Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters

Katharina Eggensperger, Matthias Feurer, Frank Hutter
Freiburg University
{eggenspk,feurerm,fh}@informatik.uni-freiburg.de

James Bergstra
University of Waterloo
james.bergstra@uwaterloo.ca

Jasper Snoek
Harvard University
jsnoek@seas.harvard.edu

Holger H. Hoos and Kevin Leyton-Brown
University of British Columbia
{hoos,kevinlb}@cs.ubc.ca

Abstract
Progress in practical Bayesian optimization is hampered by the fact that the only available standard benchmarks are artificial test functions that are not representative of practical applications. To alleviate this problem, we introduce a library of benchmarks from the prominent application of hyperparameter optimization and use it to compare Spearmint, TPE, and SMAC, three recent Bayesian optimization methods for hyperparameter optimization.

1 Introduction
The performance of many machine learning (ML) methods depends crucially on hyperparameter settings and thus on the method used to set hyperparameters. Recently, Bayesian optimization methods have been shown to outperform established methods for this problem (such as grid search and random search [1]) and to rival—and in some cases surpass—human domain experts in finding good hyperparameter settings [2, 3, 4]. As a result, hyperparameter optimization has become an active research area within Bayesian optimization, with characteristics such as low effective dimensionality [1, 5, 6] and problem variants, such as optimization across different data sets [7] being explored.

One obstacle to further progress in this nascent field is a dearth of hyperparameter optimization benchmarks and comparative empirical studies. It can be difficult to evaluate a new optimizer on benchmarks used in previous papers because (1) optimizers are written in different programming languages and use different search space representations and file formats; (2) hyperparameter optimization benchmarks that have been developed jointly with an optimizer are not typically packaged as black boxes (including the respective machine learning algorithm and its input data) that can be used with other optimizers.

To alleviate these problems, we have collected and made available a library of hyperparameter optimization benchmarks from the recent literature and used it to empirically evaluate the respective strengths and weaknesses of three prominent Bayesian optimization methods for hyperparameter optimization: Spearmint [2], TPE [8], and SMAC [9]. We thereby hope to provide an empirical foundation to facilitate the development and evaluation of future methods for this problem.

2 Bayesian Optimization Methods for Hyperparameter Optimization
Given a machine learning algorithm A having hyperparameters $\lambda_1, \ldots, \lambda_n$ with respective domains $\Lambda_1, \ldots, \Lambda_n$, we define its hyperparameter space $\Lambda = \Lambda_1 \times \cdots \times \Lambda_n$. For each hyperparameter
setting $\lambda \in \Lambda$, we use $A_\lambda$ to denote the learning algorithm $A$ using this setting. We further use $L(A_\lambda, D_{\text{train}}, D_{\text{valid}})$ to denote the validation loss (e.g., misclassification rate) that $A_\lambda$ achieves on data $D_{\text{valid}}$ when trained on $D_{\text{train}}$. The hyperparameter optimization problem under $k$-fold cross-validation is then to minimize the blackbox function

$$f(\lambda) = \frac{1}{k} \sum_{i=1}^{k} L(A_\lambda, D_{\text{train}}^{(i)}, D_{\text{valid}}^{(i)}).$$

We note that an alternative to optimizing hyperparameters is to marginalize over them in a Bayesian framework. Auto-WEKA [3] and has also been used to configure many combinatorial optimization algorithms.

The most popular acquisition function is the expected improvement defined, a term proportional to the expected improvement from Equation 2 can be computed in closed form:

$$E_{\mathcal{M}}[I_{f_{\min}}(\lambda)] = \int_{-\infty}^{f_{\min}} \max\{f_{\min} - f, 0\} \cdot p_{\mathcal{M}}(f \mid \lambda) \, df.$$

One main difference between existing Bayesian optimization algorithms lies in the model classes they employ. In this paper, we empirically compare three popular Bayesian optimization algorithms for hyperparameter optimization that are based on different model types.

**Sequential Model-based Algorithm Configuration (SMAC)** [9, 17]. SMAC uses random forests to model $p_{\mathcal{M}}(f \mid \lambda)$ as a Gaussian distribution whose mean and variance are the empirical mean and variance over the predictions of the forest’s trees. For hyperparameter optimization problems with cross-validation, SMAC evaluates the loss of configurations at single folds at a time in order to save time. Different configurations are compared based only on the folds evaluated for both. SMAC supports continuous, categorical, and conditional parameters. It was the best-performing optimizer for Auto-WEKA [3] and has also been used to configure many combinatorial optimization algorithms.

**SPEARMINT** [2, 18]. SPEARMINT uses a Gaussian process (GP) to model $p_{\mathcal{M}}(f \mid \lambda)$ and performs slice sampling over the GP’s hyperparameters [19]. It supports continuous and discrete parameters (by rounding), but does not provide a mechanism to exploit knowledge about conditional parameters.

**Tree Parzen Estimator (TPE)** [8, 20]. TPE is a non-standard Bayesian optimization algorithm. While SPEARMINT and SMAC model $p(f \mid \lambda)$ directly, TPE models $p(f < f^*)$, $p(\lambda \mid f < f^*)$, and $p(\lambda \mid f \geq f^*)$, where $f^*$ is defined as a fixed quantile of the losses observed so far, and the latter two probabilities are defined by tree-structured Parzen density estimators. With these distributions defined, a term proportional to the expected improvement from Equation 2 can be computed in closed form [8]. TPE supports continuous, categorical, and conditional parameters, as well as priors for each hyperparameter over which values are expected to perform best. It has been used successfully in several papers beyond the one in which it was introduced [4, 21, 3].
Table 1: Hyperparameter optimization benchmarks used. The loss function is the function value for Branim and Hartmann 6d, perplexity for LDA and misclassification rate for the others. The runtime column gives the 0.1 and 0.9 quantiles over all function evaluations performed by all optimizers, in minutes. For benchmarks with crossvalidation these are the runtimes for one fold.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#hyp.params</th>
<th>continuous/ discrete</th>
<th>Dataset</th>
<th>Size (Train/Valid/Test)</th>
<th>Citation</th>
<th>Runtime [min] q0.1/q0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branim</td>
<td>2(-)</td>
<td>2/-</td>
<td>-</td>
<td>-</td>
<td>[22]</td>
<td>trivial</td>
</tr>
<tr>
<td>Hartmann 6d</td>
<td>6(-)</td>
<td>6/-</td>
<td>-</td>
<td>-</td>
<td>[22]</td>
<td>trivial</td>
</tr>
<tr>
<td>Log. Reg.</td>
<td>4(-)</td>
<td>4/-</td>
<td>MNIST</td>
<td>50k/10k/10k</td>
<td>[2, 23]</td>
<td>0.3/12.5</td>
</tr>
<tr>
<td>SVM ongrid</td>
<td>3(-)</td>
<td>/3</td>
<td>UniPROBE</td>
<td>≈ 20k/≈ 20k</td>
<td>[2, 25]</td>
<td>table lookup</td>
</tr>
<tr>
<td>HP-NNET</td>
<td>14(4)</td>
<td>7/7</td>
<td>MRBI</td>
<td>10k/2k/50k</td>
<td>[8, 26]</td>
<td>0.9/13.4</td>
</tr>
<tr>
<td>HP-NNET</td>
<td>14(4)</td>
<td>7/7</td>
<td>convex</td>
<td>6.5k/1.5k/50k</td>
<td>[8, 26]</td>
<td>0.7/16.7</td>
</tr>
<tr>
<td>HP-DBNET</td>
<td>36(27)</td>
<td>19/11</td>
<td>MRBI</td>
<td>6.5k/1.5k/50k</td>
<td>[8, 26]</td>
<td>0.7/46.3</td>
</tr>
<tr>
<td>Auto-WEKA</td>
<td>788(784)</td>
<td>296/490</td>
<td>convex</td>
<td>10 fold cv, 8k Train, 50k Test</td>
<td>[27, 3, 26]</td>
<td>0.4/15.2</td>
</tr>
<tr>
<td>Log. Reg. SCV</td>
<td>4(-)</td>
<td>4/-</td>
<td>MNIST</td>
<td>5 fold cv, 60k Train, 10k Test</td>
<td>[2, 23]</td>
<td>0.3/12.1</td>
</tr>
<tr>
<td>HP-NNET SCV</td>
<td>14(4)</td>
<td>7/7</td>
<td>MRBI</td>
<td>5 fold cv, 12k Train, 50k Test</td>
<td>[8, 26]</td>
<td>0.9/10.8</td>
</tr>
<tr>
<td>HP-NNET SCV</td>
<td>14(4)</td>
<td>7/7</td>
<td>convex</td>
<td>5 fold cv, 8k Train, 50k Test</td>
<td>[8, 26]</td>
<td>0.7/10.7</td>
</tr>
</tbody>
</table>

3 Hyperparameter Optimization Benchmarks

Table 1 summarizes all of the benchmarks we collected for the first version of our hyperparameter optimization library, HPOlib, which is available at www.automl.org/hpolib. HPOlib includes simple test functions (used for convenience in many papers) as well as the following benchmarks:

Low-dimensional benchmarks. We collected three benchmarks with few parameters from [2]: simple Logistic Regression to classify the popular MNIST dataset; Online Latent Dirichlet Allocation (LDA) for Wikipedia articles, and Structured Support Vector Machines (SVM). The latter two benchmarks are defined on a grid of hyperparameter values: for each of the grid points (288 for LDA; 1400 for SVM), algorithm performance data has been precomputed by [2] to allow for very rapid experiments. Another advantage of such precomputed data is that anyone can use these benchmarks without having to compile and run the respective ML algorithms.

Medium-dimensional benchmarks. We collected two types of benchmarks of intermediate dimensionality from [8]. HP-NNET and HP-DBNET implement a simple and a deep neural network, respectively. Both run faster on GPUs than CPUs, and both include continuous and categorical parameters, some of which are conditional. For each hyperparameter in these benchmarks, an expert-defined prior over good values is defined. Since these priors cannot be expressed in SMAC’s and SPEARMINT’s formats, they get lost in translation, as does conditionality in the case of SPEARMINT.

High-dimensional benchmarks. To test the limits of current optimizers, we also investigated the AUTO-WEKA framework [3], which encodes combined model selection and hyperparameter optimization into an enormous hierarchical (i.e., highly conditional) space with 768 hyperparameters. All benchmarks in HPOlib can be accessed through the command line. This both allows for the support of optimizers written in arbitrary programming languages and allows us to control their use of resources. HPOlib includes python scripts that offer a common interface to the three Bayesian optimization algorithms used throughout this paper (and can convert between their input formats). These scripts call the optimizers and wrap their calls to the machine learning algorithm being optimized with a tool called runsolver [28] to ensure that they respect given time and memory limits. If they do not do so (consider, e.g., a call to a neural network setting the number of layers to 1000), they are terminated (using SIGTERM and SIGKILL; for HP-NNET and HP-DBNET, we use the internal termination mechanism and give it an additional 200 seconds grace period to return the quality of its current model). Terminated or otherwise crashed calls to the machine learning algorithm without valid output yield the worst possible result for the hyperparameter setting being evaluated. All calls to the machine learner and their results are stored in a uniform optimizer-independent format, to facilitate the analysis of results. Finally, the HPOlib scripts also allow restarting interrupted optimizer runs from their last state.

Since overfitting is a critical issue in hyperparameter optimization (especially as hyperparameter optimization methods improve), HPOlib supports k-fold cross-validation, either by evaluating all k folds at once, or by evaluating one fold at a time. Out of the benchmarks above only Auto-WEKA used cross-validation, so to study the importance of cross-validation we added additional versions of the Logistic Regression and HP-NNET benchmarks with 5-fold cross-validation.
We ran each optimizer with its default settings 10 times on each benchmark for the number of function evaluations. We also provide results for half the function evaluation budget to quantify the improvement over time. This work introduces a benchmark library for hyperparameter optimization and provides the first Table 2 summarizes our results. Overall, SPEARMINT performed best for the low-dimensional continuous problems. For the higher-dimensional problems, which also include conditional parameters, SMAC and TPE performed better. However, we note that based on the small number of 10 runs per optimizer, many performance differences were not statistically significant.

For benchmarks including k-fold cross-validation, SMAC evaluated one fold at a time, while the other methods (which do not yet support single fold evaluations) evaluated all k folds. SMAC thus managed to consider roughly 4 times more configurations than either TPE or SPEARMINT in the same budget of fold evaluations. Since the budget for each optimizer was expressed as a number of function evaluations, they were not penalized for choosing costly-to-evaluate hyperparameters, and in our experiments, SPEARMINT runs were sometimes up to a factor of three slower than those of TPE; in the future, we plan to use time budgets and SPEARMINT’s time-sensitive EI criterion [2]. We also studied the CPU time required by the optimizers. SMAC’s and TPE’s overhead was negligible (<1 second), but due to the cubic scaling behaviour of its GP’s, SPEARMINT required >42 seconds to select the next data point after 200 evaluations. This would prohibit its use for the optimization of cheap functions, but here this overhead was dominated by the expensive function evaluations.

## 5 Conclusion and Future Work

This work introduces a benchmark library for hyperparameter optimization and provides the first extensive comparison of three optimizers. To support further research, our software package and benchmarks are publicly available at www.automl.org/hpolib. It offers a common interface for the three optimization packages utilized in this paper and allows the easy integration of new ones. Our benchmark library is only a first step, but we are committed to making it easy for other researchers to use and to contribute their own benchmarks and optimization packages.

---

1However, it showed some robustness problems, e.g., crashing on 1 of 10 runs for the HARTMANN-6 function because of a singular covariance matrix. It also had problems with discrete parameter values: by maximizing expected improvement over a dense Sobol grid instead of the discrete input grid, it sometimes repeatedly chose values that were rounded to the same discrete values for evaluation (leading to repeated samples).
References


[17] S. Ramage and F. Hutter. SMAC source code, java version v2.06.01; URL: www.cs.ubc.ca/labs/beta/Projects/SMAC.


