed in a sequential way. However, the rest of the proposals are of less quality than in the sequential case. This is true in higher or lower degree for each method previously described in this section. Logically, with imprecise feedback from the loss corresponding to previous proposals, no technique can predict with equal accuracy

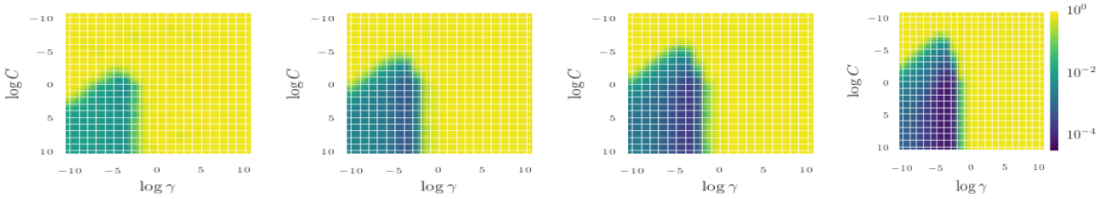
what good configurations are as compared to knowing exactly this loss. This parallelization means that (ideally) the evaluation of N simultaneously proposed configurations using of those methods can be run N times faster. Nevertheless, these configurations are expected to be of less quality than if N configurations would be proposed and evaluated sequentially. Moreover, the larger N, the worse the proposed configurations are expected to be as compared to N configurations proposed in a sequential manner. That is, the quality of configurations progressively decreases as the speedup factor increases.

2.3.5 Adaptive evaluation methods

An orthogonal approach to hyperparameter optimization is based on adapting the way configuration are evaluated rather than adapting the selection of configurations to try. This approach does not longer assume that machine learning algorithms are a black box, it instead leverages prior knowledge about the machine learning training process. Specifically, it takes advantage of the fact that training runs with less resources (be it smaller datasets, less number of iterations for iterative algorithms or a fewer of features) are representative of training runs with full resources (e.g. training until completion or with the full dataset). The main idea is to train several hyperparameter configurations with low resources or cost, discarding the least promising configurations based on this "partial" training and evaluation runs. Then, these strategies assign more resources to the most promising configurations to evaluate further. This way, the evaluation of proposed configurations is modified, spending less time for unpromising configurations and resulting in a potentially faster discovery of good hyperparameters.

Several alternatives of resources to be constrained have been studied, such as number of iterations, dataset size and (less frequently) number of features [34]. Note that not every machine learning algorithm is trained in an iterative way. Also, for the ones that are trained iteratively, their behaviour as iterations advance might differ considerably for different models. In contrast, using different training dataset sizes is possible for every algorithm. Moreover, this usually yields similar comparative results across models. Results with smaller training datasets are good representatives of the performance on bigger datasets (as shown by Klein et al. [3]). Figure 2.3 shows an example of how the same value combinations of two hyperparameters have similar good (blue) and bad (yellow) results across different dataset sizes for a certain algorithm. Additionally, as we can also observe in that figure, bigger datasets typically lead to performance improvements in a somewhat logarithmic fashion [35, 36]. On the other hand, the training run of algorithms with different dataset sizes has to be restarted from scratch. In principle, no information from previous runs with small portions of a training dataset can be used

for training faster with bigger portions. Training with a larger number of iterations, instead, can be optimized by continuing from the last iteration of previously trained models kept active after evaluation (if this is an option). Thus, both using the number of iterations and the dataset size as a resource have their own advantages and disadvantages.



(a) $s = 1/128$ (b) $s = 1/16$ (c) $s = 1/4$ (d) $s = 1$

FIGURE 2.3: Validation error of an algorithm across different subset proportions (s) according to a grid of values for two hyperparameters (taken from Klein et al. [3]).

The following subsections describe the most relevant methods that use this strategy. In their original versions, they use either random or grid search for proposing hyperparameters which are then evaluated using this adaptive strategy. We will explore methods combining adaptive configuration and adaptive evaluation later in Section 2.3.6.

2.3.5.1 Successive Halving

Successive Halving (SH) [37] was originally inspired by work in the multi-armed bandit field [38], formulating it in a suitable way for the area of hyperparameter optimization. Mulit-armed bandits study scenarios where a resource (e.g. computational power) is allocated to one out of a set of choices (e.g. hyperparameter values). As the quality of each choice is initially unknown, different strategies are proposed to learn about their quality and maximize the desired outcome. Two main objectives are studied in this area. First, simply identifying the choice with the highest reward (or lowest loss). Second, maximizing the cumulative reward (or minimizing the cumulative loss) over time. In this case, we are concerned not only on finding out what the best configuration is at the end of the execution, but also on having selected choices along the way that reported good results. While a significant part of the multi-armed bandit literature studies the second objective, the hyperparameter optimization problem fits better under the first objective. In hyperparameter optimization, we want to find out the best configuration at the end of the execution and we are not worried if to do so we have evaluated bad performing configurations along the way. Concretely, it belongs to the non-stochastic version of this problem, because we assume the loss of a certain configuration to be deterministic given the same evaluation data. Karnin et al. [39] first proposed the Successive Halving

algorithm for the stochastic setting, while Jamieson and Talwalkar [37] later proposed it for the non-stochastic setting with a focus on hyperparameter optimization. Algorithm 2 describes this method.

Algorithm 2 Successive Halving

Input: Total budget B, n initial arms (choices),
$S_0 = \{1, 2, ..., n\}$
for $k = 0, 1...\lceil log_2(n) \rceil - 1$ **do**
    Run each arm i in S_k with budget $\lfloor \frac{B}{|S_k| \lceil log_2(n) \rceil} \rfloor$ and find its loss $l_{i,k}$
    $L_k = \text{sort } (l_{0,k}...l_{|S_k|-1,k})$ in ascending order
    $S_{k+1} = \{i : l_{i,k} \in \text{First } \lfloor \frac{|S_k|}{2} \rfloor \text{ elements of } L_k\}$
return The only element of $S_{\lceil log_2(n) \rceil}$

Successive Halving evaluates all the configurations for an initial fixed budget $\frac{B}{\lceil n \cdot log_2(n) \rceil}$ per configuration. Based on the losses obtained, it sorts the configurations by loss and discards the bottom halve. Then, the budget of each remaining configuration is doubled for the next halving round, proceeding in this manner for subsequent rounds until only one configuration remains. The execution of the algorithm takes $\lceil log_2(n) \rceil$ rounds, using a constant budget per round of $\frac{B}{\lceil log_2(n) \rceil}$ which results in a total budget B utilized at the end of the execution. Apart from describing the algorithm, Jamieson and Talwalkar also analyze the theoretical guarantees of this approach [37]. Additionally, they evaluate the performance of Successive Halving in hyperparameter optimization tasks taking the number of iterations as a resource. However, as we have previously seen, any other resource such as dataset size or number of features can be used for this algorithm in the same way.

In terms of potential parallelization, the whole loop can be started in different workers at the same time. Another option is to parallelize a single loop by evaluating several models in the same round at the same time. Nevertheless, the maximum speedup factor with this strategy decreases as rounds advance and the last round ends up being a bottleneck. Li et al. (2018) describe an asynchronous way of parallelizing Successive Halving following this strategy [40]. Their strategy is conservative, meaning that it evaluates all the configurations that are evaluated during the synchronous execution, while possibly evaluating more configurations in a given round (which would have been discarded in the synchronous case). It guarantees that a configuration at least as good as in the synchronous version is returned, while potentially using more total computation time (though compensated by the parallelization leading to less wall-clock time).

Note that there are simple tweaks to this technique which seem sensible. First, the rate of choices which continue after each elimination step could be modified (e.g., 1/3 instead of 1/2) for a more or less aggressive evaluation. Second, the budget assigned for each round could be modified in different ways as well. For example, assigning a higher budget progressively as the number of rounds grows instead of keeping the budget per round constant. However, the theoretical or experimental implications of these modifications are not studied in the original work.

2.3.5.2 Hyperband

Hyperband is an adaptive evaluation algorithm based in Successive Halving. Li et al. (2016) realized that Successive Halving needs to define not only the total budget to use but also the number of configurations to be compared [34]. This number of configurations, in turn, determines how much budget is allocated for each configuration in each round. A higher number of initial configurations means that more configurations are evaluated but with smaller budget for the same round, discarding earlier the worst performing ones. A smaller number, in contrast, means that fewer configurations are evaluated but with more resources, which results in more reliable information about the final loss of each configuration. If the configurations have similar results (i.e. they have similar losses) they require more precision to differentiate their quality, so it would be reasonable to evaluate less configurations with more resources. On the other hand, if fewer resources are needed to distinguish their quality, it is more appropriate to evaluate a larger number of initial configurations. This presents a trade-off between the number of initial configurations to try against the certainty in returning the best of those configurations. Algorithm 3 formally describes Hyperband as proposed by Li et al. (2016) based on the previous observations [34].

Hyperband peforms a grid search over possible values of the number of initial configurations to evaluate for a fixed budget and returns the configuration found with the smallest loss. Apart from the maximum budget to be allocated for a single resource, it also requires as an input the rate of configurations to discard after each Successive Halving round. As described in the previous section, the original formulation of Successive Halving fixes this rate to 1/2. Proceeding this way, Hyperband makes sure that an optimal situation of the trade-off described in the previous paragraph is found. Nevertheless, this is done by applying brute force and results in approximately $\lfloor log_\eta(R) \rfloor + 1$ times more computations than Successive Halving for a single iteration, where η is the rate of configurations that are kept between rounds and R is the maximum budget allocated for a single resource. The total budget spent per Successive Halving iteration

Algorithm 3 Hyperband

Input: Maximum budget for a single configuration R, η the factor of configurations kept between rounds
for $s = \lfloor log_\eta(R) \rfloor, \lfloor log_\eta(R) \rfloor - 1, ...0$ **do**
    $n = \frac{(\lfloor log_\eta(R) \rfloor + 1) \cdot \eta^s}{(s+1)}$
    $r = R \cdot \eta^{-s}$
    $T = getHypermarameters(n)$
    for $i = 0...s$ **do**
        $n_i = \lfloor n\eta^{-i} \rfloor$
        $r_i = r\eta^i$
        Run each configuration c in T with budget r_i and find its loss l_{c,r_i}
        $L_i =$ sort $(l_{0,r_i}...l_{|T|-1,r_i})$ in ascending order
        $T = \{c : l_{c,r_i} \in$ First $\lfloor \frac{n_i}{\eta} \rfloor$ elements of $T\}$
return $c : l_{c,R} < l_{p,R} \, \forall \, p \neq c$ where c and p have been run with budget R

is $B = (\lfloor log_\eta(R) \rfloor + 1)R$. Li et al. (2016) suggest to set η to 3 or 4, in contrast to the fixed rate 2 of Successive Halving [34]. Additionally, this work explores the behavior of Hyperband taking diverse aspects as a resource: dataset size, number of iterations and number of features.

Hyperband can be trivially parallelized in a similar way to grid search, starting one Successive Halving run in each worker instead of the evaluation of a single configuration. Another option is to parallelize each single Successive Halving execution as described in Section 2.3.5.1. A detailed analysis of Hyperband, which briefly discusses this aspect, is available in a more extensive study by Li et al. (2017) [41].

2.3.6 Combined adaptive configuration and evaluation

Based on the ideas of hyperparameter optimization algorithms from sections 2.3.3 and 2.3.5, new methods have been proposed. They combine improving proposals over time based on the losses found and assigning different resources to promising and unpromising configurations while evaluating them. This subsection describes the most relevant methods of this kind.

2.3.6.1 Modelling learning curves for early stopping

Two methods combine a surrogate function to suggest new hyperparameters with a prediction of the evolution of the learning curve for (exclusively) iterative machine algorithms. Their objective is to perform early stopping, that is, stopping the training run

of a configuration before it converges.

Swersky et al. (2014) use an independentent Gaussian Process to model the evolution of learning curves (the loss after each training iteration) [42]. This GP has a kernel function acting as a prior of the behavior of these curves. Then, they select a set of candidate points according to the expected improvement acquisition function. They predict the evolution of the learning curve for each candidate on the next iteration and calculate the information gain that would result from it. With this information, they select which configuration to train for another iteration. They combine then the EI and information gain criteria to output the final proposal. This way, they combine the selection of predicted promising points in the asymptote with those that will reveal more information of their evolution by being trained one extra iteration. The evolution of different training curves is assumed to be independent. This means that one GP per learning curve has to be learned, involving more parameters than in the simple formulation and making the whole estimation process more complex. Hence, the computational time spent on the calculation of this surrogate function also increases.

Domhan et al. focus on Deep Neural Networks and model their learning curves with a different strategy than in the aforementioned GP-based method [43]. Instead of relying on Gaussian Processes to model these learning curves, they use a list of parametric models of varied form, which they then combine using a weigthed sum. Both the weights of this sum and the parameters of these models have to be estimated during the training process. The resulting learning curve model is used to predict the probability that the evolution of a specific training run will eventually yield a better result than the best one seen so far. If this probability is less than a threshold value, the training run for the corresponding configuration is terminated and the predicted loss upon termination is given as feedback to the surrogate model. This strategy is agnostic to the method used to give hyperparameter proposals. Indeed, the authors evaluate the results of this strategy using both SMAC and TPE. It is important to note that a termination of the training process with a low-quality prediction of its loss upon convergence would give poor feedback for these Bayesian methods. That is, a predicted loss far from the actual (unknown) loss upon termination could make those methods perform critically worse, giving bad hyperparameter proposals.

These methods could be parallelized following the same strategy as their corresponding Bayesian method as described in Section 2.3.4. However, this probably would have to operate on the iteration level to be effective. This parallelization would be much

more fine-grained because, instead of the whole training procedure, only single training iterations would be executed simultaenously.

2.3.6.2 Modelling losses across dataset size

The other main resource to be considered when training machine learning algorithms is the size of the dataset. Two other combined methods try to extrapolate the results after training on the full dataset by using the results after training on subsets of it, while also employing a surrogate function for proposing configurations.

Swersky et al. (2013) introduce an algorithm that actually has a more general objective than the one described above [44]. It is a general-purpose multi-task bayesian optimization technique based on Gaussian Processes. Its main idea is to learn correlations between tasks which help to extrapolate knowledge from evaluating one task to the other tasks of interest. This is facilitated by a covariance function that correlates results on different tasks for a particular configuration. This function is learned during the optimization procedure by means of slice sampling and is then combined with the usual covariance function correlating configurations. They cast the dataset sampling problem in their framework by considering one task, minimizing the loss after training on the full dataset, as the primary one to be learned. This task is however costly to directly evaluate, while evaluating on smaller subsamples is cheaper. Therefore, minimizing the loss for each subset is considered as a secondary task, in a way that evaluating a configuration with subsets helps to extrapolate its behaviour with the full dataset. Of course, the larger the subsample is, the more correlated these evaluations are expected to be. The authors propose the information gain per cost acquisition function. The cost in this case corresponds to the time spent in evaluating a configuration for a particular dataset size, which is predicted beforehand using an indepentent GP. The method works by suggesting which configuration to evaluate and on which task (here, subsample size) to evaluate. There are more applications of this multi-task approach which are interesting for hyperparameter optimization. For example, evaluating with cross-validation while not evaluating every single fold for each configuration, but instead estimating its result from the evaluation of a few folds. Also, transferring knowledge from previous optimization runs, executed on problems which are related to a current problem of interest for which there is no feedback yet (trying to solve the "cold start" problem).

Klein et al. also base their method, dubbed FABOLAS, on using a Gaussian Process function as a surrogate [3]. In contrast to the previous technique, this one is exclusively

designed for learning the quality of configurations on subsamples of a large dataset and extrapolating that knowledge. This allows the authors to add prior information of the performance across subsample size (which, in general, evolves in a regular fashion) through an appropriate kernel. This prior knowledge reduces the number of total evaluations needed on the full dataset in order to generalize (possibly even without performing any evaluation on it). The acquisition function that selects the hyperparameters to evaluate and on which subsample size to do it also trades off information gain (using the entropy search acquisition function) against computational cost (predicted using an independent GP as in the previous method). This method also differs from the previous one in that it models subsample size as a continuous variable rather than a set of discretized sizes. This characteristic allows the model to automatically select any arbitrary subsample size. Another small detail which is different is how the computation cost is defined. It does not only take into account the evaluation time spent for a configuration, but also the time spent by the surrogate function to propose that configuration.

The parallelization of these two methods can also be done in the same way as simple GP-based methods, as described in Section 2.3.4. As opposed to the previous section, the granularity of the parallelization remains the same, on the evaluation level (though possibly with smaller dataset sizes and therefore smaller computation time).

2.3.6.3 Adaptive evaluation with Bayesian proposals

Recently, a new approach has been studied in the literature: to combine existing adaptive evaluation methods (such as Hyperband and Successive Halving) with Bayesian methods to propose the points evaluated with those procedures. The method this thesis introduces in Chapter 4 also uses this idea. These methods differ in how they give feedback to the Bayesian method, which loss function is modelled by it and which particular Bayesian method and evaluation method are combined. These methods can be parallelized in the same way as the adaptive evaluation procedure they use, after receiving simultaneous proposals in an appropriate way from the corresponding Bayesian method.

Falkner et al. introduced BOHB, one of the first approaches in this direction [45, 46]. They combine the Hyperband adaptive evaluation strategy with the TPE Bayesian method (with slight modifications). Concretely, they use TPE to propose a fraction of configurations that Hyperband evaluates. Apart from those, Hyperband also evaluates a certain fraction of random configurations. Then, TPE is built with the feedback from evaluations performed on the largest available budget for which enough evaluations

have been made. Hence, the goal of TPE in this case is to predict the performance of the configurations proposed on the largest available budget. Wang et al. also combine TPE with Hyperband for the particular case of neural network hyperparameter optimization [47]. However, they do so in a different way. The feedback given to TPE is the loss achieved by a proposed configuration on the smallest budget available, rather than on the largest one. This has one benefit, which is that all the proposed configurations contribute to give feedback to the Bayesian model. On the other hand, TPE's objective in this case is to propose configurations which minimize the loss for the smallest budget. Conversely, the real objective of this task is to minimize the loss for the largest budget. Hence, these proposals might be systematically different from the best possible ones for the largest budget scenario.

Two other approaches do not restrict themselves to combining TPE with Hyperband. The first one, presented by Wistuba, combines a modification of Successive Halving with both SMAC and a GP-based method [48]. Instead of using the same budget for every halving iteration, Wistuba assigns a larger budget to initial rounds, decreasing it exponentially for later rounds. Moreover, similarly to the previous case, the feedback given for the Bayesian method is the loss achieved before the first elimination of configurations is performed. Therefore, the objective of the Bayesian surrogate is minimizing the loss on the smallest budget. Finally, Bertrand et al. combine a GP-based surrogate function with Hyperband [49]. The authors do not formally define how they are combined, though they mention that the surrogate function is trained on the results of "all evaluated models". Therefore, it might be the case that losses achieved with different budgets are used as feedback. This can cause potential problems because some configurations will be compared in an unfair way. Since some configurations are evaluated with less resources than others, they will obtain higher losses than they would with larger resources.

2.4 Previous empirical comparisons

In Section 2.3, we have reasoned about the theoretical strengths and weaknesses that different techniques have according to their formulation. However, we did not explore empirical results comparing these algorithms. In this section, we will summarize the empirical results presented in the literature in order to further justify which methods to implement. Furthermore, this will help to understand the reasoning behind new potential techniques for the large-scale regime, such as the one described in Chapter 4, and

to connect with the experiments from Chapter 6.

Bergstra and Bengio [1] perform experiments in their work to compare random search and grid search for a neural network performing classification on variations of the MNIST dataset [50] with and without preprocessing the data (preprocessing introduces additional hyperparameters). They show that random search requires considerably fewer trials to achieve the same accuracy as grid search and that for the same number of trials it results in a significantly higher accuracy. However, they also present how a sequential manual optimization process performed by an expert, helped by multi-resolution grid search and coordinate descent, can outperform random search in some problems. Specifically, they show this for various high dimensional experiments (tuning 32 hyperparameters) with deep belief networks. Their final conclusion is that random search is significantly more efficient than grid search and systematically finds better hyperparameters for an allocated number of trials. Moreover, they suggest that, instead of grid search, random search should be considered as a baseline for hyperparameter optimization performance. Apart from that, they also perform experiments on certain synthetic datasets to see how well different methods cover a given subspace of interest (with no machine learning algorithm involved). There, they show that Quasi-Random point sets, also called low-discrepancy sets, yield better results than purely random points. The advantage of these methods, such as Sobol sequences, is that they cover the space in an almost uniform way where each subspace has approximately the same number of points [51]. At the same time, these points are not dimension-aligned as is the case in grid search. Hence, they conserve the benefits of random search for hyperparameter optimization, while providing a progressively better coverage of the space as new sequence points are added (without losing their uniformity).

The original papers from TPE [18] and Spearmint [22] show how their methods outperform both random search and a human expert manual search across different machine learning tasks. They also show that, for Bayesian methods, configuration proposals for a single iteration get better over time (which obviously does not happen for random search as it does not adapt to past observations). Consequently, we can expect to observe a larger difference between random search and Bayesian methods as more iterations are performed. The first one also presents how TPE is superior to a custom Gaussian-Process based approach (which has significant differences to Spearmint) for a high dimensional space including a conditional hyperparameter. The second paper, however, shows Spearmint to clearly outperform TPE in two low-dimensional problems with numerical hyperparameters: a synthetic optimization problem (Brain-Hoo function) and a logistic regression algorithm performing classification on the MNIST dataset

[50]. Moreover, Eggensperger et al. compare TPE, SMAC and Spearmint in a range of problems with different types of hyperparameters (continous, categorical and conditional), different datasets and algorithms [52]. They show how for low-dimensional and continuous spaces, Spearmint has the best performance. However, for large and highly conditional spaces such as tuning hyperparameters of neural networks and deep belief networks, SMAC has the best performance. TPE only outperforms the other two methods for one experiment out of fifteen, being close to SMAC in that case. They also prove how the multi-objective capability of SMAC can bring advantages when cross-validation is used as an evaluation method. SMAC outperforms Spearmint with this evaluation technique in a low-dimensional problem with a continuous hyperparameter space where Spearmint was better using a simple evaluation technique. This is the case because SMAC evaluates only one fold at a time. Therefore, it considers roughly $K-1$ times more configurations than the other algorithms in the same computational time, where K is the number of folds of cross-validation (though with less confidence in the total loss due to only using one fold). This last study, as well as the TPE paper, also mention that GP-based techniques require a time in the order of minutes to output a proposal after several iterations (100-200). On the other hand, the overhead of both TPE and SMAC remains negligible (lower than a few seconds).

The extensive Auto-WEKA work conducted by Thornton et al. tries to fully automate the whole selection of a machine learning pipeline [53] (including pre-processing and feature selection, a field recently named "AutoML" [54]). For this purpose, they use hyperparamter optimization algorithms to find the best possible configuration out of all the possibilities to perform classification present in the WEKA software suite [55]. With 39 learning algorithms and 11 feature selection methods, this results in an extremely high-dimensional configuration space (considering at most 10 values for each hyperparameter results in more than 10^{47} hyperparameter settings). Additionally, this space is highly conditional and has a large proportion of categorical hyperparameters (as a consequence of selecting between so many learning algorithms). They compare the performance of TPE and SMAC in this task, probably not considering GP-based methods due to their disadvantages when handling categorical and conditional hyperparameters. Consistently with the previous study, they show that SMAC outperforms grid search (which had roughly 100 times more CPU hours allocated), random search and TPE in the large majority of the 21 datasets evaluated. In the cases where SMAC was not the best, grid search was mainly the best performing method (which, again, used two orders of magnitude more CPU hours than SMAC) and the datasets employed were relatively small.

Li et al. (2016) perform experiments with a variety of machine learning problems, datasets and resources considered [34]. They show how Hyperband outperforms TPE, Spearmint, SMAC and random search in terms of hyperparameter quality achieved per resource (and therefore, per time). Additionally, these experiments evaluate SMAC's combination with the second adaptive evaluation technique described in Section 2.3.6.1 (using iterations as a resource) and random search with twice the resources. Hyperband also outperforms these two methods. In fact, in the majority of cases, Hyperband is an order of magnitude faster than most or all of the methods evaluated. Especially striking is the experiment where dataset size is used as a resource. In it, Hyperband finds a good configuration after 70 minutes, whereas all the other methods have found worse configurations after more than 700 minutes. However, they do not mention whether all these methods are run sequentially or with some parallelism, but it is likely that the sequential version was used. Also, we can observe that Hyperband does not clearly outperform one Successive Halving run with the most aggressive configuration (i.e., with the highest number of evaluated configurations and the smallest resource per configuration). Indeed, in many cases Hyperband is slightly worse than it. This fact is also corroborated by another study by Li et al. [40]. There, we can observe how making Hyperband propose a configuration after each SH halving round ("Hyperband by rung") also outperforms FABOLAS (described in Section 2.3.6.2) in almost every experiment where they are compared. The observations described in this paragraph are one of the main inspirations of the novel method introduced by this thesis in Section 4.

The papers presenting combined adaptive configuration and evaluation methods described in sections 2.3.6.1 and 2.3.6.2 all show how they increase the speed of their adaptive configuration counterparts. Precisely, Klein et al. [3] show how FABOLAS (their method) outperforms that of Swersky et al. (2013) [44] and, to some extent, Hyperband. Hyperband takes a significant amount of time until it outputs a first proposal. However, its first proposed configuration is already of very high quality and is at least not worse than the best configuration found by FABOLAS at that particular time. Following the "Hyperband by rung" strategy would probably lead to the same results mentioned in the previous paragraph: its performance might be, at least, not worse than FABOLAS. The observation that combining adaptive configuration methods with adaptive evaluation strategies results in an improvement is the other main inspiration of the algorithm introduced in Section 4.

Falkner et al. (2018) evaluate their BOHB approach in a range of machine learning problems [46]. They show how BOHB outperforms simple Hyperband in the majority of problems, especially as the number of iterations gets larger (converging to better

solutions) while being relatively similar at the beginning (and in some cases worse). However, one experiments also compares BOHB to FABOLAS on the standard MNIST dataset in terms of valdiation error, where BOHB does not show superior performance to FABOLAS. In fact, BOHB is also outperformed by simple Hyperband in that experiment. Wang et al. show how their own TPE and Hyperband combination can lead to better results than TPE and Hyperband alone for neural networks [47]. In general, however, the difference between simple Hyperband and their combination is relatively small. The combination proposed by Wistuba does not achieve significant improvements compared to simple SH in the problems evaluated, probably due to the aggressive reduction of budget as iterations advance [48]. Finally, Bertrand et al. show, for an experiment tuning the hyperparameters of a neural network, how their combination outperforms Hyperband and a GP-based Bayesian optimization method separately [49].

Chapter 3

Data-parallel systems

Very large dataset sizes can exceed the processing capabilities of traditional computer systems, such as conventional database systems. This situation started to be very common during the last decade with the advent of Big Data, with an exponential growth in the number of data sources, quantities of data and speed at which it is generated. As an example, it has been estimated that, as of 2016, 90% of the digital data worldwide had been created in the previous two years [56]. To be able to handle this, existing computing systems need to scale. There are two approaches to scalability: To improve the resources of a single computer (scale-up) or to employ more computers of similar characteristics (scale-out). In the first case, the computer is a single point of failure. Additionally, the cost of improving the computer's resources increases in an exponential manner, and this improvement is limited by the progress of hardware technology. On the other hand, the cost of adding more computers to solve a problem increases linearly, and the disruption caused by a failure of one of those computers can be mitigated by techniques such as replication. Therefore, the preferred choice to scale resources is the second one, for example setting these computers to function as a computer cluster.

Data-parallel systems were introduced to perform efficient computations in this situation. In many cases, this can be necessary because a single system does simply not have enough storage capacity. In others, this is desirable in order to process the data much faster by using several processing units at the same time. While traditional parallel programming environments can be used for this purpose, they, in general, need tailored solutions for each particular problem which are complex to program [57]. In contrast, data-parallel systems were designed to offer flexible platforms which are general enough to be able to easily solve a large range of problems. Also, they intend to make the parallelization almost transparent to the programmer. These systems distribute data across

the nodes (machines) that are part of a cluster, possibly replicating each data partition for fault-tolerance. To process the data, each machine executes similar operations simultaneously on different partitions, finally combining the computations performed by each node to return a result. The first of these systems to be widely used was Hadoop [58], based on the map-reduce paradigm introduced by Dean et al. [59]. The idea of this paradigm is to partition the data across machines and perform the same operations across different data partitions giving key-value pairs as a result (map phase). Then, these results are combined by key in a specific way (reduce phase). This strategy makes it easy to abstract a solution from the resources used at a particular time and to provide fault-tolerance. However, Hadoop's programming model is somewhat restrictive and its performance is not optimal, especially for iterative processing tasks (which are common when training machine learning algorithms). Apache Spark [5] was designed to solve these problems, providing a flexible yet high-level programming language with improved performance by making use of in-memory computing and optimizing iterative processing computations. Apache Flink [6] is another system which brings similar benefits and which pays more emphasis on data streaming, improving Spark for those tasks.

Both Spark and Flink include machine learning libraries to be able to leverage data-parallel processing for performing or executing machine learning algorithms on large datasets. However, Spark includes an implementation of a wider range of algorithms and is the most popular choice for large-scale machine learning with batch datasets (as opposed to streaming), which is the focus of this thesis. Therefore, Spark is the system chosen to implement some of the algorithms from Section 2.3 and to evaluate them in this thesis. This section will further focus on Spark and its machine learning library to understand the implementation details and characteristics that are important for hyperparameter optimization algorithms.

It is important to note the difference that these systems imply for hyperparameter optimization as compared to traditional parallelism strategies (which are usually the focus in the literature). Those traditional parallel solutions are focused on performing independent training and validation phases simultaneously in each node with different hyperparameter configurations. For that, they assume that the full dataset is available in every node. Hence, the speedup brought by this strategy is based on the evaluation of several configurations in the same wall-clock time that a single hyperparameter evaluation takes in one machine. This strategy is commonly referred to as model-parallelism, since independent models are handled by separate machines. In contrast, data-parallel systems explicitly make different parts of the dataset available for different machines. As stated before, data-parallelism can be necessary because the dataset does not fit in

a single machine, or desirable because a single evaluation can be executed faster in this way. Therefore, the speedup in this case is a result of performing the computations necessary to train and validate a single machine learning model simultaneously (e.g. calculating the gradient with respect to training instances from different partitions at the same time). This results in smaller wall-clock time for the evaluation of a single hyperparameter configuration. For these reasons, data-parallel systems shift the focus of scaling hyperparamter optimization algorithms from strategies adapted for model parallelism to strategies adapted for optimal sequential evaluations.

3.1 Apache Spark and MLlib

As we have seen, Apache Spark was designed to be an optimized, general purpose data-parallel system which abstracts the parallelization details and resources from the programmer. At the same time, it provides fault-tolerance, making it resistant against failures in cluster's resources without having to restart computations from the beginning. Spark has interfaces for Java, Scala, R and Python. Its data model is based on the Resilient Distributed Dataset (RDD) abstraction, which is a representation of data potentially distributed across different nodes of the cluster. Currently, Spark has four libraries providing different functionalities that leverage data-parallel processing to operate in large datasets, which can be combined in the same execution. Spark Streaming [60] provides streaming capabilities, while GraphX [61] is focused on large-scale graph processing. SparkSQL [62] is a library providing a structured and semi-structured database functionality, allowing SQL-like interaction as well as including a domain-specific language for interaction. It introduces the DataFrames abstraction, based on RDDs, which represents a dataset with schema.

MLlib [7] is the distributed machine learning library of Spark, which uses (since Spark's version 2.0) the DataFrame abstraction of SparkSQL to train algorithms and predict with them on distributed data. It offers a wide range of feature extractors, transformers and selectors for different use cases, such as dimensionality reduction. Furthermore, it includes optimized implementations of several classification, regression, clustering and collaborative filtering algorithms. A chain of pre-processing components (such as feature transformers) and machine learning algorithms can be easily built with the pipeline abstraction, which facilitates the specification of end-to-end computations from raw data to predictions. Additionally, it includes an interface to perform basic hyperparameter optimization through a user-defined grid search, either using a fixed train-validation split or using cross-validation. This thesis' implementation of a selection of algorithms from Section 2.3 is built on top of MLlib. Its main purpose is

to extend the hyperparpameter optimization capabilities of Spark with state-of-the-art methods which have shown a clear superiority over grid search. We would like to tune our pipelines with respect to the largest dataset size available in order to improve our results. Depending on the application, even relatively small improvements derived from better hyperparameter values can result in large economic benefits. Given that Spark is able to process very large datasets more efficiently than traditional systems, MLlib is a very suitable candidate for this task.

We can illustrate how machine learning algorithms can be executed on distributed data in a faster way by observing the example of gradient descent, an optimization algorithm. Gradient descent can be used to train certain machine learning models, such as logistic regression or neural networks, by updating their inner parameters in an iterative way. Gradient descent calculates the gradient of the error of each training datapoint with respect to the parameters and averages the results. To update the existing parameters, it sums the averaged gradient multiplied by a certain factor (step-size) to them. Clearly, since each datapoints' gradient is calculated independently, the gradients can be calculated in any order before accumulating and averaging their results in one gradient descent iteration. This is an ideal scenario for data-parallelism and MLlib takes advantage of it. MLlib's gradient descent implementation can perform gradient calculations simultaneously on each data partition. Hence, ideally, the speedup of its parallel execution is equal to the number of workers used (given enough data partitions).

MLlib still allows to employ model-parallelism by making use of multi-threading when several cores are available in each worker. This can be implemented, for example, by using the Future abstraction present in the Scala programming language. This results into every worker performing computations for more than one model concurrently in its own data partition. Nevertheless, this means using less resources for training a single model. Therefore, the speedups brought by model-parallelism can be diminished by proportionally larger training times for each single model, especially with large datasets.

Chapter 4

Bayesian Geometric Halving

This thesis proposes a novel hyperparameter optimization algorithm, Bayesian Geometric Halving (BGH), which is based on methods from Section 2.3. BGH falls under the category of combined adaptive configuration and evaluation methods. Its idea is similar to the one of the methods presented in Section 2.3.6.3. It avoids trying to model the behavior of the loss achieved with different resources by means of a certain function. Instead, it is inspired by Successive Halving, having a similar adaptive evaluation strategy. Additionally, the configurations evaluated are proposed by a Bayesian method instead of being randomly sampled. We have seen in Section 2.4 that random configurations are systematically worse than the ones proposed by Bayesian methods. The main reason for combining these two approaches is that the better the proposed configurations are, the lower the loss of the best evaluated configuration after each iteration is expected to be. At the same time, the benefits from the adaptive evaluation procedure remain. BGH is agnostic to the method used to propose configurations, though obviously different methods will result in different performances depending on the quality of the method and the type of problem.

As we have examined in Section 2.4, Hyperband does not show better results across different experiments than SH with aggressive setups. For this reason, Hyperband was not considered as an option to perform adaptive evaluation, focusing on the simpler idea of Successive Halving instead. This is a noticeable difference as compared to three of the methods from Section 2.3.6.3, which base their combination on Hyperband. Besides, as explained in Section 2.3.5.1, Successive Halving uses the same resource r for each round before halving the number of configurations. This results in $r \cdot n$ total resources being used, where n is the number of halving rounds (including the last evaluation of the best performing configuration). Therefore, it uses n times more resources than evaluating a

single configuration with resource r (the usual proceeding for algorithms without adaptive evaluation). Although we have described ways to parallelize Successive Halving, this might not necessarily translate to a faster execution. As argued in Section 3, data-parallel systems are a very favorable option if we want to handle large quantities of data for training our machine learning algorithms. In these systems, the model-parallel approach does not bring significant improvements. Training more models in parallel does not increase the training speed of a single model. In contrast, training a single model in a data-parallel system can be faster than doing so sequentially, because computations are performed simultaneously for a single run.

Building upon these observations, the adaptive evaluation method proposed tries to improve performance by evaluating in a similar way to Successive Halving and Hyperband. At the same time, it tries to keep a minimum impact in the evaluation time as compared to methods without adaptive evaluation. This is possible because its evaluation time does not significantly change with a variable number of configurations evaluated (as it is the case for the other two methods). To achieve this, BGH halves the number of configurations evaluated between successive rounds while doubling the resources used per round. This is an important difference as compared to the only method combining Successive Halving and Bayesian optimization from Section 2.3.6.3. That method assigns exponentially more budget to the first iterations and is not able to achieve significant experimental improvements compared to SH. Supposing a finite number of rounds I and a maximum budget of B assigned to a single configuration, this strategy results in a total computation budget T as calculated in Equation 4.1:

$$T = \sum_{i=1}^{I} n_i \cdot b_i \cdot B = B(1 \cdot 1 + 2 \cdot \frac{1}{4} + 4 \cdot \frac{1}{16} + 8 \cdot \frac{1}{64} + 16 \cdot \frac{1}{256} + ...) = B(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ...) < 2B \tag{4.1}$$

Each iteration i corresponds to a single halving round (starting backwards from the last iteration with only one configuration). The fractions b_i correspond to the proportion of the maximum budget B used in round i for a single configuration. These proportions are multiplied by the number of configurations n_i evaluated in that round i. The addends are a finite part of an infinite geometric series with absolute convergence and whose sum is $2B$, hence the inequality on the right. This means that BGH's evaluation procedure requires less than twice as many computations as a single evaluation with full resources. This is the case regardless of how many halving rounds there are and, by extension, regardless of how many initial configurations. Note that it is possible to achieve this geometric series by different means than doubling resources and halving configurations. Any rate r of budget increase between successive iterations along with a rate of $\frac{2}{r}$ for

reducing the number of configurations has the same result. However, we will focus on halving configurations in order to increase the budget less aggressively across iterations. This way, we have a conservative elimination of bad configurations which takes maximum advantage of the total budget.

As this thesis focuses on large-scale data, the experiments with BGH will focus on using dataset size as a resource. This allows us to apply the method in a way which is suitable for any machine learning algorithm (not only iterative ones). As an assumption, we can estimate that the complexity of evaluating a configuration is at least linear in the number of training datapoints. This way, with the budget corresponding to the number of training datapoints, the adaptive evaluation execution satisfies Equation 4.1 with respect to time. This makes its sequential execution less than twice as costly in terms of time than the last round alone (which evaluates a single configuration with the full dataset). Certain methods (e.g. SVM) can have a higher complexity, which still respects this upper bound (by a larger margin). In comparison to SH, this approach uses smaller subsamples for a single configuration in each corresponding round, while eliminating bad performing configurations in the same way. However, we could expect smaller dataset sizes not to affect the adaptive evaluation performance very significantly. Examples like the one analyzed by Klein et al. [3] show small subsample sizes (e.g 1/128) to be good representatives of the behavior of different hyperparameter configurations on the whole dataset. Using random dataset subsamples for each SH round has an interesting consequence. The configurations which are finally proposed have performed well after being trained on a range of different data partitions. To some extent, this mimics the cross-validation procedure (although the validation data is kept constant), and could provide some help against overfitting.

BGH's adaptive evaluation procedure has another interesting feature. When execution times vary greatly for different hyperparameter configurations, BGH will only spend a large amount of time evaluating expensive configurations if they are worth it. In the case that some bad performing configurations are expensive to evaluate, this can be an important advantage compared to random search and simple Bayesian methods. SH exhibits a similar behavior, but spends a larger proportion of its total execution time in the first halving rounds. Therefore, it spends more time in evaluating bad and expensive configurations before they are discarded.

We have seen how BGH performs adaptive evaluation of configurations. Now, let us see how it adaptively proposes configurations to be evaluated. As we have mentioned,

this method is agnostic to the particular Bayesian technique used to propose hyperparameters. To provide feedback to these methods, BGH only uses the loss of the proposed configuration trained using the whole dataset. There is not an exact way to provide a comparable loss as feedback for the rest of the configurations. They have been evaluated with smaller dataset size and, in general, the losses achieved with less training data are systematically higher. This procedure can be seen as a waste of resources, because only one of the multiple configurations evaluated in each iteration is providing feedback to the model. However, it avoids having to predict a loss not actually evaluated (which could be inaccurate) like methods from Section 2.3.6.2 do. Moreover, the frequency at which the Bayesian method receives feedback is, at worst, twice as slow as in the basic Bayesian optimization loop (due to equation 4.1). It would be possible to give the loss achieved with the smallest dataset as feedback for every configuration. Nevertheless, the objective of the Bayesian method with this strategy would be to propose configurations with optimal loss on the smallest dataset. In contrast, the real objective of the optimization procedure is to find configurations with optimal loss on the largest dataset available. Algorithm 4 formally describes BGH.

Algorithm 4 Bayesian Geometric Halving

Input: Maximum budget for a single configuration evaluation B, configurations per trial n, number of trials T, acquisition function S, initial model M_0, loss function l
$H = \emptyset$
for $t = 1, 2...T$ **do**
    $X_0 = x_1...x_n = S(M_{t-1})$
    for $k = 0, 1...\lceil log_2(n) \rceil$ **do**
        Run each x_i in X_k with budget $b = \frac{B}{|S_k| 2^{\lceil log_2(n) \rceil - k}}$ and find its loss $l_b(x_i)$
        $L_k = \text{sort } (l_b(x_0)...l_b(x_{|S_k|-1}))$ in ascending order
        $X_{k+1} = \{i : l_b(x_i) \in \text{First } \lfloor \frac{|S_k|}{2} \rfloor \text{ elements of } L_k\}$
    $x_p =$The only element of $X_{\lceil log_2(n) \rceil}$
    $H = H \bigcup (x_p, l_B(x_p))$
    Fit a new Model M_t to H
return H

In this work, two Bayesian methods for proposing configurations in BGH are evaluated: SMAC and Spearmint. For SMAC, the configuration proposals can be simply taken as the top N configurations listead by its usual search procedure. On the other hand, we have seen that there are several options to output multiple proposals with GP-based methods. The choice natively implemented in Speramint, based on sequentially simulating the outcomes of pending configurations based on the current posterior distribution, might lead to low exploration. Instead, BGH favors a more explorative strategy because only the loss of one configuration is given as feedback. Hence, the drawbacks of exploring too much in some of the configurations proposed are mitigated because they

do not result in useless feedback. Otherwise, this feedback opportunity could have been better utilized by other configurations that exploited current knowledge (to better learn the shape of the loss with respect to the hyperparameters). Following this reasoning, this thesis will explore GP-based BGH with the use of the constant liar strategy (see Section 2.3.4). The exact value used as a lie is the mean loss observed so far as justified in Chapter 5.

It is still possible to parallelize this method in the model-parallel setting with shared data. Each adaptive evaluation execution can be parallelized either synchronously or asynchronously in the same way as Successive Halving. This approach, however, is constrained by the evaluation with the full dataset. Only one model is evaluated in that round, so it imposes an important bottleneck. This last round spends more than half of the adaptive evaluation budget, so this parallelization can only result in a speedup smaller than 2. Another possibility with more potential is to parallelize the whole evaluation loop, executing n evaluation runs simultaneously, each in a different worker. In order to do this, n times more proposals have to be made by the Bayesian method in each iteration. Apart from that, those proposals have to be assigned to different workers in a certain way. It would be desirable that the quality of the proposals is similar for different workers (so that the outcome of each evaluation loop is as good as possible). Also, that the quality of the proposals is as dissimilar as possible within a particular worker (to be able to differentiate them easily with the halving procedure). Bayesian methods can give a list of proposals qualitatively ordered, for example, by EI (as in SMAC) or by chronological order of proposals (as in GP with the constant liar strategy). Assigning them in a round-robin fashion based on their position in this list would be a sensible approach. This way, the first n proposals would be assigned to n different workers. Additionally, the difference in index between two consecutive proposals from the same worker would also be n.

Chapter 5

Implementation details

We have seen in Section 2.4 how both Spearmint and SMAC exhibit the best performance for different type of problems. TPE, however, is not clearly better than both of these methods in any of the scenarios where they have been compared in the literature. For this reason, we decide to implement SMAC and Spearmint on top of Spark's MLlib as the most promising Bayesian hyperparameter optimization strategies. Moreover, we have also seen how Hyperband does not significantly improve Successive Halving. Therefore, we implement Successive Halving as well as Bayesian Geometric Halving, the novel method this thesis proposes in Chapter 4. We also implement and evaluate the simple random search strategy to provide a baseline for these methods. This thesis focuses on dataset size as a resource. As a result, we skip the implementation of the methods from Section 2.3.6.1 because they focus on iterations as a resource. The methods from Section 2.3.6.2 are also not implemented, as they have not been shown to outperform adaptive evaluation methods (e.g. Hyperband by rung [40]) in previous empirical studies. Moreover, they have a complex formulation which makes their implementation intricate. The implementation of every method has been built on top of MLlib's version included in Spark 2.3.1, its latest stable release as of September 2018. We implement these methods in Scala, the language in which Spark is originally written, using its version 2.11.

SMAC uses random forest to model the performance of hyperparameters. Therefore, we use Spark's implementation of random forest, which makes SMAC's implementation relatively simple. The hyperparameters set for the random forest model are the ones described by Hutter et al. [13] and chosen by default in its recent Python implementation [63]. These values are 10 for the number of trees, 5/6 for the feature ratio to consider in each node (from the original paper) and 20 for the maximum depth of each tree (from the Python implementation, as it is not defined in the paper). Finally,

the original paper suggests to set the minimum number of training instances in a node before splitting to 10. However, this hyperparameter is not configurable in MLlib's random forest implementation. For this reason, we set the minimum number of instances in a resulting child node to 3 (as in the aforementioned Python implementation) which should have a comparable effect. In order to find the configurations which maximize the acquisition function, we implement the same local search procedure as described in Section 2.3.3.1. However, there is a difference as compared to the standard optimization strategy. Instead of evaluating the acquisition function on all the previously evaluated configurations, we evaluate the acquisition function for 1000 grid points (defined from a Sobol sequence). Then, we pick the 10 most promising configurations evaluated to start a local search, as it is done in the original strategy. This results in a similar optimization strategy to the one Spearmint uses. Furthermore, for the use-case of only proposing a single configuration per iteration, we decide to give every second proposal as a random point. This way, half of the configurations evaluated are randomly proposed, simiarly to the strategy from SMAC's original paper. Evaluating these random points gives unbiased feedback to the surrogate model. As a result, the search is less likely to fall in local optima and to fail to evaluate other regions of the hyperparameter space.

Spearmint's implementation is more complicated than in the previous case because Gaussian processes are not included in MLlib. The first Python Spearmint implementation [21] was taken as a reference. However, after performing initial experiments, we introduce slight modifications as compared to this implementation. The implementation of this thesis runs 10 Markov chain Monte Carlo (MCMC) iterations instead of 20 in order to marginalize over the hyperparameters (after discarding the first 100 samples as burn-in during the first iteration like in the original implementation). Moreover, the number of candidate points from the original grid (defined from a Sobol sequence as for SMAC) chosen to be optimized through a local search is 10 instead of 20. The reason behind these two modifications is to reduce the computation time needed in practice by the model to propose a new configuration. In early exerpiments, we observed that the model required approximately one minute to propose a new configuration after 100 iteations, so this was a concern. These two modifications reduce the computation time to approximately one forth of the original one. Half of the computations were necessary to calculate the acquisition function and the number of points optimized is also reduced to half. In practice, the surrogate function spends most of its execution time in optimizing these points. Similar to the original Spearmint implementation, our implementation executes concurrently the optimization of these points, as it is independent for each point. The calculation of the EI for the points on the initial grid was observed to have a releant contribution to the surrogate's execution time. Consequently, we also perform this calculation concurrently. We run the LBFGS-B algorithm to perform

the local search to optimize promising points (as in the original implementation). For that, we use the LBFGS-B implementation from Breeze, a numerical processing package for Scala [64]. Apart from this algorithm, Breeze was also utilized for performing the numerical computations necessary to implement the GP-based surrogate model, such as matrix multiplication and inversion. Another difference is the fixed-range uniform prior distribution used for sampling lengthscale values. The upper limit of this range is changed from the original value of 2 to a value equal to the square root of the number of dimensions (i.e., number of hyperparameters being tuned). The reason is that the lengthscale was observed to behave in a counter-intuitive manner with inappropriate upper limit values given the number of dimensions. For example, with only one or two dimensions and a maximum value of 2, long lengthscales were given high probability and sampled frequently. This, in turn, resulted in a downgrade of Spearmint's performance. The square root of the number of dimensions is equal to the maximum possible length between two points in our normalized space (with ranges between 0 and 1 for each hyperparameter). This value was chosen because the lengthscale controls the relative influence between two points according to their distance. In preliminar experiments, this value was observed to give similar or better empirical results.

In the implementations of BGH and SH, the training dataset used in each round is randomly sampled from the full training dataset. These samples are performed at the beginning of the hyperparameter optimization execution, meaning that they are the same for each iteration during the adaptive evaluation procedure. Moreover, every configuration is trained with exactly the same sample within a given halving round (i.e., for a given dataset size). The evaluation dataset is the same across every iteration and dataset size, since our objective is to optimize the results on it at the end of the execution. This can mean that, for rounds with a small dataset subsample, the evaluation dataset might be bigger than the training dataset. In principle, we could think that this would cause a significant overhead during those rounds. However, the evaluation phase typically is orders of magnitude faster than the training phase. As a result, the overhead of the evaluation phase is insignificant in practice. For the usage of SMAC in BGH, we evaluate the top proposals from its candidate list. SMAC was initially designed to propose multiple configurations at a time for evaluation, so this does not necessarily have to be changed. To propose multiple points with Spearmint in BGH, we use the constant liar approach, using a "lie value" equal to the mean of the losses seen so far. Empirically, it was observed that the difference between the maximum and the mean value for some problems is extremely large. Even though using the maximum value results in more exploration (which is favoured by BGH), this made the model fail to efficiently exploit its knowledge for subsequent proposals. Using the mean, instead,

usually makes this value decrease over time, resulting in more exploitation as the iterations advance. At the same time, it still has a good amount of exploration, especially during initial stages. This is intuitively appropriate because we have more certainty of the model's performance as iterations advance. Therefore, we would like to rely more on Spearmint's predicted good areas during final stages.

In Chapter 4 we have seen how the adaptive evaluation strategy of BGH should result in less than twice the execution time of a single evaluation with full resources. However, this limit can be violated in practice due to various overheads, a behavior observed during preliminary experiments. When the dataset size that each partition holds is relatively small, communication time can dominate computation time. That is, the largest proportion of the execution time is spent in sending information between machines and cores. This causes the training time to grow slower than the dataset size (because the communication time does not necessarily increase), making Equation 4.1 not hold. Other additional overheads coming from the management and coordination of the workers may also contribute to this behavior. For larger datasets, computation time largely dominates communication time, therefore we observe a linear increase in computation time with respect to dataset size. In order to overcome this problem for our experiments, the training and testing of each configuration was performed concurrently during each halving iteration (except, obviously, for the last one with only one configuration). We implemented this employing Scala's Futures. This strategy resulted in larger training times for individual configurations within a halving round (because they have less resources available). Nevertheless, in the case of small dataset sizes, the overall execution time for that halving round was significantly shorter. For larger subsamples, the execution time of a halving round with this strategy was practically the same. Evaluating, for example, two configurations concurrently means that each of those configurations needs twice as much execution time as evaluating them sequentially. Note that for really large datasets (e.g. 100 GB) this concurrent execution would not be necessary unless there are many halving rounds (and therefore its smallest subsample is very small, e.g. below 500MB). In that case, even for their smallest subsamples, computation time would largely dominate communication time. It would have been possible to perform experiments with much larger datasets in order to avoid the aforementioned problem. Nevertheless, those datasets imply larger total training times, making those experiments infeasible. We want to evaluate many iterations for a given method, to compare several methods and, in some cases, to execute those several times for statistical reliability. Additionally, publicly available large datasets are relatively rare.

As already mentioned, this thesis focuses on large datasets and therefore dataset

size is taken as a resource for BGH and SH. Yet, it could also be interesting to evaluate the behavior of those methods with iterations as a resource for optimizing iterative algorithms. However, implementing this in MLlib is not trivial. Machine learning algorithms (Estimators in MLlib) are implemented in such a way that they are given a training dataset as an input and return a final model (Transformer). In consequence, there is no access to intermediate steps of their training process. Therefore, it is difficult to halt a training execution of these algorithms at a certain iteration and evaluate their result on a separate test dataset. After that, it would be necessary to provide feedback to the surrogate model and restart the training phase of another arbitrary configuration. Since MLlib is not designed for stopping a training execution and restarting it later, this procedure would probably be not optimized. There is always the possibility of restarting the training execution from the beginning in each halving round. Nonetheless, this wastes a lot of time performing computations that have already been performed in previous halving rounds, so it is not a sensible approach.

The code including the implementations of random search, SMAC, Spearmint and the adaptive evaluation procedures of BGH and SH can be found in the following GitHub repository: `https://github.com/Aitor4/thesis-code`

Chapter 6

Experiments

Four experiments compare the most promising algorithms we have identified for the large-scale setting: random search, Spearmint, SMAC, Successive Halving, Bayesian Geometric Halving with random proposals (which we refer to as RGH, for Random Geometric Halving), BGH combined with Spearmint and BGH combined with SMAC. The first experiment only compares SMAC, Spearmint and random search. The purpose of this experiment is to show the importance of an appropriate scale for certain hyperparameters. However, the dataset employed is relatively small, so we do not compare the algorithms designed for large datasets in it. The rest of the experiments compare every algorithm mentioned above. Successive Halving and Bayesian Geometric Halving evaluated 8 different configurations per iteration, meaning that they run 4 halving iterations with 8, 4, 2 and 1 point respectively. For every experiment, the same random sample with 80% of the original dataset was used as training dataset, with the remaining 20% used as validation dataset. In every hyperparameter optimization run, the first configuration proposed is randomly sampled in order to provide an initial feedback to Bayesian methods (which they require to operate further). The values of the hyperparameters which are not optimized for each algorithm during the experiments (if any) correspond to the default values defined in MLlib for Spark 2.3.1.

In every plot, the y-axis shows the best metric achieved on the validation dataset by any configuration evaluated until that point. The x-axis shows either the number of iterations or the execution time after a given configuration was evaluated. The number of iterations is used in Section 6.2 because the execution of BGH did not satisfy Equation 4.1 with the resources utilized. Using larger dataset sizes or fewer resources makes this equation hold, but it would result in very time-consuming experiments. In this setting, SH is only allowed to propose a new configuration and validate it every 4 iterations

(because, in principle, its execution time is 4 times larger than Bayesian methods). Similarly, BGH is allowed to do so every 2 iterations (because its execution is, at most, twice larger). The reason to do this is to simulate the expected behavior with respect to time when optimizing a larger dataset or employing fewer computational resources. The plots of sections 6.1 and 6.2 show the average result after executing the experiments 5 times per algorithm. This way, we take advantage of their faster execution time to have a more statistically reliable result. Appendix A includes plots representing the standard deviation of the results from these experiments. For the plots of sections 6.3 and 6.4, the experiments were only run once due to the very large execution times needed.

Table 6.1 summarizes the characteristics of the datasets utilized in these experiments. The second dataset is used in two different experiments, while the other two datasets are used in one experiment each.

Dataset	**Size**	**# of instances**	**# of features**	**Task**
MovieLens 100K [65]	$\sim$ 2 MB	100,000	4	Coll. filtering
Susy [66]	$\sim$ 2.5 GB	5,000,000	18	Classification
MNIST8M [67]	$\sim$ 12 GB	8,100,000	784	Classification

TABLE 6.1: Characteristics of the datasets used

6.1 Alternating least squares

We will first present the results of running three Bayesian hyperparameter optimization algorithms for the hyperparameters of the alternating least squares (ALS) learning algorithm implemented in MLlib. ALS is an algorithm to perform collaborative filtering, a technique used in recommender systems to predict unseen user-item entries (e.g. ratings of a movie) based on the available ones. The dataset chosen for this task is the MovieLens 100K dataset [65], composed of 100,000 ratings of 1700 movies from 1000 users, and of approximately 2MB of size. Even though its size is relatively small, this algorithm is quite time-consuming. Therefore, employing a large dataset was unfeasible due to time constrains, especially because we wanted to average the results over several runs. Jamieson et al. also performed an experiment for the optimization of the hyperparameters of a collaborative filtering algorithm on this dataset to evaluate Successive Halving [37]. The main purpose of this experiment is to show the impact that different range transformations can have for Bayesian optimization algorithms. Additionally, it compares these algorithms in a new problem where they had not been compared before in the literature.

This experiment optimizes three hyperparameters: one integer (rank), one binary (whether to apply nonnegativity constrains) and one continuous (regularization) hyperparameter. Jamieson et al. [37] search for the rank in a linear range of [2,50] and for the regularization hyperparameter in a logarithmic range of [10^{-6},1]. We follow the same approach for one experiment, while performing a second one exploring the regularization parameter in a linear range of [0,1]. Figure 6.1 and Figure 6.2 show the performance of random search, SMAC and Spearmint in this optimization problem with the two variations described.

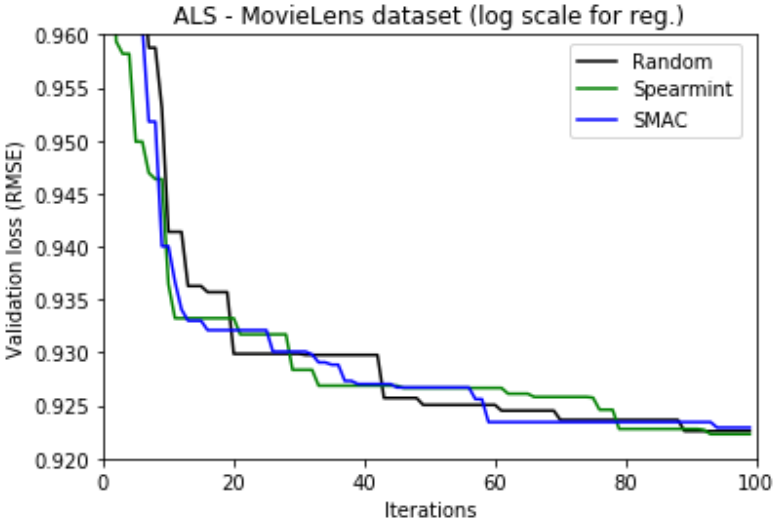


FIGURE 6.1: ALS with logarithmic transformation for the regularization range

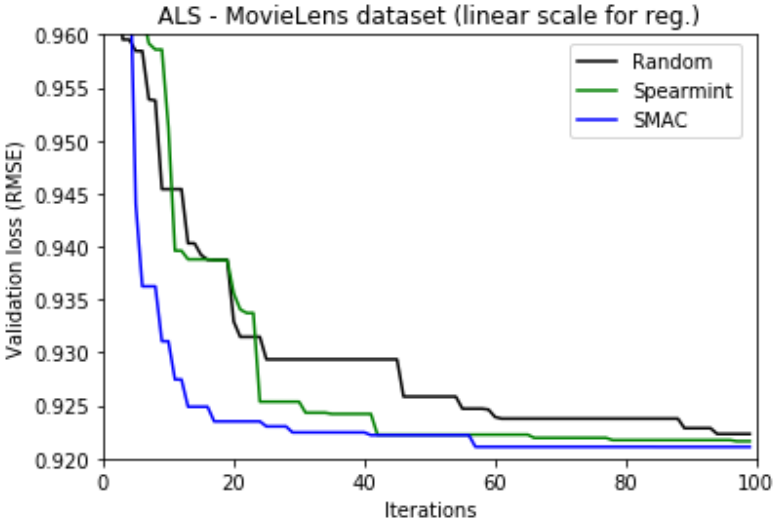


FIGURE 6.2: ALS with linear regularization range

We can see that, for a logarithmic range for the regularization hyperparameter, neither SMAC nor Spearmint outperform random search. This can be explained by the relative importance of the different hyperparameters. The best results in these experiments corresponded to configurations where the regularization parameter fell within the value range of [0.07,0.15]. When this hyperparameter was out of this range, none of the configurations had a relatively good result. After a logarithmic transformation, this promising range is transformed to a smaller range (length smaller than 0.04) and results in larger fluctuations of the validation error in its surroundings. This transformation makes it difficult for SMAC and Spearmint to propose values from this range and attribute the good results to it. For that reason, they cannot outperform random search in the first experiment, while they clearly show better performance in the second one. In the second experiment, they find a configuration with better validation error after executing 100 iterations. Also, after 42 iterations, they find a better configuration than random search does and than they did in the previous experiment after 100 iterations. Random search, on the other hand, presents no significant differences between the two experiments. This result shows the importance of finding good range transformations for an optimal behavior of hyperparameter optimization algorithms and motivates further studies such as Snoek et al. [11].

When the regularization hyperparameter was within the aforementioned range, the impact of the other hyperparameters was relatively small, though significant for very similar regularization values. This can explain why, even though there is no explicit conditional hyperparameter, SMAC is able to beat Spearmint searching in a linear range of the regularization hyperparameter. This difference is especially significant during the first iterations, where SMAC finds a better configurations after 20 iterations than Spearmint does after more than 40 iterations. In practice, only when the regularization value is within a restricted range, the other two hyperparameters matter. SMAC is able to learn this "artificial conditionality" faster and then explores different values of the other two hyperparameters given a good regularization value. Note, however, that as iterations advance (after the 40th iteration), Spearmint has relatively similar results to SMAC. Therefore, we can observe how Spearmint needs a larger number of iterations than SMAC to learn the shape of the hyperparameter space. Yet, after a while, Spearmint explores this space in a similar way.

6.2 Decision trees

We examine another hyperparameter optimization problem with decision trees, a popular classification algorithm, and the SUSY dataset [66]. This dataset has a size of approximately 2.5 GB and includes 5 million instances, each composed of 18 features. It presents a binary classification problem where the objective is to distinguish a signal process producing physical supersymmetric particles and a background process.

In this case, four hyperparameters are tuned: a boolean variable indicating which impurity measure to use (Gini or entropy-based), an integer that indicates the maximum depth of the tree (in a linear range of [3,15]), a continuous variable defining the minimum information gain for a split to be considered (in a linear range of [0,1]) and an integer indicating the minimum instances per child node for a split to be considered (in a linear range of [1,10]). Figure 6.3 shows the results of seven different hyperparameter optimization algorithms on this problem: random search, SMAC, Spearmint, Successive Halving and Bayesian Geometric Halving combined with the first three methods to propose configurations.

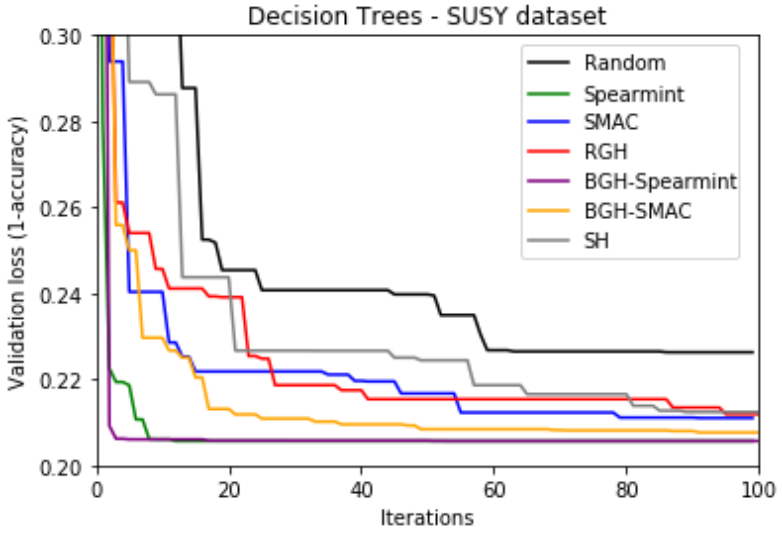


FIGURE 6.3: Decision Trees on Susy dataset

We can observe many interesting details in this experiment. Random search is, as expected, the worst performing method, having a worse result after 100 iterations than all the other methods after 25. BGH improves the results of all the methods it uses to propose configurations with their respective combination. SH also improves random search (which implicitly is the method it uses to propose configurations). RGH

outperforms SH, finding a better configuration after 40 iterations than SH does after 80. After 100 iterations, however, they converge to similar results. SMAC has slightly better results than RGH and SH, having found a better configuration than them during the majority of the iterations. BGH combined with SMAC shows significantly better results than SMAC, finding a better configuration after 25 iterations than SMAC does after 100 iterations. The best hyperparameter values are found on the boundaries of the range for the minimum information gain (bottom) and minimum instances per child (top). The optimal maximum depth of a tree was between 13 and 14, which is close to its top boundary (15). Spearmint tends to initially explore the boundaries of the search space, which can explain why it finds good values relatively fast and converges early. As a consequence, Spearmint outperforms both random search and SMAC, finding the best configuration after less than 10 iterations. BGH combined with Spearmint greatly improves Spearmint's results during the first few iterations, finding this best configuration after less than 5 iterations. By that time, Spearmint is still exploring the hyperparameter space without as much success.

6.3 Logistic regression

This section presents an experiment performed using a large-scale dataset and optimizing hyperparameters from the logistic regression classification algorithm. This dataset is an augmented version of the MNIST dataset [67] (sometimes referred to as MNIST8M), generated by applying transformations to the images of that original dataset. It has 8.1 million instances with 784 features and a size of almost 12GB. It presents a classification problem with 10 possible classes (images of digits from 0 to 9). The hyperparameters tuned are a regularization continuous hyperparameter (in a logarithmic scale of $[10^{-8},1]$), an elastic net continuous hyperparameter (in a linear scale of $[0,1]$) and a boolean indicating whether to fit an intercept term. The execution of each method was only performed once due to the large execution times needed (around 11 hours for each method). Figure 6.4 shows the results of random search, RGH, SMAC and its combination with BGH, Spearmint and its combination with BGH and SH. As in the previous cases, random search, SMAC and Spearmint were run until proposing 100 configurations, RGH and BGH until proposing 50 configurations and SH until proposing 25 configurations.

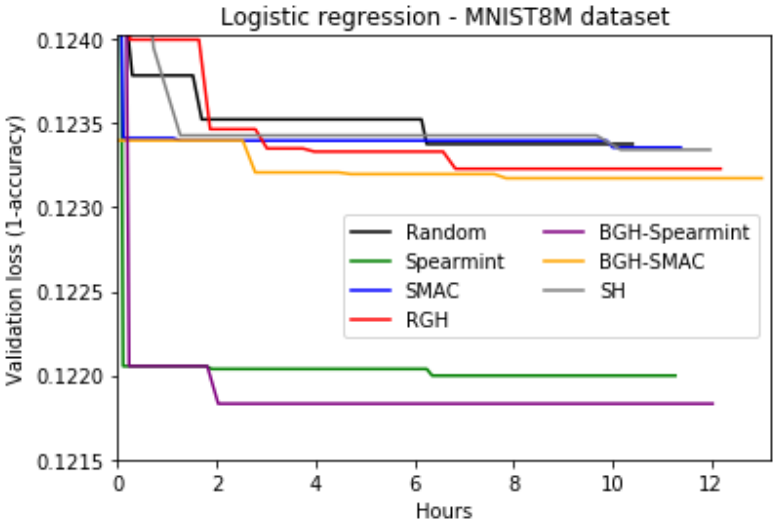


FIGURE 6.4: Logistic regression on the MNIST dataset

RGH improves the results of random search, especially as iterations advance, finding a better result after 3 hours than random search does after more than 10. SH does not improve random search much, only being significantly better during the first 6 hours. SH has worse performance than RGH, as its result after almost 12 hours is worse than that of RGH after 3 hours. Note that since the configurations tried are purely random in random search, SH and RGH and the execution of these experiments has only been done once. Hence, their comparative results get more statistically reliable over time, as they have tried a larger number of configurations. SMAC does not improve the results of random search at the end (around 11 hours), while having better results at the beginning. After some minutes, it finds a configuration almost equally good as the one random search finds upon termination, being significantly better during the first 6 hours. However, SMAC is not able to significantly improve its initial results after receiving feedback. Nevertheless, its combination with BGH achieves similar loss values in the first 2 hours but does improve its initial results over time. BGH combined with SMAC finds a better configuration after 3 hours than any of the aforementioned methods finds upon termination. The best hyperparameter configurations had the elastic net term with a value of 0 and the intercept term choice as true. Given these values, the best configuration had the regularization term with a value of approximately 10^{-6}, with similar errors for values in that order of magnitude. Spearmint is more effective than SMAC at finding configurations at the boundaries of the search space. Therefore, Spearmint is able to learn the best values of the previous two hyperparameters and finely search for optimal values of the regularization term given those. SMAC, in contrast, has more troubles exploring boundary values and then it is not able to fine-tune the regularization term in the same way. Spearmint achieves much better results than the aforementioned

methods after only a few minutes. However, similarly to SMAC, it does not improve much its own result over time. BGH combined with Spearmint takes a few minutes longer to match the best initial result of Spearmint. However, it is able to improve this result after slightly more than 2 hours, finding then a significantly better configuration than any other method upon termination.

In this experiment, we can notice that not every method finishes the execution of its corresponding iterations at the same time. Let us remember that different hyperparameter configurations can result in different training (and, therefore, evaluation) times. In these experiments, very bad performing configurations (that usually resulted from a regularization hyperparameter value very close to 1) resulted in significantly smaller execution times. We can expect worse performing methods, especially random search, to spend more time evaluating with such configurations. Adaptive evaluation methods discard them in their first rounds, while Bayesian methods learn to avoid those values over time. We can see how SMAC takes longer time than random search to finish and Spearmint takes slightly longer than SMAC. Adaptive evaluation methods evaluate good performing configurations in more rounds, spending more execution time on them. Coherently, SH, RGH and BGH (both combined with Spearmint and with SMAC) take longer than the other methods to finish. It is possible that the overheads associated with evaluating configurations on smaller datasets also contribute to these results, making Equation 4.1 not hold exactly with respect to time. Nevertheless, most of the iterations empirically fulfilled that equation, which was one of the main purposes of this experiment. Another additional overhead present in Bayesian methods and BGH is the time the surrogate model spends to propose points. However, even the most expensive proposal (made in the last iterations of BGH combined with Spearmint) required less than 10% the time needed to evaluate that proposal. Overall, this proportion was significantly smaller, making it practically negligible. The differences in execution time are not very significant, as the slowest method (BGH-SMAC) is less than 30% slower than the fastest one (random search) to finish its iterations.

6.4 Classifier choice as a hyperparameter

The last experiment includes a conditional structure in the hyperparameters. For it, we use the SUSY dataset as we did in Section 6.2. In this case, however, the choice of the classifier to use is a hyperparameter. The hyperparameter optimization procedure chooses between using logistic regression or support vector machines (SVM) to solve this problem. Apart from this hyperparameter, we tune three hyperparameters for logistic

regression and three hyperparameters for SVM. The ones corresponding to logistic regression are exactly the same as in Section 6.3, with identical ranges. For SVM, the hyperparameters tuned are a continuous regularization hyperparameter (in a logarithmic scale of $[10^{-8},1]$), a boolean indicating whether to standardize the features and a boolean indicating whether to fit an intercept term. Furthermore, the execution of different datasets for these algorithms scaled approximately as Equation 4.1 with respect to time. Therefore, we show runtime on the x-axis of our plots. This is a notable difference as compared to Section 6.2, where we used the same dataset. The reason is that both logistic regression and SVM are significantly more costly to execute than decision trees, so the overheads have less impact. Figure 6.5 shows the results of Spearmint and SMAC with and without imputation for the hyperparameters which are not active (i.e., the hyperparameters corresponding to the classifier which has not been chosen). We impute these values in the same manner as Lévesque et al. [10] do in their experiemnts, giving the middle value of the range as feedback for each inactive hyperparameter. We run this initial experiment in order to choose which version is the best for subsequent experiments with BGH. It also allows us to compare the impact of the imputation strategy employed in this problem.

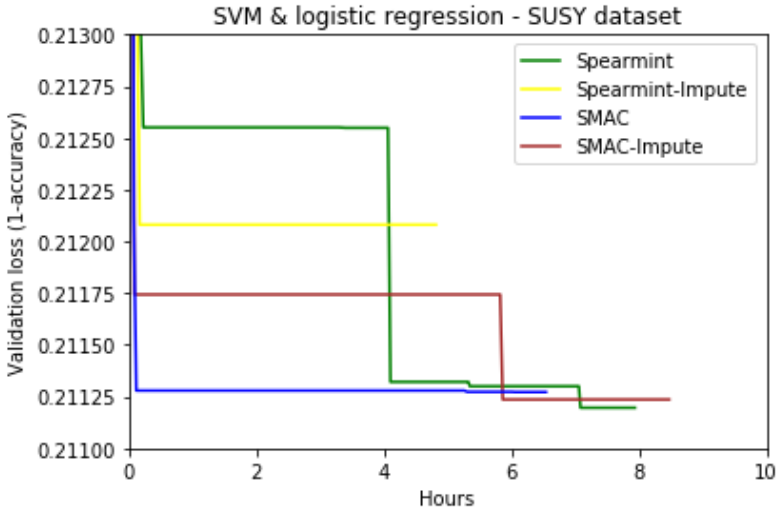


FIGURE 6.5: Results with classifier choice (SVM or logistic regression) on the SUSY dataset, with and without imputation of inactive hyperparameters

It is important to note that logistic regression and SVM had significantly different running times when they were chosen. SVM required around three times longer execution time than logistic regression to be evaluated. Even though choosing SVM gives relatively good results without being affected much by its own hyperparameters, logistic regression gives the overall best results. At the same time, logistic regression gives the overall worst results with bad values of its own hyperparameters, as it is more sensitive to them than SVM. The differences in execution time of the hyperparameter optimization

algorithms in this experiment can be explained by the different proportion of classifier choices they make. Algorithms which choose SVM more frequently are slower, due to its longer execution time. SMAC without imputation is able to find good configurations in a few minutes, although after 6 hours SMAC with imputation finds slightly better configurations. Nevertheless, we will consider SMAC without imputation as better due to the large differences until that point as compared to the small differences after it. Spearmint without imputation has relatively bad performance during the first 4 hours, when Spearmint with imputation outperforms it. However, after 4 hours, Spearmint without imputation finds much better results, being close to those of SMAC. In contrast, Spearmint with imputation does not improve its initial result over time, having much worse performance after 4 hours. Examining the cause of this behavior, we saw how Spearmint with imputation constantly tried to propose the same hyperparameter configuration where inactive hyperparameters had extreme values. The feedback given to the model always imputes inactive hyperparameters to the middle value of the corresponding range. Hence, evaluating this proposal did not decrease the variance of that configuration and caused Spearmint with imputation to continue to propose the same values. Figure 6.6 shows the same results of the algorithms without imputation shown above as well as BGH combined with them, random search, RGH and SH.

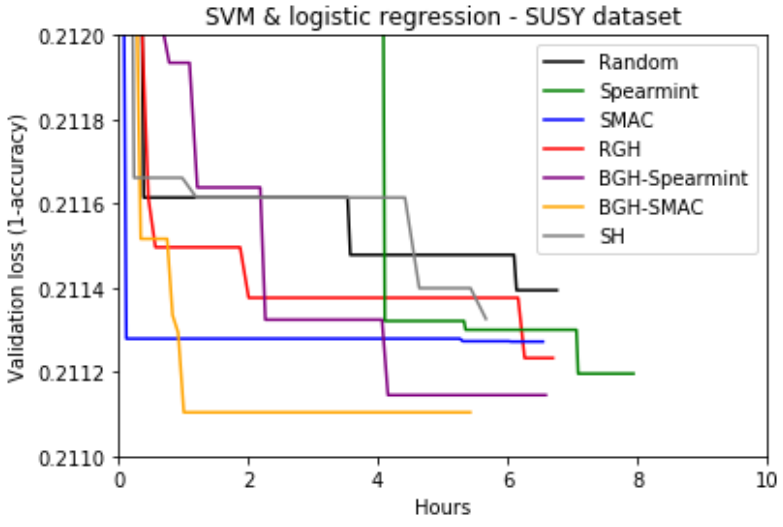


FIGURE 6.6: Results with classifier choice (SVM or logistic regression) on the SUSY dataset.

SH did not improve the results of random search during the first 4 hours, while outperforming it after less than 5 hours. After 5 hours, SH had better results than RGH for some minutes. However, soon after that point RGH found a significantly better configuration. Moreover, RGH had much better performance than SH during the first 4 hours. Note that random methods have more variability in their results (see Appendix

A), so their results are less reliable in this experiment because they had only been executed once. However, the large differences during the first 5 hours between RGH and SH makes us conclude that RGH did perform better overall in this example. SMAC is better suited to handle conditionality than Spearmint, as we argued in Section 2.3. Moreover, SMAC has been shown in previous studies to perform better than Spearmint in the presence of conditional hyperparameters, as discussed in Section 2.4. Consistently, SMAC has better performance than Spearmint in this experiment, both by themselves and combined with BGH. Similarly to previous experiments, we can observe how BGH improves the results of the strategy it employs for proposing configurations in each case. That is, BGH with SMAC outperforms SMAC, BGH with Spearmint outperforms Spearmint and RGH outperforms random search. In fact, BGH with SMAC and BGH with Spearmint give the best two results upon termination. BGH with SMAC has the overall best result, finding in slightly more than 1 hour a better configuration than all other methods do upon convergence.

SVM scales worse than logistic regression due to its higher complexity. This results in a significantly longer evaluation time for SVM, as previously discussed. This causes the increase in execution time with respect to dataset size to be larger than in Equation 4.1 for SVM. For this reason, BGH and its combinations require proportionally shorter computation time when evaluating SVM as compared to a simple evaluation strategy (it is closer to requiring x1.5 than x2 the time of executing a simple evaluation). Evaluating logistic regression, BGH sometimes required more than x2 the time of a single evaluation (around x2.3) due to overheads with the smallest datasets. Nevertheless, BGH's evaluations of SVM compensated this overhead, so on average one evaluation of BGH took less than a simple evaluation. In fact, the methods with the largest execution time are random search and Spearmint, while SH and all the BGH combinations terminate earlier. This can also be a result of those algorithms choosing logistic regression more or less frequently. We could expect the best algorithms to choose logistic regression more frequently, as it finally gives the best results. However, we have seen in Figure 6.5 how Spearmint with imputation had significantly bad performance while requiring shorter execution time. As we discussed, it chose logistic regression very frequently but with bad values for its hyperparameters. Therefore, a better validation performance does not always mean a shorter execution time in this experiment. The proportion of classifier choices and the scaling of the adaptive evaluation procedure are responsible for the variability in total execution time for different algorithms. An interesting observation about this experiment is that the best configurations are also significantly faster than others (as opposed to the experiment of Section 6.3). In this scenario, BGH avoids spending large evaluation times with unpromising configurations (i.e., choosing SVM) due to its

adaptive evaluation procedure. Yet, in some iterations (especially at the beginning) it evaluates SVM in its last rounds because no good configuration for logistic regression is proposed.

6.5 Discussion

We can observe some common insights across the previous experiments. Random search had the worst performance in every experiment, only being competitive with other algorithms in logistic regression. The comparative performance of SMAC and Spearmint varied: Spearmint was clearly superior in decision trees and logistic regression while SMAC had better performance in ALS and with classifier choice as a hyperparameter. We discussed how Spearmint performs better when optimal values are in the boundaries of the search space for all or some of the (numerical) hyperparameters being tuned. On the other hand, SMAC performs better when optimizing conditional hyperparameters. These characteristics were corroborated in the results of our experiments.

BGH outperformed each method it utilized for proposing configurations in every experiment where it was evaluated. BGH is designed for experiments which fulfill Equation 4.1 with respect to training dataset size and execution time. As we have reasoned, this should happen with relatively large datasets for the computational resources being utilized. Based on our experiments, we can expect BGH to improve the results of any base method used to propose configurations in similar conditions. Interestingly, RGH outperformed SH in every experiment where they were compared, even if their differences in performance were not very large. The only difference between these two methods lies on the adaptive evaluation procedure, with a more aggressive decrease of resources in RGH. Given a hypothetical combination of SH with Spearmint or SMAC to propose configurations, we can expect BGH to outperform that combination by a larger margin. The reason is that BGH's adaptive evaluation procedure has proposed twice as many configurations as SH has proposed at a given iteration. This implies that, with BGH, Bayesian methods have twice as much feedback at a given time and hence twice as much information available to propose new points. As a consequence, these methods should give better proposals when combined with BGH and lead to better final results. Either BGH combined with SMAC or BGH combined with Spearmint had the best performance in every experiment where they were involved. In decision trees and logistic regression, BGH with Spearmint was the best performing method, while BGH with SMAC had the best result with classifier choice as a hyperparameter. This corresponds to the relative performance of SMAC and Spearmint in those sections: BGH with

Spearmint outperformed BGH with SMAC where Spearmint outperformed SMAC and viceversa. We can expect this behavior to be reproduced in other experiments as well. Finally, the experiment with classifier choice as a hyperparameter proved experimentally the benefits of BGH's evaluation procedure when some bad configurations are expensive to evaluate. This situation can be relatively frequent depending on the meaning of the hyperparamters being tuned.

Chapter 7

Conclusions

This thesis has studied the hyperparameter optimization problem with a special focus on algorithms and how their characteristics adapt to tasks with large datasets. We have seen how some algorithms have, in general, distinct quality (e.g. random search performs worse than Successive Halving), while others perform comparatively different according to the specific characteristics of the task at hand (e.g. conditionality of the hyperparameters). We also have discussed that there are two main scaling approaches to be able to handle problems with large datasets in reasonable time. The model-parallel approach parallelizes the whole configuration evaluation independently in several machines (each of them containing the full dataset). On the other hand, the data-parallel approach distributes the data across machines. Then, it executes a single configuration evaluation at a time by performing operations on different partitions of the dataset simultaneously. The first approach is limited by the dataset size that one single machine can store. Additionally, with that approach, the quality of the Bayesian optimization process gets worse as more machines are added. In contrast, the second approach does not suffer from these problems. Therefore, we focused on data-parallel systems and how they influence these algorithms. We chose Apache Spark to implement the most promising ones because it is a data-parallel system with a an extensive machine learning library, MLlib.

Based on the previous observations and on the characteristics of different algorithms that we analyzed in Section 2.3, this thesis has proposed a new hyperparameter optimization algorithm, Bayesian Geometric Halving. It combines an adaptive evaluation procedure similar to that of Successive Halving while selecting the configurations to evaluate using Bayesian algorithms such as SMAC or Spearmint. Furthermore, it reduces the overhead of the evaluation procedure to a maximum of twice that of a single evaluation independently of the number of halving rounds. Therefore, this overhead is

independent of the number configurations evaluated per iteration even when executed sequentially (which is optimal for data-parallel systems). In contrast, Successive Halving has an overhead equal to the number of halving rounds and Hyperband has the overhead of SH multiplied by its number of brackets.

Section 6 described experiments on a variety of problems with different datasets, algorithms and hyperparameters where the implemented algorithms are compared. For the adaptive evaluation procedures, we used training dataset size as a resource, coherently with the focus on large-scale data. Random search was used as a baseline method and showed, as expected, the worst performance across experiments. We were able to observe some characteristics which affect the comparative performance of Speramint and SMAC. Depending on the type of hyperparameters and on the location of optimal values in the hyperparameter space, one method can perform better than the other. BGH improved the results of Bayesian methods that were combined with it and its variants achieved the best results in the experiments where they were compared. Hence, we can conclude that it is a promising method for the large-scale regime. Additionally, BGH's adaptive evaluation strategy combined with random selection of configurations also showed better behavior than Successive Halving in the experiments where they were compared. The comparative results of BGH with Spearmint and BGH with SMAC are similar to the comparative results of Spearmint and SMAC. That is, when Spearmint performs better than SMAC, BGH with Spearmint is expected to perform better than BGH with SMAC.

7.1 Future Work

There are many possible hyperparameter optimization methods which could be additionally implemented in Spark. An obvious improvement, especially relevant after observing the experiments of Section 6.1, is to extend Spearmint with the input warping strategy presented by Snoek et al. [11] and implemented in a newer version of Spearmint [68]. It could also be interensting to implement FABOLAS [3], as it has been showed to have relatively good performance using dataset size as a resource. Additionally, a potential modification of MLlib's algorithms to be able to easily use iterations as a resource would be an interesting extension.

The space of possible experiments to perform is practically unlimited so there are many possible interesting problems in which hyperparameter optimization algorithms could be executed and compared. This thesis has not evaluated problems with a large number of hyperparameters or very complex conditional dependencies, which are also

relevant. Also, this thesis has not evaluated the optimization of the hyperparameters of neural networks. However, this would probably have to be performed in platforms different than Spark, becase MLlib's multi-layer perceptron implementation is rather limited. Given enough computational resources, dataset size and available time, it would be possible to perform large-scale experiments with several averaged executions (e.g. 10). This would be beneficial in order to provide a more statistically reliable result in the experiments with large datasets. We did not perform experiments analyzing any modification of the algorithms to scale with the model-parallel approach. However, it would be especially interesting to see how the proposed modification of BGH for that approach, running several evaluation loops simultaneously, would perform. Lastly, utilizing different Bayesian optimization methods for different iterations of the same execution could yield promising results in certain situations. This is also an interesting direction to be further explored.

Appendix A

Standard deviations

The following plots show the same results as sections 6.1 and 6.2 but with bars representing the standard deviation of each algorithm. These bars cover the area of the averaged result plus and minus one standard deviation in each iteration. Figures A.1 and A.2 are equivalent to the plots of Section 6.1. Figures A.3 and A.4 divide the plot of Section 6.2 in two. Those two plots show the results of different algorithms for the same experiment in order to avoid occlusions provoked by the error bars.

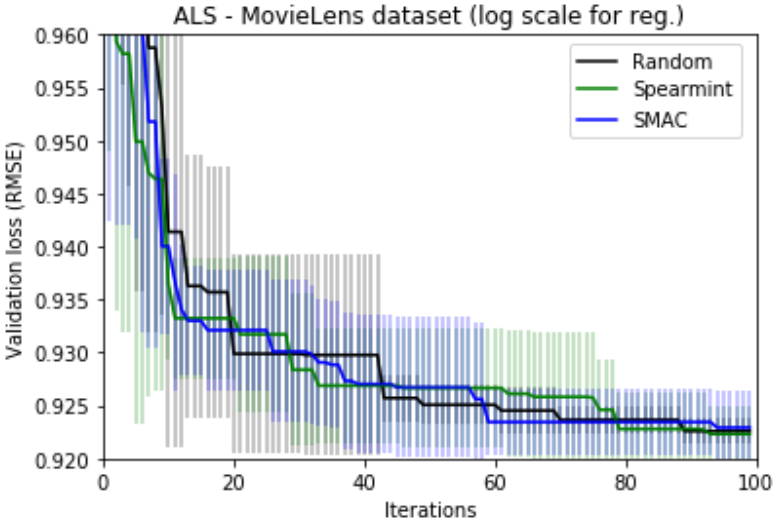


FIGURE A.1: ALS with logarithmic transformation for the regularization range, with standard deviation

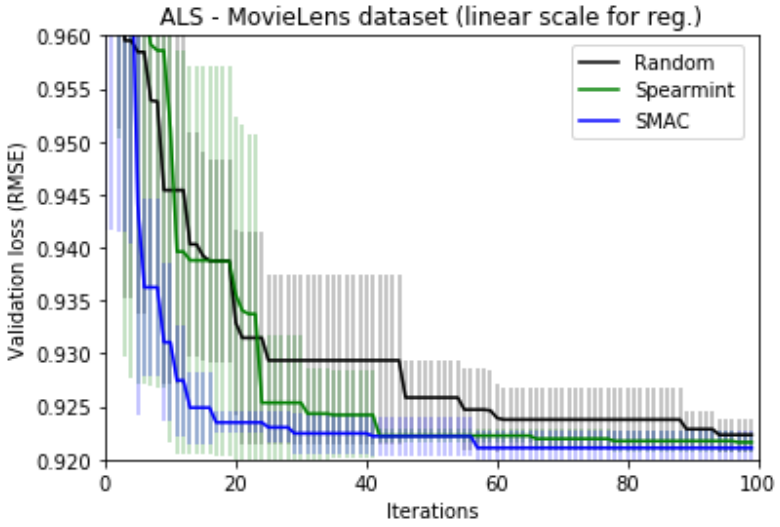


FIGURE A.2: ALS with linear regularization range, with standard deviation

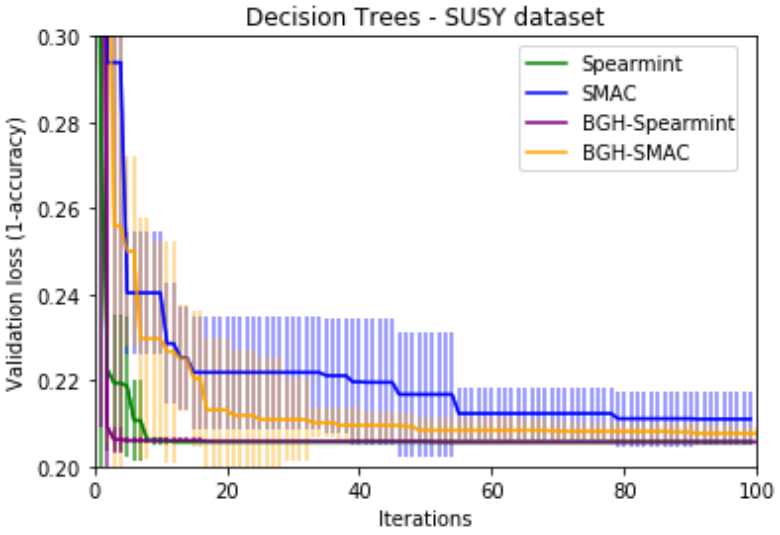


FIGURE A.3: Decision Trees on Susy dataset with standard deviation (I)

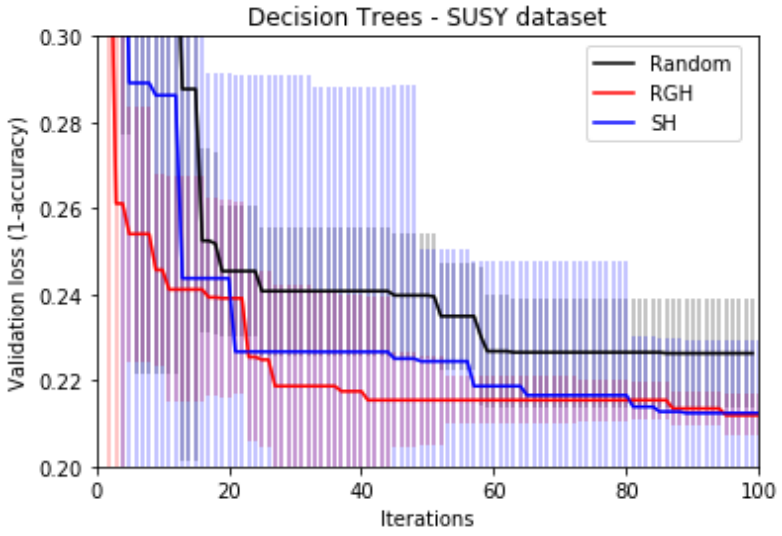


FIGURE A.4: Decision Trees on Susy dataset with standard deviation (II)

In general, we can observe how the standard deviation gets smaller as iterations advance. This shows that these algorithms tend to converge to similar solutions across different runs as iterations advance. Moreover, as expected, the standard deviation of algorithms based on proposing random configurations (random search, RGH and SH) is larger due to their nature.

Bibliography

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[2] 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. *arXiv preprint arXiv:1012.2599*, 2010.

[3] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[5] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[7] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17 (1):1235–1241, 2016.

[8] Spark Hyperparameter Tuning module. https://spark.apache.org/docs/latest/ml-tuning.html. Accessed: 2018-08-24.

[9] Tammo Krueger, Danny Panknin, and Mikio L Braun. Fast cross-validation via sequential testing. *Journal of Machine Learning Research*, 16(1):1103–55, 2015.

[10] Julien-Charles Lévesque, Audrey Durand, Christian Gagné, and Robert Sabourin. Bayesian optimization for conditional hyperparameter spaces. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 286–293. IEEE, 2017.

[11] Jasper Snoek, Kevin Swersky, Rich Zemel, and Ryan Adams. Input warping for bayesian optimization of non-stationary functions. In *International Conference on Machine Learning*, pages 1674–1682, 2014.

[12] NIPS 2014 survey. https://github.com/jaak-s/nips2014-survey. Accessed: 2018-06-21.

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

[14] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

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

[16] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. Real-time human pose recognition in parts from single depth images. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1297–1304. Ieee, 2011.

[17] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4): 455–492, 1998.

[18] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[19] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[20] Carl Edward Rasmussen and Christopher KI Williams. *Gaussian process for machine learning*. MIT press, 2006.

[21] Spearmint package. https://github.com/JasperSnoek/spearmint, . Accessed: 2018-08-26.

[22] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[23] Iain Murray and Ryan P Adams. Slice sampling covariance hyperparameters of latent gaussian models. In *Advances in neural information processing systems*, pages 1732–1740, 2010.

[24] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1): 97–106, 1964.

[25] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[26] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[27] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[28] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015.

[29] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. In *Learning and Intelligent Optimization*, pages 55–70. Springer, 2012.

[30] David Ginsbourger, Rodolphe Le Riche, and Laurent Carraro. A multi-points criterion for deterministic parallel global optimization based on gaussian processes. 2008.

[31] David Ginsbourger, Janis Janusevskis, and Rodolphe Le Riche. Dealing with asynchronicity in parallel gaussian process based global optimization. In *4th International Conference of the ERCIM WG on computing & statistics (ERCIM'11)*, 2011.

[32] Clément Chevalier and David Ginsbourger. Fast computation of the multi-points expected improvement with applications in batch selection. In *International Conference on Learning and Intelligent Optimization*, pages 59–69. Springer, 2013.

[33] David Ginsbourger, Rodolphe Le Riche, and Laurent Carraro. Kriging is well-suited to parallelize optimization. In *Computational intelligence in expensive optimization problems*, pages 131–162. Springer, 2010.

[34] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. 2016.

[35] Xiangxin Zhu, Carl Vondrick, Deva Ramanan, and Charless C Fowlkes. Do we need more training data or better models for object detection?. In *BMVC*, volume 3, page 5. Citeseer, 2012.

[36] Aitor Palacios Cuesta. A comparative study of structured prediction methods for sequence labeling, 2016.

[37] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[38] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*, pages 23–37. Springer, 2009.

[39] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.

[40] Lisha Li, Kevin Jamieson, Afshin Rostamizadeh, Katya Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. 2018.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[42] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[43] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, volume 15, pages 3460–8, 2015.

[44] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013.

[45] Stefan Falkner, Aaron Klein, and Frank Hutter. Combining hyperband and bayesian optimization. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS), Bayesian Optimization Workshop*, 2017.

[46] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

[47] Jiazhuo Wang, Jason Xu, and Xuejun Wang. Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning. *arXiv preprint arXiv:1801.01596*, 2018.

[48] Martin Wistuba. Bayesian optimization combined with incremental evaluation for neural network architecture optimization. 2017.

[49] Hadrien Bertrand, Roberto Ardon, Matthieu Perrot, and Isabelle Bloch. Hyperparameter optimization of deep neural networks: Combining hyperband with bayesian model selection. 2017.

[50] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[51] Il'ya Meerovich Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967.

[52] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013.

[53] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[54] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[55] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[56] 10 Key Marketing Trends For 2017, IBM Marketing Cloud . https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345GBEN. Accessed: 2018-09-07.

[57] William D Gropp, William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[58] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[59] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[60] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[61] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[62] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[63] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, Andr Biedenkapp, and Frank Hutter. Smac v3: Algorithm configuration in python. https://github.com/automl/SMAC3, 2017.

[64] Breeze library. https://github.com/scalanlp/breeze. Accessed: 2018-09-11.

[65] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19, 2016.

[66] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

[67] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. Training invariant support vector machines using selective sampling. *Large scale kernel machines*, 2, 2007.

[68] Spearmint package, newer implementation with input warping. https://github.com/HIPS/Spearmint, . Accessed: 2018-09-17.