Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau

Efficient Bayesian Hyperparameter Optimization

Aaron Klein

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Dekanin:
Prof. Dr. Hannah Bast
Erstgutachter und Betreuer der Arbeit:
Prof. Dr. Frank Hutter
Albert-Ludwigs-Universität Freiburg
Zweitgutachter:
Prof. Dr. Thomas Brox
Albert-Ludwigs-Universität Freiburg
Datum der mündlichen Prüfung
16.01.2020

“Chaos is order yet undeciphered”
José Saramago

Contents

Abstract

Automated machine learning emerged as a new research field inside of machine learning that tries to progressively automate different steps of common machine learning pipelines which are traditionally executed by humans. One of its core tasks is the automated search for the right hyperparameters of a given machine learning algorithm which in practice is often essential to achieve good performance. Compared to other optimization problems, hyperparameter optimization is usually particularly expensive, since in each iteration, it requires to train and validate the underlying algorithm. One of the most successful approaches for hyperparameter optimization is Bayesian optimization. At its core, Bayesian optimization fits a probabilistic model of the objective function, which together with an additional acquisition function is used to guide the search towards the global optimum.

In this thesis we present several extensions to standard Bayesian optimization to improve its performance for hyperparameter optimization problems. First, we introduce a new probabilistic model based on Bayesian neural networks, that allows to model the performance of hyperparameter configurations across different tasks and thereby scales much better with the number of data points and dimensions than Gaussian processes which are traditionally used inside Bayesian optimization. In hyperparameter optimization, often approximations, so-called *fidelities*, of the objective function are available which are much cheaper to evaluate. We present two new Bayesian optimization methods that can leverage such fidelities, such as learning curves or dataset subsets, to improve the overall search process in terms of wall-clock time by orders of magnitude. Furthermore, based on our proposed Bayesian neural network model, we present a new neural network architecture which models the learning curve of iterative machine learning methods, such as neural networks. Finally, due to the high computational cost of hyperparameter optimization, thorough benchmarking and evaluation of new developed methods is often prohibitively expensive. We show that one can approximate continuous and discrete benchmarks by surrogate benchmarks that capture the characteristics of the original benchmark but take only milliseconds to evaluate. This allows us to perform a rigorous analysis and comparison of various different hyperparameter optimization methods from the literature.

Zusammenfassung

Innerhalb des maschinellen Lernens ist automatisiertes, maschinelles Lernen ein neues Forschungsgebiet. Dabei wird schrittweise versucht, verschiedene Komponenten geläufiger Pipelines, welche normalerweise von Menschen ausgeführt werden, zu automatisieren.

Ein Hauptaufgabengebiet des automatisierten, maschinellen Lernens besteht in der Suche nach den richtigen Hyperparametern eines Lernalgorithmus', was oft ausschlaggebend für eine gute Performanz in der Praxis ist. Allerdings ist die Hyperparameteroptimierung im Vergleich zu anderen Optimierungsproblemen oft sehr zeitaufwendig, da in jeder Iteration der zugrundeliegende Algorithmus trainiert und evaluiert werden muss. Einer der erfolgreichsten Ansätze für dieses Problem ist die Bayes'sche Optimierung. Ihr Hauptbestandteil ist ein probabilistisches Modell der Zielfunktion, welches zusammen mit einer Hilfsfunktion die Suche nach dem globalen Optimum steuert.

Diese Arbeit beschäftigt sich mit verschiedenen Erweiterungen der Bayes'schen Hyperparameteroptimierung, mit dem Ziel, diese effizienter zu gestalten. Zunächst wird ein probabilistisches Modell vorgestellt, welches, basierend auf einem Bayes'schen neuronalen Netzwerkes, die Performanz von Hyperparameterkonfigurationen über verschiedene Aufgabenstellungen hinweg vorhersagt. Das Modell skaliert dabei besser in der Anzahl der Datenpunkten und Dimensionen als vergleichsweise Gaussprozesse, welche traditionell für Bayes'schen Optimierung verwendet werden.

Zudem sind für die verschiedenen Problemstellungen der Hyperparameteroptimierung Approximationen der Zielfunktion, so genannte Fidelities, verfügbar. Diese sind zwar meist ungenau, können zugleich aber sehr viel günstiger ausgewertet werden. Diese Arbeit stellt zwei neue Bayes'sche Optimierungsmethoden vor, die diese Fidelities, wie zum Beispiel Lernkurven oder Untermengen des Trainingdatensatzes, nutzen können, um den Optimierungsprozess zu beschleunigen.

Des Weiteren wird, aufbauend auf den oben genannten Bayes'schen neuronalen Netzwerken, eine neue Netzwerkarchitektur zum Modellieren von Lernkurven iterativer, maschineller Lernalgorithmen, wie zum Beispiel neuronaler Netzwerke, vorgestellt.

Wie bereits erwähnt, ist mit der Hyperparameteroptimierung meistens ein hoher Zeitaufwand verbunden. Dies erschwert häufig das gründliche Vergleichen und Evaluieren verschiedener Methoden in der Praxis. Diese Arbeit zeigt, dass kontinuierliche und diskrete Benchmarks durch Surrogatebenchmarks approximiert werden können. Einzelne Funktionsevaluierungen werden dabei in Millisekunden durchgeführt, die

Charakteristiken der Originalbenchmarks jedoch beibehalten. Eine tiefergehende Analyse sowie der Vergleich vieler verschiedener Hyperparameteroptimierungsmethoden aus der Literatur, konnte somit in dieser Arbeit durchgeführt werden.

List of Figures

List of Tables

List of Algorithms

Acknowledgements

This thesis would not have been possible without the support and help of many people, to whom I am deeply grateful.

First, I would like to thank Frank Hutter, the supervisor of this thesis. Frank always was at hand with constructive feedback and enlightening ideas. Under his guidance I was able to gather all the necessary skills to become an independent researcher. I also would like to thank Thomas Brox, who not only helped with funding most of this thesis but also often provided valuable input. Many thanks also to Joschka Boedecker and Michael Tangermann who served as chair man and observer of this thesis, respectively. Furthermore, I would like to thank Javier Gonzalez, Zhenwen Dai and Neil Lawrence for their guidance during my internship at Amazon Research Cambridge.

During my entire time as a member of the machine learning group at the University of Freiburg, I feel privileged to have worked alongside many great people which supported a stimulating and creative environment. Among them are, in alphabetic order: Noor Awad, Simon Bartels, Andre Biedenkapp, Ayush Dewan, Tobias Domhan, Alexey Dosovitsky, Katharina Eggensperger, Thomas Elsken, Andreas Eitel, Matthias Feurer, Matilde Gargiani, Maria Hügle, Numair Mansur, Ilya Loshchilov, Raghu Rajan, Jan van Rijn, Robin Schirrmeister, Manuel Watter, Abhinav Valada and Arber Zela. Many thanks also to Stefan Stäglich, Uli Jakob and the NEMO team for maintaining the technical infrastructure that was necessary for this thesis. I thank Petra Geiger and Steffi Beysel for their endless patience with me filling out paperwork. Special thanks to Tobias Springenberg, who helped me getting started with deep learning.

Furthermore, I am particularly grateful to Noha Radwan, who was a good friend during the whole time of my thesis. I am also indebted to Stefan Falkner, for his snarky comments and the endless, and often heated, discussions which were often the source of my best ideas and without which this thesis would not have been possible.

Many thanks to Ramon Busch, Philipp Rettke and Michael Helm who helped me to escape the scientific bubble every once in a while. Also thanks to my family for their unconditional support. Finally, I am especially thankful to Leonie who always stood beside me, even when everything else looked grim.

1. Introduction

With the help of an ever-increasing compute power and the availability of larger dataset, machine learning has achieved great success in recent years. On a variety of applications, such as image classification (Krizhevsky et al., 2012), speech recognition (Mikolov et al., 2010), game play (Silver et al., 2016) or machine translation (Sutskever et al., 2014) it surpassed the state-of-the-art by a considerable margin. This success in turn led to more and more software packages, such as *scikit-learn* (Pedregosa et al., 2011) or *keras* (Chollet et al., 2015), that aim to enable even novice users to apply and tinker with cutting edge methods, making it easier than ever to apply machine learning in the wild.

In practice each machine learning method comes with a set of hyperparameters that either control the capacities or the training of the underlying statistical model. Common examples are the number of units in each layer of a neural network or the regularization parameter of support vector machines. Setting these hyperparameters somewhat correctly is essential and often makes the difference between state-of-the-art performance or mediocre performance (Feurer and Hutter, 2018). For example, Melis et al. (2018) were able to outperform complex state-of-the-art methods for language modelling benchmarks by simply optimizing the hyperparameters of a standard LSTM (Hochreiter and Schmidhuber, 1997) network. Henderson et al. (2018) showed that hyperparameters play a crucial role in benchmarking reinforcement learning algorithms and need to be set carefully to allow for a fair comparison to baseline methods.

Unfortunately, manually finding the right hyperparameters is usually a tedious task which is often based on a long process of trial-and-error. In order to automate this part of the machine learning pipeline, recent approaches in *automated machine learning* cast the search of the right hyperparameters as an optimization problem (see Feurer and Hutter (2018) for an overview). In contrast to the internal optimization problem that many methods, for example neural networks, need to solve, gradient information is not available, or at least not easily obtainable. Additionally, due to the intrinsic randomness of most machine learning methods, the objective function can only be observed with noise. Moreover, what makes hyperparameter optimization particularly challenging is that single function evaluations require to train and validate a machine learning algorithm and hence can take several hours or even days. With the increasing demand for computational resources of contemporary machine learning models, this poses a major challenge not only in applying hyperparameter optimization but also to develop and benchmark new methods.

One of the most popular approaches for hyperparameter optimization is Bayesian optimization (Jones et al., 1998; Mockus et al., 1978; Shahriari et al., 2016). At its core, Bayesian hyperparameter optimization maintains a probabilistic model of the validation error of a machine learning method with respect to its hyperparameters. Based on a utility or acquisition function, in each iteration, it selects a new hyperparameter configuration which is then used to train and validate the machine learning method to obtain its performance.

Because of its probabilistic model, Bayesian optimization is a powerful and flexible framework for hyperparameter optimization problems. In this thesis, we present several extensions that make Bayesian optimization more efficient, in particular for expensive optimization problems.

1.1. Contributions

Bayesian optimization (Jones et al., 1998) as a means for automated hyperparameter optimization is an active research field which offers great practical potential (Shahriari et al., 2016). In this thesis we present several contributions to advance the state-of-the art in this field:

Bayesian neural networks: We present in Chapter 3 a flexible and powerful probabilistic model based on Bayesian neural networks that can be used inside Bayesian optimization. While this has been explored by others before (e. g. Snoek et al. (2015)), our model contains a full Bayesian treatment of the neural network weights and hence allows to more faithfully represent the epistemic uncertainty of the model. On a more practical note, it can be adapted for modelling learning curves of iterative machine learning methods as we show in Chapter 5. It also plays a crucial role for our generative meta-model to benchmark hyperparameter optimization methods presented in Chapter 8.

Multi-fidelity optimization: Hyperparameter optimization is often prohibitively expensive. One major contribution of this thesis is to speed up the whole optimization process by exploiting cheap but approximate fidelities of the objective functions. Chapter 4 presents a new Bayesian optimization variant, called Fabolas, that is able to reason across datasets to achieve 10 to 1000 fold speedups of standard Bayesian optimization. In Chapter 5 we present a probabilistic model that allows to model learning curves, a fidelity often used for iterative machine learning methods. Finally, in Chapter 6 we introduce BOHB, which, by combining Bayesian optimization with Hyperband, is able to handle arbitrary fidelities as long as they correlate with the objective function. Furthermore, BOHB is easy to parallelize and more flexible with respect to the configuration space than Gaussian process based methods, such as Fabolas.

Hyperparameter optimization benchmarks: The high computational demands of hyperparameter optimization often block researchers from systematically runing

rigorous comparisons in order to draw statistically significant conclusions. Besides that, obtaining challenging benchmarks is often hard since meaningful datasets are usually scarce. We present in Chapter 7 a set of discrete tabular benchmarks that are based on an exhaustive search of the hyperparameters of a feed forward neural network. These benchmarks allow us to perform cheap table lookups instead of training the neural networks every time we want to evaluate the objective function. For continuous benchmarks we describe in Chapter 8 a generative meta-model across tasks that, once trained on data generated offline, allows to sample an arbitrary amount of new optimization tasks. Based on this generative model, another key contribution of this thesis is an exhaustive empirical evaluation of different hyperparameter optimization methods from the literature.

1.2. Publications

Most of this thesis has already been published as conference or journal papers. In the following, we present all publications that are parts of this thesis chronologically:

- Bayesian optimization with robust Bayesian neural networks
  J. T. Springenberg and **A. Klein** and S. Falkner and F. Hutter
  Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (2016)
- Fast Bayesian optimization of machine learning hyperparameters on large datasets
  A. Klein and S. Falkner and S. Bartels and P. Hennig and F. Hutter
  Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (2017)
- Learning curve prediction with Bayesian neural networks
  A. Klein and S. Falkner and J. T. Springenberg and F. Hutter
  International Conference on Learning Representations (2017)
- Fast Bayesian hyperparameter optimization on large datasets
  A. Klein and S. Falkner and S. Bartels and P. Hennig and F. Hutter
  Electronic Journal of Statistics (2017)
- RoBO: A Flexible and Robust Bayesian Optimization Framework in Python
  A. Klein and S. Falkner and N. Mansur and F. Hutter
  NIPS 2017 Bayesian Optimization Workshop (2017)
- BOHB: Robust and efficient hyperparameter optimization at scale
  S. Falkner and **A. Klein** and F. Hutter
  Proceedings of the 35th International Conference on Machine Learning (2018)
- Tabular Benchmarks for Joint Architecture and Hyperparameter Optimization
  A. Klein and F. Hutter
  arXiv:1905.04970 [cs.LG] (2019)

- Meta-Surrogate Benchmarking for Hyperparameter Optimization
  A. Klein and Z. Dai and F. Hutter and N. Lawrence and J. Gonzalez
  arXiv:1905.12982 [cs.LG] (2019)

I was also involved in the following publications which, however, were out of the scope of this thesis:

- Efficient and robust automated machine learning
  M. Feurer and **A. Klein** and K. Eggensperger and J. Springenberg and M. Blum and F. Hutter
  Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems
- Towards automatically-tuned neural networks
  H. Mendoza and **A. Klein** and M. Feurer and J. T. Springenberg and F. Hutter
  Workshop on Automatic Machine Learning
- The sacred infrastructure for computational research
  K. Greff and **A. Klein** and M. Chovanec and F. Hutter and J. Schmidhuber
  Proceedings of the 16th Python in Science Conference
- Uncertainty estimates and multi-hypotheses networks for optical flow
  E. Ilg and O. Cicek and S. Galesso and **A. Klein** and O. Makansi and F. Hutter and T. Brox
  Proceedings of the European Conference on Computer Vision
- NAS-Bench-101: Towards Reproducible Neural Architecture Search
  C. Ying* and **A. Klein*** and E. Real and E. Christiansen and K. Murphy and F. Hutter
  Proceedings of the 36th International Conference on Machine Learning (2019)
  * contributed equally

1.3. Collaborations

Major parts of this thesis result from fruitful collaborations with other researchers. More specifically:

- Chapter 3: The original idea and implementation of using scale adaption for stochastic gradient Hamiltonian Monte-Carlo to more robust Bayesian neural networks came from Tobias Springenberg. He also conducted the empirical evaluation of Bohamiann for the multi-task, the reinforcement learning and the parallel settings. The deployment of the Bayesian neural network in Bayesian optimization and implementing it in the python framework RoBO was conducted by me. I also implemented the baseline methods, such as GP-BO, MTBO and DNGO and performed the comparisons on the synthetic

benchmarks. Stefan Falkner advised on questions about Markov Chain Monte Carlo sampling and its convergence. The whole project was supervised by Frank Hutter.

- Chapter 4: I served as the lead author of the corresponding paper and was responsible for the implementation and evaluation of the proposed method. Frank Hutter had the initial idea to develop a method that can reason across dataset subsets. Stefan Falkner helped as advisor and implemented the multi-task kernel, the basis function kernel and Hyperband. He also conducted the experiments for the heteroscedastic noise induced by using subsets of the training data. Frank Hutter and Philipp Hennig served as supervisor for this paper. Simon Bartels and Philipp Hennig improved the idea of the basis function kernel to model the performance and cost with respect to the datasets size. Fabolas originated from the joint master project of Simon Bartels and me proposed and supervised by Frank Hutter.
- Chapter 5: The proposed method is a follow-up to the work by Domhan et al. (2015). I developed and implemented the proposed method and conducted most experiments. Stefan Falkner and Tobias Springenberg helped with the Bayesian neural network part. Frank Hutter supervised the project.
- Chapter 6: Stefan Falkner implemented the software architecture with the core functionality for Hyperband and BOHB and performed the analysis of BOHB's hyperparameters on the surrogate benchmarks. I implemented the sampling strategy for BOHB, created the surrogate benchmarks, implemented some of the baseline methods (Fabolas, LC-Net, BO-GP) and conducted the experiments for the Bayesian neural network and reinforcement learning experiments. Frank Hutter proposed to combine Hyperband and Bayesian optimization and supervised the project.
- Chapter 7: I generated and analyzed the data and conducted the benchmark comparison of the various hyperparameter optimization methods. The general idea of using tabular benchmarks as replacement for expensive neural architecture search problems was originally from Frank Hutter and Kevin Murphy, which eventually resulted in another paper (see Ying et al. (2019)).
- Chapter 8: This project was mainly developed during my internship at Amazon Research Cambridge. It was the result of a very fruitful discussion between me, Zhenwen Dai and Javier Gonzales. At an early stage of the project, Zhenwen Dai developed and implemented a multi-task model based on Gaussian processes. I was in charge of implementing the method with Bayesian neural networks and executing the experiments. Frank Hutter helped with developing the statistical tests and accumulating statistics of the performance of hyperparameter optimization methods across tasks. Neil Lawrence and Frank Hutter helped in later stages with writing the paper. During the whole time Javier Gonzales supervised the project.
- RoBO: Most of the methods developed in this thesis were implemented in the

python framework RoBO (Klein et al., 2017b). Frank Hutter originally had the idea of an open-source framework for Bayesian optimization. I implemented most of the core parts and served as a leading developer. Stefan Falkner implemented the original Hyperband code and parts of the Gaussian process implementation. Numair Mansur helped with writing the documentation and setting up the integrated unit tests.

2. Background and Related Work

The goal of this chapter is to provide a solid background for the remaining chapters and to give a broad overview of the relevant literature of this thesis. We first give an introduction into hyperparameter optimization and its related fields in Section 2.1. In Section 2.2 we describe Bayesian optimization which represents the underlying framework for many of the proposed methods in this thesis. Finally, in Section 2.3 we present related work in obtaining uncertainty estimates for deep neural networks.

2.1. Hyperparameter Optimization

Hyperparameter optimization has been studied by many other researches before, and various different fields have developed their own approaches to tackle it. In this chapter we first present a formal definition of the hyperparameter optimization problem (see Section 2.1.1). Afterwards, we give in Section 2.1.2 an overview of the literature that approaches this optimization problem from different angles. In Section 2.1.3 of this chapter we discuss other research fields in machine learning that are related to hyperparameter optimization.

2.1.1. Problem Definition

Given a machine learning method A with hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$, we are interested in optimizing its generalization performance on some dataset $\mathcal{D}$. The input space $\boldsymbol{\Lambda}$ is the so-called configuration space and might consist of mixed continuous, integer or categorical hyperparameters. Furthermore, individual hyperparameters can be conditional dependent on each other which means that hyperparameter $\lambda_i \in \boldsymbol{\lambda}$ is only active if $\lambda_j \in \boldsymbol{\lambda}$ is set to a specific value. For example, the number of units in the second layer of a fully connected neural network is only relevant if the total number of layers is larger or equal than two.

We follow the formulation by Thornton et al. (2013) and define the hyperparameter optimization problem as follows:

$$\boldsymbol{\lambda}_\star \in \arg\min_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \mathbb{E}_{(\mathcal{D}_{train}, \mathcal{D}_{valid} \sim \mathcal{D})} \Big[\mathcal{L}(\boldsymbol{\lambda}, \mathcal{D}_{train}, \mathcal{D}_{valid}) \Big] \tag{2.1}$$

where $\mathcal{L}$ measures the validation error of A on the validation data $\mathcal{D}_{valid}$ after it has been trained on the training data $\mathcal{D}_{train}$ with hyperparameters $\boldsymbol{\lambda}$. We refer to $\mathcal{L}$ as the objective function.

In practice the expectation in Equation 2.1 cannot be solved in closed-form and is often approximated by either k-fold cross-validation, random data splits or one single train and validation split (Feurer and Hutter, 2018). If $\mathbf{\Lambda}$ does not only contain the hyperparameters but also the choice of algorithm A from a set of algorithms $A \in \{A_0, \dots A_N\} \subset \mathbf{\Lambda}$, Equation 2.1 is also referred as the combined algorithm selection and hyperparameter (CASH) problem (Thornton et al., 2013).

2.1.2. Literature Overview

In this section we present an overview of the literature on automated hyperparameter optimization (Feurer and Hutter, 2018). We first discuss model-free methods, such as random search or evolutionary algorithms that do not model the objective function directly. Secondly, we describe model-based methods, such as Bayesian optimization, to which the work presented in this thesis also belongs to. Finally we look at methods that do not treat hyperparameter optimization as a gradient-free optimization problem but rather try to approximate the gradients of the validation error of a machine learning method with respect to its hyperparameters such that standard gradient descent techniques can be applied.

2.1.2.1. Model-free Methods

Arguably, the simplest solution to optimize the hyperparameters of any kind of machine learning algorithm is grid search, which first discretizes the configuration space based on a fixed sized grid and then evaluates each hyperparameter configuration of this grid. Even though grid search is trivial to implement and can easily be parallized, it has two major disadvantages (Bergstra and Bengio, 2012): It is unlikely to find the global optimum particularly in continuous spaces and more importantly, due to the curse of dimensionality grid search relatively quickly becomes infeasible if the number of dimensions increases.

Instead of evaluating a predefined set, random search (Bergstra and Bengio, 2012) samples each hyperparameter configuration at random from a predefined distribution. Most commonly used are uniform distributions but any other type of parametric distribution is possible. Random search is conceptually equally simple as grid search but in the case where only a small subset of the hyperparameters effect the final performance, called the low-effective dimensionality, it is usually much more efficient (Bergstra and Bengio, 2012). Compared to the model-based or gradient-based approaches described in the next sections, random search is guaranteed to find the global optimum without making any assumptions on the objective function, if the number of function evaluations approaches infinity and each hyperparameter configuration in the configuration space has a non-zero probability mass.

More recently, Li et al. (2017) developed Hyperband which augments random search with the bandit strategy successive halving (Jamieson and Talwalkar, 2016) to dynam-

ically allocate resource to a set of randomly drawn hyperparameter configurations by exploiting cheap-to-evaluate approximation of the objective function, called fidelities. We describe Hyperband and successive halving in more detail in Section 6.1. It can be shown that the worst-case performance of Hyperband in the limit of infinite function evaluations is only a constant slower than random search.

To guide the search during optimization, population based search algorithms maintain a population of data points in the input space. Individual elements of this population are mutated to explore the search space whereas for exploitation only the well-performing elements are kept in the population. Since population based methods do not require any gradient information they can be easily applied to the hyperparameter optimization setting. For instance, Loshchilov and Hutter (2016) used the popular evolutionary algorithm CMA-ES (Hansen, 2006) to optimize the hyperparameters of neural networks and achieved comparable results to model-based methods such as Bayesian optimization. Jaderberg et al. (2017) sample a population of random hyperparameter configurations for neural networks and adapt them during the training by either randomly perturbing single hyperparameters or exploiting other well-performing settings in the population. A major advantage of these methods is that they can adapt single hyperparameters online during training which is not trivial with model-based approaches.

2.1.2.2. Model-based Methods

In order to exploit the gained knowledge from previous function evaluations, model-based approaches maintain a model of the objective function internally to guide the search towards the global optimum. If a probabilistic model is used, these types of methods are also commonly referred to as Bayesian optimization (Jones et al., 1998; Shahriari et al., 2016). Based on this probabilistic model Bayesian optimization uses an acquisition function to select points in the input space that automatically trade off exploration and exploitation of the objective function. Since Bayesian optimization represents the core framework used in this thesis, we present a more detailed introduction in Chapter 2.2.

Compared to model-free methods, using a probabilistic model allows to exploit additional sources of information. Swersky et al. (2013) augmented standard Bayesian optimization by an additional task variable to allow to warm-start the optimization process from previously optimized tasks with the same input domain. Instead of just using a predefined task variable, Feurer et al. (2015b) defined a set of meta-features that capture similarities between optimization tasks, and, hence, does not require to obtain any observations on the new task before it can measure correlation to previous tasks. Springenberg et al. (2016) (see also Chapter 3) presented a Bayesian neural network model that learns an embedding of tasks during optimization.

Often so-called fidelities of the objective function are available, which are approximations that are much cheaper to evaluate. Kandasamy et al. (2016) developed a

multi-fidelity Bayesian optimization strategy, that, given a set of independent fidelities, achieves a theoretical lower regret than non-fidelity methods. In follow up work, Kandasamy et al. (2017) extend it to continuous fidelities. For iterative machine learning methods Swersky et al. (2014) augmented the default model in Bayesian optimization by also modelling learning curves such that it can automatically decide to continue or to stop the evaluation of hyperparameter configurations. Klein et al. (2017a) developed Fabolas (see Chapter 4) which treats the training dataset size a continuous fidelity to speed up the optimization process. Instead of a fidelity scenario which assumes that the quality of the fidelities increases with the evaluation cost, and , hence, induces an order between fidelities, Poloczek et al. (2017) extended Bayesian optimization to multi-source scenario where sources can be correlated but do not necessarily have an implicit order.

2.1.2.3. Gradient-Based Methods

Recently, an alternative view on hyperparameter optimization has been established that, for specific machine learning algorithms, tries to compute the gradient of the validation error with respect to its hyperparameters, the so-called hypergradients (Maclaurin et al., 2015). In order to compute these hypergradients Franceschi et al. (2017) studied the reverse and forward mode for automated differentiation, which trade off space vs time requirements. Particularly for adapting the learning rate of neural networks, Baydin et al. (2018) revisited an old idea by Almeida et al. (1999) which approximates the gradient of the learning rate. Liu et al. (2019) used a continuous relaxation to optimize discrete architecture choices of neural networks.

A major advantage of gradient-based methods is that they allow to adjust certain hyperparameters, such as the learning rate on the fly which cannot easily be done with model-based methods that cast hyperparameter optimization as a gradient-free optimization problem. Furthermore, they usually scale much better with the number of hyperparameters than their model-based counterparts.

One caveat of gradient-based hyperparameter optimization methods is that they are only applicable for differentiable loss functions. However, popular loss functions, such as accuracy or BLEU scores, are non-differentiable, and hence, surrogate loss function such as the cross-entropy loss need to be optimized instead. Moreover, hypergradients are only computable for continuous hyperparameters and only for certain discrete hyperparameters relaxations exist that allow for an optimization.

2.1.3. Connection to Other Research Fields

Automated machine learning (AutoML) (Hutter et al., 2018) is a new research field that tries to automate different parts of machine learning pipelines that are traditionally executed by humans. Since in practice hyperparameters are often tuned manually, hyperparameter optimization can be considered as a subfield of AutoML

and is often an integral component of the most practical AutoML system (Feurer et al., 2015a; Thornton et al., 2013). For instance Thornton et al. (2013) build the automated machine learning tool Auto-Weka which internally uses the random forest based Bayesian optimization method SMAC (Hutter et al., 2011) to optimize the choice of algorithm and its hyperparameters based on the machine learning methods implemented in the Weka (Hall et al., 2009) library. In a follow-up work, Feurer et al. (2015a) build auto-sklearn which also uses SMAC but is implemented around the popular python machine learning framework sklearn (Pedregosa et al., 2011)

Algorithm configuration (Hutter et al., 2009) tries to automatically find the right parameters and design choices for any kind of algorithm, usually across a set of problem instances. It mostly, but not exclusively, focuses on hard-combinatorial problems, such as for instance SAT or TSP where singe instances corresponds to single SAT clauses or TSP graphs, respectively. In algorithm configuration one usually tries to optimize the runtime of the algorithm rather than a validation score. Hyperparameter optimization can be seen as algorithm configuration on a single instance that represents the underlying dataset.

More recently, *neural architecture search* (Zoph and Le, 2017; Real et al., 2017) (for an overview see Elsken et al. (2018)) emerged which solely focus on hyperparameters that define the architecture of the neural network and hence can be considered as a specialized form of hyperparameter optimization. Thereby, the most methods optimize other hyperparameters that effect the training of the neural network, such as the learning rate or regularization parameters in second post-hoc step. Compared to hyperparameter optimization, architectures cannot easily be described by a numerical vector, but are usually encoded as a directed acyclic graph. While most work either use reinforcement learning (Zoph and Le, 2017; Zoph et al., 2018) or evolutionary algorithms (Real et al., 2017, 2019; Elsken et al., 2019) since they can easily be adapted to handle these structured space, Kandasamy et al. (2018) defined a new kernel based on a optimal transport program to make Bayesian optimization applicable on neural architecture search tasks. Moreover, recent work (Pham et al., 2018; Liu et al., 2019) uses weight sharing between architectures to reduce the computation burden which has not been done for hyperparameter optimization yet.

Meta-learning (Vanschoren, 2018), also often referred to *learning-to-learn*, tries to improve the learning of machine learning algorithms on new tasks in terms of speed and generalization performance by exploiting knowledge accumulated on previously seen tasks. As we show in Chapter 3, meta-learning can be combined with hyperparameter optimization to speed up the optimization procedure (Swersky et al., 2013; Feurer et al., 2015b; Springenberg et al., 2016) by reusing information of previously conducted optimization runs on similar problems. To go even one step further, Chen et al. (2017) learned an entire algorithm for gradient-free optimization problem that mimics Bayesian optimization but amortizes its computational overhead.

At last, hyperparameter optimization is often treated as a “black-box“ optimization problem, since it does not assume any gradient information, it is strongly related

to literature on general *derivative-free optimization* (Conn et al., 2009). The most notable difference though, compared to other common derivative-free optimization problems, is that hyperparameter optimization assumes single function evaluations to be extremely expensive (Shahriari et al., 2016) (e. g. multiple hours or even days) and hence a larger additional overhead from the optimizer of a few seconds or minutes is acceptable if a better sample efficiency is achieved.

2.2. Bayesian Optimization

Bayesian optimization (Mockus et al., 1978; Jones et al., 1998) is a framework for model-based optimization of gradient-free optimization problems. It is designed for functions that are particularly expensive to evaluate such that only few samples can be collected. Over the years, Bayesian optimization has been successfully applied on a variety of applications, such as robotics (Calandra et al., 2014), environmental monitoring (Marchant and Ramos, 2012), sensor set selection (Garnett et al., 2010), reinforcement learning (Brochu et al., 2010), drug discovery (Gómez-Bombarelli et al., 2018) or gene design (González et al., 2014). However, maybe its biggest success story is the hyperparameter optimization of machine learning methods. For instance, Snoek et al. (2012) used Gaussian process based Bayesian optimization to optimize the hyperparameters of a convolutional neural networks and improve at that point state-of-the-art performance on the CIFAR-10 benchmark. Melis et al. (2018) used the Bayesian optimization based hyperparameter tuner Vizier (Golovin et al., 2017) to achieve state-of-the-art performance on language modelling tasks. Auto-sklearn Feurer et al. (2015a) won the first AutoML challenge (Sun-Hosoya et al., 2018) by using Bayesian optimization to search for the right algorithm and hyperparameters in the sklearn framework (Pedregosa et al., 2011). With the help of Bayesian optimization, Chen et al. (2018) substantially improved the famous AlphaGo system which defeated top human Go players.

In this chapter we give a brief introduction into Bayesian optimization by first describing its general concept and then its two core components: the probabilistic model (Section 2.2.2) and the acquisition function (Section 2.2.3). More comprehensive tutorials are presented by Brochu et al. (2010), Frazier (2018) and Shahriari et al. (2016).

2.2.1. The Main Loop

Given a black-box function $f : \mathbb{X} \to \mathbb{R}$, Bayesian optimization aims to find points in the input space $\boldsymbol{x}_\star \in \arg\min_{\boldsymbol{x}\in\mathbb{X}} f(\boldsymbol{x})$ that globally minimize f. The input or configuration space $\mathbb{X}$ can be an arbitrary space with continuous and discrete parameters, but is usually defined as a compact set $\mathbb{X} \subset \mathbb{R}^d$ of the real space.

The general idea of Bayesian optimization is to define a prior distribution $p(f)$ - usually referred to as the model - over the objective function f which is updated to a posterior distribution $p(f \mid \mathcal{D})$ if data $\mathcal{D}$ is observed. The intuition behind $p(f \mid \mathcal{D})$ is that it captures our current belief of f based on the knowledge that we obtained through previous evaluations. As we are going to see in the next section, popular choices for the model of the objective function $p(f)$ are Gaussian processes (Snoek et al., 2012) or random forests (Hutter et al., 2011). We also discuss how Bayesian neural networks (Snoek et al., 2015; Springenberg et al., 2016) can be applied in Chapter 3.

The second important component is the so-called acquisition function $a_{p(f|\mathcal{D})} : \mathbb{X} \to \mathbb{R}$ that quantifies the utility of evaluating the objective function at any $\boldsymbol{x} \in \mathbb{X}$ in the input space and thereby trades off exploration and exploitation. For a new query point one can use any numerical optimization technique to select the point that maximizes the acquisition function and, if the model provides gradient information, one can even use off-the-shelf gradient-based optimization methods. Importantly, the acquisition function is solely based on the model and, hence, compared to the actual objective function, cheap-to-evaluate.

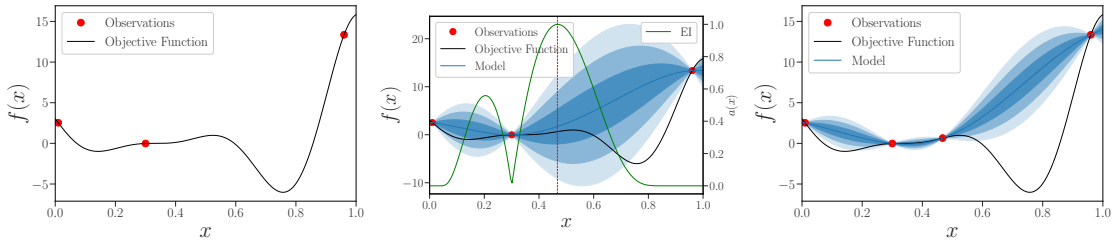
With these two ingredients, Bayesian optimization iterates the following steps (Jones et al., 1998; Mockus et al., 1978) (see also Figure 2.1 for a visualization and Algorithm 1 for pseudo code) after observing some initial design $\mathcal{D}_t = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_t, y_t)\}$:

1. update the probabilistic model $p(f \mid \mathcal{D}_t)$ and acquisition function $a_{p(f|\mathcal{D}_t)}$ with the data $\mathcal{D}_t = (\boldsymbol{x}_j, y_j)_{j=1\ldots t}$ that has been observed so far
2. find the most promising candidate $\boldsymbol{x}_t$ by maximizing the acquisition function $a_{p(f|\mathcal{D}_t)}(\boldsymbol{x})$
3. evaluate the noisy objective function $y_{t+1} \sim f(\boldsymbol{x}_{t+1}) + \mathcal{N}(0, \sigma^2)$ and add the resulting data point $(\boldsymbol{x}_{t+1}, y_{t+1})$ to the set of observations $\mathcal{D}_t$

Algorithm 1 Bayesian optimization

1: Initialize data $\mathcal{D}_0$ using an initial design.
2: **for** $t = 1, 2, \ldots$ **do**
3: Fit probabilistic model $p(f \mid \mathcal{D}_{t-1})$ for $f(\boldsymbol{x})$ on data $\mathcal{D}_{t-1}$
4: Choose $\boldsymbol{x}_{t+1} \in \arg\max a_{p(f|\mathcal{D}_t)}(\boldsymbol{x})$
5: Evaluate $y_t \sim f(\boldsymbol{x}_t) + \mathcal{N}(0, \sigma^2)$, and augment the data: $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(\boldsymbol{x}_t, y_t)\}$
6: Estimate incumbent $\hat{\boldsymbol{x}}_t$
7: **end for**

To allow for an anytime comparison, in each iteration t, we maintain an estimate $\hat{\boldsymbol{x}}_t$, dubbed the incumbent, of the global optimizer $\boldsymbol{x}_\star$. The most robust choice to select $\hat{\boldsymbol{x}}_t$ in the case of low noise that we typically see hyperparameter optimization is to simply pick the point with the lowest observed function value: $\min\{y_1, \ldots y_t\}$. Alternatively, one could also optimize the posterior mean of $p(f \mid \mathcal{D}_t)$ which is particularly appealing in the case of high noise but strongly hinges on a robust model,



(a) **(b)** **(c)**

Figure 2.1.: Visualization of the core steps of Bayesian optimization: (a) the true unknown objective function and some initial observations ; (b) a Gaussian process trained on the observed data to model $p(f \mid \mathcal{D})$ (solid blue line denotes the mean and the shaded area one, two and three times the standard deviation), the expected improvement acquisition function (green solid line) and its maximizer (vertical red dashed line) ; (c) the updated model after we evaluated the objective function at the maximizer of the acquisition function.

which might be brittle especially in the beginning of the search when not a sufficient amount of data points has been observed yet.

2.2.2. Common Probabilistic Models

As described above one of the core components of Bayesian optimization is the probabilistic model of the objective function. In general, every method that, given a test data point, provides a predictive distribution could be used as such as model. In this section we look at the two most commonly used methods: Gaussian processes and random forests.

2.2.2.1. Gaussian Processes

Gaussian processes (GP) are probably the most popular choice for $p(f)$, thanks to their descriptive power and analytic tractability (e.g. Rasmussen and Williams, 2006). Formally, a GP is a collection of random variables, such that every finite subset of them follows a multivariate normal distribution. A GP is solely defined by a mean function m (often set to $m(\boldsymbol{x}) = 0 \; \forall \boldsymbol{x} \in \mathbb{X}$), and a positive definite covariance function, called kernel $k(\boldsymbol{x}, \boldsymbol{x}')$ which measures the similarity between two points in the input space. Given some observations $\mathcal{D}_t = (\boldsymbol{x}_j, y_j)_{j=1\ldots t} = (\boldsymbol{X}, \boldsymbol{y})$ with a joint Gaussian likelihood $p(\boldsymbol{y} \mid \boldsymbol{X}, f(\boldsymbol{X}))$, the posterior $p(f \mid \mathcal{D}_t)$ follows another GP, whose mean and covariance functions have also a tractable and analytic form.

The most frequently used kernel is the squared exponential or radial basis function (RBF) kernel:

$$k_{\text{RBF}}(\boldsymbol{x}, \boldsymbol{x}') = \nu \exp\left(\frac{-||\boldsymbol{x} - \boldsymbol{x}'||^2}{\boldsymbol{\lambda}}\right). \tag{2.2}$$

Here, the covariance amplitude ν and the lengthscale $\boldsymbol{\lambda}$ are free parameters, i. e. hyperparameters of the GP.

More popular for applications in AutoML though is the stationary and twice-differentiable Matérn $5/2$ kernel (Matérn, 1960), in its Automatic Relevance Determination form (MacKay and Neal, 1994):

$$k_{\text{Matern}5/2}(\boldsymbol{x}, \boldsymbol{x}') = \nu \left(1 + \sqrt{5} d_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{x}') + 5/3 d^2_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{x}')\right) e^{-\sqrt{5} d_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{x}')}. \tag{2.3}$$

where $d_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x} - \boldsymbol{x}')^\intercal \operatorname{diag}(\boldsymbol{\lambda})(\boldsymbol{x} - \boldsymbol{x}')$ is the Mahalanobis distance.

In contrast to the RBF kernel, it makes less restrictive smoothness assumptions, which can be helpful in the optimization setting (Snoek et al., 2012) (see Figure 2.2 for a visualization of both kernels).

An additional hyperparameter of the GP model is the overall noise σ_{noise} which is added to the diagonal of the covariance matrix in order to handle noisy observations. For clarity: These GP hyperparameters are *internal* hyperparameters of the Bayesian optimizer, as opposed to those of the target machine learning algorithm to be tuned. All hyperparameters $\xi = \{\boldsymbol{\lambda}, \nu, \sigma_{noise}\}$ are determined by maximizing the marginal log-likelihood distribution of the Gaussian process (Rasmussen and Williams, 2006; Snoek et al., 2012):

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \xi) = \mathcal{N}(\boldsymbol{y} | m, \boldsymbol{\Sigma}_{\boldsymbol{\lambda}, \nu} + \sigma_{noise} \mathbb{I}) \tag{2.4}$$

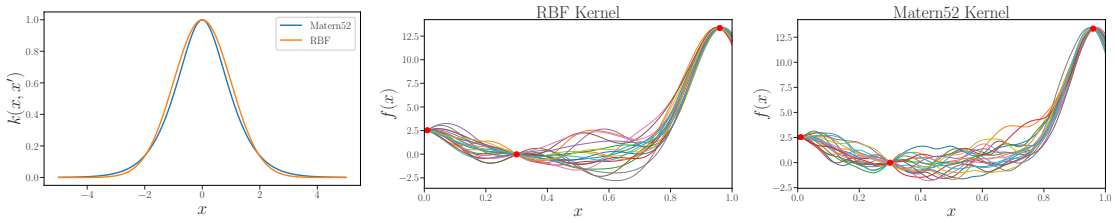
However, since just using a single maximum a-posterior estimate of the marginal likelihood might be sensitive to local optima, Snoek et al. (2012) argued for a Bayesian treatment of the GP's hyperparameters. They propose to use the integrated acquisition function:

$$a_{p(f|\mathcal{D})}(\boldsymbol{x}) = \int a_{p(f|\mathcal{D}, \xi)}(\boldsymbol{x}) p(\xi \mid \mathcal{D}) d\xi \tag{2.5}$$

which can approximated by a Monte-Carlo approach by sampling different ξ from the marginal likelihood in Equation 2.4 with Markov-Chain Monte-Carlo sampling algorithms.

2.2.2.2. Random Forests

Alternatively to Gaussian processes, random forests (Breimann, 2001) have been proposed as a probabilistic model for Bayesian optimization (Hutter et al., 2011). A random forest consists of a set of T regression trees where each tree splits the input space into disjoint regions $S_0, \dots S_{L-1}$ with L being the number of leafs in each tree. A regression tree is build recursively by splitting the training data at each node based on a splitting criterion that decides whether the current data point



(a) **(b)** **(c)**

Figure 2.2.: Difference between the RBF and Matérn5/2 kernel: (a) the kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$ with $\boldsymbol{x}' = 0$ and varying $\boldsymbol{x}$. Function samples drawn from $p(f \mid \mathcal{D})$ based on the function shown in Figure 2.1 with (b) a RBF kernel and (c) a Matérn5/2 kernel for the GP.

goes to the left or the right subtree until a leaf is reached. Instead of storing the datapoints in the leafs directly, a regression tree usually stores only a constant c_l in a leaf l such as, for example, the mean of the training datapoints that fall into this leaf. The mean prediction for a new unseen test datapoint is then computed by: $\tilde{\mu}(\boldsymbol{x}) = \sum_{l=1}^{L} c_l \cdot \mathbb{I}(\boldsymbol{x} \in S_i)$ with $\mathbb{I}$ as the indicator function that returns 1 if $\boldsymbol{x} \in S_i$ and 0 otherwise.

To increase the diversity of the regression trees such that the ensemble becomes more robust, random forest usually induce an additional source of randomness. For that different strategies exist, such as subsampling the training data for each tree with replacement, called bagging or randomly selecting the splitting criterion (Breimann, 2001).

During test time, the mean prediction of the random forest for a new test datapoint is simply the average of the T individual tree predictions:

$$\mu(\boldsymbol{x}) = \frac{1}{T} \sum_{t=1}^{T} \tilde{\mu}_t(\boldsymbol{x}) \tag{2.6}$$

For the predictive variance, Hutter et al. (2014b) suggested to additionally store the empirical variance $\tilde{\sigma}_t^2(\boldsymbol{x})$ of the training data in each leaf and, based on the law of total variance, compute:

$$\sigma^2(\boldsymbol{x}) = \Big(\frac{1}{T} \sum_{t=1}^{T} \tilde{\sigma}_t^2(\boldsymbol{x}) + \tilde{\mu}_t^2(\boldsymbol{x}) \Big) - \mu(\boldsymbol{x}) \tag{2.7}$$

Compared to Gaussian processes, random forests seem to work better in higher dimensional discrete spaces (Eggensperger et al., 2013). Furthermore, training a random forest scales in $\mathcal{O}(NlogN)$ with N as the number of training data points whereas Gaussian processes scale cubically, i. e. $\mathcal{O}(N^3)$. However, random forests achieve less smooth uncertainty estimates than Gaussian processes (see Figure 2.3 for a visual comparison) which is crucial in an active learning scenario, such as Bayesian optimization since otherwise the acquisition function becomes more difficult to optimize.

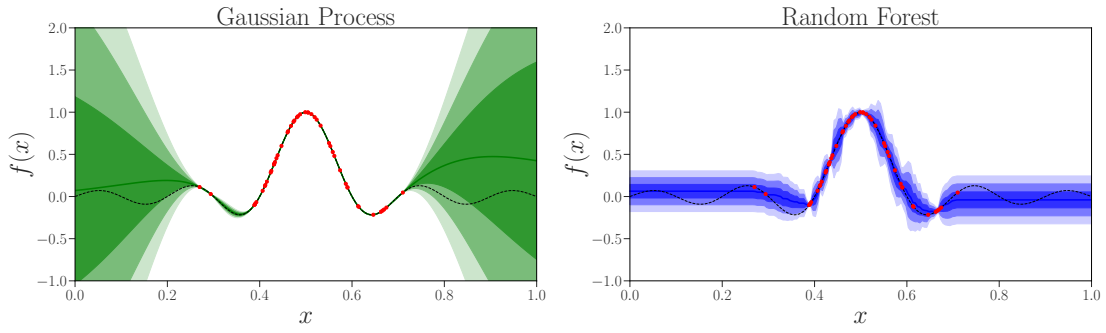


Figure 2.3.: A visual comparison of the two popular models for Bayesian optimization: Gaussian process (a) and random forest (b). As one can see, the Gaussian process achieves smoother and more reliable uncertainty estimates, i. e. the uncertainty increases away from data. However, random forests scale more gracefully with the number of datapoints and tend to work better in high-dimensional discrete spaces.

2.2.3. Acquisition Functions

The role of the acquisition function is to balance between exploration in regions of the input space where our model is uncertain and exploitation in the good regions of the space where our model assumes the global optimum. Among the most popular acquisition functions are:

Thompson Sampling (TS): a well-known strategy from the multi-armed bandit literature. Given a function sample from our model $f \sim p(f \mid \mathcal{D}_t)$ it computes:

$$a^{\text{TS}}_{p(f|\mathcal{D}_t)}(\boldsymbol{x}) = f(\boldsymbol{x})$$

However, while TS is conceptually very simple it requires further approximations if $p(f \mid \mathcal{D})$ is implemented as a Gaussian process (see for example Hernández-Lobato et al. (2014)).

Upper Confidence Bound (UCB) (Srinivas et al., 2010)[1]: which, inspired from the well-known Bandit policy, computes:

$$a^{\text{UCB}}_{p(f|\mathcal{D}_t)}(\boldsymbol{x}) = \mu(\boldsymbol{x}) + \beta_t \sigma(\boldsymbol{x})$$

Here, $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ are the mean and standard deviation of $p(f \mid \mathcal{D})$ provided by our model and the parameter β_t controls the exploration / exploitation trade-off. Given some assumptions on the objective function and adapting β_t over time t one can proof sublinear regret bounds (Srinivas et al., 2010) for UCB.

[1] UCB assumes that the objective function is maximized. We can easily adapt it to our setting by simply maximizing the negative lower bound instead: $-(\mu(\boldsymbol{x}) - \beta_t \sigma(\boldsymbol{x}))$

Probability of Improvement (PI) (Jones et al., 1998): computes the probability of improving over the currently best observed function value $y_\star \in \min\{y_0, \dots, y_t\}$:

$$a^{\text{PI}}_{p(f|\mathcal{D}_t)}(\boldsymbol{x}) = \Phi(\gamma(\boldsymbol{x}))$$

where $\gamma(\boldsymbol{x}) = \frac{y_\star - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}$ and Φ is the CDF of a standard normal distribution

Expected Improvement (EI) (Mockus et al., 1978) is arguably the most often used acquisition function

$$a^{\text{EI}}_{p(f|\mathcal{D}_t)}(\boldsymbol{x}) = \mathbb{E}_p[\max(y_\star - f(\boldsymbol{x}), 0)]\,. \tag{2.8}$$

If we assume the predictive distribution of $p(f \mid \mathcal{D})$ to be Gaussian we can compute the above expectation in closed form $a_{\text{EI}} = \sigma(\boldsymbol{x})(\gamma(\boldsymbol{x})\Phi(\gamma(\boldsymbol{x}))) + \phi(\gamma(\boldsymbol{x}))$. Here $\gamma(\boldsymbol{x}) = \frac{y_\star - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}$ and Φ is the CDF and ϕ is the PDF of a standard normal distribution.

All acquisition functions above share the same characteristic that they try to collect points with low function values. In contrast, information-theoretic acquisition functions, such as Entropy Search (ES) (Hennig and Schuler, 2012; Villemonteix et al., 2008) and Predictive Entropy Search (PES) (Hernández-Lobato et al., 2014) model the probability of the minimum and search for points that reduce the entropy of this distribution. That allows to evaluate in regions of the input space where the optimum is unlikely to be located but still additional information about it can be gained. This is useful in the multi-fidelity setting as we are going to see in Chapter 4, where we also give a more detailed description of entropy search developed by Hennig and Schuler (2012) (see Section 4.1).

2.3. Uncertainty in Deep Learning

A fundamental aspect of any sequential decision making application, such as for example Bayesian optimization, is to trade off exploration of the environment and exploiting what presumably works best at the current time step. A key requirement for any machine learning method used to tackle this dilemma, is to obtain reliable uncertainty estimates of its own prediction. While some methods, such as Gaussian processes, naturally provide predictive uncertainties, capturing the intrinsic uncertainty of a neural network requires additional work.

In this Section we give a brief overview of different approaches to obtain uncertainty estimates with neural networks. We first explain how uncertainty can be decomposed in two parts which capture different sources of variability. Afterwards we describe Bayesian deep learning methods, which, based on Bayes Theorem, try to approximate the posterior distribution of the weights after data has been observed. Instead of using a Bayesian treatment of the parameters, we also look at methods that use a more frequentist perspective to obtain uncertainty estimates.

2.3.1. Decomposing Uncertainty

In general uncertainty of a probabilistic model can be classified as *aleatoric* or *epistemic* (Kiureghian and Ditlevsen, 2009). Uncertainty is referred to aleatoric if it describes the variability inherent to observations, such as for example the noise of sensory inputs, and does not diminish with an increasing number of data points. On the other hand, epistemic uncertainty represents a lack of knowledge of the probabilistic model, which implies, that it becomes smaller with more data. Epistemic uncertainty can be seen as the uncertainty within a model, for example the uncertainty of the parameters of a neural network.

Kendall and Gal (2017) studied the effect of aleatoric and epistemic uncertainty for computer vision tasks, such as image segmentation and depth regression. They showed that modelling aleatoric uncertainty helps in large data regimes as well as real time applications where epistemic uncertainty either becomes negligible or too expensive to model. However, epistemic uncertainty helps especially for safety-critical applications to identify data points too far away from the training data.

Depeweg et al. (2018) showed that decomposing the prediction distribution into an aleatoric and epistemic component allows to locate more informative points in an active learning scenario. This is particularly useful when the observation noise of the objective function is rather complex, for example heteroscedastic or bimodal, and cannot be described by a Gaussian.

2.3.2. Bayesian Deep Learning

Bayesian deep learning (Neal, 1996) uses a Bayesian treatment of the parameters of neural networks, rather than a maximum likelihood approach, which allows to approximate the predictive distribution for unseen test points. Therefore, a prior distribution is assign to all parameters in the neural network such that, combined with the likelihood of the training data, the posterior distribution can be computed. Unfortunately, due to the non-linearities of the activation functions, exact inference of the posterior distribution is analytically intractable and, hence, needs additional approximations. The rest of this section discusses two popular frameworks for Bayesian inference of neural networks, Markov-Chain Monte Carlo and variational inference.

2.3.2.1. Markov Chain Monte Carlo Methods

Markov Chain Monte Carlo (MCMC) represents a powerful framework for approximate inference which can also be used for Bayesian deep learning (Neal, 1996). In one of the earliest works on Bayesian deep learning, Neal (1996) used Hamiltonian Monte Carlo to sample from the posterior distribution of the weights. While this

was for a long time considered the gold standard for Bayesian neural networks, it requires the full batch gradient and cannot handle stochastic gradients.

More recently, Welling and Teh (2011) introduced stochastic gradient Langevin dynamics (SGLD) which can be applied to the stochastic mini-batch setting of contemporary neural networks. Welling and Teh (2011) showed that by appropriately decaying the step size, SGLD samples asymptotically from the true posterior. While SGLD is simple and easy to implement, its mixing rate, i. e. the time it requires to approach the true distribution, is slow. This can be attributed to the fact that parameters often change on a different scale. As cure Ahn et al. (2012) proposed to use the Fisher information matrix as a preconditioner to scale the gradient in order to improve the efficiency of SGLD. Chen et al. (2014) adapted Hamiltonian Monte Carlo to the stochastic gradient setting, which in practice, due to its additional momentum, often achieves a faster mixing rate than SGLD. More recently, Mandt et al. (2017) showed that standard stochastic gradient descent with a fixed learning rate performs Bayesian inference and simulates a Markov chain where the stationary distribution minimized the Kullback-Leibler divergence to the posterior.

While MCMC are asymptotically unbiased (with appropriated step sizes see (Nagapetyan et al., 2017)) and in practice often lead to reliable uncertainty estimate (see also Chapter 3), they require to store and evaluate many weight samples during test time which becomes infeasible with larger neural network architectures.

2.3.2.2. Variational Inference

Variational inference methods (Murphy, 2013) aim to minimize the discrepancy between the true posterior distribution and an analytical tractable distribution and thereby cast Bayesian inference as an optimization problem. Compared to MCMC methods, variational inference methods usually increase the number of parameters that need to be stored during inference time by only a small constant factor. Additionally, they can make use of recent advances developed for stochastic gradient descent techniques to converge faster than their MCMC counterparts.

In early work, Hinton and van Camp (1993) introduced an information-theoretic loss function motivated by the minimum description length principle to perform variational inference for neural networks. In follow up work, Graves (2011) applied Gaussian mean-field variational inference to deep neural networks. Later, Blundell et al. (2015) showed how to obtain unbiased gradients of the expected lower bound of the marginal log-likelihood with help of the reparameterization trick (Kingma and Welling, 2014). Gal and Ghahramani (2016) demonstrated that using dropout (Srivastava et al., 2014) during test time corresponds to variational inference and allows to easily obtain uncertainty estimates. In follow up work, Gal et al. (2017) applied this technique in an active learning setting, where in each iteration, based on the uncertainty estimate of a convolutional neural network, the task is to decide which data points should be labeled by a human. Hernández-Lobato and Adams (2015) used expectation propagation

to perform variational inference and improved upon previous methods (Graves, 2011). Khan et al. (2018) developed a new natural gradient method that performs variational inference and can be easily implemented in popular optimizers, such as Adam (Kingma and Ba, 2015) by simply perturbing the weights during optimization. To increase the robustness of variational inference, Wu et al. (2019) proposed a deterministic method to model the uncertainty of the activations of each layer in a neural network. Also, instead of using fixed predefined parameters for the priors, they use an empirical Bayes (Robbins, 1956) procedure to automatically determine the prior variances of the parameters.

2.3.3. Ensemble-based Methods

Instead of using Bayesian inference, which often makes additional assumptions about the posterior, a simpler way to obtain uncertainty estimates is to compute the empirical mean and variance of the predictions of an ensemble (Murphy, 2013) of neural networks. However, ensemble-based methods usually do not come with the strong theoretical support as Bayesian deep learning methods. For example, Lakshminarayanan et al. (2017) demonstrated that one can easily obtain an ensemble by training multiple neural networks which only differ in the seed of the random number generator. These ensembles often achieve better uncertainty estimates than more sophisticated Bayesian neural network techniques and, since they do not try to approximate any distribution, they are much simpler to implement and scale better to deeper neural network architectures. Furthermore, Lakshminarayanan et al. (2017) showed that the uncertainty estimates can be further improved when each individual neural network models a predictive distribution, i.e the network outputs a mean and variance and is trained by minimizing the negative log-likelihood, instead of just using the empirical variance of the networks' predictions.

Instead of learning multiple neural networks independently, Huang et al. (2017) trained a single neural networks and ensembles the weights at several local optima along the optimization trajectory. In order to converge to different local optima, Huang et al. (2017) used stochastic gradient with restarts as proposed by Loshchilov and Hutter (2017). Ilg et al. (2018) showed that such snapshot ensembles obtain a performance for optical flow tasks which is comparable to ensembles based on bootstrapping or different random initializations. However, note that, compared to the most MCMC methods, there are no theoretical guarantees that stochastic gradient descent with restarts asymptotically approximates the targets distribution.

Ensemble-based approaches showed also competitive performance in sequential decision making settings. For example, for contextual bandit problems, Riquelme et al. (2018) investigated and benchmarked different approaches to obtain uncertainty estimates for neural networks in order to apply Thompson Sampling, which is one of the most successful approach to tackle the exploration-exploitation dilemma (see also Section 2.2.3 which describes how Thompson Sampling can be used in

the context of Bayesian optimization). Notably, they showed that ensemble based methods combined with Thompson sampling are on-par with Bayesian deep learning methods that performed well in the supervised case but tend to underperform in sequential decision making scenario. For reinforcement learning problems, Osband et al. (2016) proposed bootstrapped neural networks to enable better exploration in deep Q-learning (DQN) (Mnih et al., 2015) in order to improve the performance in Atari games.

3. Bayesian optimization with Bayesian neural networks

Arguably, the most popular method to model the posterior distribution $p(f \mid \mathcal{D})$ for Bayesian optimization (BO) are Gaussian processes (Shahriari et al., 2016). Due to their mathematical elegance, they provide smooth and reliable uncertainty estimates (Rasmussen and Williams, 2006) which are essential to trade-off exploration and exploitation in BO. However, Gaussian processes scale cubically with the number of data points and hence are not efficient in regimes where many observations are available, such as for instance in the multi-task setting where we warm-start the optimization process by reusing previously collected data on similar datasets.

In this chapter we will explore Bayesian neural networks (Neal, 1996) as a model for BO. The key idea of Bayesian neural networks is to use a Bayesian treatment of the parameter of a neural network in order to obtain sensible uncertainty estimates while keeping the efficient scaling with respect to the number of datapoints. In Section 3.1 we give a general model definition for using Bayesian neural networks in the single as well as multi-task setting of BO.

There are different ways to obtain Bayesian neural networks, however, for BO it is essential that we obtain reliable uncertainties and that our model is robust enough such that it does not open up another internal hyperparameter optimization problem. In Section 3.3 we compare different existing methods and show that stochastic gradient Hamiltonian Monte-Carlo obtains well-calibrated uncertainty estimates. We present a robust version of stochastic gradient Hamiltonian Monte-Carlo (Chen et al., 2014) in Section 3.4 that automatically adapts its own hyperparameters and show in Section 3.5 that it improves upon the standard version. Finally, using our method – that we dub **B**ayesian **O**ptimization with **Hami**ltonian Monte Carlo **A**rtificial **N**eural **N**etworks (BOHamiANN) – we demonstrate in Section 3.6 state-of-the-art performance for a wide range of optimization tasks such as synthetic objective functions from the literature as well as multi-task hyperparameter optimization problems.

3.1. Model Definition for Single and Multi-Task Optimization

As described in Section 2.2, given a function $f : \mathbb{X} \to \mathbb{R}$ defined over some configuration space $\boldsymbol{x} \in \mathbb{X}$ where each function evaluation yields a noisy observation $y \sim \mathcal{N}(f(\boldsymbol{x}), \sigma^2_{\text{obs}})$, Bayesian optimization aims to find $\boldsymbol{x}^* \in \arg\min_{\boldsymbol{x}\in\mathbb{X}} f(\boldsymbol{x})$.

In the more generalized case of *multi-task Bayesian optimization* (Swersky et al., 2013), there are K related black-box functions, $\mathcal{F} = \{f_1, \ldots, f_K\}$, each with the same domain $\mathbb{X}$, and the goal is to find $\boldsymbol{x}^* \in \arg\min_{\boldsymbol{x}\in\mathbb{X}} f_t(\boldsymbol{x})$, for a given t.[1] In this case, the initial design is augmented with previous evaluations on the related functions. That is, $\mathcal{D} = \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_K$ with $\mathcal{D}_i = \{(\boldsymbol{x}^1, y_i^1), \ldots (\boldsymbol{x}^n, y_i^n)\}$, where $y_i^j = f_i(\boldsymbol{x}^j)$ and n points have already been evaluated for function f_i. BO then requires a joint probabilistic model $p(f \mid \mathcal{D})$ across the K functions, which can be used to transfer knowledge from the related tasks to the target task t (and thus reduce the required number of function evaluations on t).

Formally, using the assumption that the observed function values – conditioned on $\boldsymbol{x}$ – are normally distributed (with unknown mean and variance) we can first define our probabilistic function model as

$$p(f_t(\boldsymbol{x}) \mid \boldsymbol{x}, \theta) = \mathcal{N}(\hat{f}(\boldsymbol{x}, t; \theta_\mu), \theta_{\sigma^2}), \tag{3.1}$$

where $\theta = [\theta_\mu, \theta_{\sigma^2}]^T$, $\hat{f}(\boldsymbol{x}, t; \theta_\mu)$ is the output of a parametric neural network model with parameters θ_μ, and where we assume a homoscedastic noise model.[2] A single-task model can trivially be obtained from this definition:

Single-task model. In the single-task setting we simply model the function mean $\hat{f}(\boldsymbol{x}, t; \theta_\mu) = h(\boldsymbol{x}; \theta_\mu)$ as the output of a neural network, where h denotes the network output.

Multi-task model. For the multi-task model we use a slightly adapted network architecture. As additional input, the network is provided with a task-specific embedding vector. That is, we have $\hat{f}(\boldsymbol{x}, t; \theta_\mu) = h\left([\boldsymbol{x}, \psi_t]^T, \theta_h\right)$, where $h(\cdot)$, again, denotes the output of the neural network (here with parameters θ_h) and ψ_t is the t-th row of an embedding matrix $\psi \in \mathbb{R}^{K\times L}$ (we choose $L = 5$ for our experiments). This embedding matrix is learned alongside all other parameters. Additionally, if information about the dataset (such as data-set size etc.) is available it can be appended to this embedding vector. The full vector of the network parameters then becomes $\theta_\mu = [\theta_h, \text{vec}(\psi)]$, where $\text{vec}(\cdot)$ denotes vectorization. Instead of using a learned embedding we could have chosen to represent the tasks through a one-out-of K encoding vector, which functionally would be equivalent but would induce a large

[1] The standard single-task case is recovered when $K = t = 1$.

[2] We note that, if required, we could model heteroscedastic functions by defining the observation noise variance θ_{σ^2} as a deterministic function of $\boldsymbol{x}$ (e.g. as the second output of the neural network).

number of additional learned parameters for large K. With these definitions, the joint probability of the model parameters and the observed data is then

$$p(\mathcal{D}, \theta) = p(\theta_\mu)p(\theta_{\sigma^2}) \prod_{k=1}^{K} \prod_{i=1}^{|\mathcal{D}_k|} \mathcal{N}(y_k^i | \hat{f}(\boldsymbol{x}^i, k; \theta_\mu), \theta_{\sigma^2}), \tag{3.2}$$

where $p(\theta_\mu)$ and $p(\theta_{\sigma^2})$ are priors on the network parameters and on the variance, respectively.

For BO, we need to be able to compute the acquisition function at given candidate points $\boldsymbol{x}$. This relies on the predictive posterior $p(f_t(\boldsymbol{x}) \mid \boldsymbol{x}, \mathcal{D})$ (marginalized over the model parameters θ). Unfortunately, for our choice of modeling f_t with a neural network, evaluating this posterior exactly is intractable. Let us, for now, assume we can generate samples from the posterior for the model parameters given the data; we will show how to do this with stochastic gradient Hamiltonian Monte Carlo (SGHMC) in Section 3.4.

We can then use these samples $\theta^i \sim p(\theta \mid \mathcal{D})$ to approximate the predictive posterior $p(f_t(\boldsymbol{x})|\boldsymbol{x}, \mathcal{D})$ as

$$p(f_t(\boldsymbol{x}) \mid \boldsymbol{x}, D) = \int_\theta p(f_t(\boldsymbol{x}) \mid \boldsymbol{x}, \theta) p(\theta \mid D) d\theta \approx \frac{1}{M} \sum_{i=1}^{M} p(f_t(\boldsymbol{x}) \mid \boldsymbol{x}, \theta^i). \tag{3.3}$$

Using the same samples $\theta^i \sim p(\theta \mid \mathcal{D})$, we make a Gaussian approximation to this predictive distribution to obtain mean and variance to compute the acquisition functions described in Section 2.2.3:

$$\begin{aligned} \mu(f_t(\boldsymbol{x})|\mathcal{D}) &= \frac{1}{M} \sum_{i=1}^{M} \hat{f}(\boldsymbol{x}, t; \theta_\mu^i), \\ \sigma^2(f(\boldsymbol{x})|\mathcal{D}) &= \frac{1}{M} \sum_{i=1}^{M} \left(\hat{f}(\boldsymbol{x}, t; \theta_\mu^i) - \mu(f_t(\boldsymbol{x})|\mathcal{D}) \right)^2 + \theta_{\sigma^2}^i. \end{aligned} \tag{3.4}$$

Notably, we can compute partial derivatives of these acquisition functions (with respect to $\boldsymbol{x}$) via backpropagation through all functions $\hat{f}(\boldsymbol{x}, t; \theta_\mu^i)$. This allows for a gradient-based maximization of the acquisition function.

3.2. Related Work

The ability to combine the flexibility and scalability of (deep) neural networks with properly-calibrated uncertainty estimates would be very useful in many contexts. Consequently, there are many approaches for this problem, including early work on (non-scalable) Hamiltonian Monte Carlo (Neal, 1996), recent work on variational

inference methods (Graves, 2011; Blundell et al., 2015) and expectation propagation (Hernández-Lobato and Adams, 2015), re-interpretations of dropout as approximate inference (Gal and Ghahramani, 2016; Kingma et al., 2015), as well as stochastic gradient MCMC methods based on Hamiltonian Monte Carlo (Chen et al., 2014) and stochastic gradient Langevin MCMC (Welling and Teh, 2011).

While any of these methods could, in principle, be used for BO, we found most of them to result in suboptimal uncertainty estimates. Our preliminary experiments (see Section 3.3) suggest that they often conservatively estimate the uncertainty for points far away from the training data, particularly when based on little training data. This is problematic for BO, which crucially relies on well-calibrated uncertainty estimates based on few function evaluations. One family of methods that consistently resulted in good uncertainty estimates in our tests were Hamiltonian Monte Carlo (HMC) methods, which we will thus use throughout this Chapter. Concretely, we will build on the scalable stochastic MCMC method from Chen et al. (2014).

Using neural networks as underlying model for Bayesian optimization has been explored by others before. Snoek et al. (2015) proposed DNGO which uses a simple feed forward network to learn basis functions for Bayesian linear regression. Compared to Gaussian processes, Bayesian linear regression scales cubically with the number of input dimension. However, DNGO uses a point estimate of the weights and, in comparison to our method, does not allow to capture the epistemic uncertainty of the whole neural network.

3.3. Obtaining Well Calibrated Uncertainty Estimates

As mentioned in the related work section above, there exists a large body of work on Bayesian methods for neural networks. In preliminary experiments, we tried several of these methods to determine which algorithm was capable of providing well calibrated uncertainty estimates. All approximate inference methods we looked at (except for the MCMC variants) exhibited one of two problems (including the variational inference method from Blundell et al. (2015), the method from Gal and Ghahramani (2016) as well as the expectation propagation based approach from Hernández-Lobato and Adams (2015)): either they did severely underfit the data, or they poorly predicted the uncertainty in regions far from observed data points. The latter behaviour is exemplified in Figure 3.1 where we regressed the sinc function from 20 observations with a three layer neural network (50 tanh units each). In contrast, a fit of the same data with stochastic gradient Hamiltonian Monte-Carlo (SGHMC) (Chen et al., 2014) more faithfully represents model uncertainty as depicted in Figure 3.1 (bottom right). Based on this observation we decided to use SGHMC to approximate the posterior of our model. However, in its standard version it is quite sensitive to its hyperparameters in particular to the step length. We present a more robust version of SGHMC that automatically adjust its hyperparameters in the next section.

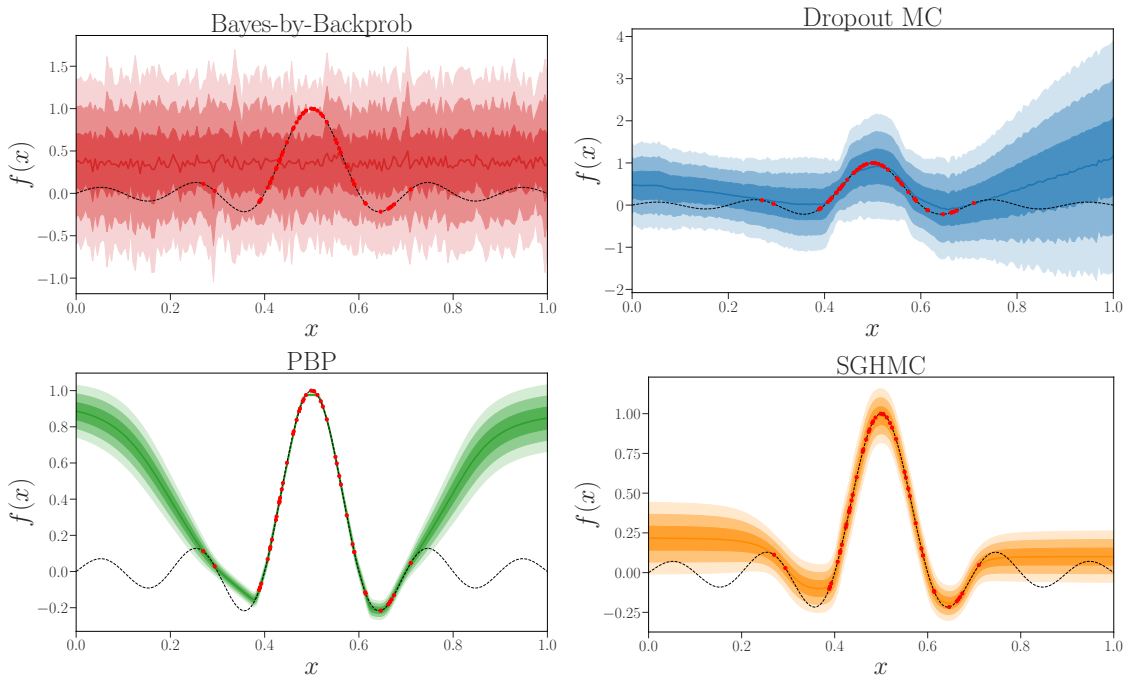


Figure 3.1.: Four fits of the sinc function from 20 data-points. On the top-left the regression task was solved using a re-implementation of the Bayes by Backprop (BBB) approach from Blundell et al. (2015). Note that, even though we were able to reproduce the toy regression example described in Blundell et al. (2015) with an third party open-source implementation, we were not able to find a better fit for this task even after tuning hyperparameters. On the top-right we used a re-implementation of the Dropout MC approach from Gal and Ghahramani (2016) and in the bottom-left the original implementation of probabilistic Backpropagation (Hernández-Lobato and Adams, 2015). On the bottom-right is a fit using SGHMC (Chen et al., 2014). As it can be observed all variational inference methods are overly confident in large regions of the input space or poorly extrapolate away from data. Note that this function has no observation noise.

3.4. Robust Stochastic Gradient Hamiltonian Monte-Carlo via Scale Adaptation

In this section, we show how to use stochastic gradient Hamiltonian Monte Carlo (Chen et al., 2014) (SGHMC) to sample from the model defined by Equation 3.2. We first summarize the general formalism behind SGHMC and then derive a robust variant suitable for BO.

3.4.1. Stochastic Gradient Hamiltonian Monte-Carlo

Hamiltonian Monte-Carlo (Neal, 1996) (HMC) introduces a set of auxiliary variables, $\boldsymbol{r}$, and samples from the joint distribution

$$p(\theta, \boldsymbol{r} \mid \mathcal{D}) \propto \exp\left(-U(\theta) - \frac{1}{2}\boldsymbol{r}^T\mathbf{M}^{-1}\boldsymbol{r}\right), \tag{3.5}$$

with $U(\theta) = -\log p(\mathcal{D}, \theta)$ by simulating a fictitious physical system described by a set of differential equations, called Hamilton's equations. In this system, the negative log-likelihood $U(\theta)$ plays the role of a potential energy, $\boldsymbol{r}$ corresponds to the momentum of the system, and $\mathbf{M}$ represents the (arbitrary) mass matrix (Duane et al., 1987).

Classically, the dynamics for θ and $\boldsymbol{r}$ depend on the gradient $\nabla U(\theta)$ whose evaluation is too expensive for our purposes, since it would involve evaluating the model on all data-points. By introducing a user-defined *friction* matrix $\mathbf{C}$, Chen et al. (2014) showed how Hamiltonian dynamics can be modified to sample from the correct distribution if only a noisy estimate $\nabla\tilde{U}(\theta)$, e.g. computed from a mini-batch, is available. In particular their discretized system of equations reads

$$\Delta\theta = \epsilon\mathbf{M}^{-1}\boldsymbol{r}\,, \qquad \Delta\boldsymbol{r} = -\epsilon\nabla\tilde{U}(\theta) - \epsilon\mathbf{C}\mathbf{M}^{-1}\boldsymbol{r} + \mathcal{N}(0, 2(\mathbf{C} - \hat{\mathbf{B}})\epsilon)\,, \tag{3.6}$$

where, in a suggestive notation, we write $\mathcal{N}(0, \Sigma)$ representing the addition of a sample from a multivariate Gaussian with zero mean and covariance matrix Σ. Besides the estimate for the noise of the gradient evaluation $\hat{\mathbf{B}}$, and an undefined step length ϵ, all that is required for simulating the dynamics in Equation 3.6 is a mechanism for computing gradients of the log likelihood (and thus of our model) on small subsets (or batches) of the data. This makes SGHMC particularly appealing when working with large models and data-sets. Furthermore, Equation 3.6 can be seen as an MCMC analogue to stochastic gradient descent (with momentum) (Chen et al., 2014). Following these update equations, the distribution of $(\theta, \boldsymbol{r})$ is the one in Equation 3.5, and θ is guaranteed to be distributed according to $p(\theta \mid \mathcal{D})$.

3.4.2. Scale Adapted Stochastic Gradient Hamiltonian Monte-Carlo

Like many Monte Carlo methods, SGHMC does not come without caveats, most importantly the correct setting of the user-defined quantities: the friction term $\mathbf{C}$, the estimate of the gradient noise $\hat{\mathbf{B}}$, the mass matrix $\mathbf{M}$, the number of MCMC steps, and – most importantly – the step-size ϵ. We found the friction term and the step-size to be highly model and data-set dependent, which is unacceptable for BO, where we require robust estimates across many different functions $\mathcal{F}$ with as few parameter choices as possible.

A closer look at Equation 3.6 shows why the step-size crucially impacts the robustness of SGHMC. For the popular choice $\mathbf{M} = \mathbf{I}$, the change in the momentum is proportional to the gradient. If the gradient elements are on vastly different scales (and potentially correlated), then the update effectively assigns unequal importance to changes in different parameters of the model. This, in turn, can lead to slow exploration of the target density. To correct for unequal parameter scales (and respect their correlation), we would ideally like to use $\mathbf{M}$ as a pre-conditioner, reflecting the metric underlying the model's parameters. This would lead to a stochastic gradient analogue of Riemann Manifold Hamiltonian Monte Carlo (Girolami and Calderhead, 2011), which has been studied before by Ma et al. (2015) and results in an algorithm called generalized stochastic gradient Riemann Hamiltonian Monte Carlo (gSGRHMC). Unfortunately, gSGRHMC requires the computation (and storage) of the full Fisher information matrix of U and its gradient, which is prohibitively expensive for our purposes.

As a pragmatic approach, we consider a pre-conditioning scheme increasing SGHMCs robustness with respect to ϵ and $\mathbf{C}$, while avoiding the costly computations of gSGRHMC. We want to note that recently – and directly related to our approach – adaptive pre-conditioning using ideas from SGD methods has been combined with stochastic gradient Langevin dynamics in Li et al. (2016) and to derive a hybrid between SGD optimization and HMC sampling in Chen et al. (2016), these however either come with additional hyperparameters that need to be set or do not guarantee unbiased sampling. The rest of this section shows how all remaining parameters in our method are determined automatically.

Choosing M. For the mass matrix, we take inspiration from the connection between SGHMC and SGD. Specifically, the literature (Tieleman and Hinton, 2012; Duchi et al., 2011) shows how normalizing the gradient by its magnitude (estimated over the whole dataset) improves the robustness of SGD. To perform the analogous operation in SGHMC, we propose to adapt the mass matrix *during the burn-in phase*. We set $\mathbf{M}^{-1} = \text{diag}\left(\hat{V}_\theta^{-1/2}\right)$, where $\hat{V}_\theta$ is an estimate of the (element-wise) uncentered variance of the gradient: $\hat{V}_\theta \approx \mathbb{E}[(\nabla\tilde{U}(\theta))^2]$. We estimate $\hat{V}_\theta$ using an exponential moving average during the burn-in phase which yields the update equation:

$$\Delta\hat{V}_\theta = -\tau^{-1}\hat{V}_\theta + \tau^{-1}\nabla(\tilde{U}(\theta))^2, \tag{3.7}$$

where τ is a free parameter vector specifying the exponential averaging windows. Note all multiplications above are element-wise and τ is a vector with the same dimensionality as θ.

Automatically choosing τ. To avoid adding τ as a new hyperparameter – that we would have to tune – we automatically determine its value. For this purpose, we use an adaptive estimate previously derived for adaptive learning rate procedures for SGD (Schaul et al., 2013). We maintain an additional smoothed estimate of the

gradient $g_\theta \approx \nabla U(\theta)$ and consider the element-wise ratio $g_\theta^2/\hat{V}_\theta$ between the squared estimated gradient and the gradient variance. This ratio will be large if the estimated gradient is large compared to the noise – in which case we can use a small averaging window – and it will be small if the noise is large compared to the average gradient – in which case we want a larger averaging window. We formalize these desiderata by simultaneously updating Equation 3.7,

$$\Delta\tau = -g_\theta^2 \hat{V}_\theta^{-1}\tau + 1\,, \qquad \text{and} \qquad \Delta g_\theta = -\tau^{-1} g_\theta + \tau^{-1}\nabla\tilde{U}(\theta)\,. \tag{3.8}$$

Estimating $\hat{\mathbf{B}}$. While the above procedure removes the need to hand-tune $\mathbf{M}^{-1}$ (and will stabilize the method for different $\mathbf{C}$ and ϵ), we have not yet defined an estimate for $\hat{\mathbf{B}}$. Ideally, $\hat{\mathbf{B}}$ should be the estimate of the empirical Fisher information matrix that, as discussed above, is too expensive to compute. We therefore resort to a diagonal approximation yielding $\hat{\mathbf{B}} = \frac{1}{2}\epsilon\hat{V}_\theta$ which is readily available from Equation 3.7.

Scale adapted update equations. Finally, we can combine all parameter estimates to formulate our automatically scale adapted SGHMC method. Following Chen et al. (2014), we introduce the variable substitution $\mathbf{v} = \epsilon\mathbf{M}^{-1}\boldsymbol{r} = \epsilon\hat{V}_\theta^{-1/2}\boldsymbol{r}$ which leads us to the dynamical equations

$$\Delta\theta = \mathbf{v}\,, \tag{3.9}$$

$$\Delta\mathbf{v} = -\epsilon^2\hat{V}_\theta^{-1/2}\nabla\tilde{U}(\theta) - \epsilon\hat{V}_\theta^{-1/2}\mathbf{C}\mathbf{v} + \mathcal{N}\left(0, 2\epsilon^3\hat{V}_\theta^{-1/2}\mathbf{C}\hat{V}_\theta^{-1/2} - \epsilon^4\mathbf{I}\right)\,, \tag{3.10}$$

using the quantities estimated in Equations 3.7 and 3.8 during the burn-in phase, and then fixing the choices for all parameters. Note that the approximation of $\hat{\mathbf{B}}$ cancels with the square of our estimate of $\mathbf{M}^{-1}$. In practice, we choose $\mathbf{C} = C\mathbf{I}$, i.e. the same independent noise for each element of θ. In this case, Equations 3.9 and 3.10 constrain the choices of C and ϵ, as we need them to fulfill the relation $\min(V_\theta^{-1})C \geq \epsilon$. For the remainder of the paper, we fix $\epsilon = 10^{-2}$ (a robust choice in our experience) and chose C such that we have $\epsilon\hat{V}_\theta^{-1/2}\mathbf{C} = 0.05\mathbf{I}$ (intuitively this corresponds to a constant decay in momentum of 0.05 per time step) potentially increasing it to satisfy the mentioned constraint at the end of the burn-in phase.

We want to emphasize that our estimation/adaptation of the parameters only changes the HMC procedure during the burn-in phase. After it, when actual samples are recorded, all parameters stay fixed. In particular, this entails that as long as our choice of ϵ and C satisfies $\min(\hat{V}_\theta^{-1})C \geq \epsilon$, our method samples from the correct distribution. Our choices are compatible with the constraints on the free parameters of the original SGHMC (Chen et al., 2014). We also note that the scale adaptation technique is agnostic to the parametric form of the density we aim to sample from. As such, it could thus potentially also simplify SGHMC sampling for models beyond those considered in this paper.

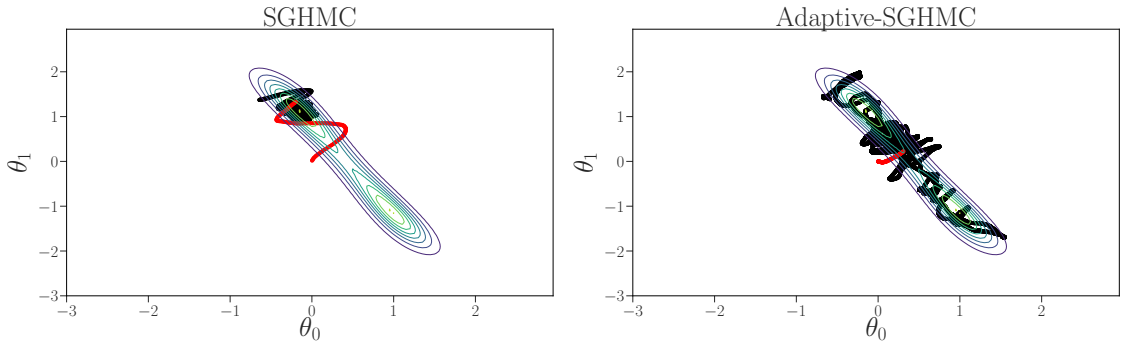


Figure 3.2.: The sampling trajectories for SGHMC (left) and our scale-adapted SGHMC (right). See text for the details about the energy landscape. The red points indicate samples drawn during the burn-in phase where as the black points represent samples draw during the sampling phase. Both methods start with the same step length, but our method can adapt and thus explores the full distribution.

3.5. Experiments on the Effects of Scale Adaptation

Before we evaluate how well our method works in Bayesian optimization, we first study the effects of our scale-adapted version of stochastic gradient Hamiltonian Monte-Carlo. In the first experiments, we present a simple two dimensional case where we can visualize the full MCMC trajectory. Afterwards, in the second set of experiments we investigate the performance of Bayesian neural networks on some standard regression benchmarks.

3.5.1. Toy Function

To visualize the effect of our scale adaption we consider the Gaussian mixture model example by Welling and Teh (2011) which is defined as:

$$x_i \sim \frac{1}{2}\mathcal{N}(\theta_0, \sigma_x^2) + \frac{1}{2}\mathcal{N}(\theta_0 + \theta_1, \sigma_x^2)$$

We generated 100 data points from the model with $\theta_0 = 0$, $\theta_1 = 1$ and $\sigma_x^2 = 2$. During the sampling, we kept σ_x^2 fix and sampled different values for θ_0 and θ_1 with standard SGHMC and our scale-adapted version. We ran both methods with a step length of $\epsilon = 10^{-2}$ and batch size of 10 for 10000 steps where we used the first 100 steps as burn-in. Figure 3.2 shows the trajectory for both samplers. One can see that the step length for SGHMC is too small to make sufficient progress and it gets stuck in one mode of the distribution. Even though our methods starts with the same step length as SGHMC, it can adapt such that it captures the full distribution eventually.

3.5.2. Regression Datasets

To test the efficacy of our novel scale adaptation technique for neural networks we performed an evaluation on five common UCI regression datasets (Lichman, 2013). The neural network architecture consisted of one fully-connected layer with 50 units and tanh activation functions followed by a linear output layer. For each dataset we picked a 90% training split and 10% test split randomly as described by Hernández-Lobato and Adams (2015). We compare standard SGHMC with different step lengths to our scale adaption with a fixed steps length of 10^{-2}. Both methods were evaluated for 15000 steps where the first 1000 steps are treated as burn-in.

Table 3.1 shows the log-likelihood and Table 3.2 the root mean-squared-error averaged over 50 independent runs for SGHMC with and without our scale adaption. For each run we sampled a new training and test split. The comparison shows that SGHMC requires different step lengths for different datasets and our adaption works as well or better than SGHMC with the best step length.

Table 3.1.: Log likelihood for UCI regression datasets.

Method / Dataset	Boston-Housing	Concrete	Yacht	Wine	Power-Plant
SGHMC 10^{-6}	-444.87 ± 145.27	-442.20 ± 77.25	-475.58 ± 199.29	-425.61 ± 56.01	-39.36 ± 7.75
SGHMC 10^{-5}	-79.53 ± 41.20	-80.19 ± 12.13	-94.61 ± 29.65	-80.69 ± 10.28	-12.35 ± 0.74
SGHMC 10^{-4}	-6.71 ± 2.29	**-4.84 ± 0.39**	-9.70 ± 2.60	**-1.73 ± 0.28**	-7.37 ± 0.99
SGHMC 10^{-3}	-8.03 ± 3.27	-5.88 ± 1.64	-4.44 ± 1.31	-3.83 ± 1.86	nan ± nan
adaptive SGHMC	**-6.34 ± 2.40**	-5.18 ± 0.70	**-2.32 ± 1.36**	-9.09 ± 1.66	**-3.17 ± 0.14**

Table 3.2.: Root mean squared error for UCI regression datasets.

Method / Dataset	Boston-Housing	Concrete	Yacht	Wine	Power-Plant
SGHMC 10^{-6}	8.92 ± 1.52	16.35 ± 1.47	14.56 ± 2.78	0.79 ± 0.05	6.00 ± 0.52
SGHMC 10^{-5}	4.88 ± 1.07	9.92 ± 0.66	9.10 ± 1.67	0.64 ± 0.04	4.44 ± 0.14
SGHMC 10^{-4}	4.44 ± 0.84	8.23 ± 0.56	8.45 ± 1.26	0.64 ± 0.05	508.53 ± 369.37
SGHMC 10^{-3}	545.12 ± 640.04	216.14 ± 491.04	19.22 ± 17.74	35.56 ± 64.34	nan ± nan
adaptive SGHMC	**3.24 ± 0.64**	**6.22 ± 0.56**	**0.93 ± 0.31**	**0.63 ± 0.04**	**4.21 ± 0.18**

3.6. Bayesian Optimization Experiments

We now show BO experiments for BOHamiANN. Unless noted otherwise, we used a three layer neural network with 50 units and tanh activation functions for all experiments. Since we are iteratively collecting more data we adapted the number of burnin steps over time by a constant factor of 100 times the number of data points. To obtain samples from the posterior we used a constant chain length of 10000 steps with thinning such that we only kept every 100th sample. For the priors we let $p(\theta_\mu) = \mathcal{N}(0, \sigma_\mu^2)$ be normally distributed and we chose a log-normal prior $p(\theta_\sigma^2)$.

Since other BO methods, such as random forests, do not provide gradients, we used the gradient-free method differential evolution (Storn and Price, 1997) to optimize the acquisition function.

3.6.1. Synthetic Objective Functions

As a first experiment we compare BOHamiANN to BO with random forests and Gaussian processes as probabilistic models as well as random search on a set of 8 different synthetic functions from the global optimization literature. For each method we report the mean (solid line) regret to the global optimum and the standard error of the mean (shaded areas) of 50 independent runs. All optimizers achieve acceptable performance, but GP based methods were found to perform best on these low-dimensional benchmarks and we thus take them as a point of reference. Overall, on the 8 benchmarks BOHamiANN matched the performance of GP based BO on 2, performed better on 2 and performed worse on 4, indicating that even in the low-data regime Bayesian neural networks (BNNs) are a feasible model class for BO. See Figure 3.3 for three example benchmarks. A detailed listing of all the results is given in Section A.1 in the supplementary material.

Additionally, we compare to our re-implementation of the recently proposed DNGO method (Snoek et al., 2015), which uses features extracted from a maximum likelihood fit of a neural network as the basis for a Bayesian linear regression fit (and was also proposed as a replacement of GPs for scalable BO). For the benchmark tasks we found BOHamiANN to perform consistently better and being slightly more robust to different architecture choices.

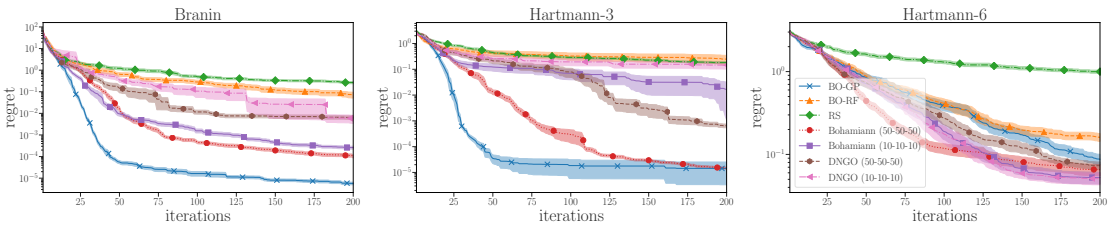


Figure 3.3.: Evaluation on common benchmark problems with different dimensionality. Immediate mean regret of various optimizers averaged over 50 runs on the Branin (left), Hartmann3 (middle) and Hartmann6 (right) function (shaded areas represent the standard error of the mean). For DNGO and BOHamiANN, we denote the layer sizes for the (3 layer) networks in parenthesis.

3.6.2. Multi-Task Hyperparameter Optimization

Next, we consider the hyperparameter optimization of the popular gradient-boosting method XGBoost (Chen and Guestrin, 2016) over a range of different regression

problems. Concretely, we consider a set of 10 different regression datasets downloaded from the UCI repository (Lichman, 2013). The definition of the configuration space for XGBoost is given in Table A.1 in the supplementary material (see Section A.2). All hyperparameters are treated as continuous parameters, where integer parameters, such as the number of estimators, are rounded before we pass them to XGBoost.

We compare BOHamiANN to Gaussian process based BO (GP-BO) and random search (RS). Additionally we also include the multi-task version of BOHamiANN (MT-Bohamiann) as well as the multi-task BO (MT-BO) procedure from Swersky et al. (2013). MT-BO is based on Gaussian processes and infers the Cholesky decomposition to model the correlations across tasks with the other GP hyperparameters. For BOHamiANN, we use an embedding matrix, which gets a one-hot encoded task variable as input and maps to a $L = 5$ dimensional feature vector. Both multi-task methods are warm-started from 10 random configurations evaluated on a randomly picked dataset out of the other 9 remaining datasets. We performed 30 independent runs of each method and report the mean and the standard error of the mean for 3 benchmarks in Figure 3.4 and for all benchmarks in Section A.2 in the supplement material.

First of all one can see that BOHamiANN works consistently better than GP-BO and RS on these type of benchmarks. Furthermore, using previously acquired knowledge always helped to speed up the optimization process with the Wine dataset being the only exception. We attribute this to the different scale of the function values compared to the auxiliary datasets (Parkinson Telemonitoring).

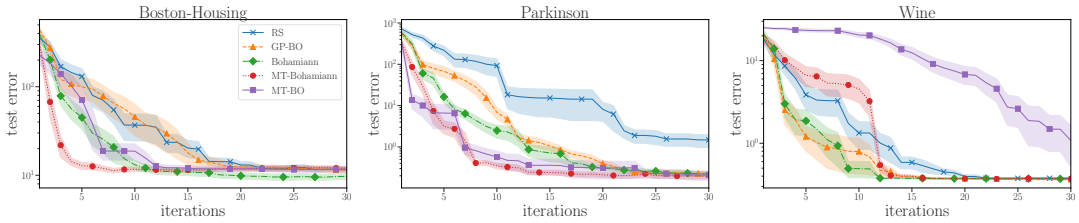


Figure 3.4.: Comparison of Bohamiann to Gaussian process based Bayesian optimization (GP-BO) and random search (RS). Additionally, we also include the multi-task variant of the two Bayesian optimization methods (denoted as MT-Bohamiann and MT-BO respectively).

3.7. Chapter Conclusion

We proposed BOHamiANN, a scalable and flexible Bayesian optimization method based on Bayesian neural networks. It natively supports multi-task optimization, and scales to high dimensions and many function evaluations. At its heart lies Bayesian inference for neural networks via stochastic gradient Hamiltonian Monte Carlo, and we improved the robustness thereof by means of a scale adaptation technique.

Due to their efficient scaling Bayesian neural networks are a natural candidate for multi-fidelity modelling such as learning curves which can be considered as a fidelity of the asymptotic performance of hyperparameter configurations. In Chapter 5 we will show how our model can be adapted to model the learning curves of iterative machine learning algorithms with respect to their hyperparameters. Besides that, we use BOHamiANN in Chapter 8 to learn a genrative meta-model across tasks.

4. Bayesian Optimization on large Datasets

As we have seen in the previous two chapters, Bayesian optimization is an efficient method for hyperparameter optimization. In its traditional setting, the loss of a machine learning algorithm with hyperparameters $\boldsymbol{x} \in \mathbb{X}$ is treated as the "black-box" problem of finding $\arg\min_{\boldsymbol{x}\in\mathbb{X}} f(\boldsymbol{x})$, where the only mode of interaction with the objective f is to evaluate it for inputs $\boldsymbol{x} \in \mathbb{X}$. If individual evaluations of f on the entire dataset require days or weeks, only very few evaluations are possible, limiting the quality of the best found value. Human experts instead often study performance on subsets of the data first, to become familiar with its characteristics before gradually increasing the subset size (Bottou, 2012). This approach can still outperform contemporary Bayesian optimization methods.

Motivated by the experts' strategy, here we leverage dataset size as an additional degree of freedom enriching the representation of the optimization problem. We treat the size of a randomly subsampled dataset N_{sub} as an additional input to the blackbox function, and allow the optimizer to actively choose it at each function evaluation. This allows Bayesian optimization to mimic and improve upon human experts when exploring the hyperparameter space. In the end, N_{sub} is not a hyperparameter itself, but the goal remains a good performance on the full dataset, i.e. $N_{sub} = N$.

While in this chapter we focus on hyperparameter optimization for large datasets, in principle, our method could also be applied to other scenarios where cheap but potentially biased and noisy approximations of the actual objective function are available, such as, for instance, in the work by Kandasamy et al. (2016), which introduces a Bayesian optimization variant that can optimize expensive functions by exploiting cheaper fidelities. Our method's only assumption is that one can define a proper basis function to describe the similarity between the objective function and its approximations. Another interesting application would be to likelihood-free inference, where Bayesian optimization has been successfully applied before (Gutmann and Corander, 2016).

Hyperparameter optimization for large datasets has been explored by other authors before. Our approach is similar to Multi-Task Bayesian optimization by Swersky et al. (2013), where knowledge is transferred between a finite number of correlated tasks. If these tasks represent manually-chosen subset-sizes, this method also tries to find the best configuration for the full dataset by evaluating smaller, cheaper subsets. However, the discrete nature of tasks in that approach requires evaluations on the

entire dataset to learn the necessary correlations. Instead, our approach exploits the regularity of performance across dataset size, enabling generalization to the full dataset without evaluating it directly.

Other approaches for hyperparameter optimization on large datasets include work by Nickson et al. (2014), who estimated a configuration's performance on a large dataset by evaluating several training runs on small, random subsets of fixed, manually-chosen sizes. Krueger et al. (2015) showed that, in practical applications, small subsets can suffice to estimate a configuration's quality, and proposed a cross-validation scheme that sequentially tests a fixed set of configurations on a growing subset of the data, discarding poorly-performing configurations early.

Li et al. (2017) proposed a multi-arm bandit strategy, called Hyperband, which dynamically allocates more and more resources to randomly sampled configurations based on their performance on subsets of the data. Hyperband assures that only well-performing configurations are trained on the full dataset while discarding bad ones early. Despite its simplicity, in their experiments the method was able to outperform well-established Bayesian optimization algorithms.

The remainder of the chapter is structured as follow: We first review Entropy Search (Hennig and Schuler, 2012) on which our method is based in Section 4.1. In Section 4.2 we show that subsets of the training data are often sufficient to reason about the performance of a hyperparameter configuration. In Section 4.3 we then present previous approaches such as Multi-task Bayesian optimization and Hyperband. In Section 4.4, we introduce our new Bayesian optimization method FABOLAS for hyperparameter optimization on large datasets. In each iteration, FABOLAS chooses the configuration $\boldsymbol{x}$ and dataset size N_{sub} predicted to yield most information about the loss-minimizing configuration on the full dataset per *unit time spent*. Finally, in Section 4.5, a broad range of experiments with support vector machines and convolutional deep neural networks show that FABOLAS often identifies good hyperparameter settings 10 to 100 times faster than state-of-the-art Bayesian optimization methods acting on the full dataset, as well as Hyperband.

4.1. Entropy Search

Entropy Search (Hennig and Schuler, 2012; Villemonteix et al., 2008) is a more recent acquisition function that selects evaluation points based on the predicted *information gain* about the optimum, rather than aiming to evaluate near the optimum. At the heart of ES lies the probability distribution $p_{\min}(\boldsymbol{x} \mid \mathcal{D}) := p(\boldsymbol{x} \in \arg\min_{\boldsymbol{x}' \in \mathbb{X}} f(\boldsymbol{x}') \mid \mathcal{D})$, the belief about the function's minimum given the prior on f and observations $\mathcal{D}$. Given $p(f)$, the probability that a point is the minimum is defined with suggestive

notation as

$$\begin{aligned} p_{min}(\boldsymbol{x}|\mathcal{D}) &= p(\boldsymbol{x} \in \arg\min_{\boldsymbol{x}' \in \mathcal{X}} f(\boldsymbol{x}')|D) \\ &= \int p(f|\mathcal{D}) \prod_{\substack{\tilde{\boldsymbol{x}} \in \mathcal{X} \\ \tilde{\boldsymbol{x}} \neq \boldsymbol{x}}} \Theta[f(\tilde{\boldsymbol{x}}) - f(\boldsymbol{x})] \, df \end{aligned} \tag{4.1}$$

where Θ is the Heaviside step function. The product in this equation is over an infinite domain (yet well-defined if $p(f|D)$ is sufficiently regular). In practice, it has to be represented in a finite form. We follow the approach of Hennig and Schuler (2012), who approximate $p(f|D)$ by a finite-dimensional Gaussian over an irregular grid of points $\boldsymbol{r}_1, ..., \boldsymbol{r}_Z$, which are designed heuristically to provide good interpolation resolution on $p_{\min}$. Like Hennig and Schuler (2012), we sample these so called representer points using Expected Improvement (see Section 2.2.3). This step reduces $p_{\min}$ to a discrete distribution, and turns the infinite product in Equation 4.1 into a finite one. That distribution itself is still analytically intractable, but an analytically tractable (in particular, differentiable) approximation $q_{\min}(\boldsymbol{r}_j)$ of good empirical quality can be computed using *Expectation Propagation (EP)* (Minka, 2001), of computational cost $\mathcal{O}(Z^4)$. EP does not only yield $p_{\min}$, but also the gradient with respect to means and covariances of the model at the representer points allowing efficient computations after an expensive initial calculation of these quantities. This particular application of EP (dubbed EPMGP) to Gaussian integrals was introduced by Cunningham et al. (2011) where all the details can be found.

The *information gain* at $\boldsymbol{x}$ is then measured by the expected Kullback-Leibler divergence (relative entropy) between $p_{\min}(\cdot \mid \mathcal{D} \cup \{(\boldsymbol{x}, y)\})$ and the uniform distribution $u(\boldsymbol{x})$, with expectations taken over the measurement y to be obtained at $\boldsymbol{x}$:

$$a_{\mathrm{ES}}(\boldsymbol{x}) := \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D})} \left[\int p_{\min}(\boldsymbol{x}' \mid \mathcal{D}') \cdot \log \frac{p_{\min}(\boldsymbol{x}' \mid \mathcal{D}')}{u(\boldsymbol{x}')} \, \mathrm{d}\boldsymbol{x}' \right], \tag{4.2}$$

where $\mathcal{D}' = \mathcal{D} \cup \{(\boldsymbol{x}, y)\}$. The primary numerical challenge in this framework is the computation of $p_{\min}(\cdot \mid \mathcal{D}')$ and the integral above. Due to the intractability, several approximations have to be made.

Algorithm 2 provides pseudocode for our implementation of Entropy Search. Lines 1-12 precompute various quantities that are needed for evaluating the acquisition function, which is optimized in line 13. Specifically, after sampling K hyperparameter settings from the marginal loglikelihood for the GP using MCMC (line 1), for every hyperparameter setting $\boldsymbol{\theta}_i$, the algorithm

- fits a GP (line 4),
- samples representer points with respect to a_{EI} (line 5),
- stores the representer points and their logarithmic EI values (lines 6 and 7),
- computes $\boldsymbol{\mu}$ and Σ for the joint predictive distribution at the representer points (line 8),

- computes p_{min} given $\boldsymbol{\mu}$ and Σ, using EPMGP (line 9),
- draws random points from a normal distribution centered at 0 and unit variance (line 10) for the innovation in Algorithm 4, and stores them (line 11) for later usage.

Algorithm 2 Selection of next point by Entropy Search

Require: $\mathcal{D}_n = (\boldsymbol{x}_j, y_j)_{j=1\ldots n}$

1: Sample K instantiations of the GP hyperparameters $\boldsymbol{\Theta} = [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K]$ w.r.t. marginal likelihood
2: $p_{min} \leftarrow [], \Omega \leftarrow [], \boldsymbol{R} \leftarrow [], \boldsymbol{U} \leftarrow []$
3: **for** $i = 1 \ldots K$ **do**
4: Fit GP model $\mathcal{M}^{(i)}$ on $\mathcal{D}_n$ with hyperparameter $\boldsymbol{\theta}_i$
5: $(\boldsymbol{r}_1, a_{EI}(\boldsymbol{r}_1)) \ldots, (\boldsymbol{r}_Z, a_{EI}(\boldsymbol{r}_Z)) \sim a_{EI}(\boldsymbol{x}|\mathcal{M}^{(i)})$ ▷ Sample Z representer points
6: $\boldsymbol{R}[i] \leftarrow \boldsymbol{r}_1, \ldots, \boldsymbol{r}_Z$ ▷ Store representer points $R \in \mathbb{R}^{K \times Z \times D}$
7: $\boldsymbol{U}[i] \leftarrow a_{EI}(\boldsymbol{r}_1), \ldots, a_{EI}(\boldsymbol{r}_Z)$ ▷ Store EI values of the representer points $U \in \mathbb{R}^{K \times Z}$
8: Let $\boldsymbol{\mu}, \Sigma$ be the mean and covariance matrix at $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_Z$ based on $\mathcal{M}^{(i)}$
9: $p_{min}[i] \leftarrow$ computePmin $(\boldsymbol{\mu}, \Sigma)$ ▷ Probability of each $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_Z$ to be the minimum.
10: For $p = 1, \ldots, P$: $\boldsymbol{\omega}_p \sim \mathcal{N}(0, \boldsymbol{I}_Z)$ ▷ Stochastic change to hallucinate P values at representer points
11: $\Omega[i] \leftarrow [\boldsymbol{\omega_1}, \ldots, \boldsymbol{\omega}_P]$ ▷ Store stochastic change for the innovations $\Omega \in \mathbb{R}^{K \times Z \times P}$
12: **end for**
13: $\boldsymbol{x}_{n+1} \leftarrow \arg\max_{\boldsymbol{x} \in \mathcal{X}}$ InformationGain$(\boldsymbol{x}, \mathcal{D}_n, \boldsymbol{R}, \boldsymbol{U}, \Omega, \boldsymbol{\Theta})$
14: **return** $\boldsymbol{x}_{n+1}$

Given these quantities, Algorithm 3 then computes the ES acquisition function from Equation 4.2.

For each hyperparameter $\boldsymbol{\theta}_i$ of the GP, it then carries out the following steps:

- train a model $\mathcal{M}^{(i)}$ on the data $\mathcal{D}$ by computing the Cholesky decomposition (line 3)
- based on this model $\mathcal{M}^{(i)}$, compute the mean and the variance for the test point $\boldsymbol{x}$ and the mean and covariance for the representer points $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_Z$ (line 4 and 5)
- For each of the P stochastic change vectors $\boldsymbol{\omega}_p$ sampled in Algorithm 1 (line 10) and stored in $\Omega[i, j, :]$,
  - fantasize the change $\Delta\boldsymbol{\mu}, \Delta\Sigma$ of the current posterior $p(f|D)$ (line 7) with Algorithm 4
  - estimate the p_{min} distribution of this updated posterior (line 8)

  - compute the relative change in entropy (line 9)
- take the expectation over $p(y|\boldsymbol{x}, D)$ of Equation (3) (line 10)
- marginalize the acquisition function $a_{\text{ES}}(\boldsymbol{x})$ over all hyperparameters $\boldsymbol{\Theta}$ (line 13)

Algorithm 3 InformationGain

Require: $\boldsymbol{x}, \mathcal{D}, \boldsymbol{R}, \boldsymbol{U}, \Omega, \boldsymbol{\Theta}$

1: $a(\boldsymbol{x}) \leftarrow 0$
2: **for** $i = 1, \dots K$ **do** ▷ Marginalization over $\boldsymbol{\Theta}$
3: Let $\mathcal{M}^{(i)}$ be the trained model on $\mathcal{D}$ with hyperparameters $\boldsymbol{\theta}_i$
4: Let μ, σ^2 be the predictive mean and variance at $\boldsymbol{x}$ based on $\mathcal{M}^{(i)}$
5: Let $\boldsymbol{\mu}, \Sigma$ be the mean and covariance matrix at $\boldsymbol{r}_1, \dots, \boldsymbol{r}_Z$ based on $\mathcal{M}^{(i)}$
6: **for** $j = 0, \dots P$ **do** ▷ Averages over all hallucinated values.
7: $\Delta\boldsymbol{\mu}, \Delta\Sigma \leftarrow$ Innovations$(\boldsymbol{x}, \mathcal{M}^{(i)}, \boldsymbol{R}[i, :, :], \sigma^2, \Omega[i, j, :])$ ▷ Change in the posterior believe at $\boldsymbol{r}_1, \dots, \boldsymbol{r}_Z$ if we would evaluate at $\boldsymbol{x}$
8: $q_{min} \leftarrow$ computePmin $(\boldsymbol{\mu} + \Delta\boldsymbol{\mu}, \Sigma + \Delta\Sigma)$ ▷ New Pmin of the updated posterior
9: $d\,\mathrm{H} \leftarrow -\sum_j q_{min}\Big(\log(q_{min}) + \boldsymbol{U}[i]\Big) + \sum_j p_{min}[i]\Big(\log(p_{min}[i]) + \boldsymbol{U}[i]\Big)$
10: $a(\boldsymbol{x}) \leftarrow a(\boldsymbol{x}) + \frac{1}{P} dH$
11: **end for**
12: **end for**
13: **return** $\frac{1}{K} a(\boldsymbol{x})$

This algorithm in turns makes use of Algorithm 4 to compute the innovations, which

- computes the change in the mean $\Delta\boldsymbol{\mu}$ by first computing the correlation $\Sigma(\boldsymbol{x}, \boldsymbol{r})$ of $\boldsymbol{x}$ and the representer points $\boldsymbol{r}_1, \dots, \boldsymbol{r}_Z$ and multiplying it with the Cholesky decomposition of the $k(\boldsymbol{x}, \boldsymbol{x})$ and the vector $\omega \in \boldsymbol{\Omega}$. Note that this change is stochastic (line 1).
- computes the change of the covariance (line 2) which is deterministic

Algorithm 4 Innovations

Require: $\boldsymbol{x}, \mathcal{M}, \boldsymbol{r}_1, \dots, \boldsymbol{r}_Z, \sigma^2, \boldsymbol{\omega}$

1: $\Delta\boldsymbol{\mu}(\boldsymbol{x}) = \Sigma(\boldsymbol{x}, \boldsymbol{r}) \cdot \sigma^2 \cdot C[\sigma^2 + \sigma^2_{noise}]\boldsymbol{\omega}$ ▷ $\Sigma(\boldsymbol{x}, \boldsymbol{x}')$ denotes the correlation between $\boldsymbol{x}$ and $\boldsymbol{x}'$ based on $\mathcal{M}$
2: $\Delta\Sigma(\boldsymbol{x}) = \Sigma(\boldsymbol{x}, \boldsymbol{r}) \cdot \sigma^2 \cdot \Sigma(\boldsymbol{x}, \boldsymbol{r})^T$
3: **return** $\Delta\boldsymbol{\mu}(\boldsymbol{x}), \Delta\Sigma(\boldsymbol{x})$

Despite the conceptual and computational complexity of ES, it offers a well-defined concept for information gained from function evaluations, which can be meaningfully traded off against other quantities, such as the evaluations' cost.

4.2. Reasoning Across Dataset Subsets

The runtime of machine learning algorithms usually scales polynomially with the number of data points N_{sub}, i.e. $\mathcal{O}(N_{\text{sub}}^{\alpha})$ for some positive α. While the computational cost of training grows, the loss of machine learning methods usually decreases with the number of training samples. The computational cost is often largely independent of the hyperparameter values, but the loss depends crucially on them (which is the reason we want to optimize them in the first place).

For an intuition on how performance changes with dataset size, we evaluated a grid of 400 configurations of a support vector machine (SVM) on subsets of the MNIST dataset (LeCun et al., 2001) ; MNIST has $N = 50\,000$ data points and we evaluated relative subset sizes $s := {}^{N_{sub}}/_{N} \in \{{}^{1}/_{512}, {}^{1}/_{256}, {}^{1}/_{128}, \ldots, {}^{1}/_{4}, {}^{1}/_{2}, 1\}$.

Figure 4.1 visualizes the validation error (top) and training time (bottom) of these configurations on $s = {}^{1}/_{128}$, ${}^{1}/_{16}$, ${}^{1}/_{4}$, and 1. Evidently, just ${}^{1}/_{128}$ of the dataset is quite representative and sufficient to locate a reasonable configuration. Additionally, there are no deceiving local optima on smaller subsets. The training time, however increases substantially with the number of datapoints, single configurations take only a few seconds to train on $s = {}^{1}/_{128}$ but can take up to a few hours on the full dataset. Based on these observations, we expect that relatively small fractions of the dataset yield representative performances and therefore vary our relative size parameter s on a logarithmic scale.

4.3. Previous Work

Making use of dataset subsets to seed up hyperparameter optimization has been investigated by others before. In this Section we will present two approaches that are similar to ours, namely Multi Task Bayesian Optimization and Hyperband.

4.3.1. Multi-Task Bayesian Optimization

The *Multi-Task* Bayesian optimization (MTBO) method by Swersky et al. (2013) refers to a general framework for optimizing in the presents of different, but correlated tasks. Given a set of such tasks $\mathbb{T} = \{1, \ldots, T\}$, the objective function $f : \mathbb{X} \times \mathbb{T} \to \mathbb{R}$ corresponds to evaluating a given $\boldsymbol{x} \in \mathbb{X}$ on one of the tasks $t \in \mathbb{T}$. The relation between points in $\mathbb{X} \times \mathbb{T}$ is modeled via a GP using a product kernel:

$$k_{\text{MT}}((\boldsymbol{x}, t), (\boldsymbol{x}', t')) = k_T(t, t') \cdot k_{5/2}(\boldsymbol{x}, \boldsymbol{x}')\,. \tag{4.3}$$

The kernel k_T is represented implicitly by the Cholesky decomposition of $k(\mathbb{T}, \mathbb{T})$ whose entries are sampled via MCMC together with the other hyperparameters of the GP. By considering the distribution over the optimum on the target task

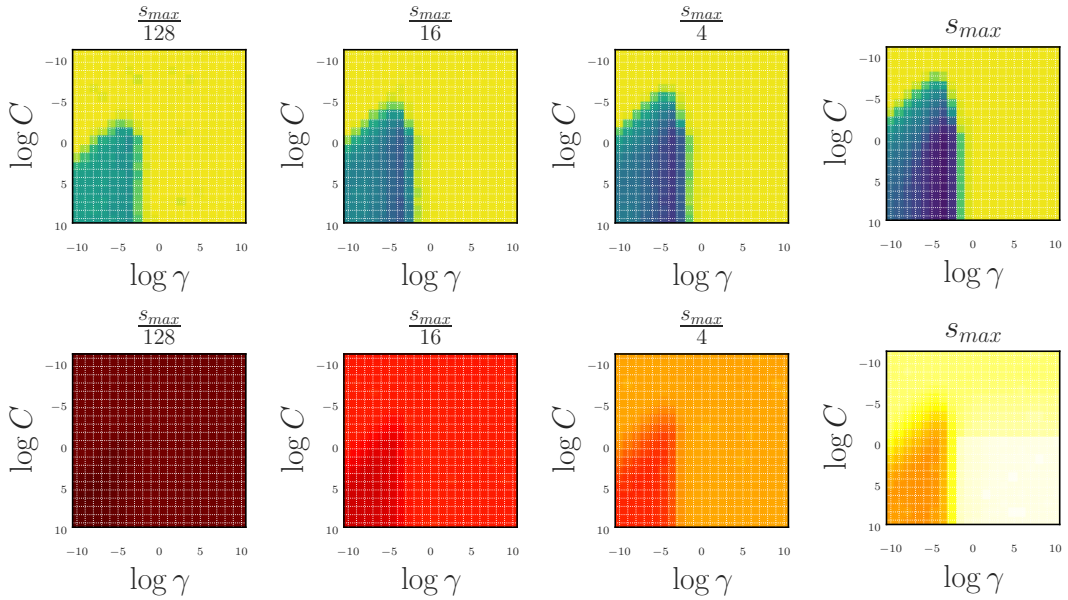


Figure 4.1.: Validation error (top row) and training time (bottom row) of a grid of 400 SVM configurations (20 settings of each of the regularization parameter C and kernel parameter γ, both on a log-scale in $[-10, 10]$) for subsets of the MNIST dataset (LeCun et al., 2001) of various sizes N_{sub}. Small subsets are quite representative: The validation error of bad configuration (yellow) remains constant at around 0.9, whereas the region of good configurations (blue) does not change drastically with s. Both hyperparameters also have an influence on the training time, even though it is less dramatic than the influence of the dataset size.

$t_* \in \mathbb{T}$, $p^{t_*}_{\min}(\boldsymbol{x} \mid \mathcal{D}) := p(\boldsymbol{x} \in \arg\min_{\boldsymbol{x}' \in \mathbb{X}} f(\boldsymbol{x}', t = t_*) \mid \mathcal{D})$, and computing any information w.r.t. it, Swersky et al. (2013) use the information gain per unit cost as their acquisition function[1]:

$$a_{\text{MT}}(\boldsymbol{x}, t) := \frac{1}{c(\boldsymbol{x}, t)} \mathbb{E}_{p(y|\boldsymbol{x},t,\mathcal{D})} \left[\int p^{t_*}_{\min}(\boldsymbol{x}' \mid \mathcal{D}') \cdot \log \frac{p^{t_*}_{\min}(\boldsymbol{x}' \mid \mathcal{D}')}{u(\boldsymbol{x}')} \, \mathrm{d}\boldsymbol{x}' \right], \quad (4.4)$$

where $\mathcal{D}' = \mathcal{D} \cup \{(\boldsymbol{x}, t, y)\}$. The expectation represents the information gain on the target task averaged over the possible outcomes of $f(\boldsymbol{x}, t)$ based on the current model. If the cost $c(\boldsymbol{x}, t)$ of a configuration $\boldsymbol{x}$ on task t is not known a priori it can be modelled the same way as the objective function.

This model supports machine learning hyperparameter optimization for large datasets by using discrete dataset sizes as tasks. Swersky et al. (2013) indeed studied this

[1] In fact, Swersky et al. (2013) deviated slightly from this formula (which follows the ES approach of Hennig and Schuler (2012)) by considering the difference in information gains in $p^{t_*}_{\min}(\boldsymbol{x} \mid \mathcal{D})$ and $p^{t_*}_{\min}(\boldsymbol{x} \mid \mathcal{D} \cup \{(\boldsymbol{x}, y)\})$. They stated this to work better in practice, but we did not find evidence for this in our experiments and thus, for consistency, use the variant presented here throughout.

approach for the special case of $\mathbb{T} = \{0, 1\}$, representing a small and a large dataset; this will be a baseline in our experiments.

4.3.2. Hyperband

Hyperband (Li et al., 2017) is a multi-arm bandit strategy based on random search. It was developed concurrently with our method, and, similar to it, makes uses of the principle that hyperparameter configurations performing poorly on subsets of the data are very likely to also perform poorly on the full datasets.

In each iteration i, Hyperband samples n_i configurations randomly and uses successive halving (Jamieson and Talwalkar, 2016) to discard hyperparameter configurations after evaluating them on subsets of the data. Hyperband iteratively calls successive halving with different tradeoffs between breadth (i.e., number of configurations) and depth (i.e., subset size), such that each iteration takes roughly the same time. Hyperband returns its first suggested hyperparameter setting after its first run of successive halving.

4.4. Fabolas

Here, we introduce our new approach for FAst Bayesian Optimization on LArge data Sets (Fabolas). While traditional Bayesian hyperparameter optimizers model the loss of machine learning algorithms on a given dataset as a blackbox function f to be minimized, Fabolas models loss and computational cost *across dataset size* and uses these models to carry out Bayesian optimization with an extra degree of freedom. The blackbox function $f : \mathbb{X} \times \mathbb{R} \to \mathbb{R}$ now takes another input representing the data subset size; we will use relative sizes $s = N_{sub}/N \in [0, 1]$, with $s = 1$ representing the entire dataset. While the eventual goal is to minimize the loss $f(\boldsymbol{x}, s = 1)$ for the entire dataset, evaluating f for smaller s is usually cheaper, and the function values obtained correlate across s. Unfortunately, this correlation structure is initially unknown, so the challenge is to design a strategy that trades off the cost of function evaluations against the benefit of learning about the scaling behavior of f and, ultimately, about which configurations work best on the full dataset. Following the nomenclature of Williams et al. (2000), we call $s \in [0, 1]$ an *environmental variable* that can be changed freely *during* optimization, but that is set to $s = 1$ (i.e., the entire dataset size), at evaluation time.

We propose a principled rule for the automatic selection of the next $(\boldsymbol{x}, s)$ pair to evaluate. In a nutshell, where standard Bayesian optimization would always run configurations on the full dataset, we use ES to reason about, how much can be learned about performance on the full dataset from an evaluation at any s. In doing so, Fabolas automatically determines the amount of data necessary to (usefully) extrapolate to the full dataset.

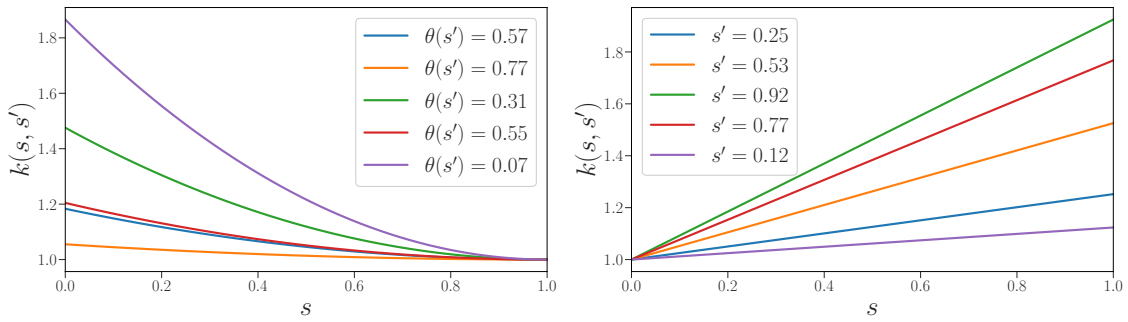


Figure 4.2.: Kernel values across different s with quadratic basis functions to model the objective function (left) and linear basis function to model the cost (right).

4.4.1. Modelling Loss and Computational Cost

To transfer the insights from the illustrative example in Section 4.2 into a formal model for the loss and cost across subset sizes, we extend the GP model by an additional input dimension, namely $s \in [0, 1]$. This allows the surrogate to extrapolate to the full data set at $s = 1$ without necessarily evaluating there. We chose a factorized kernel, consisting of the standard stationary kernel over hyperparameters, multiplied with a finite-rank ("degenerate") covariance function in s:

$$k\left((\boldsymbol{x}, s), (\boldsymbol{x}', s')\right) = k_{5/2}\left(\boldsymbol{x}, \boldsymbol{x}'\right) \cdot \left(\phi^T(s) \cdot \Sigma_\phi \cdot \phi(s')\right) . \tag{4.5}$$

Since any choice of the basis function ϕ yields a positive semi-definite covariance function, this provides a flexible language for prior knowledge relating to s. We use the same form of kernel to model the loss f and cost c, respectively, but with different basis functions ϕ_f and ϕ_c .

The loss of a machine learning algorithms usually decreases with more training data. We incorporate this behavior by choosing $\phi_f(s) = (1, (1-s)^2)^T$ to enforce monotonic predictions with an extremum at $s = 1$. This kernel choice is equivalent to Bayesian linear regression with these basis functions and Gaussian priors on the weights.

To model computational cost c, we note that the complexity usually grows with relative dataset size s. To fit polynomial complexity $\mathcal{O}(s^\alpha)$ for arbitrary α and simultaneously enforce positive predictions, we model the log-cost and use $\phi_c(s) = (1, s)^T$. As above, this amounts to Bayesian linear regression with shown basis functions.

Figure 4.2 shows some examples of our basis functions. Figure 4.3 visualizes the scaling of loss and cost with s for some random SVM configurations from Section 4.2.

4.4.2. Algorithm Description

Fabolas starts with an initial design, described in more detail in Section 4.4.3. Af-

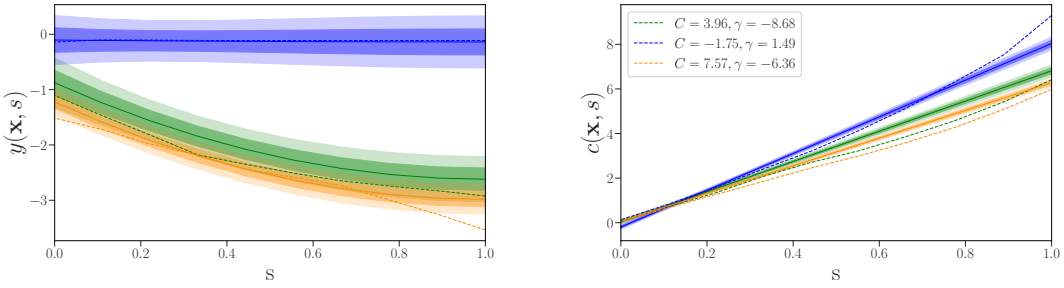


Figure 4.3.: Gaussian process model prediction (solid line) and the actual values (dashed) for the objective function (left) and the cost function (right) based on a dot product of our Bayesian linear regression kernel and a Matern kernel.

terwards, at the beginning of each iteration it fits GPs for loss and computational cost across dataset sizes s using the kernel from Eq. 4.5. Then, capturing the distribution of the optimum for $s = 1$ using $p_{\min}^{s=1}(\boldsymbol{x} \mid \mathcal{D}) := p(\boldsymbol{x} \in \arg\min_{\boldsymbol{x}' \in \mathbb{X}} f(\boldsymbol{x}', s = 1) \mid \mathcal{D})$, it selects the maximizer of the following acquisition function to trade off information gain versus cost:

$$a_{\mathrm{F}}(\boldsymbol{x}, s) := \frac{\mathbb{E}_{p(y|\boldsymbol{x},s,\mathcal{D})}\left[\int p_{\min}^{s=1}(\boldsymbol{x}' \mid \mathcal{D}') \cdot \log \frac{p_{\min}^{s=1}(\boldsymbol{x}'|\mathcal{D}')}{u(\boldsymbol{x}')} \, \mathrm{d}\boldsymbol{x}'\right]}{c(\boldsymbol{x}, s) + c_{\text{overhead}}}, \tag{4.6}$$

where $\mathcal{D}' = \mathcal{D} \cup \{(\boldsymbol{x}, s, y)\}$. Algorithm 5 shows pseudocode for FABOLAS. Additionally, we provide an open-source implementation at: https://github.com/automl/RoBO.

Algorithm 5 Fast BO for Large Datasets (FABOLAS)

1: Initialize data $\mathcal{D}_0$ using an initial design.
2: **for** $t = 1, 2, \ldots$ **do**
3: Fit GP models for $f(\boldsymbol{x}, s)$ and $c(\boldsymbol{x}, s)$ on data $\mathcal{D}_{t-1}$
4: Choose $(\boldsymbol{x}_t, s_t)$ by maximizing the acquisition function in Equation 4.6.
5: Evaluate $y_t \sim f(\boldsymbol{x}_t, s_t) + \mathcal{N}(0, \sigma^2)$, also measuring cost $z_t \sim c(\boldsymbol{x}_t, s_t) + \mathcal{N}(0, \sigma_c^2)$, and augment the data: $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(\boldsymbol{x}_t, s_t, y_t, z_t)\}$
6: Choose incumbent $\hat{\boldsymbol{x}}_t$ based on the predicted loss at $s = 1$ of all $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t\}$.
7: **end for**

Our proposed acquisition function resembles the one used by MTBO (Eq. 4.4), with two differences: First, MTBO's discrete tasks t are replaced by a continuous dataset size s (allowing to learn correlations without evaluations at $s = 1$, and to choose the appropriate subset size automatically). Second, the prediction of computational cost is augmented by the overhead of the Bayesian optimization method. This inclusion of the reasoning overhead is important to appropriately reflect the information gain per unit time spent: it does not matter whether the time is spent with a function evaluation or with reasoning about which evaluation to perform. In practice, due to

cubic scaling in the number of data points of GPs and the computational complexity of approximating $p_{\min}^{s=1}$, the additional overhead of FABOLAS is within the order of minutes, such that differences in computational cost in the order of seconds become negligible in comparison.[2]

Being an anytime algorithm, FABOLAS keeps track of its incumbent at each time step. To select a configuration that performs well on the full dataset, it predicts the loss of all evaluated configurations at $s = 1$ using the GP model and picks the minimizer. We found this to work more robustly than globally minimizing the posterior mean, or similar approaches.

4.4.3. Initial Design

It is common in Bayesian optimization to start with an initial design of points chosen at random or from a Latin hypercube design to allow for reasonable GP models as starting points. To fully leverage the speedups we can obtain from evaluating small datasets, we bias this selection towards points with small (cheap) datasets in order to improve the prediction for dependencies on s: We draw k random points in $\mathbb{X}$ ($k = 10$ in our experiments) and evaluate them on different subsets of the data (for instance on the support vector machine experiments we used $s \in \{1/64, 1/32, 1/16, 1/8\}$). This provides information on scaling behavior, and, assuming that costs increase linearly or superlinearly with s, these k function evaluations cost less than $\frac{k}{4}$ function evaluations on the full dataset. This is important as the cost of the initial design, of course, counts towards FABOLAS' runtime.

4.4.4. Implementation Details

The presentation of FABOLAS above omits some details that impact the performance of our method. As it has become standard in Bayesian optimization (Snoek et al., 2012), we use Markov-Chain Monte Carlo (MCMC) integration to marginalize over the GPs hyperparameters (we use the emcee package (Foreman-Mackey et al., 2013)). To accelerate the optimization, we use hyper-priors to emphasize meaningful values for the parameters, chiefly adopting the choices of the SPEARMINT toolbox (Snoek et al., 2012): a uniform prior between $[-10, 2]$ for all length scales $\boldsymbol{\lambda}$ in log space, a lognormal prior ($\mu_a = 0$, $\sigma_a^2 = 1$) for the covariance amplitude θ, and a horseshoe prior with length scale of 0.1 for the noise variance σ^2.

We used the original formulation of ES by Hennig and Schuler (2012) rather than the recent reformulation of PES by Hernández-Lobato et al. (2014). The main reason

[2] The same is true for standard ES and MTBO, but was never exploited as no emphasis was put on the total wall clock time spent for the hyperparameter optimization. We want to emphasize that we express budgets in terms of wall clock time (not function evaluations) since this is natural in most practical applications.

for this is that the latter prohibits non-stationary kernels due to its use of Bochner's theorem for a spectral approximation. PES could in principle be extended to work for our particular choice of kernels (using an Eigen-expansion, from which we could sample features); since this would complicate making modifications to our kernel, we leave it as an avenue for future work, but note that in any case it may only further improve our method. To maximize the acquisition function we used the blackbox optimizer DIRECT (Jones, 2001).

4.4.5. Heteroscedastic Noise

When making the subset size a parameter, we shuffle the data before an evaluation to prevent bias incurred by repeatedly using the same subset. This shuffling introduces additional noise which could be particularly high for small subsets. To investigate this, we again used the SVM grid of 400 configurations from the Section 4.2. We repeated each run with a given subset size $K = 10$ times using different subsets, and estimate the observation noise variance at each point as:

$$\sigma^2_{obs}(\boldsymbol{x}_j, s_i) = \frac{1}{K}\sum_{k=1}^{K}(y_k(\boldsymbol{x}_j, s_i) - \mu_{i,j})^2\,, \tag{4.7}$$

where $\mu_{i,j} = K^{-1}\sum_{k=1}^{K} y_k(\boldsymbol{x}_j, s_i)$. The red points in Figure 4.4 show the mean and standard deviation of $\sigma^2_{obs}(\boldsymbol{x}_j, s_i)$ over all configurations for all s_i values considered. As expected, the noise decreases with an increasing s, to a point where σ^2_{obs} is zero for $s = 1$.

In contrast to this heteroscedastic noise intrinsic to the random subsampling, the commonly used noise hyperparameter σ^2 of a GP (call it σ^2_{GP}) is fixed and typically estimated using MCMC sampling. To compare these two noise values, for each fixed size s, we also trained a GP to predict losses and plotted its estimates σ^2_{GP} as blue markers in Figure 4.4. To obtain a good estimate of the GP's hyperparameters, we used a relatively long MCMC chain compared to the ones used during Bayesian optimization. Figure 4.4 clearly shows that the estimated variance σ^2_{GP} is always larger than the observation noise σ^2_{obs}. This might indicate a certain misfit between the true objective and the space of functions the GP can model (Sollich, 2001). Consequently, we believe the heteroscedastic noise from subsampling the data to often be negligible compared to the noise estimated by the MCMC sampling.

4.5. Experiments

For our empirical evaluation of Fabolas, we compared it to standard Bayesian optimization (using EI and ES as acquisition functions), MTBO, and Hyperband. For

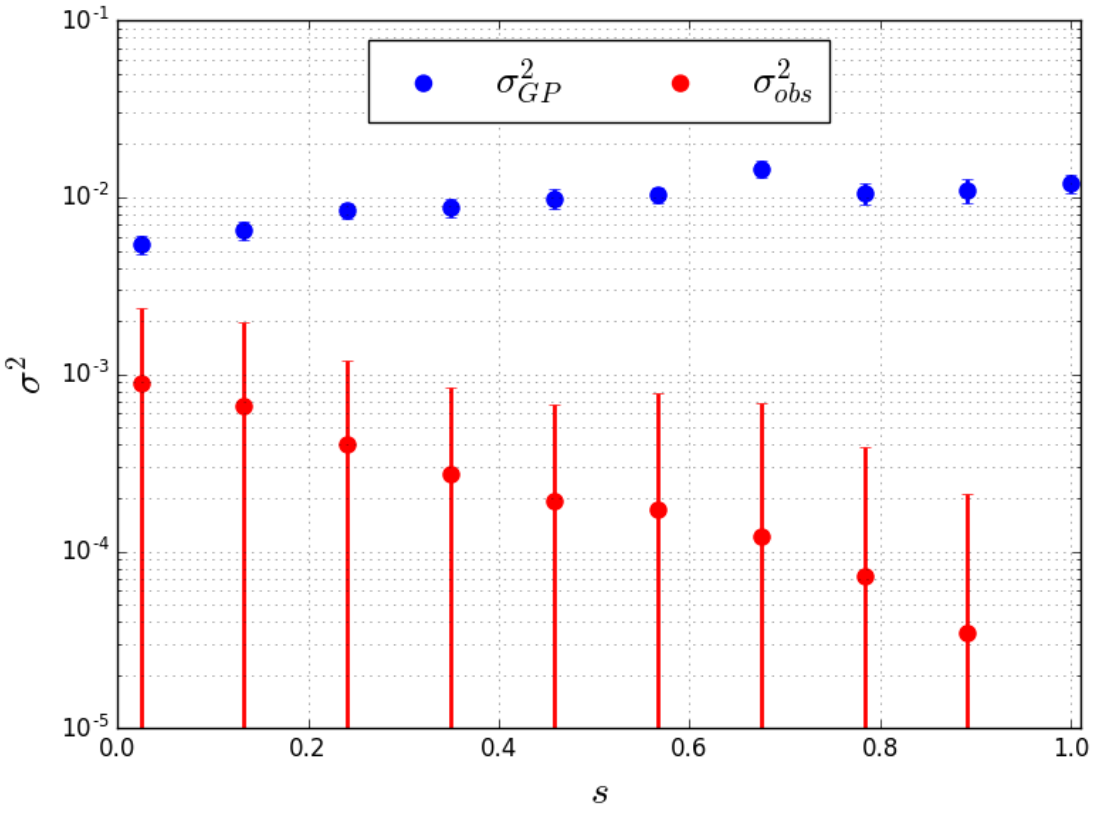


Figure 4.4.: Evaluating a configurations on a shuffled subset of the data induces an additional noise, σ^2_{obs} that depends on the dataset size s. The noise parameter σ^2_{GP} estimated by MCMC sampling for fixed dataset sizes.

each method, we tracked wall clock time (counting both optimization overhead and the cost of function evaluations, including the initial design), storing the incumbent returned after every iteration. In an offline validation step, we then trained models with all incumbents on the full dataset and measured their test error. To obtain error bars, we performed 10 independent runs of each method with different seeds (except on the grid experiment, where we could afford 30 runs per method) and plot mean and standard deviation for all experiments. Each optimization trajectory starts after all of its runs have evaluated at least one configuration.[3]

We implemented Hyperband following Li et al. (2017) using the recommended setting for the parameter $\eta = 3$ that controls the intermediate subset sizes. For each experiment, we adjusted the budget allocated to each Hyperband iteration to allow the same minimum dataset size as for FABOLAS: 100 datapoints for the support vector machine benchmarks and the maximum batch size for the neural network benchmarks. We also followed the prescribed incumbent estimation after

[3]This way we avoid assigning a performance to unfinished runs, but we loose information about the runtime distribution across independent runs.

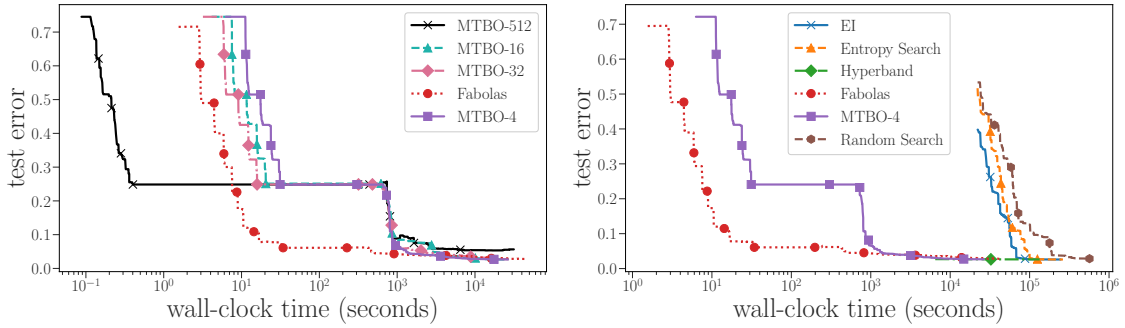


Figure 4.5.: Evaluation on SVM grid on MNIST. (Left) Test performance over time for variants of MTBO with different dataset sizes for the auxiliary task. (Right) Baseline comparison of test performance of the methods' selected incumbents over time. We only plot means to avoid clutter.

each iteration as the configuration with the best performance on the full dataset size.

4.5.1. Support Vector Machine Surrogate

First, we considered a benchmark allowing the comparison of the various Bayesian optimization methods on ground truth: we trained a random forest surrogate (Eggensperger et al., 2015) on our SVM grid on MNIST (described in Section 4.2), for which we had performed all function evaluations beforehand.

We used this benchmark to adjust the number of data points for MTBO's auxiliary task. Figure 4.5 (left) evaluates MTBO variants with a single auxiliary task with a relative size of $1/4$, $1/16$, $1/32$, and $1/512$, respectively. We found that the smaller the auxiliary task, the faster MTBO improved initially, but the slower it converged to the optimum. In the plot, MTBO with an auxiliary task of relative size $s = 1/512$ did not achieve the same performance as the other variants in the end. Given the global structure of the error surface (see Figure 4.1) and the super-linear scaling of the SVM, we chose a very conservative auxiliary task with $s = 1/4$ for the remaining experiments. This value worked consistently in our experience, although the convergence to the best solution in some of the later benchmarks was still rather slow.

At first glance, one might expect many tasks (e.g., with a task for each s value above) to work best, but quite the opposite is true. In preliminary experiments, we evaluated MTBO with up to 3 auxiliary tasks ($s = 1/4$, $1/32$, and $1/512$), but found performance to strongly degrade with a growing number of tasks. We suspect that the $\binom{|T|}{2}$ kernel parameters that have to be learned for the discrete task kernel for $|T|$ tasks are the main reason. If the MCMC sampling is too short, the correlations are not appropriately reflected, especially in early iterations; and an adjusted longer sampling creates a large computational overhead that dominates wall-clock time. We consistently obtained the best performance with only one auxiliary task.

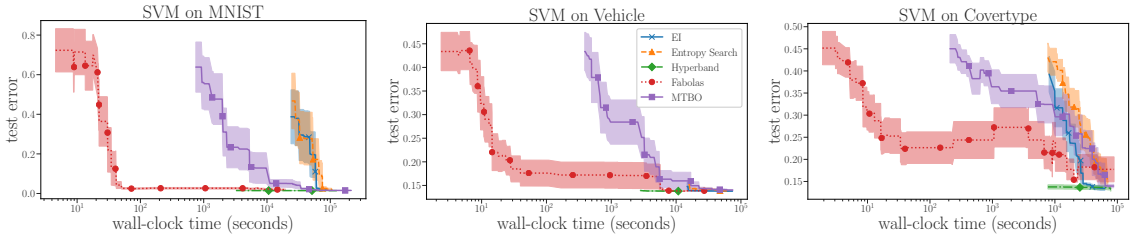


Figure 4.6.: SVM hyperparameter optimization on the datasets MNIST(left), vehicle (middle) and covertype (right). At each time, the plots show test performance of the methods' respective incumbents. FABOLAS can find good configurations between 10 and 1000 times faster than the other methods, but the is not always the fastest to find the true optimum.

We can now proceed to compare the different methods on this benchmark. The right panel of Figure 4.5 shows results using EI, ES, random search, Hyperband, MTBO and FABOLAS. EI and ES performed equally well and found the best configuration (which yields an error of 0.014, or 1.4%) after around 10^5 seconds, roughly three times faster than random search. Hyperband outperformed EI and ES by roughly one order of magnitude. MTBO achieves good performance faster, requiring only around 2×10^3 seconds to find close-to-optimal solutions. FABOLAS was roughly another order of magnitude faster than MTBO in finding good configurations, and found close-to-optimal solutions at the same time.

4.5.2. Support Vector Machines

For a more realistic scenario, we optimized the same SVM hyperparameters (see Table 4.1) without a surrogate on MNIST and two other prominent UCI datasets (gathered from OpenML (Vanschoren et al., 2014)), vehicle registration (Siebert, 1987) and forest cover types (Blackard and Dean, 1999) with more than 50000 data points. Training SVMs on these datasets can take several hours, and Figure 4.6 shows that FABOLAS found good configurations for them between 10 and 1000 times faster than the other methods. On the other hand, both FABOLAS and MTBO sometimes converged more slowly to the true optimum after their initial improvement. This could be a consequence of the GP model and the respective assumptions about the correlation across dataset sizes. Hyperband constitutes a very competitive optimizer on these benchmarks; the super-linear complexity of the SVM and lower cost of good configurations allow Hyperband to recommend its first incumbent faster than the BO methods operating on the full data set.

Table 4.1.: Hyperparameters for all support vector machine tasks.

Hyperparameter	lower bound	upper bound	log
Regularization C	e^{-10}	e^{10}	X
Kernel parameter γ	e^{-10}	e^{10}	X

4.5.3. Convolutional Neural Networks

Convolutional neural networks (CNNs) have shown superior performance on a variety of computer vision and speech recognition benchmarks, but finding good hyperparameter settings remains challenging, and almost no theoretical guarantees exist. Tuning CNNs for modern, large datasets is often infeasible via standard Bayesian optimization; in fact, this motivated the development of Fabolas.

We experimented with hyperparameter optimization for CNNs on two well-established object recognition datasets, namely CIFAR10 (Krizhevsky, 2009) and SVHN (Netzer et al., 2011). We used the same setup for both datasets (a CNN with three convolutional layers, with batch normalization (Ioffe and Szegedy, 2015) in each layer, optimized using Adam (Kingma and Ba, 2015)). We considered a total of five hyperparameters: the initial learning rate, the batch size and the number of units in each layer (see Table 4.2).

Table 4.2.: Hyperparameters for the convolutional neural network task.

Hyperparameter	lower bound	upper bound	log
Initial learning rate	10^{-6}	10^{0}	X
Batch size	32	512	
# units layer 1	2^4	2^8	X
# units layer 2	2^4	2^8	X
# units layer 3	2^4	2^8	X

For CIFAR10, we used 40000 images for training, 10000 to estimate validation error, and the standard 10000 hold-out images to estimate the final test performance of incumbents. For SVHN, we used 6000 of the 73257 training images to estimate validation error, the rest for training, and the standard 26032 images for testing.

The results in Figure 4.7 show that—compared to the SVM tasks—Fabolas' speedup was smaller because CNNs scale linearly in the number of datapoints. Nevertheless, it found good configurations about 10 times faster than vanilla Bayesian optimization. For the same reason of linear scaling, Hyperband was substantially slower than vanilla Bayesian optimization to make a recommendation, but it did find good hyperparameter settings when given enough time.

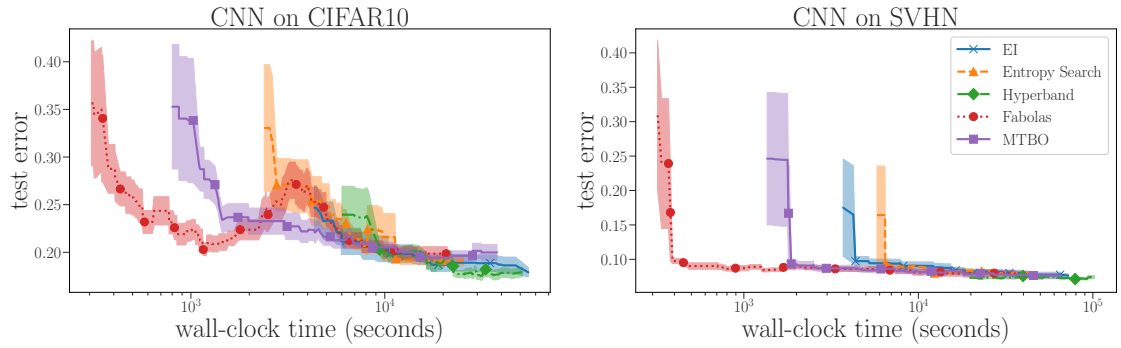


Figure 4.7.: Test performance of a convolutional neural network on CIFAR10 (left) and SVHN (right).

4.6. Chapter Conclusion

We presented Fabolas, a new Bayesian optimization method based on Entropy Search that mimics human experts in evaluating algorithms on subsets of the data to quickly gather information about good hyperparameter settings. Fabolas extends the standard way of modelling the objective function by treating the dataset size as an additional continuous input variable. This allows the incorporation of strong prior information. It models the time it takes to evaluate a configuration and aims to evaluate points that yield—per time spent—the most information about the globally best hyperparameters for the full dataset. In various hyperparameter optimization experiments using support vector machines and deep neural networks, Fabolas often found good configurations 10 to 100 times faster than the related approach of Multi-Task Bayesian optimization, Hyperband and standard Bayesian optimization.

5. Probabilistic Prediction of Learning Curves

As we have seen in the last chapter, one can tremendously speed up the optimization procedure by exploiting cheap-to-evaluate fidelities of the objective function, such as subsets of the training data. In this chapter, we will have a look at another fidelity that many machine learning algorithms exhibit: learning curves, i. e. the loss of iterative machine learning such as neural networks over time or epochs. Compared to using dataset subset as fidelities, learning curves automatically lead to substantially more data points, which makes it challenging for models that struggle with large datasets such as Gaussian process. Therefore, in this chapter we describe a general framework to model learning curves of iterative machine learning methods based on the Bayesian neural networks described in Chapter 3. We first show in Section 5.1 the approach by Domhan et al. (2015) (dubbed LC-Extrapolation) which uses parametric functions to model individual learning curves. Afterwards in Sections 5.2 and 5.3, we discuss a more general joint model across time steps and hyperparameters that can exploit similarities between hyperparameter configurations and predict for unobserved learning curves. Finally, we present an empirical comparison to other methods in Section 5.4.

5.1. Learning Curve Prediction with Basis Functions

An intuitive model for learning curves proposed by Domhan et al. (2015) uses a set of k different parametric functions $\phi_i(\boldsymbol{\theta}_i, t) \in \{\phi_1(\boldsymbol{\theta}_1, t), ..., \phi_k(\boldsymbol{\theta}_k, t)\}$ to extrapolate learning curves $(y_1, \ldots, y_n)$ from the first n time steps. Each parametric function ϕ_i depends on a time step $t \in [1, T]$ and on a parameter vector $\boldsymbol{\theta}_i$. The individual functions are combined into a single model by a weighted linear combination

$$\hat{f}(t|\vec{\Theta}, \vec{w}) = \sum_{i=1}^{k} w_i \phi_i(t, \boldsymbol{\theta}_i)\,, \tag{5.1}$$

where $\vec{\Theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k)$ denotes the combined vector of all parameters $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k$, and $\vec{w} = (w_1, \ldots, w_k)$ is the concatenated vector of the respective weights of each function. Assuming observation noise around the true but unknown value $f(t)$, i.e., assuming $y_t \sim \mathcal{N}(\hat{f}(t|\vec{\Theta}, \vec{w}), \sigma^2)$, Domhan et al. (2015) define a prior for all

parameters $P(\vec{\Theta}, \vec{w}, \sigma^2)$ and use a gradient-free MCMC method (Foreman-Mackey et al., 2013) to obtain S samples, $(\vec{\Theta}_1, \vec{w}_1, \sigma_1^2), \ldots, (\vec{\Theta}_S, \vec{w}_S, \sigma_S^2)$, from the posterior

$$P(\vec{\Theta}, \vec{w}, \sigma^2 \mid y_1 \ldots, y_n) \propto P(y_1, \ldots, y_n \mid \vec{\Theta}, \vec{w}, \sigma^2) P(\vec{\Theta}, \vec{w}, \sigma^2) \tag{5.2}$$

using the likelihood

$$P(y_1, \ldots, y_n \mid \vec{\Theta}, \vec{w}, \sigma^2) = \prod_{t=1}^{n} \mathcal{N}(y_t; \hat{f}(t \mid \vec{\Theta}, \vec{w}), \sigma^2)\,. \tag{5.3}$$

These samples then yield probabilistic extrapolations of the learning curve for future time steps m, with mean and variance predictions

$$\begin{aligned} \hat{y}_m &= \mathbb{E}[y_m | y_1, \ldots y_n] \approx \frac{1}{S} \sum_{s=1}^{S} \hat{f}(m \mid \vec{\Theta}_s, \vec{w}_s)\,, \text{ and} \\ \mathrm{var}(\hat{y}_m) &\approx \frac{1}{S} \sum_{s=1}^{S} (\hat{f}(m \mid \vec{\Theta}_s, \vec{w}_s) - \hat{y}_m)^2 + \sum_{s=1}^{S} \sigma_s^2\,. \end{aligned} \tag{5.4}$$

For our experiments, we use the original implementation by Domhan et al. (2015) with one modification: the original code included a term in the likelihood that enforced the prediction at $t = T$ to be strictly greater than the last value of that particular curve. This biases the estimation to never underestimate the accuracy at the asymptote. We found that in some of our benchmarks, this led to instabilities, especially with very noisy learning curves. Removing it cured that problem, and we did not observe any performance degradation on any of the other benchmarks.

The ability to include arbitrary parametric functions makes this model very flexible, and Domhan et al. (2015) used it successfully to terminate evaluations of poorly-performing hyperparameters early for various different architectures of neural networks (thereby speeding up Bayesian optimization by a factor of two). However, the model's major disadvantage is that it does not use previously evaluated hyperparameters at all and therefore can only make useful predictions after observing a substantial initial fraction of the learning curve.

5.2. Learning Curve Prediction with Bayesian Neural Networks

In practice, similar hyperparameter configurations often lead to similar learning curves, and modelling this dependence would allow predicting learning curves for new configurations without the need to observe their initial performance. Swersky et al. (2014) followed this approach based on an approximate Gaussian process model. Their *Freeze-Thaw* method showed promising results for finding good hyperparameters of iterative machine learning algorithms using learning curve prediction to allocate most

resources for well-performing configurations during the optimization. The method introduces a special covariance function corresponding to exponentially decaying functions to model the learning curves. This results in an analytically tractable model, but using different functions to account for cases where the learning curves do not converge exponentially is not trivial.

Here, we formulate the problem using Bayesian neural networks. We aim to model the validation accuracy $g(\boldsymbol{x}, t)$ of a configuration $\boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^d$ at time step $t \in (0, 1]$ based on noisy observations $y(\boldsymbol{x}, t) \sim \mathcal{N}(g(\boldsymbol{x}, t), \sigma^2)$. For each configuration $\boldsymbol{x}$ trained for $T_{\boldsymbol{x}}$ time steps, we obtain $T_{\boldsymbol{x}}$ data points for our model; denoting the combined data by $\mathcal{D} = \{(\boldsymbol{x}_1, t_1, y_{11})), (\boldsymbol{x}_1, t_2, y_{12}), \ldots, (\boldsymbol{x}_n, T_{x_n}, y_{nT_{x_n}})\}$ we can then write the joint probability of the data $\mathcal{D}$ and the network weights W as

$$P(\mathcal{D}, W) = P(W)P(\sigma^2) \prod_{i=1}^{|D|} \prod_{j=1}^{T_{x_i}} \mathcal{N}(y_{ij}; \hat{g}(\boldsymbol{x}_i, t_j \mid W), \sigma^2)\,. \tag{5.5}$$

where $\hat{g}(\boldsymbol{x}_i, t_i|W)$ is the prediction of a neural network. It is intractable to compute the posterior weight distribution $p(W \mid \mathcal{D})$, but we can use MCMC to sample it, in particular stochastic gradient MCMC methods, such as SGLD (Welling and Teh, 2011) or SGHMC (Chen et al., 2014) (see also Chapter 3). Given M samples $W^1, \ldots, W^M$, we can then obtain the mean and variance of the predictive distribution $p(g|\boldsymbol{x}, t, \mathcal{D})$ as

$$\begin{aligned} \hat{\mu}(\boldsymbol{x}, t \mid \mathcal{D}) &= \frac{1}{M} \sum_{i=1}^{M} \hat{g}(\boldsymbol{x}, t \mid W^i)\,, \text{ and} \\ \widehat{\sigma^2}(\boldsymbol{x}, t \mid \mathcal{D}) &= \frac{1}{M} \sum_{i=1}^{M} \Big(\hat{g}(\boldsymbol{x}, t \mid W^i) - \hat{\mu}(\boldsymbol{x}, t \mid \mathcal{D})\Big)^2 + \sigma_i^2., \end{aligned} \tag{5.6}$$

respectively. Here, σ_i^2 is an additional trainable parameter of the network i that models the observation noise. We will write the above equations shorthand as $\hat{\mu}(\boldsymbol{x}, t)$ and $\widehat{\sigma^2}(\boldsymbol{x}, t)$. This is similar to Eqs. 5.4 and exactly the model that we used for (blackbox) Bayesian optimization with Bayesian neural networks in Chapter 3; the only difference is in the input to the model: here, there is a data point for every time step of the curve, whereas in Chapter 3 we only used a single data point per curve (for its final time step).

5.3. New Basis Function Layer for Learning Curve Prediction

We now combine Bayesian neural networks with parametric functions (see Figure 5.1 for some examples and Section B.2 in the supplement material for a formal description) to incorporate more knowledge about learning curves into the network itself. Instead of obtaining the parameters $\vec{\Theta}$ and $\vec{w}$ by sampling from the posterior, we use a Bayesian neural network to learn several mappings simultaneously:

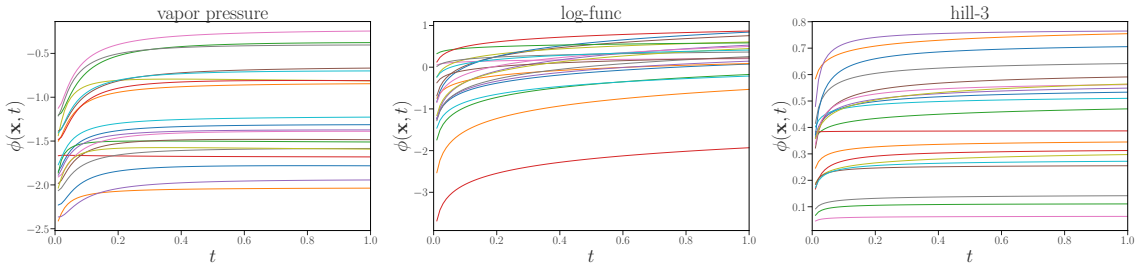


Figure 5.1.: Example functions generated with our $k = 5$ basis functions (formulas are given in Section B.2 in the appendix). For each function, we drew 50 different parameters $\vec{\theta}_i$ uniformly at random in the output domain of the hidden layer(s) of our model. This illustrates the type of functions used to model the learning curves.

1. $\hat{\mu}_\infty$: $\mathcal{X} \to \mathbb{R}$, the asymptotic value of the learning curve
2. $\vec{\Theta}$: $\mathcal{X} \to \mathbb{R}^K$, the parameters of a parametric function model (see Figure 5.1 for some example curves from our basis functions)
3. $\vec{w}$: $\mathcal{X} \to \mathbb{R}^k$, the corresponding weights for each function in the model
4. $\sigma^2 \in \mathbb{R}^+$, the observational noise

With these quantities, we can compute the likelihood in Equation 5.3 which allows training the network.

Phrased differently, we use a neural network to predict the model parameters $\vec{\Theta}$ and weights $\vec{w}$ of our parametric functions, yielding the following form for our network's mean predictions:

$$\hat{g}(\boldsymbol{x}_i, t_i \mid W) = \hat{f}(t_i \mid \vec{\Theta}(\boldsymbol{x}_i, W), \vec{w}(\boldsymbol{x}_i, W)). \tag{5.7}$$

A schematic of this is shown in Figure 5.2. For training, we will use the stochastic gradient Hamiltonian Monte-Carlo with scale adaption as described in Section 3.4.

5.4. Experiments

We now empirically evaluate the predictive performance of Bayesian neural networks, with and without our special learning curve layer. For both networks, we used a 3-layer architecture with tanh activations and 50 units per layer.

As baselines, we compare to other approaches suitable for this task. Besides the aforementioned work by Domhan et al. (2015), we also compare against random forests (Breimann, 2001) with empirical variance estimates (Hutter et al., 2014b), a Gaussian process (GP) using the learning curve kernel from Swersky et al. (2014), and the simple heuristic of using the last seen value (LastSeenValue) of each learning curve for extrapolation. The last model has been successfully used by Li et al. (2017) despite its simplicity.

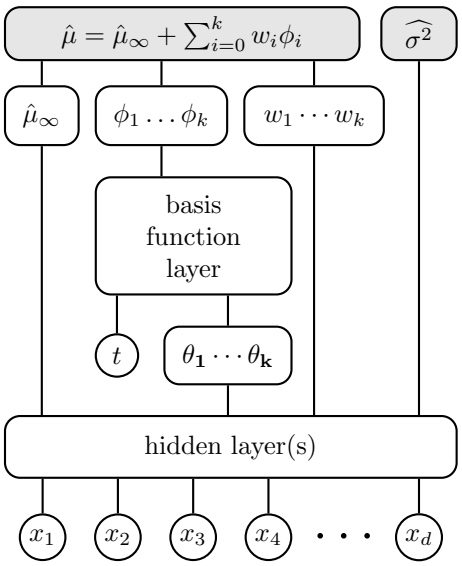


Figure 5.2.: Our neural network architecture to model learning curves. A common hidden layer is used to simultaneously model $\hat{\mu}_\infty$, the parameters $\vec{\Theta}$ of the basis functions, their respective weights $\vec{w}$, and the noise $\widehat{\sigma^2}$.

5.4.1. Datasets

For our empirical evaluation, we generated the following four datasets of learning curves (see Figure 5.3 for some examples), in each case sampling hyperparameter configurations at random from the hyperparameter spaces detailed in Table B.1 in the appendix (see also Section B.3 for some characteristic of these datasets):

- **CNN**: We sampled 256 configurations of 5 different hyperparameters of a 3-layer convolutional neural network (CNN) and trained each of them for 40 epochs on the CIFAR10 (Krizhevsky, 2009) benchmark.
- **FCNet**: We sampled 4096 configurations of 10 hyperparameters of a 2-layer feed forward neural network (FCNet) on MNIST (LeCun et al., 2001), with batch normalization, dropout and ReLU activation functions, annealing the learning rate over time according to a power function. We trained the neural network for 100 epochs.
- **LR**: We sampled 1024 configurations of the 4 hyperparameters of *logistic regression* (LR) and also trained it for 100 epochs on MNIST.
- **VAE**: We sampled 1024 configurations of the 4 hyperparameters of a *variational auto-encoder* (VAE) (Kingma and Welling, 2014). We trained the VAE on MNIST, optimizing the approximation of the lower bound for 300 epochs.

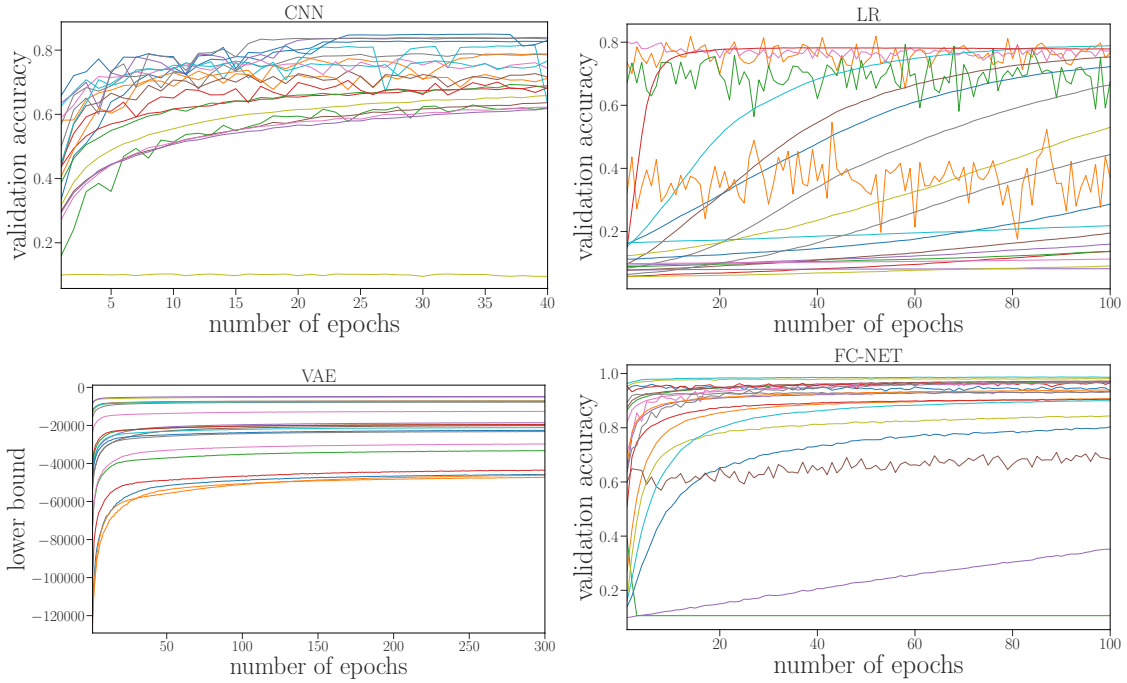


Figure 5.3.: Random learning curves from our 4 datasets. Even though all learning curve follow a similar pattern, based on the underlying model they differ in certain characteristics such as noise or steepness.

5.4.2. Predicting Asymptotic Values of Partially Observed Curves

We first study the problem of predicting the asymptotic values of partially-observed learning curves tackled by Domhan et al. (2015). The LC-Extrapolation method by Domhan et al. (2015), the GP, and the last seen value work on individual learning curves and do not allow to model performance across hyperparameter configurations. Thus, we trained them separately on individual partial learning curves. The other models, including our Bayesian neural networks, on the other hand, can use training data from different hyperparameter configurations. Here, we used training data with the same number of epochs for every partial learning curve.[1]

The left panel of Figure 5.4 visualizes the extrapolation task, showing a learning curve from the CNN dataset and the prediction of the various models trained only using the first 16 of 40 epochs of the learning curve. The right panel shows the corresponding predictive distributions obtained with these models. LastSeenValue does not yield a distribution and uncertainties are not defined.

For a more quantitative evaluation, we used all models to predict the asymptotic

[1]We note that when used inside Bayesian optimization, we would have access to a mix of fully-converged and partially-converged learning curves as training data, and could therefore expect better extrapolation performance.

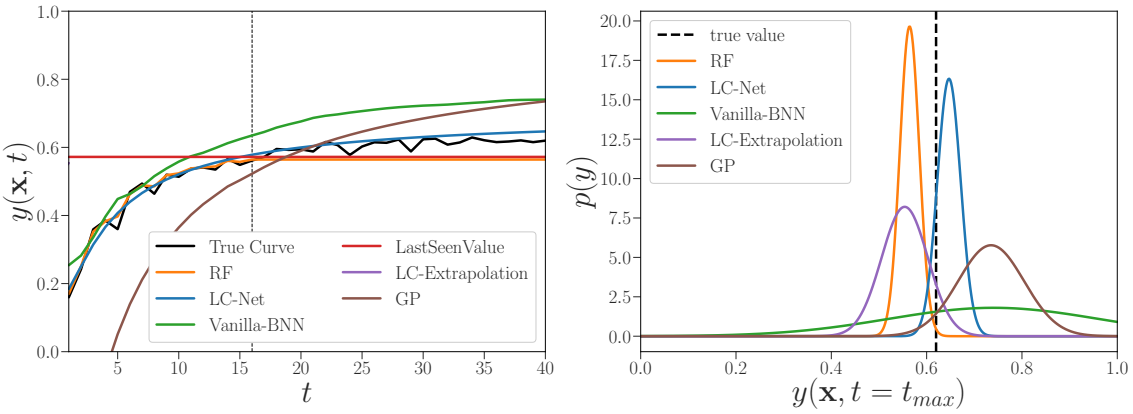


Figure 5.4.: Qualitative comparison of the different models. The left panel shows the mean predictions of different methods on the CNN benchmark. All models observed the validation error of the first 12 epochs of the true learning curve (black). On the right, the posterior distributions over the value at 40 epochs is plotted.

value of all learning curves, evaluating predictions based on observing only the beginning of learning curves. Figure 5.5 shows the mean squared error between the true asymptotic value and the models' predictions (left) and the median log-likelihood of the true value given each model as a function of how much of the learning curves has been observed. Due to the intrinsic randomness of most of the methods, we report for each method the average and standard deviation (vertical bars) over 100 independent runs. We notice several patterns:

1. Throughout, our specialized network architecture performs better than the standard Bayesian neural networks.
2. If no uncertainties are required, LastSeenValue is a competitive baseline. This is because many configurations approach their final performance quite quickly.
3. The GP uncertainty predictions are very competitive, leading to high log-likelihood values but the asymptotic mean predictions are worse than for LC-Net or LastSeenValue especially for short learning curves. We assume that the prior assumption of an exponential function is not flexible enough in practice.
4. Unsurprisingly, the random forest's mean predictions match the quality of LastSeenValue since they do not extrapolate. However, its empirical uncertainty estimates are dramatically worse than LC-Net or GPs
5. LC-Extrapolation poor performance in early stages where only a small amount of datapoints have been observed, can be attributed to a few outliers which cause an high mean square error. LCNet seems to suffer less under these outliers, since it can exploit knowledge from other learning curves.
6. Local models (i. e. LC-Extrapolation and GPs) for single curves start outper-

forming global models for almost complete learning curves in terms of mean square error but struggle if only a few data points are given. Global models, like our BNN approach, on the other hand, are trained on many configurations and need to generalize across these, yielding somewhat worse performance in later stages but obtain better prediction in earlier stages.

5.4.3. Predicting Unobserved Learning Curves

As mentioned before, training a joint model across hyperparameters and time steps allows us to make predictions for completely unobserved learning curves of new configurations. To estimate how well Bayesian neural networks perform in this task, we used the datasets from Section 5.4.1 and split all of them into 16 folds, allowing us to perform cross-validation of the predictive performance. For each fold, we trained all models on the full learning curves in the training set and let them predict the full held-out learning curves.

We compare the two neural networks architectures from Section 5.2 and 5.3 with a random forest baseline. Figure 5.6 shows for each dataset the predicted mean value versus the true values, colored by the log-likelihood based on the model's predictive distribution. Note, to avoid clutter we only plot a random subset of 5000 points out of the prediction for all 16 folds. We make two observations: firstly, both neural network architectures lead to reasonable mean predictions and log-likelihoods. Compared to the previous experiments our additional learning curve layer only helps marginally which is to be expected, since a sufficient amount of data is available and prior information becomes less important. Secondly, the random forest performed consistently worse both in terms of mean prediction as well as log-likelihood, which we assume is due to its limited representational power.

5.4.4. LC-Net with Hyperband

In this section, we show how our model can be used to improve hyperparameter optimization of iterative machine learning algorithms. For this, we extended the multi-armed bandit strategy Hyperband (Li et al., 2017), which in each iteration i first samples N_i hyperparameter configurations $C = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{N_i}\}$ and then uses successive halving (Jamieson and Talwalkar, 2016) to iteratively discard poorly-performing configurations from C. While the original Hyperband method samples configurations C from a uniform distribution over hyperparameter configurations, our extension instead samples them based on our model, with all other parts remaining unchanged. More precisely, we use Thompson sampling (see Section 2.2.3) by first sampling a weight vector $W \sim P(\mathcal{D}, W)$ from our Bayesian neural network and using a stochastic local search to select a hyperparameter configuration $\boldsymbol{x}_\star \in \arg\max_{\boldsymbol{x} \in \mathbb{X}} \hat{g}(\boldsymbol{x}, t = 1 \mid W)$. Note, Hyperband evaluates hyperparameter configurations on different budges t, but since we are interested in finding the best

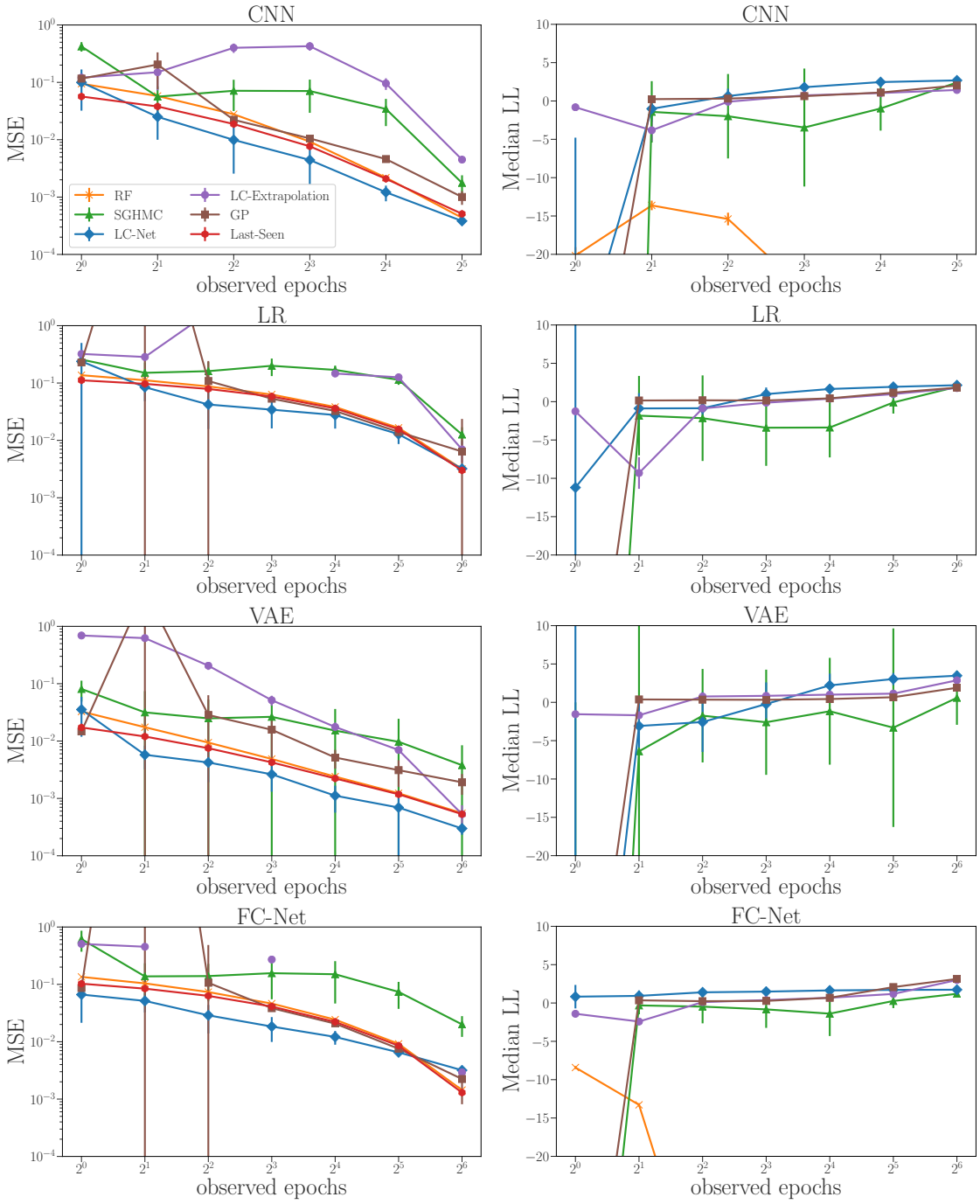


Figure 5.5.: Assessment of the predictive quality based on partially observed learning curves on all 4 benchmarks. The panels on the left show the mean squared error of the predicted asymptotic value (y-axis) after observing all learning curves up to a given fraction of maximum number of epochs (x-axis). The panels on the right show the median log-likelihood based on the predictive mean and variance of the asymptotic value. Note that LastSeenValue does not provide a predictive variance.

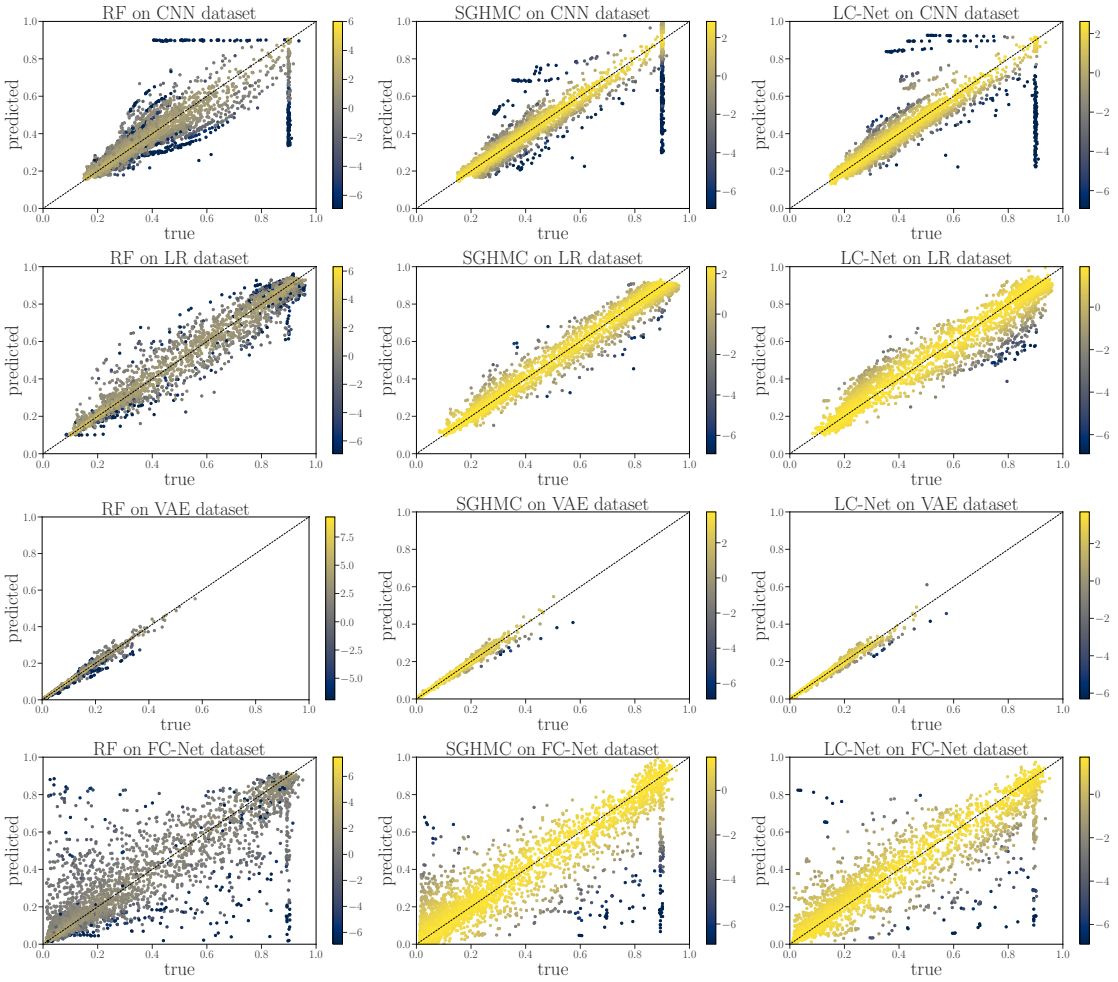


Figure 5.6.: On the horizontal axis, we plot the true value and on the vertical axis the predicted mean value. Each point is colored by its log-likelihood (the brighter the higher). To reduce clutter we plot only a subset of 5000 randomly chosen points out of all predictions from the 16 fold cross-validation.

configuration on the full budget $t = 1$, we perform Thompson sampling only on the full budget. It has been shown (Hernández-Lobato et al., 2017) that each weight sample from the Bayesian neural network can be interpreted as a function sample $f \sim p(f \mid \mathcal{D})$ and that it can be trivially adapted to the parallel setting by simply using a different weight sample every time a new hyperparameter configuration is queried.

For a thorough empirical evaluation and to reduce computational requirements we used the tabular benchmark from Chapter 7. After each iteration we report the final performance of the best observed configuration so far, along with the wall clock time that would have been needed for optimizing the true objective function.

Figure 5.7 shows the immediate regret on the HPO-Bench-Protein and HPO-Bench-

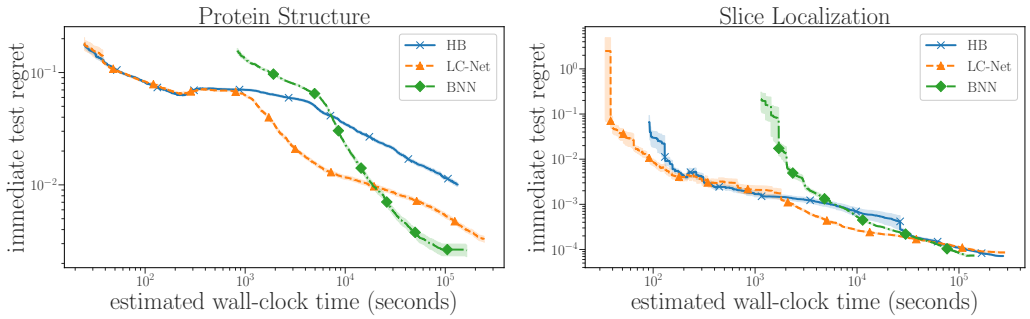


Figure 5.7.: Comparison of Hyperband, Hyperband with our model, and standard Bayesian optimization with Bayesian neural networks to model the objective function on the HPO-Bench benchmarks described in Chapter 7. Hyperband finds a good configuration faster than standard Bayesian optimization, but it approaches the global optimum quicker when extended with our model.

Slice benchmark as a function of wall-clock time, for three optimizers: Hyperband, our model-based extension of Hyperband, as well as standard Bayesian optimization with Bayesian neural networks (see Chapter 3). For the other benchmarks see Section B.4 in the supplemental material. Standard Bayesian optimization does not make use of learning curves and thus needs to evaluate each configuration for the full amount of epochs. In accordance with results by Li et al. (2017), in this experiment Hyperband found a configuration with good performance faster than standard Bayesian optimization, but its random sampling did not suffice to quickly approach the best configuration; given enough time Bayesian optimization performed better. However, extended by our model, Hyperband approaches the global optimum much faster.

5.5. Chapter Conclusion

We studied Bayesian neural networks for modelling the learning curves of iterative machine learning methods, such as stochastic gradient descent for convolutional neural networks. Based on the parametric learning curve models of Domhan et al. (2015), we also developed a specialized neural network architecture with a learning curve layer that improves learning curve predictions. In future work, we aim to study recurrent neural networks for predicting learning curves and will extend Bayesian optimization methods with Bayesian neural networks based on our learning curve models.

6. Combining Bayesian Optimization with Hyperband

We have seen in the previous chapters that, while the objective function $f : \mathcal{X} \to \mathbb{R}$ is typically expensive to evaluate (since it requires training a machine learning model with the specified hyperparameters), in most applications it is possible to define cheap-to-evaluate approximate versions $\tilde{f}(\cdot, b)$, called a fidelity of $f(\cdot)$ that are parameterized by a so-called *budget* $b \in [b_{min}, b_{max}]$. With the maximum budget $b = b_{max}$, we have $\tilde{f}(\cdot, b_{max}) = f(\cdot)$, whereas with $b < b_{max}$, $\tilde{f}(\cdot, b)$ is only an approximation of $f(\cdot)$ whose quality typically increases with b. Common examples for different fidelities are the number of iterations for an iterative algorithm (see Chapter 5), the number of data points used (see Chapter 4), the number of steps in an MCMC chain, and the number of trials in deep reinforcement learning.

Hyperband (Li et al., 2017), even though often outperformed by multi-fidelity Bayesian optimization methods (see Chapter 4) showed still a robust performance and more importantly, it makes less parametric assumptions about the relation between fidelities than its model-based counterparts. However, one major disadvantage of Hyperband it that randomly samples hyperparameter configuration and, hence, it often takes much longer to approach the global optimum than methods that maintain a model of the objective function. In this section we investigate a principal way to combine Bayesian optimization with Hyperband to get the best of two worlds: efficient anytime performance and superior final performance.

We first describe the Bayesian optimization method TPE (Bergstra et al., 2011) and Hyperband in more detail (Section 6.1 and 6.2) and then show how to combine them in our new method BOHB (short for Bayesian optimization and Hyperband), as well as how to effectively parallelize the resulting system (Section 6). Our extensive empirical evaluation (Section 6.4) demonstrates that our method combines the best aspects of Bayesian optimization and Hyperband: it often finds good solutions over an order of magnitude faster than Bayesian optimization and converges to the best solutions orders of magnitudes faster than Hyperband.

In this work, we focus only on combining Hyperband and Bayesian optimization, but we would like to mention that methods improving BO are potentially applicable to BOHB as well, such as meta learning (Swersky et al., 2013; Feurer et al., 2015b; Poloczek et al., 2016; Springenberg et al., 2016), active ensembling to combine models found during the optimization (Lévesque et al., 2016), and using multiple fidelities (Swersky et al., 2013; Kandasamy et al., 2017). The data gathered by BOHB on

different budgets could also be used to quantify the importance of hyperparameters (Hutter et al., 2014a; Biedenkapp et al., 2017; Golovin et al., 2017; van Rijn and Hutter, 2018). We leave these for future work.

6.1. Hyperband

Hyperband (HB) (Li et al., 2017) is a multi-armed bandit strategy for hyperparameter optimization that takes advantage of these different budgets b by repeatedly calling SuccessiveHalving (SH) (Jamieson and Talwalkar, 2016) to identify the best out of n randomly sampled configurations. It balances very aggressive evaluations with many configurations on the smallest budget, and very conservative runs that are directly evaluated on b_{max}. The exact procedure for this trade-off is shown in Algorithm 6 (with pseudocode for SH shown in Section C.2 in the appendix). Line 1 computes the geometrically spaced budget $\in [b_{min}, b_{max}]$. The number of configurations sampled in line 3 is chosen such that every SH run requires the same total budget. SH internally evaluates configurations on a given budget, ranks them by their performance, and continues the top η^{-1} (usually the best-performing third) on a budget η times larger. This is repeated until the maximum budget is reached. In practice, HB works very well and typically outperforms random search and Bayesian optimization methods operating on the full function evaluation budget quite easily for small to medium total budgets. However, its convergence to the global optimum is limited by its reliance on randomly-drawn configurations, and with large budgets its advantage over random search typically diminishes.

6.2. Tree Parzen Estimator

The Tree Parzen Estimator (TPE) (Bergstra et al., 2011) is a Bayesian optimization method that uses a kernel density estimator to model the densities

$$\begin{aligned} l(\boldsymbol{x}) &= p(y < \alpha | \boldsymbol{x}, D) \\ g(\boldsymbol{x}) &= p(y > \alpha | \boldsymbol{x}, D) \end{aligned} \tag{6.1}$$

over the input configuration space instead of modeling the objective function f directly by $p(f|D)$ (see Chapter 2.2). To select a new candidate $\boldsymbol{x}_{new}$ to evaluate, it maximizes the ratio $l(\boldsymbol{x})/g(\boldsymbol{x})$; Bergstra et al. (2011) showed that this is equivalent to maximizing EI in Equation 2.8. Due to the nature of kernel density estimators, TPE easily supports mixed continuous and discrete spaces, and model construction scales linearly in the number of data points (in contrast to the cubic-time Gaussian processes (GPs) predominant in the BO literature(see Chapter 2.2)).

Algorithm 6 Pseudocode for Hyperband using SuccessiveHalving (SH) as a subroutine.

Require: budgets b_{min} and b_{max}, η
1: $s_{max} = \lfloor \log_\eta \frac{b_{max}}{b_{min}} \rfloor$
2: **for** $s \in \{s_{max}, s_{max} - 1, \ldots, 0\}$ **do**
3: sample $n = \lceil \frac{s_{max}+1}{s+1} \cdot \eta^s \rceil$ configurations
4: run SH on them with $\eta^{-s} \cdot b_{max}$ as initial budget
5: **end for**

6.3. Model-Based Hyperband

We now introduce our new practical HPO method, which we dub *BOHB* since it combines Bayesian optimization (BO) and Hyperband (HB).

6.3.1. Algorithm Description

BOHB relies on HB to determine how many configurations to evaluate with which budget, but it replaces the random selection of configurations at the beginning of each HB iteration by a model-based search. Once the desired number of configurations for the iteration is reached, the standard successive halving procedure is carried out using these configurations. We keep track of the performance of all function evaluations $g(\boldsymbol{x}, b) + \epsilon$ of configurations $\boldsymbol{x}$ on all budgets b to use as a basis for our models in later iterations.

We follow HB's way of choosing the budgets and continue to use SH, but we replace the random sampling by a BO component to guide the search. We construct a model and use BO to select a new configuration, based on the configurations evaluated so far. In the remainder of this section, we will explain this procedure summarized by the pseudocode in Algorithm 7.

The BO part of BOHB closely resembles TPE, with one major difference: we opted for a single multidimensional KDE compared to the hierarchy of one-dimensional KDEs used in TPE in order to better handle interaction effects in the input space. To fit useful KDEs (in line 4 of Algorithm 7), we require a minimum number of data points N_{min}; this is set to $d + 1$ for our experiments, where d is the number of hyperparameters. To build a model as early as possible, we do not wait until $N_b = |D_b|$, the number of observations for budget b, is large enough to satisfy $q \cdot N_b \geq N_{min}$. Instead, after initializing with $N_{min} + 2$ random configurations (line 3), we choose the

$$\begin{aligned} N_{b,l} &= \max(N_{min}, q \cdot N_b) \\ N_{b,g} &= \max(N_{min}, N_b - N_{b,l}) \end{aligned} \tag{6.2}$$

best and worst configurations, respectively, to model the two densities. This ensures that both models have enough datapoints and have the least overlap when only a

limited number of observations is available. We used the KDE implementation from statsmodels (Seabold and Perktold, 2010), estimating the KDE's bandwidth with the default estimation procedure (Scott's rule of thumb), which is efficient and performed well in our experience. Details on our KDE are given in Section C.3 in the appendix.

As the optimization progresses, more configurations are evaluated on bigger budgets. Given that the goal is to optimize on the largest budget, BOHB always uses the model for the largest budget for which enough observations are available (line 2). This enables it to overcome wrong conclusions drawn on smaller budgets by eventually relying on results with the highest fidelity only.

To optimize EI (lines 5-6), we sample N_s points from $l'(\boldsymbol{x})$, which is the same KDE as $l(\boldsymbol{x})$ but with all bandwidths multiplied by a factor b_w to encourage more exploration around the promising configurations. We observed that this improves convergence especially in the late stages of the optimization, when the model on the biggest budget is queried frequently but updated rarely.

In order to keep the theoretical guarantees of HB, we also sample a constant fraction ρ of the configurations uniformly at random (line 1). Besides global exploration, this guarantees that after $m \cdot (s_{max} + 1)$ SH runs, our method has (on average) evaluated $\rho \cdot m \cdot (s_{max} + 1)$ random configurations on b_{max}. As every SH run consumes a budget of at most $(s_{max} + 1) \cdot b_{max}$, in the same time random search evaluates $(\rho^{-1} \cdot (s_{max} + 1))$-times as many configuration on the largest budget. This means, that in the worst case (when the lower fidelities are misleading), BOHB is at most this factor times slower than RS, but it is still guaranteed to converge eventually. The same argument holds for HB, but in practice both HB and BOHB substantially outperform RS in our experiments.

No optimizer is free of hyperparameters itself, and their effects have to be studied carefully. We therefore include a detailed empirical analysis of BOHB's hyperparameters in Section C.6 in the appendix that shows each hyperparameter's effect when

Algorithm 7 Pseudocode for sampling in BOHB

Require: observations D, fraction of random runs ρ, percentile q, number of samples N_s, minimum number of points N_{min} to build a model, and bandwidth factor b_w

1: **if** rand() $< \rho$ **then**
2: **return** random configuration
3: **end if**
4: $b = \arg\max \{D_b : |D_b| \geq N_{min} + 2\}$
5: **if** $b = \emptyset$ **then**
6: **return** random configuration
7: **end if**
8: fit KDEs according to Eqs. 6.1 and 6.2
9: draw N_s samples according to $l'(\boldsymbol{x})$ (see text) **return** sample with highest ratio $l(\boldsymbol{x})/g(\boldsymbol{x})$

all others are fixed to their default values (these are also listed there). We find that BOHB is quite insensitive to its hyperparameters, with the default working robustly across different scenarios.

6.3.2. Parallelization

Modern optimizers must be able to take advantage of parallel resources effectively and efficiently. BOHB achieves that by inheriting properties from both TPE and HB. The parallelism in TPE is achieved by limiting the number of samples to optimize EI, purposefully not optimizing it fully to obtain diversity. This ensures that consecutive suggestions by the model are diverse enough to yield near-linear speedups when evaluated in parallel. On the other hand, HB can be parallelized by (a) starting different iterations at the same time (a parallel for loop in Alg. 6), and (b) evaluating configurations concurrently within each SH run.

Our parallelization strategy of BOHB is as follows. We start with the first SH run that sequential HB would perform (the most aggressive one, starting from the lowest budget), sampling configurations with the strategy outlined in Algorithm 7 until either (a) all workers are busy, or (b) enough configurations have been sampled for this SH run. In case (a), we simply wait for a worker to free up and then sample a new configuration. In case (b), we start the next SH run in parallel, sampling the configurations to run for it also according to Algorithm 7; observations D (and therefore the resulting models) are shared across all SH runs. BOHB is an anytime algorithm that at each point in time keeps track of the configuration that achieved the best validation performance; it can also be given a maximum budget of SH runs.

We note that SH has also been parallelized in independent work by Li et al. (2018). Next to parallelizing SH runs (by filling the next free worker with the ready-to-be-executed run with the largest budget), that work mentioned that HB can trivially be parallelized by running its SH runs in parallel. In contrast to this approach of parallelizing HB by having separate pools of workers for each SH run, we rather join all workers into a single pool, and whenever a worker becomes available preferentially execute waiting runs with smaller budgets. New SH runs are only started when the SH runs currently executed are not waiting for a worker to free up. This strategy (a) allows us to achieve better speedups by using all workers in the most aggressive (and often most effective) bracket first, and (b) also takes full advantage of models built on smaller budgets. Figure 6.1 demonstrates that our method of parallelization can effectively exploit many parallel workers.

6.4. Experiments

We now comprehensively evaluate BOHB's empirical performance in a wide range of tasks, including a high-dimensional toy function, as well as optimizing the hyperpa-

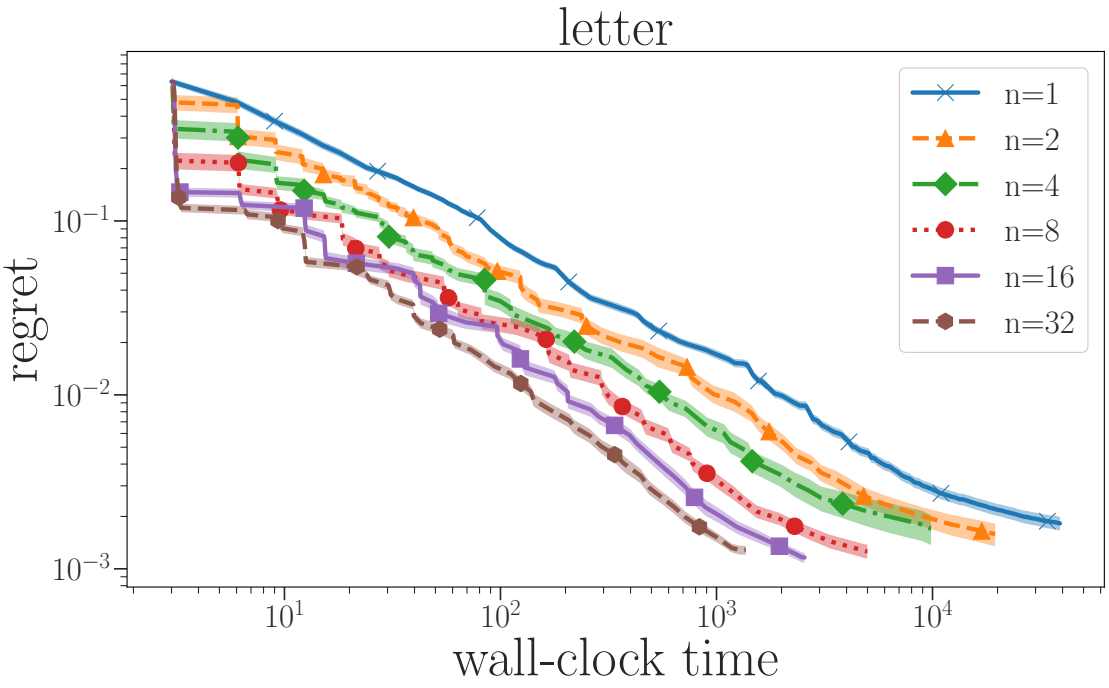


Figure 6.1.: Performance of our method with different number of parallel workers on the letter surrogate benchmark (see Sec. 6.4) for 128 iterations. The speedup for two and four workers is close to linear, for more workers it becomes sublinear. For example, the speedup to achieve a regret of 10^{-2} for one vs. 32 workers is ca. $2000s/130s \approx 15$. We plot the mean and twice the standard error of the mean over 128 runs.

rameters of support vector machines, feed-forward neural networks, Bayesian neural networks, deep reinforcement learning agents and convolutional neural networks. Code for BOHB and our benchmarks is publicly available at https://github.com/automl/HpBandSter

To compare against TPE, we used the Hyperopt package (Bergstra et al., 2011), and for all GP-BO methods we used the RoBO python package (Klein et al., 2017b). In all experiments we set $\eta = 3$ for HB and BOHB as recommended by Li et al. (2017). If not stated otherwise, for all methods we report the mean performance and the standard error of the mean of the best observed configuration so far (incumbent) at a given budget.

6.4.1. Artificial Toy Function: Stochastic Counting Ones

In this experiment we investigated BOHB's behavior in high-dimensional mixed continuous / categorical configuration spaces. Since GP-BO methods tend to not work well on such configuration spaces (Eggensperger et al., 2013) we do not include them in this experiment. However, we do use SMAC (Hutter et al., 2011), since its random

forest is known to perform well in high-dimensional categorical spaces (Eggensperger et al., 2013).

Given a set of N_{cat} categorical variables $x_i \in \{0, 1\}$ and N_{cont} continuous variables $x_j \in [0, 1]$, we defined a variant of the counting ones problem as follows: We sum the values of all categoricals x_i and add the sample means of Bernoulli distributions with parameters x_j given by the continuous variables. The number of samples used to estimate the mean represents the budget. Figure 6.2 shows the result of 512 independent runs of various optimizers in a 16-dimensional space with $N_{cat} = N_{cont} = 8$ parameters. We plot the normalized immediate regret of the noise free function, i.e. $|f(\boldsymbol{x}_{inc}) - d|/d$ where $d = N_{cat} + N_{cont}$ and $\boldsymbol{x}_{inc}$ is the incumbent at a specific time step.

Random search worked very poorly on this benchmark and was quickly dominated by SMAC and TPE. Even though HB worked better in the beginning, SMAC and TPE clearly outperformed it after having obtained a sufficiently informative model. BOHB worked as well as HB in the beginning and then quickly started to perform better.

We obtained similar results for other dimensionalities (see Figures 8 and 9 in the supplementary material), but the picture is not always as clear. In higher dimensions, SMAC seems to outperform TPE, hinting at the limitations of TPE's KDE compared to SMAC's random forest. As BOHB still evaluates configurations on small budgets even in late stages of the optimization, convergence can be slowed down compared to SMAC and TPE. A formal description of the problem, the budgets, and a more detailed discussion of the results can be found in Section C.7 in the appendix.

6.4.2. Comprehensive Experiments on Surrogate Benchmarks

For the next experiments we constructed a set of surrogate benchmarks based on offline data following Eggensperger et al. (2015). Optimizing a surrogate instead of the real objective function is substantially cheaper, which allows us to afford many independent runs for each optimizer and to draw statistically more meaningful conclusions. A more detailed discussion of how we generated these surrogates can be found in Section C.8 in the supplementary material. To better compare the convergence towards the true optimum, we again computed the immediate regret of the incumbent.

6.4.2.1. Support Vector Machine on MNIST

To compare against GP-BO, we used the support vector machine on MNIST surrogate from Klein et al. (2017a) (described in Section 4.5.1). This surrogate imitates the hyperparameter optimization of a support vector machine with a RBF kernel with two hyperparameters: the regularization parameter C and the kernel parameter

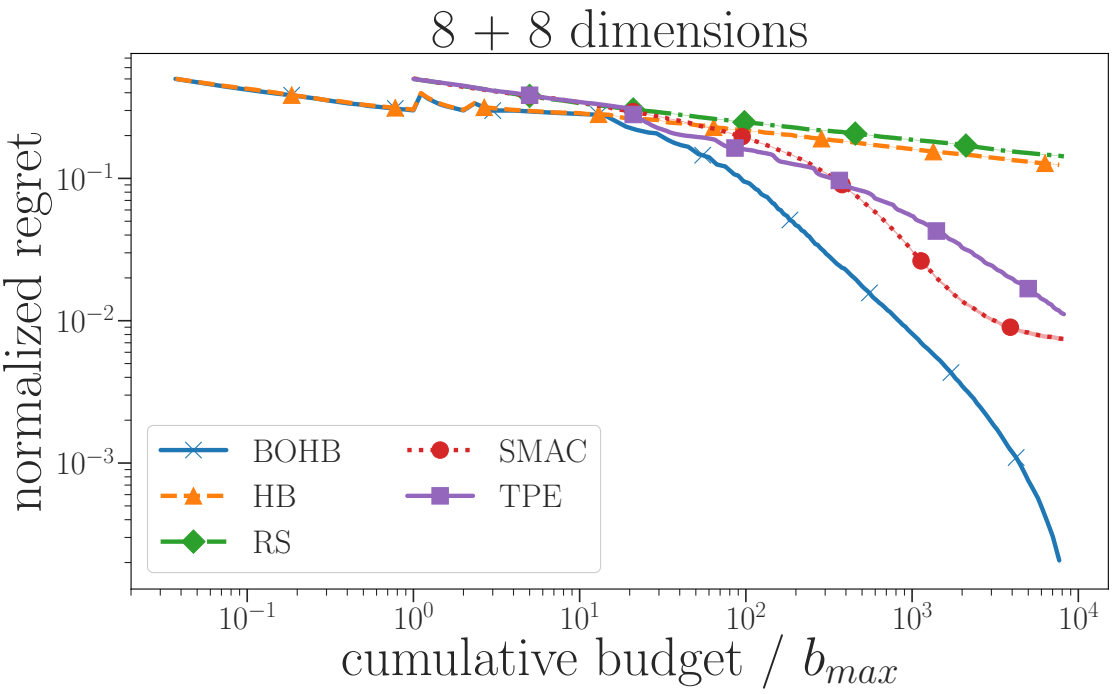


Figure 6.2.: Results for the counting ones problem in 16 dimensional space with 8 categorical and 8 continuous hyperparameters. In higher dimensional spaces RS-based methods need exponentially more samples to find good solutions.

γ. The budget is given by the number of training datapoints, where the minimum budget is $1/512$ of the training data and the maximum budget is the full training data. For further details, see Section 4.5.1.

Figure 6.3 compares BOHB to various BO methods, such as Fabolas (Klein et al., 2017a), multi-task Bayesian optimization (MTBO) (Swersky et al., 2013), GP-BO with expected improvement (Snoek et al., 2012; Klein et al., 2017b), RS and HB. We follow the evaluation protocol of Klein et al. (2017a) and plot the performance of each configuration when retrained using the full dataset. Both BOHB and HB identified the best configuration within their first iterations, making them competitive to Fabolas and MTBO. We note that this is despite the fact that GP-BO methods usually work particularly well on such low-dimensional continuous problems (Eggensperger et al., 2013).

6.4.2.2. Feed-forward Neural Networks on OpenML Datasets

We optimized six hyperparameters that control the training procedure (initial learning rate, batch size, dropout, exponential decay factor for learning rate) and the architecture (number of layers, units per layer) of a feed forward neural network for six different datasets gathered from OpenML (Vanschoren et al., 2014): Adult (Kohavi, 1996), Higgs (Baldi et al., 2014), Letter (Frey and Slate, 1991),

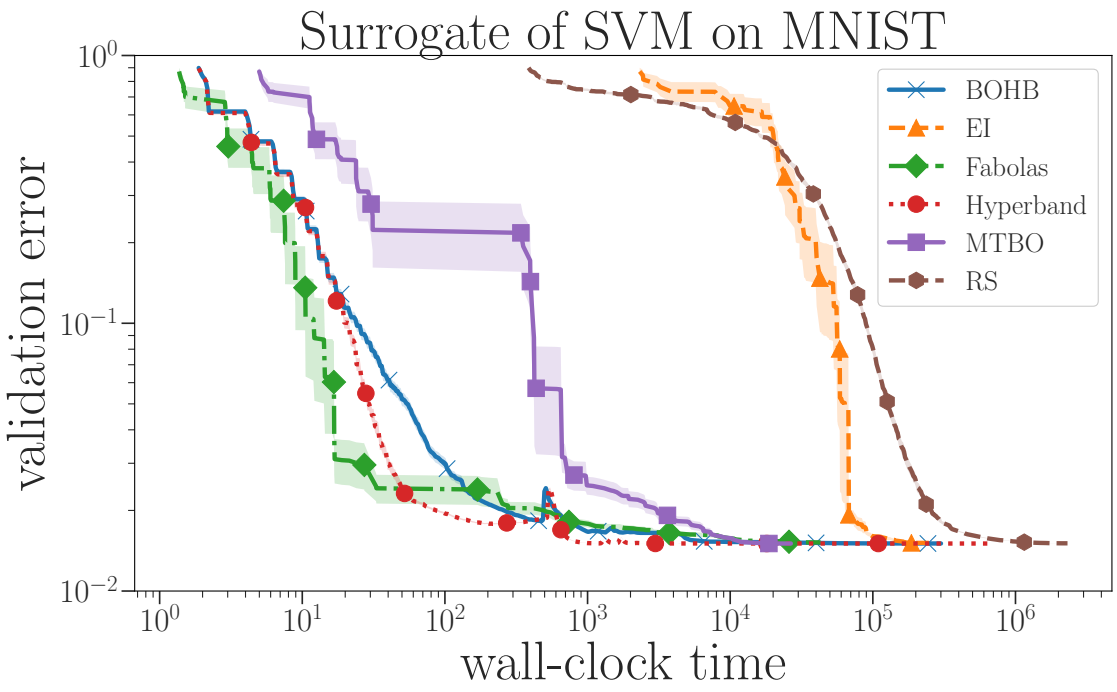


Figure 6.3.: Comparison on the SVM on MNIST surrogates as described in Klein et al. (2017a). BOHB and HB work comparably to Fabolas on this benchmark outperforming MTBO and GP-BO.

MNIST (LeCun et al., 2001), Optdigits (Lichman, 2013), and Poker (Cattral et al., 2002). A detailed description of all hyperparameter ranges and training budgets can be found in Section C.8 in the appendix.

We ran random search (RS), TPE, HB, GP-BO, Hyperband with LC-Net (HB-LCNet, see Klein et al. (2017c) and Section 5) and BOHB on all six datasets and summarize the results for one of them in Figure 6.4. Figures for the other datasets are shown in Appendix E.

We note that HB initially performed much better than the vanilla BO methods and achieved a roughly three-fold speedup over RS. However, for large enough budgets TPE and GP-BO caught up in all cases, and in the end found better configurations than HB and RS. HB and BOHB started out identically, but BOHB achieved the same final performance as HB 100 times faster, while at the same time yielding a final result that was better than that of the other BO methods. All model-based methods substantially outperformed RS at the end of their budget, whereas HB approached the same performance. Interestingly, the speedups that TPE and GP-BO achieved over RS are comparable to the speedups that BOHB achieved over HB. Finally, HB-LCNet performed somewhat better than HB alone, but consistently worse than BOHB, even when tuning HB-LCNet. We only compare to HB-LCNet on this benchmark, since it is the only one that includes full learning curves (for which the parametric functions in HB-LCNet were designed). Also, HB-LCNet requires

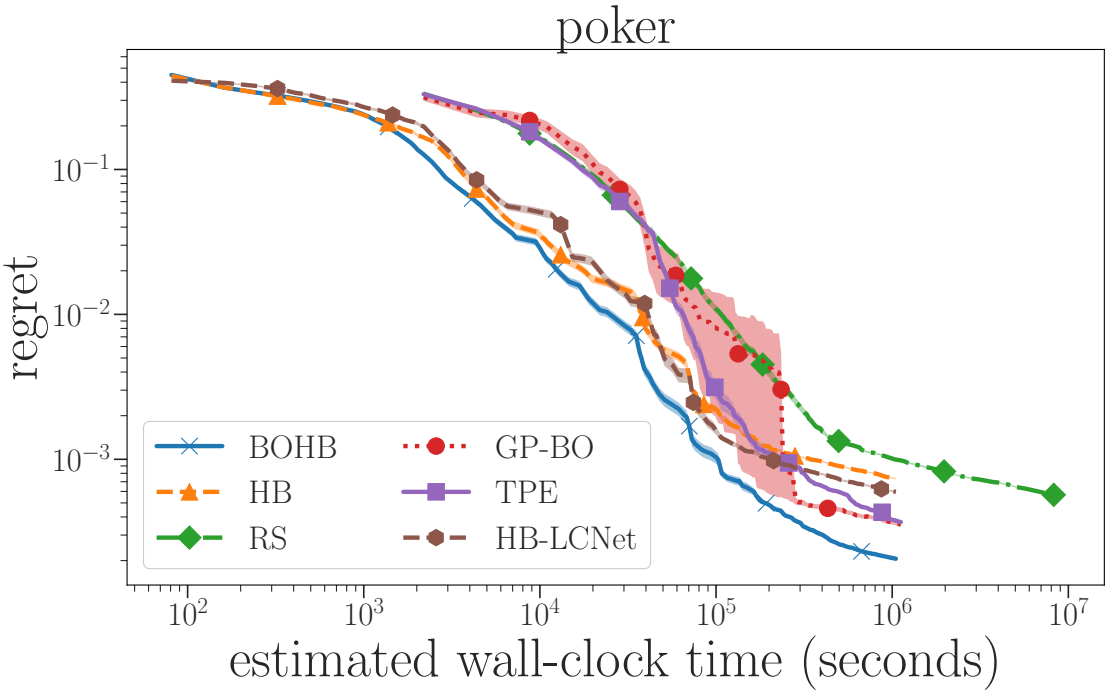


Figure 6.4.: Optimizing six hyperparameter of a feed-forward neural network on featurized datasets; results are based on surrogate benchmarks. Results for the other 5 datasets are qualitatively similar and are shown in Figure C.2 in the supplementary material.

access to performance values for all budgets, which we do not obtain when, e.g., using data subset sizes as a budget, and we thus expect HB-LCNet to perform poorly in the other cases.

6.4.3. Bayesian Neural Networks

For this experiment we optimized the hyperparameters and the architecture of a two-layer fully connected Bayesian neural network trained with Markov Chain Monte-Carlo (MCMC) sampling. We used stochastic gradient Hamiltonian Monte-Carlo sampling (SGHMC) (Chen et al., 2014) with scale adaption (Springenberg et al., 2016) as described in Chapter 3 to sample the parameter vector of the network. Note that to the best of our knowledge, this is the first application of hyperparameter optimization for Bayesian neural networks.

As tunable hyperparameters, we exposed the step length, the length of the burn-in period, the number of units in each layer, and the decay parameter of the momentum variable. A detailed description of the configuration space can be found in Section C.9 in the appendix. We used the Bayesian neural network implementation provided in the RoBO python package (Klein et al., 2017b) as described by Springenberg et al. (2016).

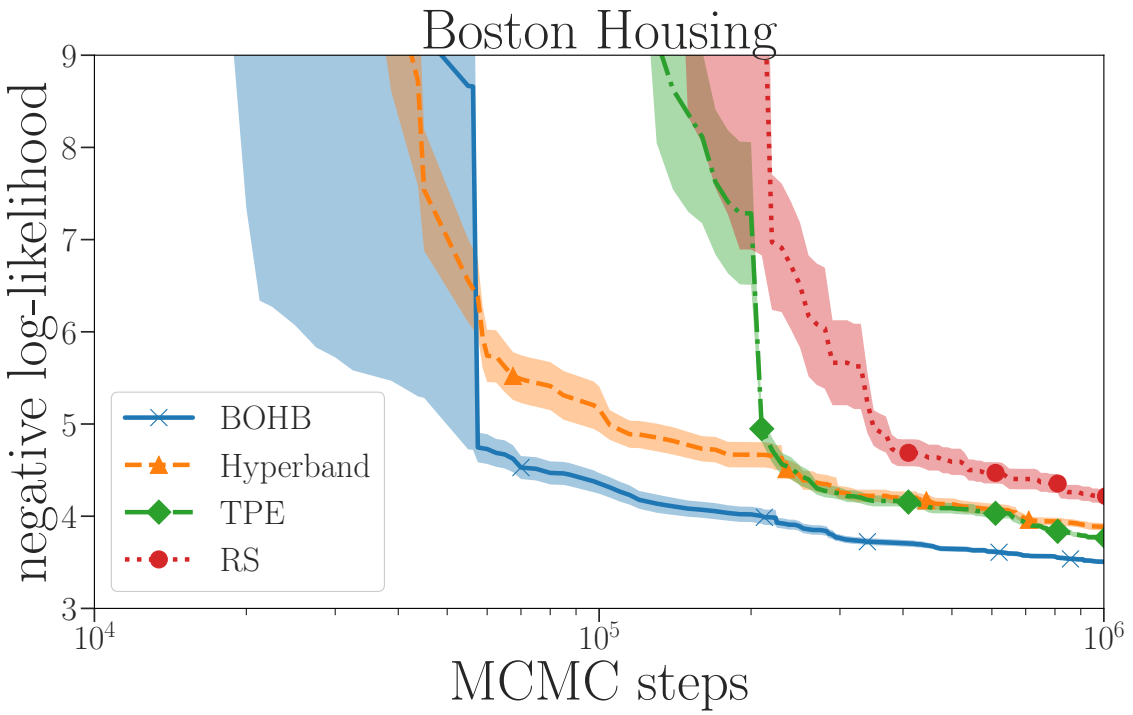


Figure 6.5.: Optimization of 5 hyperparameters of a Bayesian neural network trained with SGHMC. BOHB quickly outperforms both TPE and HB.

We considered two UCI (Lichman, 2013) regression datasets, *Boston housing* and *protein structure* as described by Hernández-Lobato and Adams (2015) and report the negative log-likelihood of the validation data. For BOHB and HB, we set the minimum budget to 500 MCMC steps and the maximum budget to 10000 steps. RS and TPE evaluated each configuration on the maximum budget. For each hyperparameter optimization method, we performed 50 independent runs to obtain statistically significant results.

As Figure 6.5 shows, HB initially performed better than TPE, but TPE caught up given enough time. BOHB converged faster than both HB and TPE and even found a better configuration than the baselines on the Boston housing dataset.

6.4.4. Reinforcement Learning

Next, we optimized eight hyperparameters of proximal policy optimization (PPO) (Schulman et al., 2017) to learn the *cartpole swing-up* task. For PPO, we used the implementation from the TensorForce framework developed by Schaarschmidt et al. (2017) and we used the implementation from OpenAI Gym (Brockman et al., 2016) for the cartpole environment. The configuration space for this experiment can be found in Section C.10 in the appendix.

To find a configuration that not only converges quickly but also works robustly, for

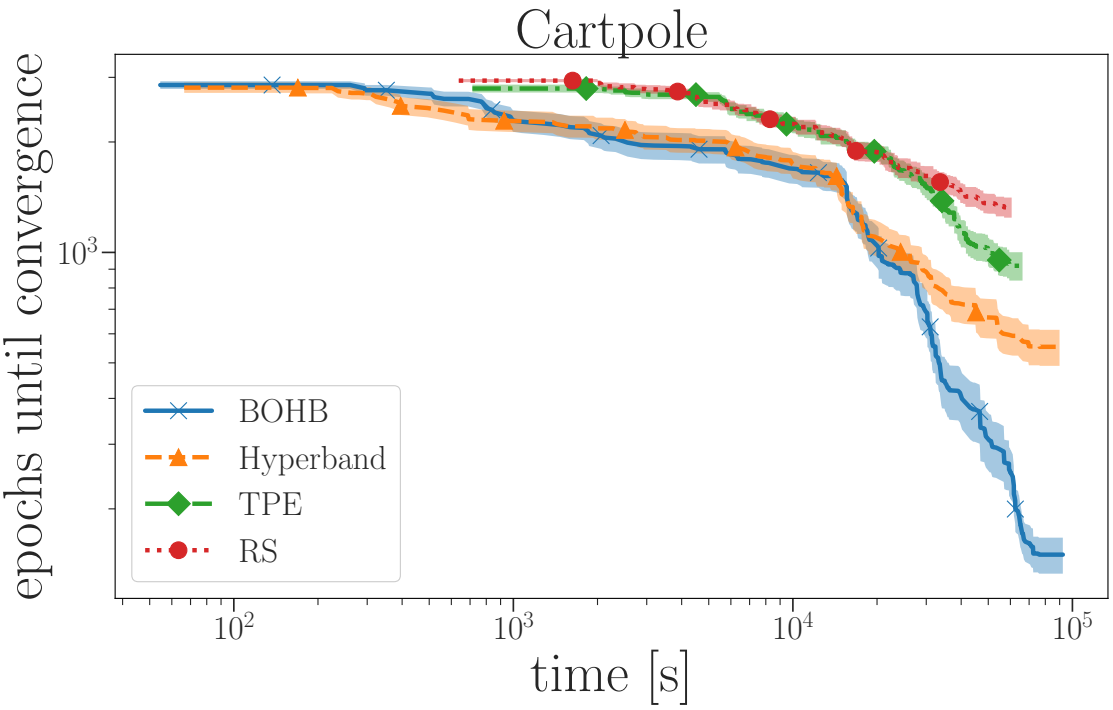


Figure 6.6.: Hyperparameter optimization of 8 hyperparameters of PPO on the cartpole task. BOHB starts as well as HB but converges to a much better configuration.

each function evaluation we ran a configuration for nine individual trials with a different seed for the random number generator. We returned the average number of episodes until PPO has converged to the optimum, defining convergence to mean that the reinforcement learning agent achieved the highest possible reward for 20 consecutive episodes. For each hyperparameter configuration we stopped training after the agent has either converged or ran for a maximum of 3000 episodes. The minimum budget for BOHB and HB was one trial and the maximum budget were nine trials, and all other methods used a fixed number of nine trials. As in the previous benchmark, for each hyperparameter optimization method we performed 50 independent runs.

Figure 6.6 shows that HB and BOHB worked equally well in the beginning, but BOHB converged to better configurations in the end. Apparently, the budget for this benchmark was not sufficient for TPE to find the same configuration.

6.4.5. Convolutional Neural Networks on CIFAR-10

For a final evaluation, we optimized the hyperparameters of a medium-sized residual network (depth 20 and basewidth of 64; roughly 8.5M parameters) with Shake-Shake (Gastaldi, 2017) and Cutout (DeVries and Taylor, 2017) regularization. To perform

hyperparameter optimization, we split off 5 000 training images as a validation set. As hyperparameters, we optimized learning rate, momentum, weight decay, and batch size.

We ran BOHB with budgets of 22, 66, 200, and 600 epochs, using 19 parallel workers. Each worker used 2 NVIDIA TI 1080 GPUs for parallel training, which resulted in runs with the longest budget taking approximately 7 hours (on 2 GPUs). The complete BOHB run of 16 iterations required a total of 33 GPU days (corresponding to a cost of less than 3 full function evaluations on each of the 19 workers) and achieved a test error of $2.78\% \pm 0.09\%$ (which is better than the error Gastaldi (2017) obtained with a slightly larger network). While we note that the performance numbers from different papers are not directly comparable due to the use of different optimization and regularization approaches, it is still instructive to compare this result to others in the literature. Our result is better than that of last year's state-of-the-art neural architecture search by reinforcement learning (3.65% (Zoph and Le, 2017)) and the recent paper on liu-eccv18 neural architecture search (3.41% (Liu et al., 2018)), but it does not quite reach the state-of-the-art performance of 2.4% and 2.1% reported in recent arXiv papers on reinforcement learning (Zoph et al., 2018) and evolutionary search (Real et al., 2019). However, since these approaches used 60 to 95 times more compute resources (2 000 and 3 150 GPU days, respectively!), as well as networks with 3-4 times more parameters, we believe that our results are a strong indication of the practical usefulness of BOHB for resource-constrained optimization.

6.5. Chapter Conclusions

We introduced BOHB, a simple yet effective method for hyperparameter optimization satisfying the desiderata outlined above: it is robust, flexible, scalable (to both high dimensions and parallel resources), and achieves both strong anytime performance and strong final performance. We thoroughly evaluated its performance on a diverse set of benchmarks and demonstrated its improved performance compared to a wide range of other state-of-the-art approaches. Our easy-to-use open-source implementation (available under https://github.com/automl/HpBandSter) should allow the community to effectively use our method on new problems. To further improve BOHB, we will consider an automatic adaptation of the budgets used to alleviate the problem of misspecification by the user while maintaining the versatility and robustness of the current version.

7. Tabular Benchmarks for Hyperparameter Optimization

Due to the high computational demands executing a rigorous comparison between hyperparameter optimization (HPO) methods is often cumbersome. The goal of this Chapter is to facilitate a better empirical evaluation of HPO methods by providing benchmarks that are cheap to evaluate, but still represent realistic use cases. We believe these benchmarks provide an easy and efficient way to conduct reproducible experiments for neural hyperparameter search.

Our benchmarks consist of a large grid of configurations of a feed forward neural network (see Section 7.1) on four different regression datasets including architectural hyperparameters and hyperparameters concerning the training pipeline. We first performed an in-depth analysis of the data to gain a better understanding of the properties of the optimization problem, as well as of the importance of different types of hyperparameters (see Section 7.2 and 7.3). Then, in Section 7.4 we compared various different state-of-the-art methods from the hyperparameter optimization literature on these benchmarks.

7.1. Setup

We use 4 popular UCI (Lichman, 2013) datasets for regression: protein structure (Rana, 2013), slice localization (Graf et al., 2011), naval propulsion (Coraddu et al., 2014) and parkinsons telemonitoring (Tsanas et al., 2010). We call them HPO-Bench-Protein, HPO-Bench-Slice, HPO-Bench-Naval and HPO-Bench-Parkinson, respectively. For each dataset we used 60% for training, 20% for validation and 20% for testing (see Table 7.1 for an overview) and removed features that were constant over the entire dataset. Afterwards, all features and targets values were normalized by subtracting the mean and dividing by the variance of the training data. These datasets do not require deeper neural network architectures which means we can train them on CPUs rather than GPUs and hence we can afford to run many configurations.

As the base architecture, we used a two layer feed forward neural network followed by a linear output layer on top. The configuration space (denoted in Table 7.2) only includes a modest number of 4 architectural choice (number of units and activation functions for both layers) and 5 hyperparameters (dropout rates per layer,

Table 7.1.: Dataset splits

Dataset	# training datapoints	# validation datapoints	# test datapoints	# features
HPO-Bench-Protein	27 438	9 146	9 146	9
HPO-Bench-Slice	32 100	10 700	10 700	385
HPO-Bench-Naval	7 160	2 388	2 388	15
HPO-Bench-Parkinson	3 525	1 175	1 175	20

batch size, initial learning rate and learning rate schedule) in order to allow for an exhaustive evaluation of all the 62 208 configurations resulting from discretizing the hyperparameters as in Table 7.2. We encode numerical hyperparameters as ordinals and all other hyperparameters as categoricals. Each network was trained with Adam (Kingma and Ba, 2015) for 100 epochs, optimizing the mean squared error. We repeated the training of each configuration 4 independent times with a different seed for the random number generator and recorded for each run the training / validation / test accuracy, training time and the number of trainable parameters. We provide full learning curves (i. e. validation and training error for each epoch) as an additional fidelity that can be used to benchmark multi-fidelity algorithms with the number of epochs as the budget. The dataset, as well as the code to reproduce the experiments, are publicly available at `https://github.com/automl/nas_benchmarks`.

Table 7.2.: Configuration space of the fully connected neural network

Hyperparameters	Choices
Initial LR	$\{.0005, .001, .005, .01, .05, .1\}$
Batch Size	$\{8, 16, 32, 64\}$
LR Schedule	$\{\text{cosine}, \text{fix}\}$
Activation/Layer 1	$\{\text{relu}, \tanh\}$
Activation/Layer 2	$\{\text{relu}, \tanh\}$
Layer 1 Size	$\{16, 32, 64, 128, 256, 512\}$
Layer 2 Size	$\{16, 32, 64, 128, 256, 512\}$
Dropout/Layer 1	$\{0.0, 0.3, 0.6\}$
Dropout/Layer 2	$\{0.0, 0.3, 0.6\}$

7.2. Dataset Statistics

We now analyze the properties of these datasets. First, for each dataset we computed the empirical cumulative distribution function (ECDF) of the test, validation and training error after 100 epochs and the total training time. For each metric, we averaged over the 4 repetitions. Additionally we computed the ECDF for the number of trainable parameters of each neural network architecture. To avoid clutter, we show here only the results for the HPO-Bench-Protein which we found to be consistent with the other datasets and present all results in Section D.1 in the supplemental material.

One can see in Figure 7.1 that the mean-squared-error (MSE) for training, validation and test is spread over an order of magnitude or more. On one side only a small

subset of configurations achieve a final MSE lower than 0.3 and on the other many outliers exist that achieve errors orders of magnitude above the average. Furthermore, due to the changing number of parameters, also the training time varies dramatically across configurations.

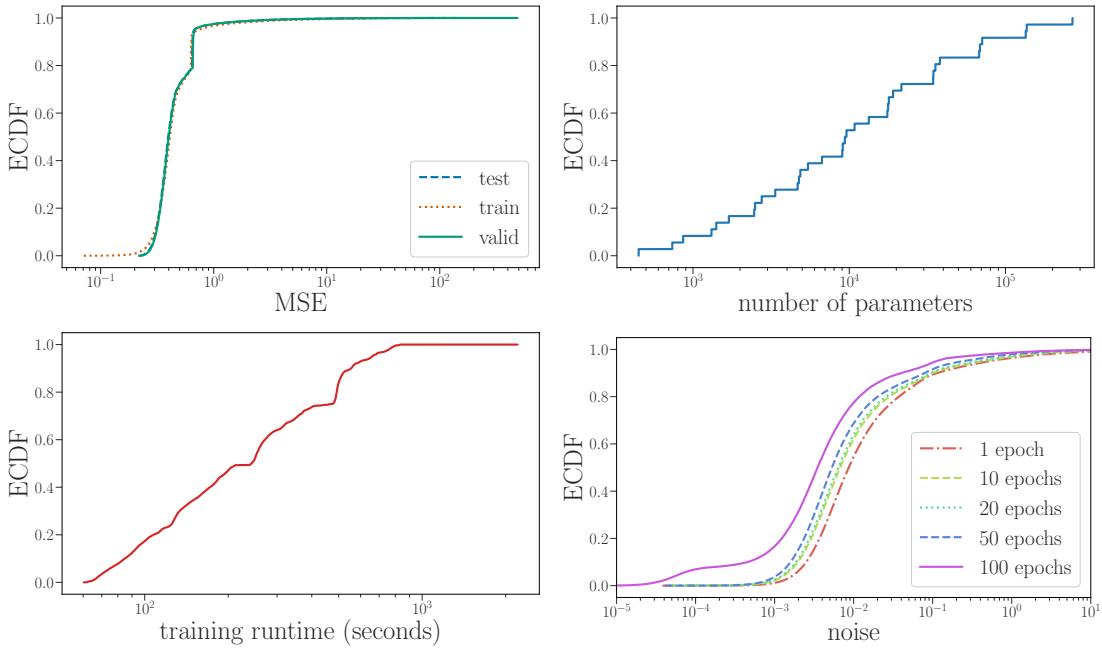


Figure 7.1.: The empirical cumulative distribution (ECDF) of the average train/valid/test error after 100 epochs of training (upper left), the number of parameters (upper right), the training runtime (lower left) and the noise for different number of epochs (lower right) computed on HPO-NAS-bench-Protein. See Section D.1 in the supplemental material for the ECDF plots of all datasets.

Figure 7.1 bottom right shows the empirical cumulative distribution of the noise, defined as the standard deviation between the 4 repetitions for different number of epochs. We can see that the noise is heteroscedastic. That is, different configurations come with a different noise level. As expected, the noise decreases with an increasing number of epochs.

For many multi-fidelity hyperparameter optimization methods, such as Hyperband (Li et al., 2017) or BOHB (Falkner et al., 2018a), it is essential that the ranking of configurations on smaller budgets to higher budgets is preserved. In Figures 7.2, we visualize the Spearman rank correlation between the performance of all hyperparameter configurations across different number of epochs and the highest budget of 100 epochs. Since every hyperparameter optimization method needs to mainly focus on the top performing configurations, we also show the correlation for only the top 1%, 10%, 20%, and 50% of all configurations. As expected the correlation to the highest budget increases with increasing budgets. If only top-performing configurations are

considered, the correlation decreases, since their final performances are closer to each other.

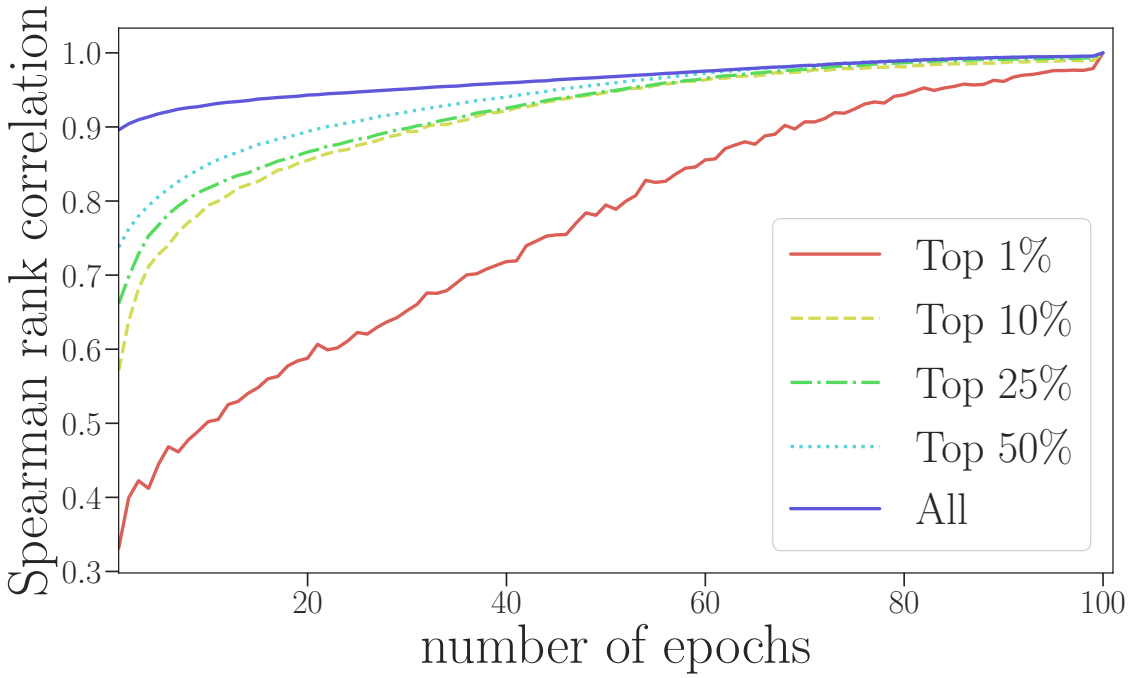


Figure 7.2.: The Spearman rank correlation between different number of epochs to the highest budget of 100 epochs for the HPO-Bench-Protein when we consider all configurations or only the top 1%, 10%, 20%, and 50% of all configurations based on their average test error. Results for other datasets are presented in Section D.1 in the supplemental material.

7.3. Hyperparameter Importance

We now analyze how the different hyperparameters affect the final performance, first globally with help of the functional ANOVA (Sobol, 1993; Hutter et al., 2014a) and then from a more local point of view. Finally, we show how the top performing hyperparameter configurations correlate across the different datasets. As in the previous section, we show here only the results for HPO-Bench-Protein and for all other dataset in Section D.2 in the supplemental material.

7.3.1. Functional ANOVA

To analyze the importance of hyperparameters, assessing the change of the final error with respect to changing a single hyperparameter at a time, we used the fANOVA

tool by Hutter et al. (2014a). It quantifies the importance of a hyperparameter by marginalizing the error obtained by setting it to a specific value over all possible values of all other hyperparameters. The importance of a hyperparameter is then the variation in error that is explained by this hyperparameter. In default setting this tool fits a random forest model on the observed function values in order to compute the marginal predictions. However, since we already evaluated the full configuration space, we do not even need to use a model and can compute the required integrals directly.

As can be seen in Figure 7.3 (upper right), on average across the entire configuration space, the initial learning rate obtained the highest importance value. However, the importance of individual hyperparameters is very small due to a few outliers with very high errors, which only happen for a few combinations of several hyperparameter values. We also computed the importance values of hyperparameter configuration pairs (see Figure 7.3 lower right for the ten most important pairs), which obtain slightly higher values. This indicates that there are higher order interaction effects between hyperparameters. Unfortunately, computing higher than second order interaction effect is computational infeasible.

A better estimate of hyperparameter importance in a region of the configuration space with reasonable performance can be obtained by only using the best performing configuration for the fANOVA. Figure 7.3 (left) shows the results of this procedure with the 1 percentile and Figure 7.3 (middle) with the 10 percentile of all configuration. This shows that in this more interesting part of the configuration space, other hyperparameters also become important.

7.3.2. Local Neighbourhood

While the fANOVA takes the whole configuration space into account, we now focus on a more local view around the best configuration (incumbent) to see how robust it is against small perturbations. We show in Table 7.3 the change in performance if we flip single hyperparameters of the incumbent while keeping all other hyperparameters fixed. Additionally, we also show in the rightmost column the relative change $\frac{y_{new}-y_\star}{y_\star}$ between the error of the incumbent $y_\star$ and the new observed error y_{new}.

Interestingly, the highest drop in performance occurs by changing the activation function of the first and the second layer from relu to tanh. This is despite the fact that tanh is a much more common activation function for regression than relu. In contrast, increasing the batch size only has a marginal effect on the performance.

7.3.3. Ranking across Datasets

We now analyse hyperparameter configurations across the four different datasets. In Table 7.4 we can see that the best configuration in terms of average test error changes

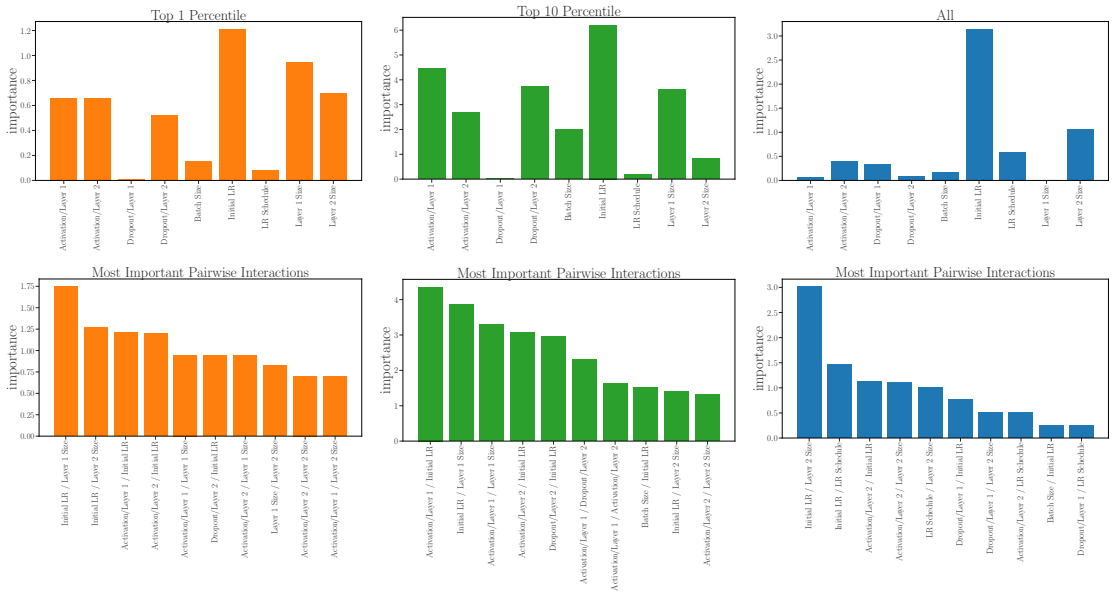


Figure 7.3.: Top row: Importance of the different hyperparameter based on the fANOVA for: (left) only the top 1% ; (middle) top 10% ; (right) all configurations. Bottom row: most important hyperparameter pairs with (left) only the top 1% ; (middle) top 10% ; (right) all configurations.

only slightly across datasets. For some hyperparameters, such as the learning rate, a certain value, 0.0005 in this case, can be used for all all datasets whereas other hyperparameters, for example the activation functions, need to be set differently.

To see how the performance of all hyperparameter configurations correlates across datasets, we computed for every configuration on every dataset its rank in terms of final average test performance. Figure 7.4 shows the Spearman rank correlation between the different datasets if we consider the first percentile (left), the 10th percentile (middle) or all configurations (right). The correlation decreases if we only consider the best-performing configurations, which implies that it does not suffice to reuse a good configuration from a different datasets to achieve top performance on a new dataset. Nevertheless, the correlation for all configurations is high which indicates that multi-task methods could be able to exploit previously collected data.

7.4. Comparison

In this section we use the generated benchmarks to evaluate different HPO methods. To mimic the randomness that comes with evaluating a configuration, in each function evaluation we randomly sample one of the four performance values. To obtain a realistic estimate of the wall-clock time required for each optimizer, we accumulated the stored runtime of each configuration the optimizer evaluated. We do not take the additional overhead of the optimizer into account since it is negligible compared to

Table 7.3.: Performance change if single hyperparameters of the incumbent (average test error 0.2153) are flipped.

Hyperparameter	Change	Test Error	Relative Change
Batch Size	$8 \rightarrow 16$	0.2163	0.0042
Initial LR	$0.0005 \rightarrow 0.001$	0.2169	0.0072
Layer 2 Size	$512 \rightarrow 256$	0.2203	0.0231
Layer 1	$512 \rightarrow 256$	0.2216	0.0288
Dropout/Layer 2	$0.3 \rightarrow 0.6$	0.2257	0.0478
LR Schedule	*cosine* $\rightarrow$ *const*	0.2269	0.0534
Dropout/Layer 2	$0.3 \rightarrow 0.0$	0.2280	0.0587
Dropout/Layer 1	$0.0 \rightarrow 0.3$	0.2307	0.0711
Activation/Layer 2	*relu* $\rightarrow$ *tanh*	0.2875	0.3351
Activation/Layer 1	*relu* $\rightarrow$ *tanh*	0.3012	0.3987

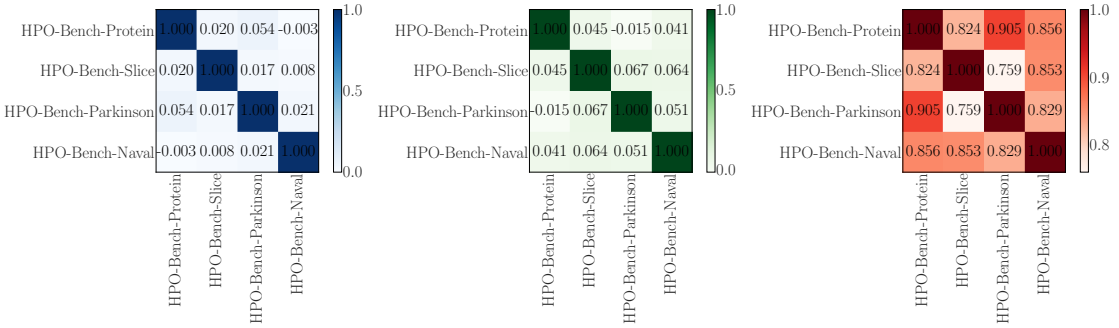


Figure 7.4.: Correlation of the ranks for (left) top-1% / (middle) top-10% and all hyperparameter configurations across all four datasets.

Table 7.4.: Best configurations in terms of average test error for each dataset

Hyperparameters	HPO-Bench-Protein	HPO-Bench-Slice	HPO-Bench-Naval	HPO-Bench-Parkinson
Initial LR	0.0005	0.0005	0.0005	0.0005
Batch Size	8	32	8	8
LR Schedule	cosine	cosine	cosine	cosine
Activation/Layer 1	relu	relu	tanh	tanh
Activation/Layer 2	relu	tanh	relu	relu
Layer 1 Size	512	512	128	128
Layer 2 Size	512	512	512	512
Dropout/Layer 1	0.0	0.0	0.0	0.0
Dropout/Layer 2	0.3	0.0	0.0	0.0

the training time of the neural network. After each function evaluation we estimate the incumbent as the configuration with the lowest observed error and compute the regret between the incumbent and the globally best configuration in terms of test error. We performed 500 independent runs of each method and report the median and the 25th and 75th quantiles.

7.4.1. Performance over Time

We compared the following HPO methods from the literature (see Figure 7.5): random search (Bergstra and Bengio, 2012), SMAC (Hutter et al., 2011)[1], Tree Parzen Estimator (TPE) (Bergstra et al., 2011)[2], Bohamiann (Springenberg et al., 2016)[3] (see also Chapter 3), Regularized Evolution (Real et al., 2019), Hyperband (HB) (Li et al., 2017) and BOHB (Falkner et al., 2018a)[4] (see also Chapter 6). Inspired by the recent success of reinforcement learning for neural architecture search (Zoph and Le, 2017), we also include a similar reinforcement learning strategy (RL), which however does not use an LSTM as controller but instead uses REINFORCE (Williams, 1992) to optimize the probability of each categorical variable directly (Ying et al., 2019). Each method that operates on the full budget of 100 epochs was allowed to perform 500 function evaluations. For BOHB and HB we set the minimum budget to 3 epochs, the maximum budget to 100, η to 3 and the number of successive halving iterations to 125 (which leads to roughly the same amount of function evaluation time as the other methods). More details about the meta-parameters of the different optimizers are described in Appendix D.3.

Figure 7.5 left show the performance over time for all methods. Results for the other datasets can be found in Appendix D.3. We can make the following observations:

- As expected, Bayesian optimization methods, i. e. SMAC, TPE and Bohamiann worked as well as RS in the beginning but started to perform superior once they obtained a meaningful model. Interestingly, while all Bayesian optimization methods start improving at roughly the same time, they converge to different optima, which we attribute to their different internal models.

- The same holds for BOHB, which is in the beginning as good as HB but starts outperforming it as soon as it obtains a meaningful model.

- HB achieved a reasonable performance relatively quickly but only slightly improves over simple RS eventually.

- RE needed more time than Bayesian optimization methods to outperform RS; however, it often achieved the best final performance, since, compared to Bayesian optimization methods, it does not suffer from any model missmatch.

- RL requires even more time to improve upon RS than RE or Bayesian optimization and seems to be too sample inefficient for these tasks.

[1] We used SMAC3 from `https://github.com/automl/SMAC3`

[2] We used Hyperopt from `https://github.com/hyperopt/hyperopt`

[3] We used the implementation from Klein et al. (2017b)

[4] For both HB and BOHB we used the implementation from `https://github.com/automl/HpBandSter`

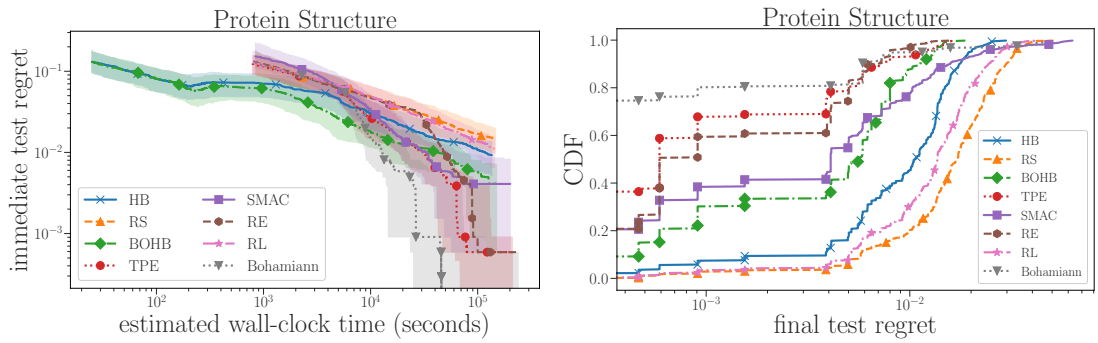


Figure 7.5.: Left: Comparison of various HPO methods on the HPO-Bench-Protein datasets. For each method, we plot the median and the 25th and 75th quantiles (shaded area) of the test regret of the incumbent (determined based on the validation performance) across 500 independent runs. Right: Empirical cumulative distribution of the final performance over all runs of each methods after 10^5 seconds.

7.4.2. Robustness

Besides achieving good performance, we argue that robustness plays an important role in practice for HPO methods. Figure 7.5 shows the empirical cumulative distribution of the test regret for the final incumbent after 10^5 seconds for HPO-Bench-Protein across all 500 runs of each method.

While RE achieves a lower mean test regret than TPE it seems to be less robust with respect to its internal randomness. Interestingly, while all methods have non-zero probability to achieve a final test regret of 10^{-3} within 10^5 seconds, only Bohamiann, RE and TPE are able to achieve this regret in more than 60% of the cases. Also none of the methods is able to converge consistently to the same final regret.

7.5. Chapter Conclusions

We presented new tabular benchmarks for neural architecture and hyperparameter search that are cheap to evaluate but still recover the original optimization problem, enabling us to rigorously compare various methods from the literature. Based on the data we generated for these benchmarks, we had a closer look at the difficulty of the optimization problem and the importance of different hyperparameters.

In future work, we will generate more of these benchmarks for other architectures and datasets. Ultimately, we hope that such benchmarks will help the community to easily reproduce experiments and evaluate new developed methods without spending enormous compute resources.

8. Meta-Surrogate Benchmarking for Hyperparameter Optimization

Despite recent progress (see e. g. the review by Feurer and Hutter (2018)), during the phases of developing and evaluating new HPO methods one frequently faces the following problems:

- Evaluating the objective function is often expensive in terms of wall-clock time; *e.g.*, the evaluation of a single hyperparameter configuration may take several hours or days. This renders extensive HPO or repeated runs of HPO methods computationally infeasible.
- Even though repositories of datasets such as OpenML (Vanschoren et al., 2014) provide thousands of datasets, a large fraction cannot meaningfully be used for HPO since they are too small or too easy (in the sense that even simple methods achieve top performance). Hence, useful available datasets are scarce, making it hard to produce a comprehensive evaluation of how well a HPO method will generalize across tasks.

Due to these two problems researchers can only carry out a limit number of comparisons within a reasonable computational budget. This delays the progress of the field as statistically significant conclusions about the performance of different HPO methods may not be possible to draw. See Figure 8.1 for an illustrative experiment of the HPO of XGBoost (Chen and Guestrin, 2016). It is well known that Bayesian optimization with Gaussian processes (BO-GP) (Shahriari et al., 2016) outperforms naive random search (RS) in terms of number of function evaluations on most HPO problems. While we show clear evidence for this in Appendix E.2 on a larger set of datasets, this conclusion cannot be reached when optimizing on the three "unlucky" picked datasets in Figure 8.1. Surprisingly, the community has not paid much attention to this issue of proper benchmarking, which is a key step required to generate new scientific knowledge but also to foster reproducibility.

In this Chapter we present a generative meta-model that, conditioned on off-line generated data, allows to sample an unlimited number of new tasks that share properties with the original ones. There are several advantages to this approach. First, the new problem instances are inexpensive to evaluate as they are generated with a parameteric form, which drastically reduces the resources needed to compare HPO methods, bounded only by the optimizer's computational overhead. See Figure 8.2 for an example. Second, there is no limit in the number of tasks that can be

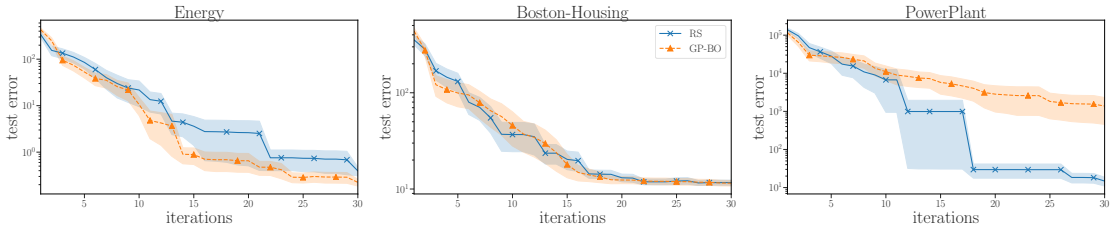


Figure 8.1.: Common pitfalls in the evaluation of HPO methods: we compare two different HPO methods for optimizing the hyperparameters of XGBoost on three UCI regression datasets (see Appendix B for more datasets). The small number of tasks makes it hard to draw any conclusions, since the ranking between the methods varies between the tasks. Furthermore, a full run might take several hours which makes it prohibitively expensive to average across a large number of runs.

generated, which helps to draw statistically more reliable conclusions. Third, the shape and properties of the tasks are not predefined but learned using a few real tasks of an HPO problem. While the *global* properties of the initial tasks are preserved in the samples, the generative model allows the exploration of instances with diverse *local* properties making comparisons more robust and reliable (see Appendix E.4 for some example tasks).

In light of the recent call for more reproducibility, we are convinced that our meta-surrogate benchmarks enable more reproducible research in AutoML: First of all, these cheap-to-evaluate surrogate benchmarks allows researches to reproduce experiments or perform many repeats of their own experiments without relying on tremendous computational resources. Second, based on our-proposed method, we provide a more thorough benchmarking protocol that reduces the risk of extensively tuning an optimization method on single tasks. Third, surrogate benchmarks in general are less dependent on hardware and technical details, such as complicated training routines or preprocessing strategies.

8.1. Related Work

The use of meta-models that learn across tasks has been investigated by others before. To warm-start HPO on new tasks from previously optimized tasks, Swersky et al. (2013) extended Bayesian optimization to the multi-task setting by using a Gaussian process that also models the correlation between tasks. Instead of a Gaussian process, Springenberg et al. (2016) used a Bayesian neural network inside multi-task Bayesian optimization which learns an embedding of tasks during optimization. Similarly Perrone et al. (2018) used Bayesian linear regression, where the basis functions are learned by a neural network, to warm-start the optimization from previous tasks.

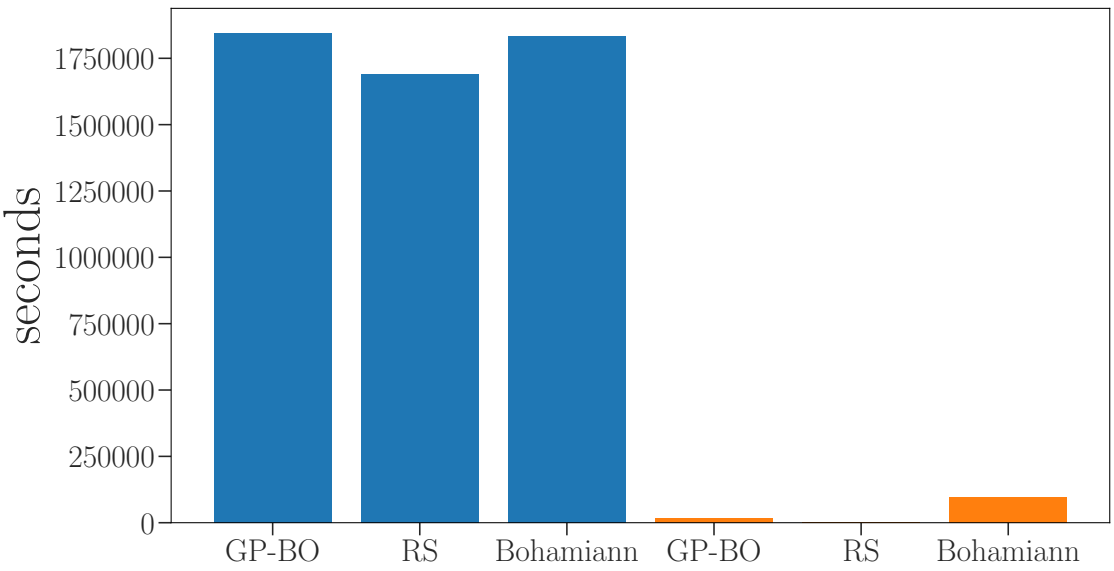


Figure 8.2.: The three blue bars on the left show the total wall-clock time of executing 20 independent runs of GP-BO, RS and Bohamiann (see Section 8.4) with 100 function evaluation for the HPO of a feed forward neural network on MNIST. The orange bars show the same for optimizing a tasks sampled from our proposed meta-model, where benchmarking is orders of magnitude cheaper in terms of wall-clock time than the original benchmarks, thereby the computational time is almost exclusively spend for the optimizer overhead (hence the larger bars for GP-BO and Bohamiann compared to RS).

Feurer et al. (2015b) used a set of dataset statistics as meta-features to measure the similarity between tasks, such that hyperparameter configurations that were superior on previously optimized similar tasks can be evaluated during the initial design before the actual optimization procedure starts. This technique is also applied inside the auto-sklearn framework (Feurer et al., 2015a). In a similar vein Fusi et al. (2018) proposed to use a probabilistic matrix factorization approach to exploit knowledge gathered on previously seen tasks. van Rijn and Hutter (2018) evaluated random hyperparameter configurations on a large range of tasks to learn priors for a support vector machine, random forest and Adaboost. The idea of using a latent variable to represent correlation among multiple outputs of Gaussian process has been exploited by Dai et al. (2017).

In the context of benchmarking HPO methods, HPOlib (Eggensperger et al., 2013) is a benchmarking library that provides a fixed and rather small set of problems that have been used to compare several Bayesian optimization tools. In earlier work, Eggensperger et al. (2015) also used surrogates to speed up the empirical benchmarking of HPO methods. Similar to our work, these surrogates are trained on data generated in an off-line step. Afterwards, function evaluations require only prediction of the surrogate model instead of actually running the benchmark. However,

these surrogates only mimic one particular task and do not allow for generating new tasks as presented in this work. Recently, tabular benchmarks were introduced for neural architecture search (Ying et al., 2019) and hyperparameter optimization (Klein and Hutter, 2019), which first perform an exhaustive search of a discrete benchmark problem to store all results in a database and then replace expensive function evaluations by efficient table lookups. While this does not introduce any bias due to a model (see Section 8.5 for a more detailed discussion), tabular benchmarks are only applicable for problems with few, discrete hyperparameters. Related to our work, but for benchmarking general blackbox optimization methods, is the COCO platform (Hansen et al., 2016b). However, compared to our approach, it is based on handcrafted synthetic functions that do not resemble real world HPO problems.

8.2. Benchmarking HPO methods with Generative Models

We now describe the generative meta-model to create HPO tasks. First we give a formal definition of benchmarking HPO methods across tasks sampled from a unknown distribution and then describe how we can approximate this distribution by our new proposed meta-model.

8.2.1. Problem Definition

We denote $t_1, \ldots, t_M$ to be a set of related objectives/tasks with the same input domain $\mathcal{X}$. We assume that each t_i for $i = 1, \ldots M$, is an instantiation of an unknown distribution of tasks $t_i \sim p(t)$. Every task t has an associated objective function $f_t : \mathcal{X} \subset \mathbb{R}^d \to \mathbb{R}$ where $\boldsymbol{x} \in \mathcal{X}$ represents a hyperparameter configuration and we assume that we can observe f_t only through noise: $y_t \sim \mathcal{N}(f_t(\boldsymbol{x}), \sigma_t^2)$.

Let us denote by $r(\alpha, t)$ the performance of an optimization method α on a task t; for instance, a common example for r is the regret of the best observed solution (called incumbent). To compare two different methods α_A and α_B, the standard practice is to compare $r(\alpha_A, t_i)$ with $r(\alpha_B, t_i)$ on a set of hand-picked tasks $t_i \in \{t_0, \ldots t_M\}$. However, to draw statistically more significant conclusions, we would ideally like to integrate over all tasks:

$$S_{p(t)}(\alpha) = \int r(\alpha, t) p(t) dt. \tag{8.1}$$

Unfortunately, the above integral is intractable as $p(t)$ is unknown. The main contribution of this paper is to approximate $p(t)$ with a generative meta-model $\hat{p}(t \mid \mathcal{D})$ based on some off-line generated data $\mathcal{D} = \left\{ \{(\boldsymbol{x}_{tn}, y_{tn})\}_{n=1}^{N} \right\}_{t=1}^{T}$. This enables us to sample an arbitrary amount of tasks $t_i \sim \hat{p}(t \mid \mathcal{D})$ in order to perform a Monte-Carlo approximation of Equation 8.1.

8.2.2. Meta-Model for Task Generation

In order to reason across tasks, we define a probabilistic encoder $p(\mathbf{h}_t \mid \mathcal{D})$ that learns a latent representation $\mathbf{h}_t \in \mathbb{R}^Q$ of a task t.

More precisely, we use Bayesian GP-LVM (Titsias and Lawrence, 2010) which assumes that the target values that belong to the task t, stacked into a vector $\boldsymbol{y}_t = (y_{t1}, \ldots, y_{tN})$ follow the generative process:

$$\boldsymbol{y}_t = g(\mathbf{h}_t) + \epsilon, \quad g \sim \mathcal{GP}(0, k), \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{8.2}$$

where k is the covariance function of the GP. By assuming that the latent variable $\mathbf{h}_t$ has an uninformative prior $\mathbf{h}_t \sim \mathcal{N}(0, \mathbf{I})$, the latent embedding of each task is inferred as the posterior distribution $p(\mathbf{h}_t \mid \mathcal{D})$. The exact formulation of the posterior distribution is intractable, but following the variational inference presented in Titsias and Lawrence (2010), we can estimate a variational posterior distribution $q(\mathbf{h}_t) = \mathcal{N}(m_t, \Sigma_t) \approx p(\mathbf{h}_t \mid \mathcal{D})$ for each task t.

Similar to Multi-Task Bayesian Optimization (Swersky et al., 2013; Springenberg et al., 2016), we define a probabilistic model for the objective function $p(y_t \mid \boldsymbol{x}, \mathbf{h}_t)$ across tasks which gets as an additional input a task embedding based on our independently trained probabilistic encoder. Following (Springenberg et al., 2016), we use a Bayesian neural network with M weight vectors $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M\}$ to model

$$p(y_t \mid \boldsymbol{x}, \mathbf{h}_t, \mathcal{D}) = \int p(y_t \mid \boldsymbol{x}, \mathbf{h}_t, \boldsymbol{\theta}) p(\theta \mid \mathcal{D}) d\boldsymbol{\theta} \approx \frac{1}{M} \sum_{i=1}^{M} p(y_t \mid \boldsymbol{x}, \mathbf{h}_t, \boldsymbol{\theta}_i). \tag{8.3}$$

where $\theta_i \sim p(\theta \mid \mathcal{D})$ is sampled from the posterior of the neural network weights.

By approximating $p(y_t \mid \boldsymbol{x}, \mathbf{h}_t) = \mathcal{N}\Big(\mu(\boldsymbol{x}, \mathbf{h}_t), \sigma^2(\boldsymbol{x}, \mathbf{h}_t)\Big)$ to be Gaussian, we can compute the predictive mean and variance by (Springenberg et al., 2016):

$$\mu(\boldsymbol{x}, \mathbf{h}_t) = \frac{1}{M} \sum_{i=1}^{M} \hat{\mu}(\boldsymbol{x}, \mathbf{h}_t \mid \boldsymbol{\theta}_i)$$

$$\sigma^2(\boldsymbol{x}, \mathbf{h}_t) = \frac{1}{M} \sum_{i=1}^{M} \Big(\hat{\mu}(\boldsymbol{x}, \mathbf{h}_t \mid \boldsymbol{\theta}_i) - \mu(\boldsymbol{x}, \mathbf{h}_t)\Big)^2 + \hat{\sigma}^2_{\boldsymbol{\theta}_i},$$

where $\hat{\mu}(\boldsymbol{x}, \mathbf{h}_t \mid \boldsymbol{\theta}_i)$ and $\hat{\sigma}^2_{\boldsymbol{\theta}_i}$ are the output of a single neural network with parameters $\boldsymbol{\theta}_i$[1]. To get a set of weights $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M\}$, we use stochastic gradient Hamiltonian Monte-Carlo (Chen et al., 2014) to sample $\boldsymbol{\theta}_i \sim p(\boldsymbol{\theta}, \mathcal{D})$ from:

$$p(\boldsymbol{\theta}, \mathcal{D}) = \frac{1}{N} \sum_{n} \frac{1}{H} \sum_{j} \log p(y_n \mid \boldsymbol{x}_n, \mathbf{h}_{nj})$$

with $N = |\mathcal{D}|$ and H the number of samples we draw from the latent space $\mathbf{h}_{tj} \sim q(\mathbf{h}_t)$.

[1]Note that we model an homoscedastic noise, because of that, $\hat{\sigma}^2_{\boldsymbol{\theta}_i}$ does not depend on the input

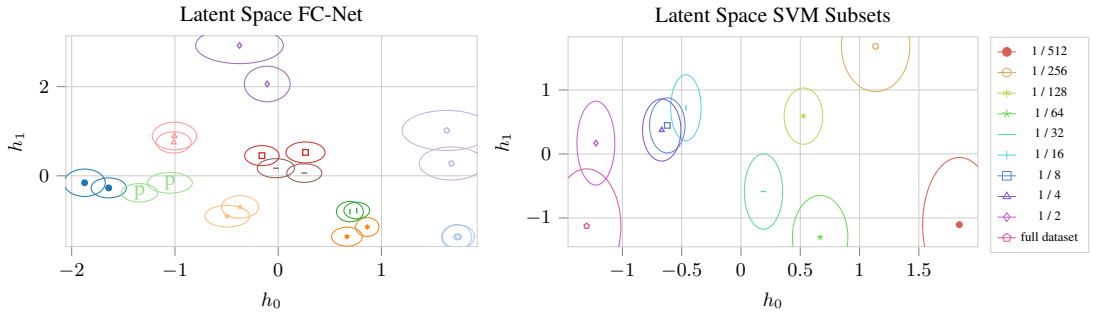


Figure 8.3.: Latent space representations of two different problem classes. *Left*: Representation of eleven pairs of datasets generated by partitioning eleven datasets from the fully connected networks benchmark detailed in Section 8.3.1. Pairs of tasks are represented with the same colour. The mean of the task are represented with different markers. The ellipses represent 4 standard deviations around the mean of the tasks. *Right*: Latent space learned for a model where the input tasks are generated by training a support vector machine on subsets of a target dataset (approximated by a random forest surrogate from Klein et al. (2017a)). One can see that our probabilistic encoder learns a meaningful embedding of different tasks.

8.2.3. Sampling New Tasks

In order to generate a new task $t_\star \sim \hat{p}(t \mid \mathcal{D})$, we need the associated objective function $f_{t_\star}$ in a parameteric from such that we can evaluate it later on any $\boldsymbol{x} \in \mathcal{X}$.

Given the meta-model above, we perform the following steps: (i) we sample a new latent task vector $\mathbf{h}_{t_\star} \sim q(\mathbf{h}_t)$; (ii) given $\mathbf{h}_{t_\star}$ we pick a random $\boldsymbol{\theta}_i$ from the set of weights $\{\boldsymbol{\theta}_1, \dots \boldsymbol{\theta}_M\}$ of our Bayesian neural network and set the new task to be $f_{t_\star}(\boldsymbol{x}) = \hat{\mu}(\boldsymbol{x}, \mathbf{h}_{t_\star} \mid \boldsymbol{\theta}_i)$.

Note that using $f_{t_\star}(\boldsymbol{x})$ makes our new task unrealisticly smooth. Instead, we can emulate the typical noise appearing in HPO benchmarks by returning $y_{t_\star}(\boldsymbol{x}) \sim \mathcal{N}\left(\hat{\mu}(\boldsymbol{x}, \mathbf{h}_{t_\star} \mid \boldsymbol{\theta}_i), \hat{\sigma}^2_{\boldsymbol{\theta}_i}\right)$, which can be done at an insignificant cost.

8.3. Profet

We now present our probabilistic data-efficient experimentation tool, called Profet, a benchmarking suite for HPO methods. The following section describes first how we collected the data to train our meta-model based on three typical HPO problems classes. We then explain how we generated $T = 1000$ different tasks for each problem class from our meta-model. As described above, we provide a noisy and noiseless version of each task. Last, we discuss two ways that are commonly used in the literature to assess and aggregate the performance of HPO methods across tasks. To reproduce our experiments as well as benchmarking and developing

new HPO methods, an open-source implementation of PROFET is available here: https://github.com/aaronkl/emukit.

8.3.1. Data Collection

We consider three different HPO problems, two for classification and one for regression, with varying dimensions D. For classification we considered a support vector machine (SVM) with $D = 2$ hyperparameters and a feed forward neural network (FC-Net) with $D = 6$ hyperparameters on 16 OpenML (Vanschoren et al., 2014) tasks each. We used gradient boosting (XGBoost)[2] with $D = 8$ hyperparameters for regression on 11 different UCI datasets (Lichman, 2013). For further details about the datasets and the configuration spaces see Appendix E.1. To make sure that our meta-model learns a descriptive representation we need a solid coverage over the whole input space. For that we drew $100D$ pseudo randomly generated configurations from a Sobol grid (Sobol, 1967).

Details of our meta-model are described in Appendix E.6. We show some qualitative examples of our probabilistic encoder in Section 8.4.1. We can also apply the same machinery to model the cost in terms of computation time for evaluating a hyperparameter configuration to use time rather than function evaluations as budget. This enables future work to benchmark or develop HPO methods that explicitly take the cost into account (e. g. EIperSec by Snoek et al. (2012)).

8.3.2. Performance Assessment

To assess the performance of a HPO method aggregate over tasks, we consider two different ways commonly used in the literature. First, we measure the **runtime** $r(\alpha, t, y_{target})$ that a HPO method α needs to find a configuration that achieves a performance that is equal or lower than a certain target value y_{target} on task t (Hansen et al., 2016a). Here we define runtime either in terms of function evaluations or estimated wall-clock time predicted by our meta-model. Using a fixed target approach allows us to make quantitative statements, such as: *method A is, on average, twice as fast than method B*. See Hansen et al. (2016a) for a more detailed discussion. We average across target values with a different complexity by evaluating the Sobol grid from above on each generated task. We use the corresponding function values as targets, which, with the same argument as described in Section 8.3.1, provides a good coverage of the error surface. To aggregate the runtime we use the empirical cumulative distribution function (ECDF) (Moré and Wild, 2009), which, intuitively, shows for each budget on the x-axis the fraction of solved tasks and target pairs on the y-axis (see Figure 8.5 left for an example).

[2]We used the implementation from Chen and Guestrin (2016)

Another common way to compare different HPO methods is to compute the **average ranking score** in every iteration and for every task (Bardenet et al., 2013). We follow the procedure described by Feurer et al. (2015b) and compute the average ranking score as follows: assuming we run K different HPO methods M times for each task, we draw a bootstrap sample of 1000 runs out of the K^M possible combinations. For each of these samples, we compute the average fractional ranking (ties are broken by the average of the ordinal ranks) after each iteration. At the end, all the assigned ranks are further averaged over all tasks. Note that averaged ranks are a relative performance measurement and can worsen for one method if another method improves (see Figure 8.5 right for an example).

8.4. Experiments

In this section we present: (i) some qualitative insights of our meta-model by showing how it is able to coherently represent a sets of tasks in its latent space, (ii) an illustration of why PROFET helps to obtain statistically meaningful results and (iii) a comparison of various methods from the literature on our new benchmark suite. In particular, we show results for the following state-of-the-art Bayesian optimization (BO) methods for HPO as well as two popular evolutionary algorithms for general continuous black-box optimization:

- BO with Gaussian processes (BO-GP) (Jones et al., 1998). We used expected improvement as acquisition function and marginalize over the Gaussian process' hyperparameters as described by Snoek et al. (2012).
- SMAC (Hutter et al., 2011): which is a variant of BO that uses random forests to model the objective function and stochastic local search to optimize expected improvement.
  We use the implementation from https://github.com/automl/SMAC3.
- The BO method TPE by Bergstra et al. (2011) which models the density of good and bad configurations in the input space with a kernel density estimators. We used the implementation provided from the Hyperopt package (Komer et al., 2014)
- BO with Bayesian neural networks (BOHAMIANN) as described by Springenberg et al. (2016). To avoid introducing any bias, we used a different architecture with less parameters (3 layers, 50 units in each) than we used for our meta-model (see Section 8.2).
- Differential Evolution (DE) (Storn and Price, 1997) (we used our own implementation) with rand1 strategy for the mutation operators and a population size of 10.
- Covariance Matrix Adaption Evolution Strategy (CMA-ES) by Hansen (2006) where we used the implementation from https://github.com/CMA-ES/pycma

- Random Search (RS) (Bergstra and Bengio, 2012) which samples configurations uniformly at random.

For BO-GP, BOHAMIANN and RS we used the implementation provided by the RoBO package (Klein et al., 2017b). We provide more details for every method in Appendix E.5.

8.4.1. Tasks Representation in the Latent Space

We demonstrate the interpretability of the learned latent representations of tasks in two examples. For the first experiment we used the fully connected network benchmark described in Section 8.3.1. To visualize that our meta-model learns a meaningful latent space, we doubled 11 out of the 18 original tasks to train the model by splitting each one of them randomly in two of the same size. Thereby, we guarantee that there are pairs of tasks that are similar to each other. In Figure 8.3 (left), each color represents the partition of the original task and each ellipse represents the mean and four times the standard deviation of the latent task representations. One can see that the closest neighbour of each task is the other task that belongs to the same original task.

The second experiment targets multi-fidelity experiments that arise when a machine learning model needs to be trained on a very large dataset and approximate versions of the target objective are generated by considering subsamples of different sizes. For this experiment we used the SVM surrogate for different dataset subsets from Klein et al. (2017a). The surrogate consists of a random forest trained on a grid of hyperparameter configurations of a SVM evaluated on different subsets of the training data. In particular, we defined the following subsets: $\{1/512, 1/256, 1/128, 1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1\}$ as tasks and sampled 100 configurations per task to train our meta-model. Note that we only provide the observed targets and not the subset size to our model. Figure 8.3 (right) shows the latent space of the trained meta-model: the latent representation of the model captures that similar data subsets are also close in the latent space. In particular, the first latent dimension h_0 coherently captures the sample size, which is learned using exclusively the correlation between the datasets and with no further information about their size.

8.4.2. Benchmarking with Profet

Comparing HPO methods using a small number of instances affects our ability to properly perform statistical tests. To illustrate this we consider a distribution of tasks that are variations of the Forrester function $f(x) = ((\alpha x - 2)^2)\sin(\beta x - 4)$ for parameters α and β. We generated 1000 tasks by uniformly sampling random α and β in $[0, 1]$ and compared six HPO methods: RS, DE, TPE, SMAC, BOHAMIANN and BO-GP (we left CMA-ES out because the python version does not support 1-dimensional optimization problems).

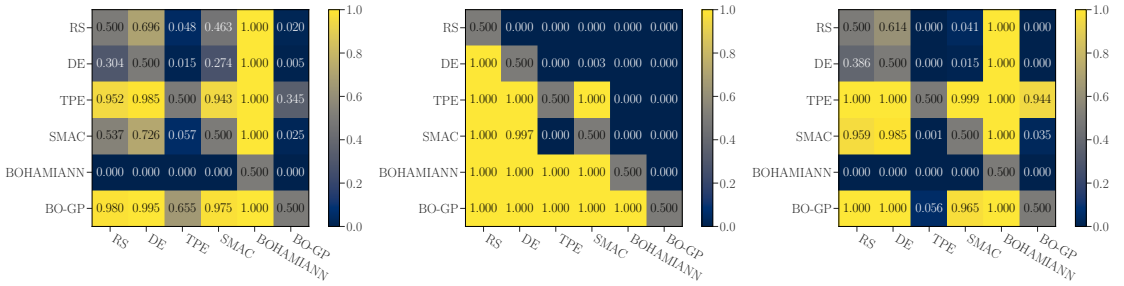


Figure 8.4.: Heatmaps of the p-values of the pairwise comparisons across the methods in the three scenarios using a Mann-Whitney U test. Small p-values should be interpreted as the test finding evidence that the method in the column improves the method in the row. Using tasks from our meta-model instead lead to results that are very close to using the large set of original tasks from the original distribution. *Left*: results with 1000 real tasks. *Middle*: subset of only 9 reals tasks. *Right:* results with 1000 tasks generated from our meta-model.

Figure 8.4 (left) shows the p-values of all pairwise comparisons with the null hypothesis 'Method$_{column}$ achieves a higher error after 50 function evaluations averaged over 20 runs than Method$_{row}$' for the Mann-Whitney U test. Squares in the figure with a p-value smaller than 0.05 are comparisons in which with a 95% confidence we have evidence to show that the method in the column is better that the method in the row (we have evidence to reject the null hypothesis). To reproduce a realistic setting where one has access to only a small set of tasks, we picked 9 out of the 1000 tasks randomly. Now, in order to acquire a comparable number of samples to perform a statistical test, we performed 2220 runs of each method on every task, and then computed the average of groups of 20 runs, such that we obtained 999 samples per method to compute the statistical test. One can see in Figure 8.4 (middle), that although the results are statistically significant, they are misleading: for example, BOHAMIANN is dominating all other methods (except BO-GP), whereas it is significantly worse than all other methods if we consider all 1000 tasks.

To solve this issue and obtain more information from the same limited number of a subset of 9 tasks, we use PROFET. We first train the meta-model on the same 9 selected tasks and then use it to generate 1000 new surrogate tasks (see Appendix E.3 for a visualization). Next, we use these tasks to run the comparison of the HPO methods. Results are shown in Figure 8.4 (right). The heatmap of statistical comparisons reaches very similar conclusions to those obtained with the original 1000 tasks, contrary to what happened when we did the comparisons with 9 tasks only (i. e. p-values are closer to the original ones). We conclude that using samples from the meta-model (generated based on a subset of tasks) allows us to draw conclusion that are more in line with experiments on the full dataset of tasks than running directly on the subset of tasks.

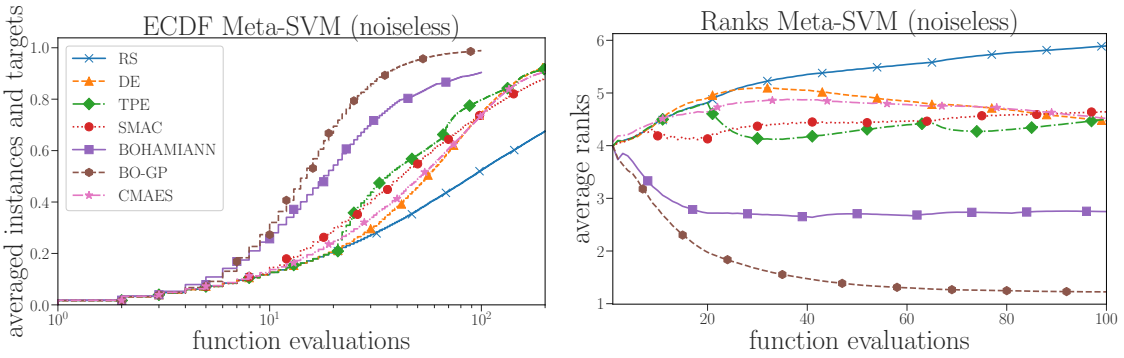


Figure 8.5.: Comparison of various HPO methods on 1000 tasks of the noiseless SVM benchmark. See Appendix D for the results on all benchmark problems. *Left:* the ECDF for the runtime. *Right*: the ranking of each method averaged across all tasks.

8.4.3. Comparing State-of-the-art HPO Methods

We conducted 20 independent runs for each method on each task of all three problem classes described in Section 8.3.1 with a different random seed. Each method had a budget of 200 function evaluations per task, except for BO-GP and BOHAMIANN, where, due to their computational overhead, we were only able to perform 100 function evaluations. Note that conducting this kind of comparison on the original benchmarks would have been prohibitively expensive. In Figure 8.5 we show the ECDF curves and the average ranking for the noiseless version of the SVM benchmark. The results for all other benchmarks are shown in Appendix E.5. We can make the following observations:

- Given enough budget, all methods are able to outperform RS. BO approaches can exploit their internal model such that they start to outperform RS earlier than evolutionary algorithms (DE, CMA-ES). Thereby, more sophisticated models, such as Gaussian processes or Bayesian neural networks are more *sample efficient* than somewhat simpler methods, e. g. random forests or kernel density estimators.
- The performance of BO methods that model the objective function (BO-GP, BOHAMIANN, SMAC) instead of just the distribution of the input space (TPE) decays if we evaluate the function through noise. Also evolutionary algorithms struggle with noise.
- Standard BO (BO-GP) works superior on these benchmarks but its performance decays rapidly with the number of dimensions.
- Runner-up is BOHAMIANN which works slightly worse than BO-GP but seems to suffer less under noisy function values. Note that this result can only be achieved by using Profet as we could not have evaluated with and without noise on the original datasets.

- Given a sufficient budget, DE starts to outperform CMA-ES as well as BO with simpler (and cheaper) models of the objective function (SMAC, TPE), making it a competitive baseline particularly for higher dimensional benchmarks.

8.5. Chapter Conclusions and Future Work

We presented PROFET, a new tool for benchmarking HPO algorithms. The key idea is to use a generative meta-model, trained on offline generated data, to produce new tasks, possibly perturbed by noise. The new tasks retain the properties of the original one but can be evaluated inexpensively, which represents a major advance to speed up comparisons of HPO methods. In a battery of experiments we have illustrated the representation power of PROFET and its utility when comparing HPO methods in families of problems where only a few tasks are available. While in this work we have focused on HPO methods, the same idea can be generalized to other optimization problems.

Besides these strong benefits, there are certain drawbacks of our proposed method that we would like to explicitly mention. First, since we encode new tasks based on a machine learning model, our approach is based on the assumptions that come with this surrogate model. Second, while we show in Section 8.4 empirical evidence that conclusions based on our proposed method are virtually identical to the one based on the original tasks, there are no theoretical guarantees that results translate one-to-one to the original benchmarks. Nevertheless, we believe that PROFET sets the ground for further research in this direction to provide much more realistic use-cases than commonly used synthetic functions, e. g. Branin, such that future work on HPO can rapidly perform reliable experiments during development and only execute the final evaluation on expensive real benchmarks. Ultimately, we think this is an important step towards more reproducibility, which is paramount in such an empirical-driven field as AutoML.

A possible extension of PROFET would be to consider multi-fidelity benchmarks (Klein et al., 2017a; Kandasamy et al., 2017; Klein et al., 2017c) where cheap, but approximate fidelities of the objective function are available, e. g. learning curves or dataset subsets. Furthermore, since PROFET also provides gradient information it could serve as a training distribution for learning-to-learn approaches (Chen et al., 2017; Volpp et al., 2019).

9. Conclusions

In this thesis we presented several advancements for Bayesian hyperparameter optimization in the Chapters 3, 4, 5, 6 and 8). Additionally, we provided as background material an overview on hyperparameter optimization in Chapter 2.1 and Bayesian optimization in Chapter 2.2. In the next Section 9.1 we summarize and discuss the central findings of this thesis. Afterwards we provide in Section 9.2 an outlook into potential future work.

9.1. Summary and Discussion

We first looked in Chapter 3 at Bayesian Neural Networks as a probabilistic model for Bayesian optimization . Important requirements for a model in any active learning setting, such as Bayesian optimization, are robustness in terms of the model's own hyperparameters as well as reliable uncertainty estimates. For that we proposed an adaptive scaling version of stochastic gradient Hamiltonian Monte-Carlo that automatically adapts the preconditioning of the gradients during the burn-in phase of sampling. Furthermore, we presented a flexible neural network architecture for single and multi-task optimization problems which can be applied to continuous and discrete configuration spaces. The resulting Bayesian optimization method BOHamiANN performed on-par with Gaussian process based Bayesian optimization methods on continuous hyperparameter optimization problems and better than random forest based methods on discrete problems.

In Chapters 4, 5 and 6 we looked at different ways to exploit fidelities of the objective function to speed up the optimization process. First, we presented the Gaussian process based method Fabolas in Chapter 4 which showed superior performance to standard Bayesian optimization on low-dimensional continuous spaces where our fidelity of the objective function represents the training dataset size. While Fabolas could be potentially also used for other fidelities, one would need to define suitable basis functions that describe the change in performance and cost with respect to this fidelity. Besides that, it suffers under the same problem as the most other Gaussian process based methods, i. e. scaling with the number of datapoints and the number of dimensions.

Inspired by the strong performance of Bayesian neural networks observed in Chapter 3, we developed LC-Net (Chapter 5), a neural network architecture that, given a hyperparameter configuration, predicts the parameters of a set of basis functions

to describe learning curves. LC-Net showed strong performance in predicting the final performance of hyperparameter configurations when only the first few points of the learning curve have been observed. However, similar to Fabolas, LC-Net makes parametric assumptions about the fidelity and hence can not necessarily be applied to new fidelities out-of-the-box.

The third multi-fidelity methods we proposed is BOHB (Chapter 6) which is a combination of Hyperband and Bayesian optimization. It uses successive halving to reason across fidelities and hence it is applicable to any fidelity setting as long as smaller budgets are sufficiently representative for the next higher budget. BOHB keeps the strong performance of Hyperband at the beginning of the optimization process but converges to the usually better final performance of Bayesian optimization, instead, like Hyperband, falling back to random search. Importantly, BOHB can efficiently use parallel resources and thereby achieves an almost linear speed up with the number of workers, which is not trivial with, for example, Gaussian process based methods.

To make benchmarking of hyperparameter optimization methods more efficient, we present tabular benchmarks in Chapter 7 that mimic the discrete hyperparameter optimization of feed forward neural networks but are much cheaper since single function evaluation consist only in table lookups. At last, we proposed *profet* a generative meta-model for hyperparameter optimization tasks in Chapter 8. Profet allows to generate an arbitrary amount of functions from the same underlying distribution. These surrogate models resemble common hyperparameter optimization problems but take only milliseconds to evaluate. Based on Profet we conducted an exhaustive empirical evaluation of several state-of-the-art blackbox optimization methods.

9.2. Future Work

In this thesis we established new ways to make Bayesian optimization more efficient for automated hyperparameter optimization. But, of course, many questions remain unsolved and we hope that future work will address them to push the current state-of-the-art even further. In the following, we will outline a few potential ideas which we think might be worth to be explored.

Multi-fidelity Bayesian optimization:

- In this thesis we mostly focused on learning curves or datasets subsets, but of course many other potential fidelities exist. For example, on image classification benchmarks such as Imagenet one could treat the image resolutions as fidelity, since smaller resolutions lead to less units in the neural networks and are hence faster to train. Another potential fidelity could be the neural network size itself.

For instance, current neural architecture search methods, such as DARTS (Liu et al., 2019), conduct the actual search on neural networks with a reduced amount of units and increase them for the final evaluation. While in the case of DARTS these are manually specified, future work could cast this as fidelity and try to automatically increase the network size if more information about the error landscape is obtained.

- As described above for the most machine learning algorithms usually multiple fidelities are available. Potential future work could try to exploit multiple fidelities at the same time to gain further improvements. However, care needs to be taken with the modelling which we assume is not trivial and might come with an increased amount of noise of the objective function
- As we have seen in Chapter 3 warm-starting the optimization procedure can lead to faster improvements. Interesting future work could combine this multi-task setting with a multi-fidelity setting. Also here, correct modelling is key and might be challenging due to the complexity of the problem.

Surrogate Benchmarking

- A natural next step for meta-surrogate benchmarking of hyperparameter optimization described in Chapter 8 is to extend it to the multi-fidelity setting. A straightforward way would be to use the machinery presented in Chapter 4 and Chapter 5 to obtain a robust meta-model.
- Recently, Chen et al. (2017) proposed a meta-learning approach to learn new blackbox optimization methods. These learning-to-learn methods require a large amount of tasks as training distribution. Profet might be particularly appealing for these kind of methods since it not only provides an arbitrary amount of tasks, every task is encoded as a neural network and hence also allows to compute gradients which is essential for learning-to-learn methods that cast the search for new optimizers as a supervised learning problem.

General Hyperparameter Optimization

- Recently, a new subfield in AutoML emerged, called neural architecture search that tries to automatically design neural architectures (Elsken et al., 2018). So far, work in this field concentrates only on choices of the architecture and keeps other hyperparameters that control the training, such as learning rates or regularization parameters fix (Zoph and Le, 2017; Real et al., 2017). Obviously, both, architecture and hyperparameters, are strongly connected and future work could try to optimize them jointly. Also, some work in neural-architecture search (Liu et al., 2019; Pham et al., 2018) uses a shared set of weights between architectures which drastically improves the computational efficiency. While it might not be trivial, due to the non-stationary function values, future work in

hyperparameter optimization could also try to exploit shared weights to gain additional speed ups.

- A current limitation of hyperparameter optimization is that during a function evaluation hyperparameters are fixed. However, certain hyperparameters, such as for instance the learning rate might need to be adapted during training. Recent work by Jaderberg et al. (2017) maintains a population of configurations and mutates poorly performing configurations to improve performance. Those mutations are performed randomly and, with the same argument for Bayesian optimization, potential future work could use a probabilistic model to guide the search more informatively.

A. Supplementary Material to Chapter 3

A.1. Synthetic Functions

Figure A.1 shows the comparison between Bayesian optimization with Gaussian processes (BO-GP), random forests (BO-RF), random search, DNGO (Snoek et al., 2015) and our proposed methods Bohamiann on 8 different synthetic functions from the literature. For Bohamiann and DNGO we used two different feed forward neural networks architectures with 3 layers and either 50 or 10 units in each layer (denoted in parentheses). All Bayesian optimization methods used expected improvement as acquisition function and we used differential evolution (Storn and Price, 1997) to optimize it. We sampled 2 points uniformly at random as initial design before we started the Bayesian optimization process. Every method had for each run on each benchmark a total budget of 200 function evaluations and we report the mean and the standard error of the mean over 50 independent runs of each method. See main text for an analysis of the results.

A.2. XGBoost

We optimized the 8 hyperparameters (denoted in Table A.1) of XGBoost on 10 different UCI regression datasets. In Figure A.2 we show the results for random search (RS), Bayesian optimization with Gaussian processes (BO-GP), Bohamiann, Bohamiann with warmstarting from 50 points of another datasets (MT-Bohamainn) and Multi-task Bayesian optimization (MTBO) (Swersky et al., 2013), which warm-starts from the same points, for all datasets. Each method had for every datasets a budget of 30 function evaluation and we report the mean and standard error of the mean across 30 runs of each method. See main text for an discussion of the results.

Name	Range	log scale
learning rate	$[10^{-6}, 10^{-1}]$	✓
gamma	$[0, 2]$	-
L1 regularization	$[10^{-5}, 10^{3}]$	✓
L2 regularization	$[10^{-5}, 10^{3}]$	✓
number of estimators	$[10, 500]$	-
subsampling	$[0.1, 1]$	-
max. depth	$[1, 15]$	-
min. child weight	$[0, 20]$	-

Table A.1.: Hyper-parameter configuration space of the gradient tree boosting (XGBoost) benchmark.

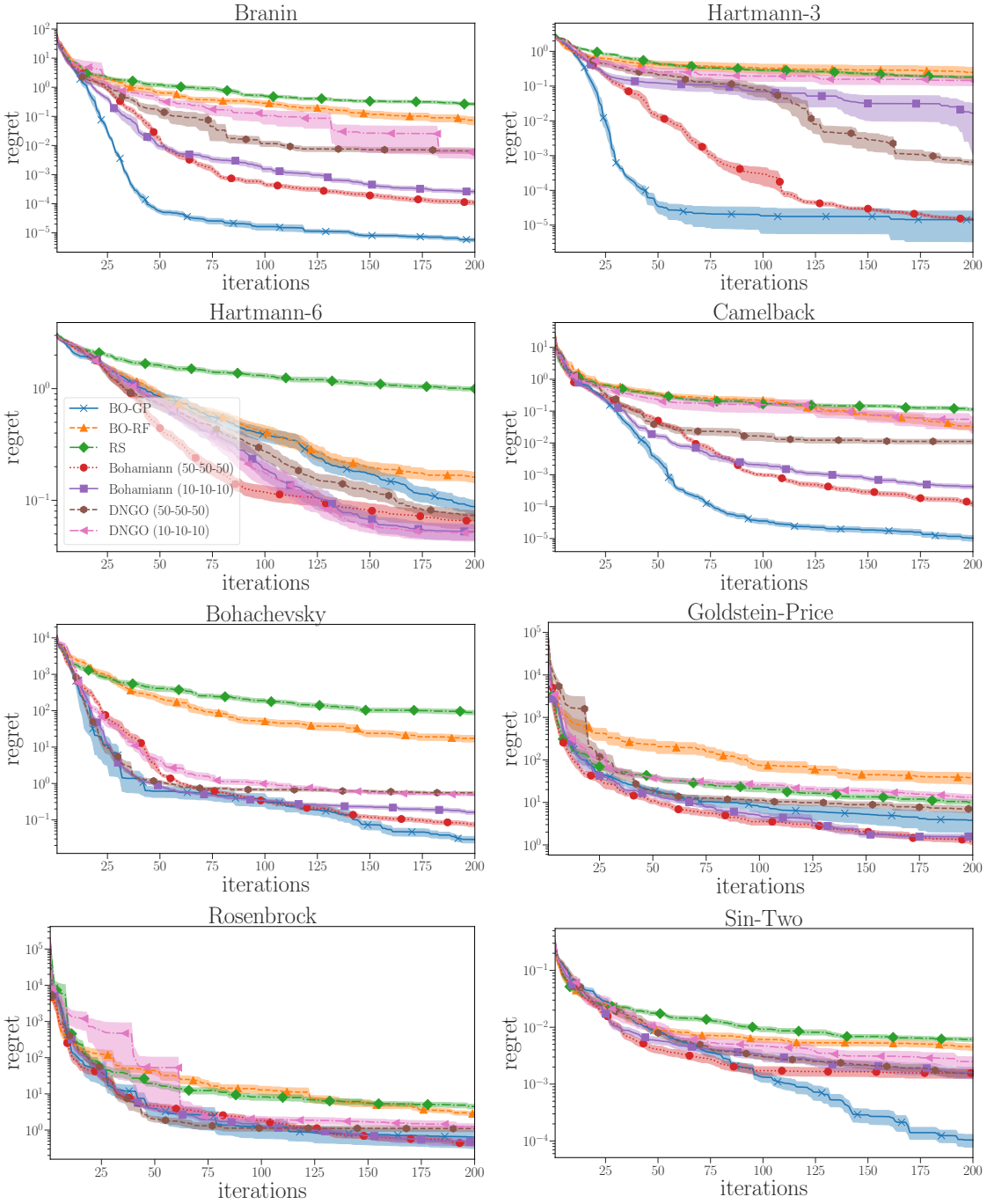


Figure A.1.: Comparison of Bayesian optimization with different models: Gaussian processes (BO-GP), random forests (BO-RF), Bohamiann, DNGO and random search(RS). We used two different architectures for Bohamiann and DNGO (3 layers 50 units and 3 layers 10 units)

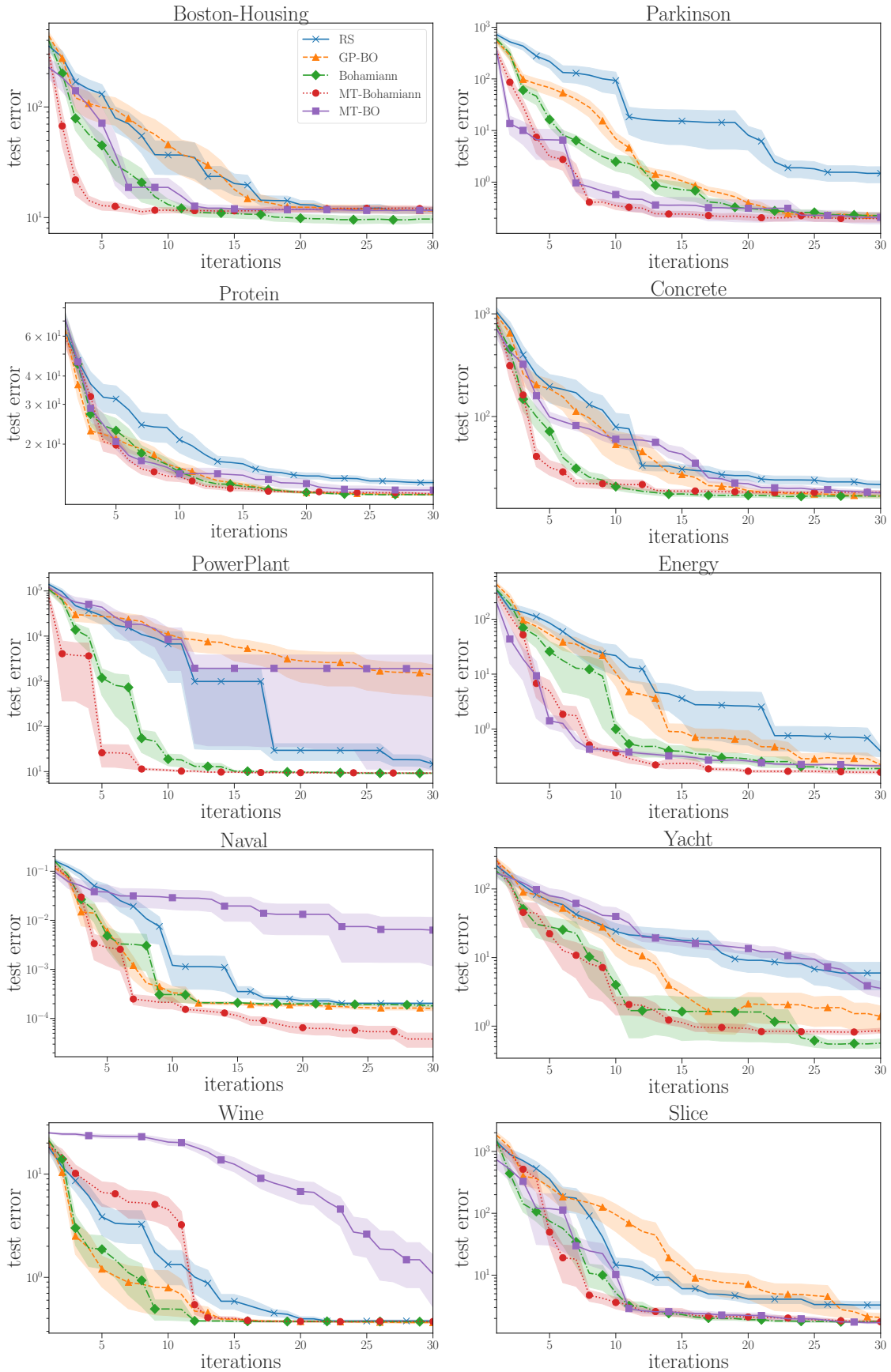


Figure A.2.: Comparison of random search (RS), Gaussian process based Bayesian optimization (BO-GP), Bohamiann, multi-task Bayesian optimization (MT-BO) and multi-task Bohamiann (MT-Bohamiann) where both started from the same 50 random points of a randomly chosen other dataset.

B. Supplementary Material to Chapter 5

B.1. Experimental Setup – Details

Table B.1 shows the hyperparameters of our 4 benchmarks described in Section 5.4.1 and their ranges.

B.2. Description of the Basis Functions

To reduce complexity, we used a subset of the basis function from Domhan et al. (2015) which we found to be sufficient for learning curve prediction. We adapted these functions to model the residual between the asymptotic value y_∞ and we scaled the parameters Θ to be in $[0, 1]$. Table B.2 shows the exact equations we used.

B.3. Dataset Characteristics

Figure B.1 shows the distributions over runtimes for all random configurations of different benchmarks. As it can be seen there is a high variance of the runtime between different configurations for all benchmarks and some configurations need order of magnitudes longer than others.

Figure B.2 shows the empirical cumulative distribution of the asymptotic performance of random configurations. These plots give an intuition about the difficulty of a benchmark as they show how many random configurations one has to sample in order to achieve a good performance.

B.4. Optimization on Tabular Benchmarks

We used the four tabular benchmarks described in Chapter 7 to compared Hyperband against Hyperband augmented with our model. Hyperparameter configurations from our model are selected based on Thompson sampling, were we interpret as single weight vector as a function sampled from our probabilistic model over the objective

	Name	Range	log scale
CNN	batch size	$[32, 512]$	-
	number of units layer 1	$[2^4, 2^{10}]$	✓
	number of units layer 2	$[2^4, 2^{10}]$	✓
	number of units layer 3	$[2^4, 2^{10}]$	✓
	learning rate	$[10^{-6}, 10^{-0}]$	✓
FCNet	initial learning rate	$[10^{-6}, 10^{0}]$	✓
	L_2 regularization	$[10^{-8}, 10^{-1}]$	✓
	batch size	$[32, 512]$	-
	γ	$[-3, -1]$	-
	κ	$[0, 1]$	-
	momentum	$[0.3, 0.999]$	-
	number units 1	$[2^5, 12^2]$	✓
	number units 2	$[2^5, 2^{12}]$	✓
	dropout rate layer 1	$[0.0, 0.99]$	-
	dropout rate layer 2	$[0.0, 0.99]$	-
LR	learning rate	$[10^{-6}, 10^{0}]$	✓
	L_2	$[0.0, 1.0]$	-
	batch size	$[20, 2000]$	-
	dropout rate on inputs	$[0.0, 0.75]$	-
VAE	L	$[1, 3]$	-
	number of hidden units	$[32, 2048]$	-
	batch size	$[16, 512]$	-
	z dimension	$[2, 200]$	-

Table B.1.: Hyperparameter configuration space of the four different iterative methods. For the FCNet we decayed the learning rate by a $\alpha_{decay} = (1 + \gamma * t)^{-\kappa}$ and also sampled different values for γ and κ.

function. We used an simple stochastic local search method, which starting from a random configuration, evaluates the one step neighborhood and jumps to neighbor with the highest acquisition value until it either converged or hit a maximum number of 10 steps. After each round of successive halving, we return the current best observed configuration and its asymptotic performance. As additional baseline we use Bayesian optimization with Bayesian neural networks as probabilistic model and expected improvement as acquisition function as described in Chapter 3. For each method we performed 500 independent runs and report the mean and the standard error of the mean across all these runs. Figure B.3 shows that our specialized neural network architecture helps to speed up the convergence of Hyperband and often finds a better final performance. The only exception if the Parkinson Telemonitoring benchmark, where it helps to find better configurations faster than Hyperband, but

Name	**Formula**
vapor pressure	$\phi(t,a,b,c) = exp\left[-a - {}^{b}/_{10} \cdot t^{-1} - {}^{c}/_{10} \cdot \log(t)\right] - exp\left[a + {}^{b}/_{10}\right]$
log function	$\phi(t,a,b,c) = 10 \cdot c + {}^{a}/_{2} \log(b \cdot t)$
hill-3	$\phi(t,a,b,c) = a \cdot \left[({}^{c}/_{100 \cdot t})^{b} + 1\right]^{-1}$

Table B.2.: The formulas of our 5 basis functions. We assume that for all basis functions $a, b, c \in [0, 1]$.

eventually converges to a worse final performance.

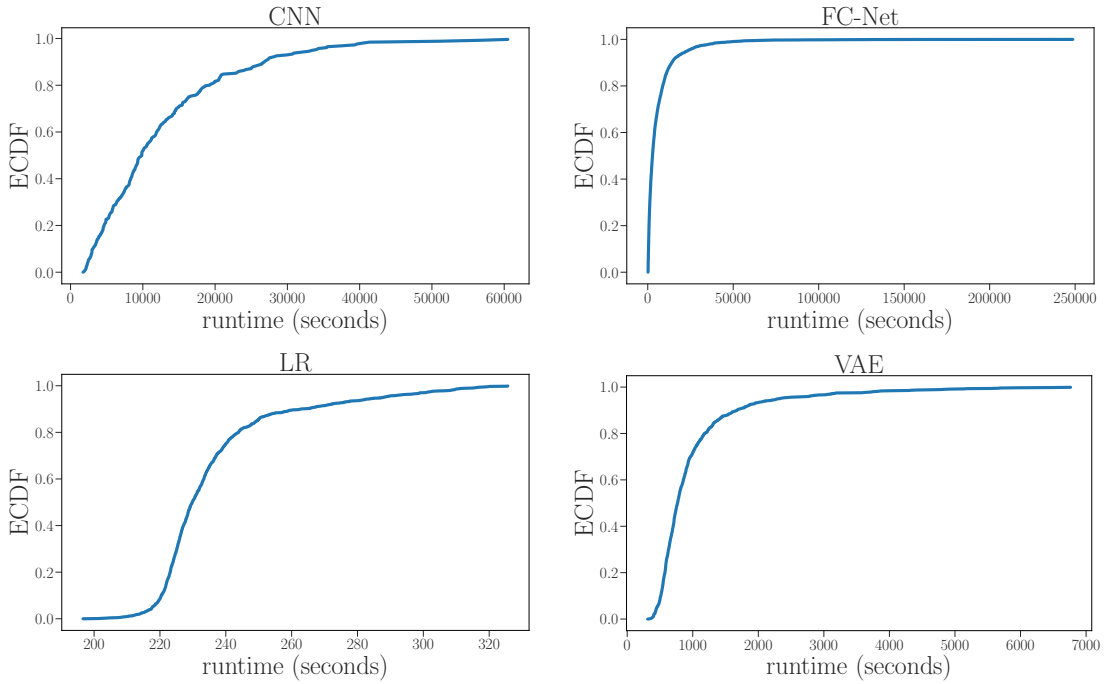


Figure B.1.: Distributions over runtimes for different random configurations for the CNN, FCNet, LR and VAE benchmark described in Section 5.4.1. One can see that all distributions are long tailed and that especially on the FCNet benchmark some configuration need order of magnitudes longer than others.

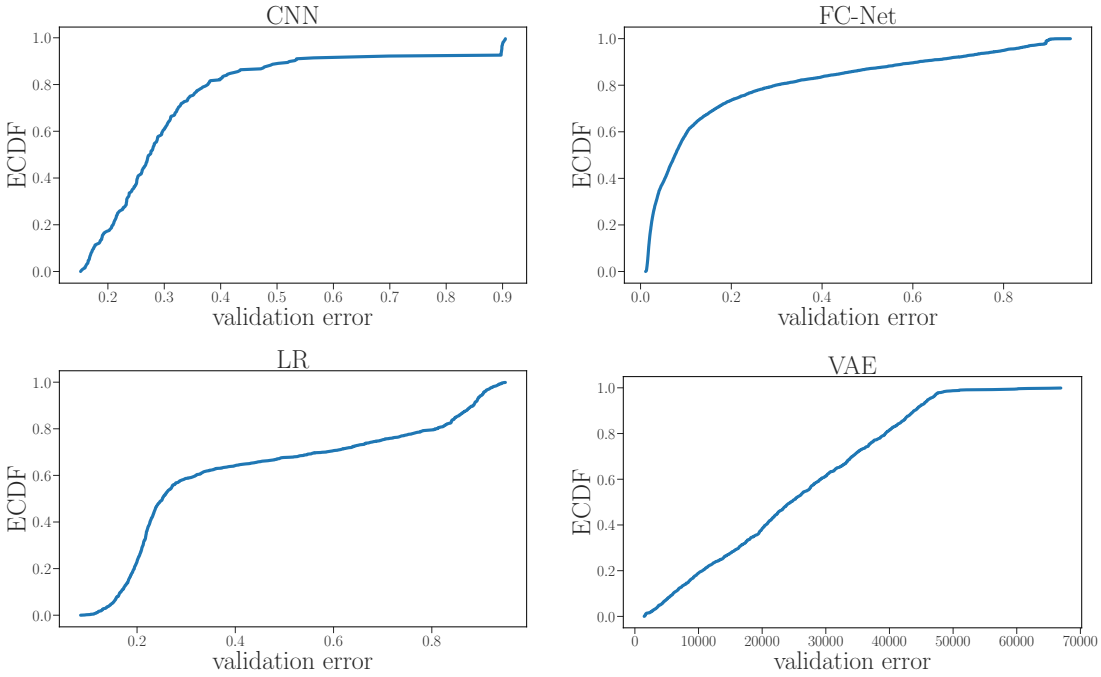


Figure B.2.: Empirical cumulative distributions for the CNN, FCNet, LR and VAE benchmark.

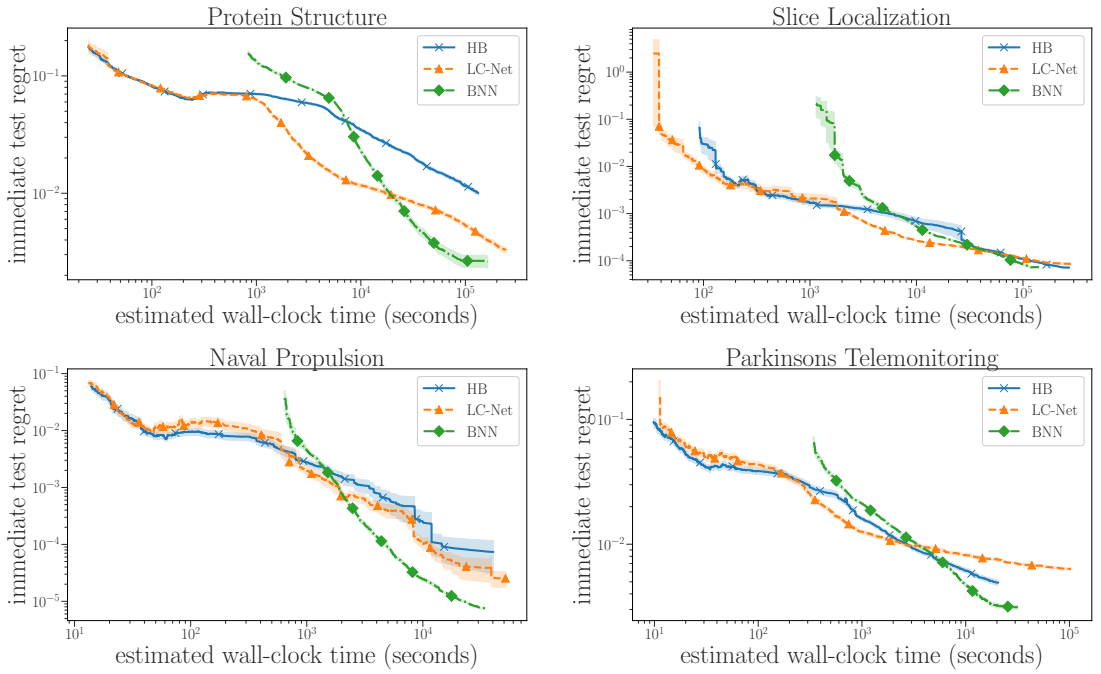


Figure B.3.: Different optimizers on the tabular benchmarks described in Chapter 7. Note how sampling from the model improves Hyperband's performance by converging to the optimum faster.

C. Supplementary Material to Chapter 6

C.1. Comparison to Other Work on Bayesian Optimization and Hyperband

Here we discuss the differences between our method and the related approaches of Bertrand et al. (2017) and Wang et al. (2018) in more detail. We note that these works are independent and concurrent; our work extends our preliminary short workshop papers at NIPS 2017 (Falkner et al., 2017) and ICLR 2018 (Falkner et al., 2018b).

While the general idea of combining Hyperband and Bayesian optimization by Bertrand et al. (2017) is the same as in our work, they use a Gaussian process for modeling the performance. The budget is modeled like any other dimension of the search space, without any special treatment. Based on our experience with Fabolas (Klein et al., 2017a), we expect that the squared exponential kernel might not extrapolate well, which would hinder performance. Also, the small evaluation provided by Bertrand et al. (2017) does not allow strong conclusions about the performance of their method.

Wang et al. (2018) also independently combined TPE and Hyperband, but in a slightly different way than we did. In their method, TPE is used as a subroutine in every iteration of Hyperband. In particular, a new model is built from scratch at the beginning of every SuccessiveHalving run (Algorithm 3, line 8 in Wang et al. (2018)). This means that in later iterations of the algorithm, the model does not benefit from any of the evaluations in previous iterations. In contrast, BOHB collects all evaluations on all budgets and uses the largest budget with enough evaluations (admittedly a heuristic, but we would argue a reasonable one) as a base for future evaluations. This way, BOHB aggregates more knowledge into its models for the different budgets as the optimization progresses. We believe this to be a crucial part of the strong performance of our method. Empirically, Wang et al. (2018) did not achieve the consistent and large speedups across a wide range of applications BOHB achieved in our experiments.

C.2. Successive Halving

SuccessiveHalving is a simple heuristic to allocate more resources to promising candidates. For completeness, we provide pseudo code for it in Algorithm 8. It is initialized with a set of configurations, a minimum and maximum budget, and a scaling parameter η. In the first *stage* all configurations are evaluated on the smallest budget (line 3). The losses are then sorted and only the best $1/\eta$ configurations are kept in the set C (line 4). For the following stage, the budget is increased by a factor of η (line 5). This is repeated until the maximum budget for a single configuration is reached (line 2). Within Hyperband, the budgets are chosen such that all SuccessiveHalving executions require a similar total budget.

Algorithm 8 Pseudocode for SuccessiveHalving used by Hyperband as a subroutine.

Require: initial budget b_0, maximum budget b_{max}, set of n configurations $C = \{c_1, c_2, \dots c_n\}$

1: $b = b_0$
2: **while** $b \leq b_{max}$ **do**
3: $\quad L = \{\tilde{f}(c, b) : c \in C\}$
4: $\quad C = \text{top}_\text{k}(C, L, \lfloor |C|/\eta \rfloor)$
5: $\quad b = \eta \cdot b$
6: **end while**

C.3. Details on the Kernel Density Estimator

We used the MultivariateKDE from the statsmodels package Seabold and Perktold (2010), which constructs a factorized kernel, with a one-dimensional kernel for each dimension. Note that using this product of 1-d kernels differs from the original TPE, which uses a pdf that is the product of 1-d pdfs. Figure C.1 visualizes the differences for two small problems. For the continuous parameters a Gaussian kernel is used, whereas the Aitchison-Aitken kernel is the default for categorical parameters. We used Scott's rule for efficient bandwidth estimation, as preliminary experiments with maximum-likelihood based bandwidth selection did not yield better performance but caused a significant overhead.

C.4. Performance of All Methods on All Surrogates

Figure C.2 shows the performance of all methods we evaluated on all our surrogate benchmarks. Random search is clearly the worst optimizer across all datasets when the budget is large enough for GP-BO and TPE to leverage their model. Hyperband and the two methods based on it (HB-LCNet) and BOHB improve much more

quickly due to the smaller budgets used. On all surrogate benchmarks, BOHB starts to outperform HB after the first couple of iterations (sometimes even earlier, e.g., on dataset letter). The same dataset also shows that traditional BO methods can still have an advantage for very large budgets, since in these late stages of the optimization process the low fidelity evaluations of BOHB can cause a constant overhead without any gain.

C.5. Performance of Parallel Runs

Figure C.3 shows the performance of BOHB when run in parallel on all our surrogate benchmarks. The speed-ups are quite consistent, and almost linear for a small number of workers (2-8). For more workers, more random configurations are evaluated in parallel before the first model is built, which degrades performance. But even for 32 workers, linear speedups are possible (see, e.g., dataset letter, for reaching a regret of 2×10^{-3}).

We note that in order to carry out this evaluation of parallel performance, we actually simulated the parallel optimization by making each worker wait for the given budget before returning the corresponding performance value of our surrogate benchmark. (The case of one worker is an exception, where we can simply reconstruct the trajectory because all configurations are evaluated serially.) By using this approach in connection with threads, each evaluation of a parallel algorithm still only used 1 CPU, but the run actually ran in real time. For this reason, we decided to not evaluate all possible numbers of workers for dataset poker, for which each run with less than 16 workers would have taken more than a day, and we do not expect any different behavior compared to the other datasets.

C.6. Evaluating the Hyperparameters of BOHB

In this section, we evaluate the importance of the individual hyperparameters of BOHB, namely the number of samples used to optimize the acquisition function (Figure C.4), the fraction of purely random configuration ρ (Figure C.5), the scaling parameter η (Figure C.6), and the bandwidth factor used to encourage exploration (Figure C.7).

Additionally, we want to discuss the importance of η, b_{min} and b_{max} already present in HB. The parameter η controls how aggressively SH cuts down the budget and the number of configurations evaluated. Like HB Li et al. (2017), BOHB is also quite insensitive to this choice in a reasonable range. For our experiments, we use the same default value ($\eta = 3$) for HB and BOHB.

More important for the optimization are b_{min} and b_{max}, which are problem specific and inputs to both HB and BOHB. While the maximum budget is often naturally

defined, or is constrained by compute resources, the situation for the minimum budget is often different. To get substantial speedups, an evaluation with a budget of b_{min} should contain some information about the quality of a configuration with larger budgets; for example, when subsampling the data, the smallest subset should not be one datum, but rather enough points to fit a meaningful model. This requires knowledge about the benchmark and the algorithm being optimized.

C.7. Stochastic Counting Ones

We now formally define the stochastic variant of the counting ones function we use and present the results for different dimensions. The objective function to be minimized can be written as

$$f(\boldsymbol{x}) = -\left(\sum_{x \in X_{cat}} x \quad + \sum_{x \in X_{cont}} \mathbb{E}_b[(B_{p=x})]\right), \tag{C.1}$$

where B_p is the Bernoulli distribution with parameter p, and $\mathbb{E}_b$ denotes the expectation estimated using b independent draws from the distribution.

The problem consists of a deterministic discrete part (the standard counting ones problem), and a stochastic component whose noise is controlled by the budget b. To keep the noise consistent across different dimensions, we chose the budgets such that the total number of samples used remains constant. Specifically, we picked $b_{min} = 576/d$ and $b_{max} = 93312/d$ where $d = N_{cat} + N_{cont}$. For $N_{cat} = N_{cont} = 4$ this results in a minimum of 144 and a maximum of 11664 samples for each Bernoulli distribution. BOHB and HB evaluated between these budgets, where as TPE and SMAC operated always with the maximum budget.

This function has some noteworthy properties:

1. By design, we know the best value, $-d$, and worst value, 0. For easier comparison between different numbers of dimensions, we plot the normalized regret, which in our case is $(f(\boldsymbol{x}) + d)/d$.
2. The optimum $\boldsymbol{x} = [1]^d$ is at the boundary of the search space. This could be problematic for both the random forests in SMAC, and the KDEs in TPE and BOHB.

Figure C.8 shows the mean performance of all applicable methods in $d = 8$, 16, 32 and 64 dimensions for a budget of 4000 full function evaluations. The median performances are shown in Figure C.9.

We draw the following conclusions from the results:

1. Despite its simple definition, this problem is quite challenging for the methods we applied to it. RS and HB both suffer from the fact that drawing configurations at random performs quite poorly in this space. The model-based

approaches SMAC and TPE performed substantially better, especially with large budgets. We would like to mention that SMAC and TPE treated the problem as a blackbox optimization problem; the results for SMAC could likely be improved further by treating individual samples as "instances" and using SMAC's intensification mechanism to reject poor configurations based on few samples and evaluate promising configurations with more samples.

2. BOHB struggles to converge for the eight dimensional example. The most probable reason is the number of samples required to build a model, which was set to 9 here. This led to a consistently slow convergence of BOHB. The median plots (Figure C.9) show that there is quite some variability in BOHB's performance for this dimensionality. We attribute this at least in part to the small number of samples used to build the initial model; combined with the poor performance of random configurations, this leads to unstable performance.
3. Given a large enough budget, BOHB's evaluations on small budgets lead to a constant overhead over only using the more reliable evaluations on larger budgets. This slows down convergence.
4. Since the optimization problem is perfectly separable (there are no interaction effects between any dimensions), we expect TPE's univariate KDE to perform better than BOHB's multivariate one, which might also explain the relatively slow convergence compared to TPE towards the end of the trajectories.

C.8. Feed Forward Network Surrogates

C.8.1. Constructing the Surrogates

To build a surrogate, we sampled 10 000 random configurations for each dataset, trained them for 50 epochs, and recorded their classification error after each epoch, along with their total training time. We fitted two independent random forests that predict these two quantities as a function of the hyperparameter configuration used. This enabled us to predict the classification error as a function of time with sufficient accuracy. As almost all networks converged within the 50 epochs, we extend the curves by the last obtained value if the budget would allow for more epochs.

The surrogates enable cheap benchmarking, allowing us to run each algorithm 256 times. Since evaluating a configuration with the random forest is inexpensive, we used a global optimizer (differential evolution) to find the true optimum. We allowed the optimizer 10 000 iterations which should be sufficient to find the true optimum.

Besides these positive aspects of benchmarking with surrogates, there are also some drawbacks that we want to mention explicitly:

- There is no guarantee that the surrogate actually reflects the important properties of the true benchmark.

Table C.1.: The hyperparameters and architecture choices for the fully connected networks.

Hyperparameter	Range	Log-transform
batch size	$[2^3, 2^8]$	yes
dropout rate	$[0, 0.5]$	no
initial learning rate	$[10^{-6}, 10^{-2}]$	yes
exponential decay factor	$[-0.185, 0]$	no
# hidden layers	$\{1, 2, 3, 4, 5\}$	no
# units per layer	$[2^4, 2^8]$	yes

- The presented results show the optimized classification error on the validation set used during training. There is no test performance that could indicate overfitting.
- Training with stochastic gradient descent is an inherently noisy process, i.e. two evaluations of the same configuration can result in different performances. This is not at all reflected by our surrogates, making them a potentially easier to optimize than the true benchmark they are based on.
- By fixing the budgets (see below) and having deterministic surrogates, the global minima might be the result of some small fluctuations in the classification error in the surrogates' training data. That means that the surrogate's minimizer might not be the true minimizer of the real benchmark.

None of these downsides necessarily have substantial implications for comparing different optimizers; they simply show that the surrogate benchmarks are not perfect models for the real benchmark they mimic. Nevertheless, we believe that, especially for development of novel algorithms, the positive aspects outweigh the negative ones.

C.8.2. Determining the Budgets

To choose the largest budget for training, we looked at the best configuration as predicted by the surrogate and its training time. We chose the closest power of 3 (because we also used $\eta = 3$ for HB and BOHB) to achieve that performance. We chose the smallest budget for HB such that most configurations had finished at least one epoch. Table C.2 lists the budgets used for all datasets.

C.9. Bayesian Neural Networks

We optimized the hyperparameters described in Table C.3 for a Bayesian neural network trained with SGHMC on two UCI regression datasets: Boston Housing and Protein Structure. The budget for this benchmark was the number of steps for the

Table C.2.: The budgets used by HB and BOHB; random search and TPE only used the last budget

Dataset	Budgets in seconds for HB and BOHB
Adult	9, 27, 81, 243
Higgs	9, 27, 81, 243
Letter	3, 9, 27, 81
Poker	81, 243, 729, 2187

Table C.3.: The hyperparameters for the Bayesian neural network task.

Hyperparameter	Range	Log-transform
# units layer 1	$[2^4, 2^9]$	yes
# units layer 2	$[2^4, 2^9]$	yes
step length	$[10^{-6}, 10^{-1}]$	yes
burn in	$[0, .8]$	no
momentum decay	$[0, 1]$	no

MCMC sampler. We set the minimum budget to 500 steps and the maximum budget to 10000 steps. After sampling 100 parameter vectors, we computed the log-likelihood on the validation dataset by averaging the predictive mean and variances of the individual models. The performance of all methods for both datasets is shown in Figure C.10.

C.10. Reinforcement Learning

Table C.4 shows the hyperparameters we optimized for the PPO Cartpole task.

Table C.4.: The hyperparameters for the PPO Cartpole task.

Hyperparameter	Range	Log-transform
# units layer 1	$[2^3, 2^7]$	yes
# units layer 2	$[2^3, 2^7]$	yes
batch size	$[2^3, 2^8]$	yes
learning rate	$[10^{-7}, 10^{-1}]$	yes
discount	$[0, 1]$	no
likelihood ratio clipping	$[0, 1]$	no
entropy regularization	$[0, 1]$	no

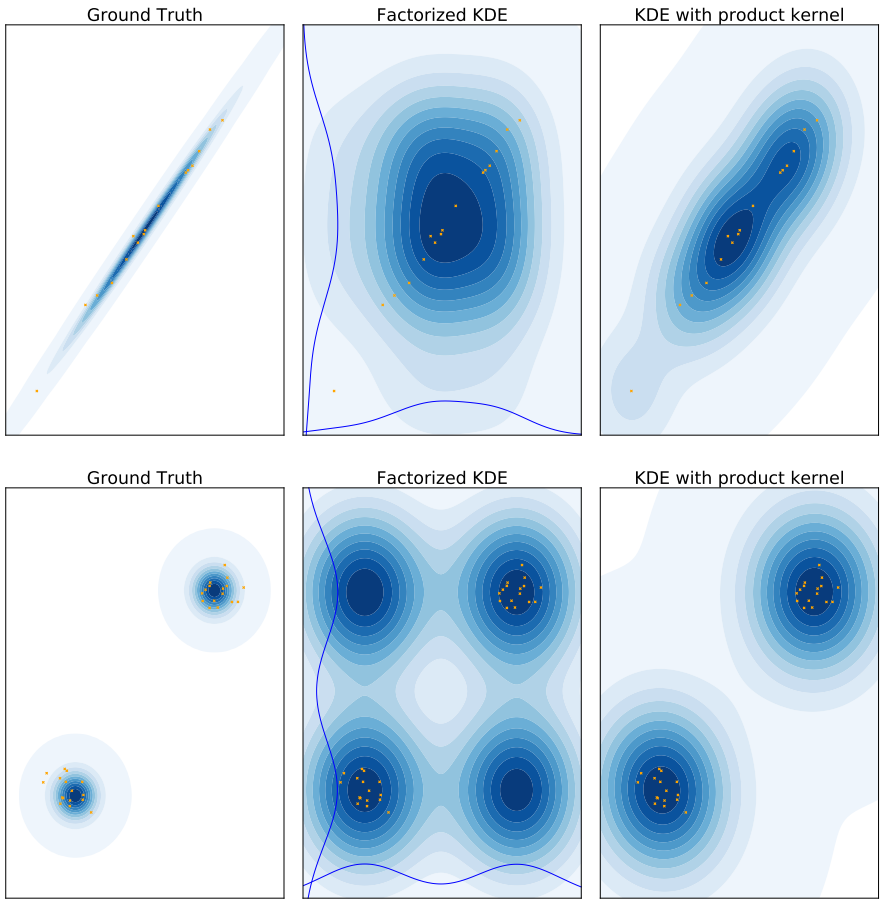


Figure C.1.: Visualization of the two different KDE approaches. The left column shows the true distribution (blue shaded area) from which 16 samples (orange crosses) were drawn. The middle column shows how a KDE that factorizes the PDF (as in TPE) models the density. The right column demonstrates how the KDE used in BOHB handles the data by factorizing the kernels instead of the PDF. The top row is a single two dimensional Gaussian probability with a strong correlation between the variables. The example in the bottom row is a mixture of two such Gaussians.

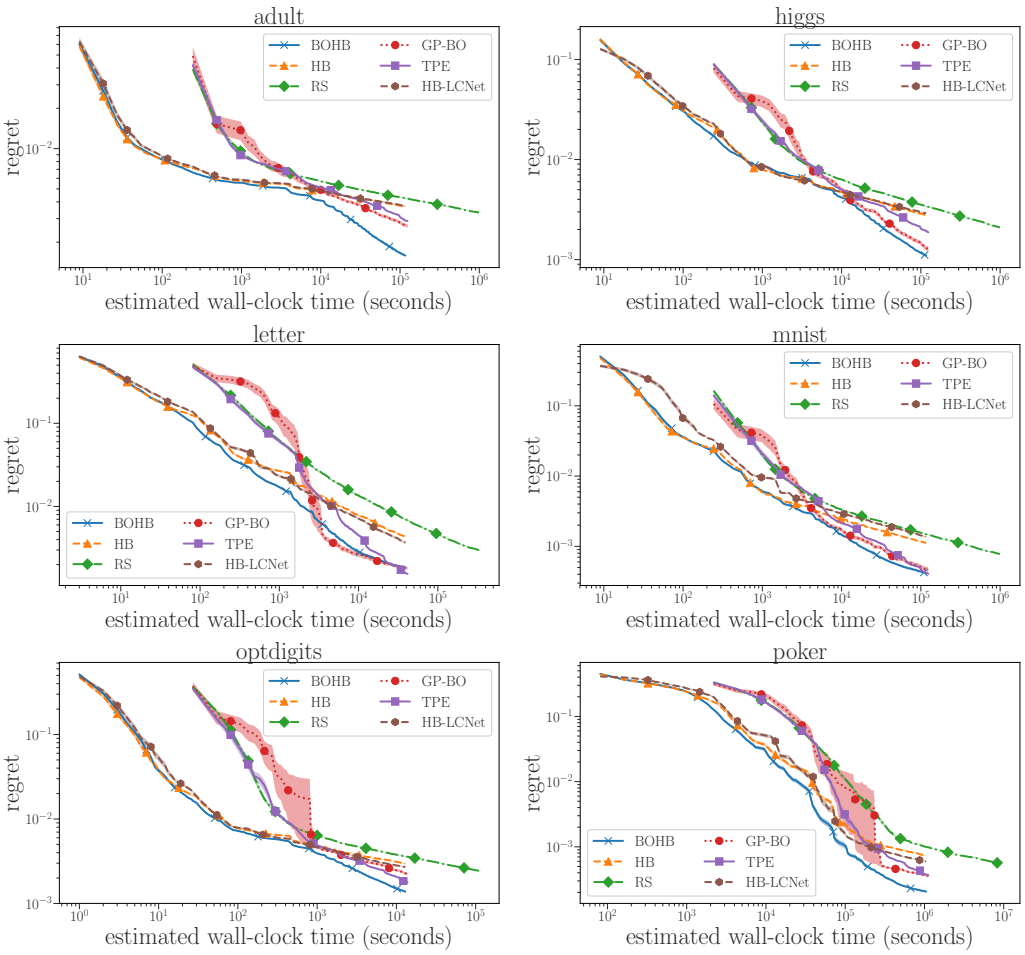


Figure C.2.: Mean performance on the surrogates for all six datasets. As uncertainties, we show the standard error of the mean based on 512 runs (except for GP-BO, which has only 50 runs).

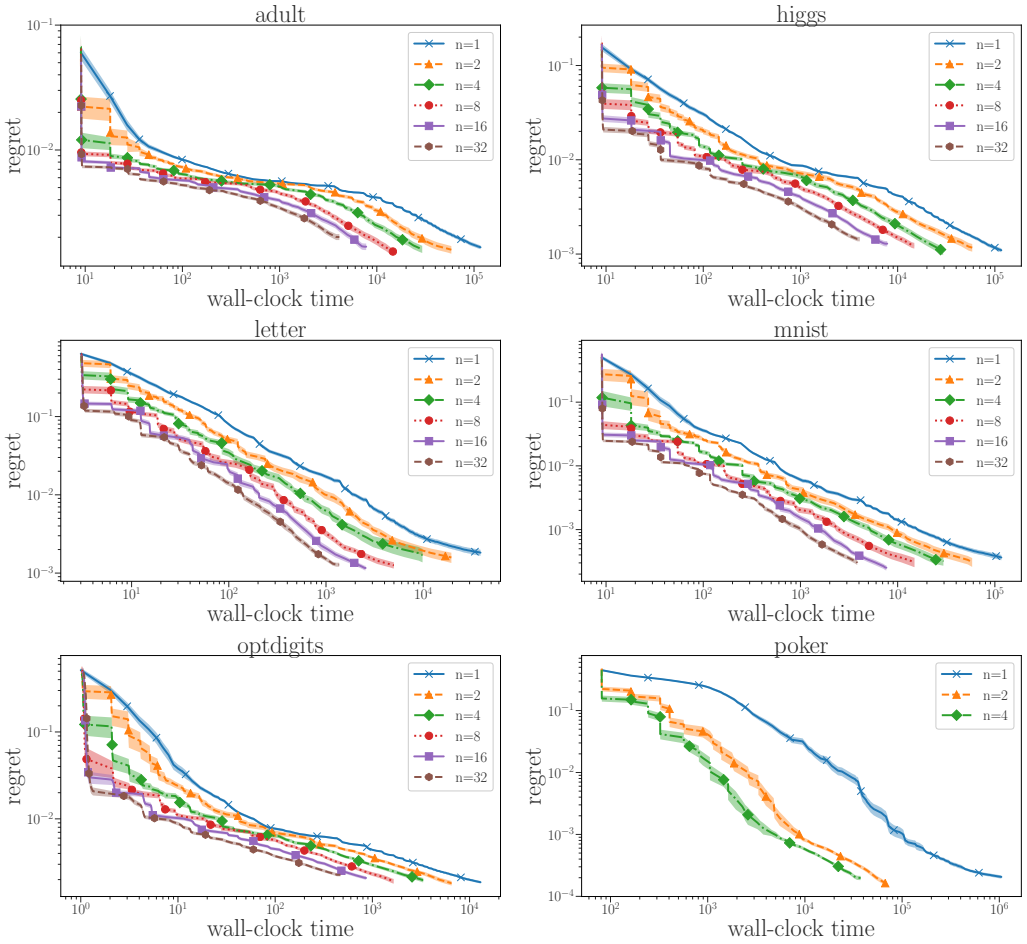


Figure C.3.: Mean performance on the surrogates for all six datasets with different numbers of workers n. As uncertainties, we show the standard error of the mean based on 128 runs. Because we simulated them in real time to capture the true behavior, poker is too expensive to evaluate with less than 16 workers within a day.

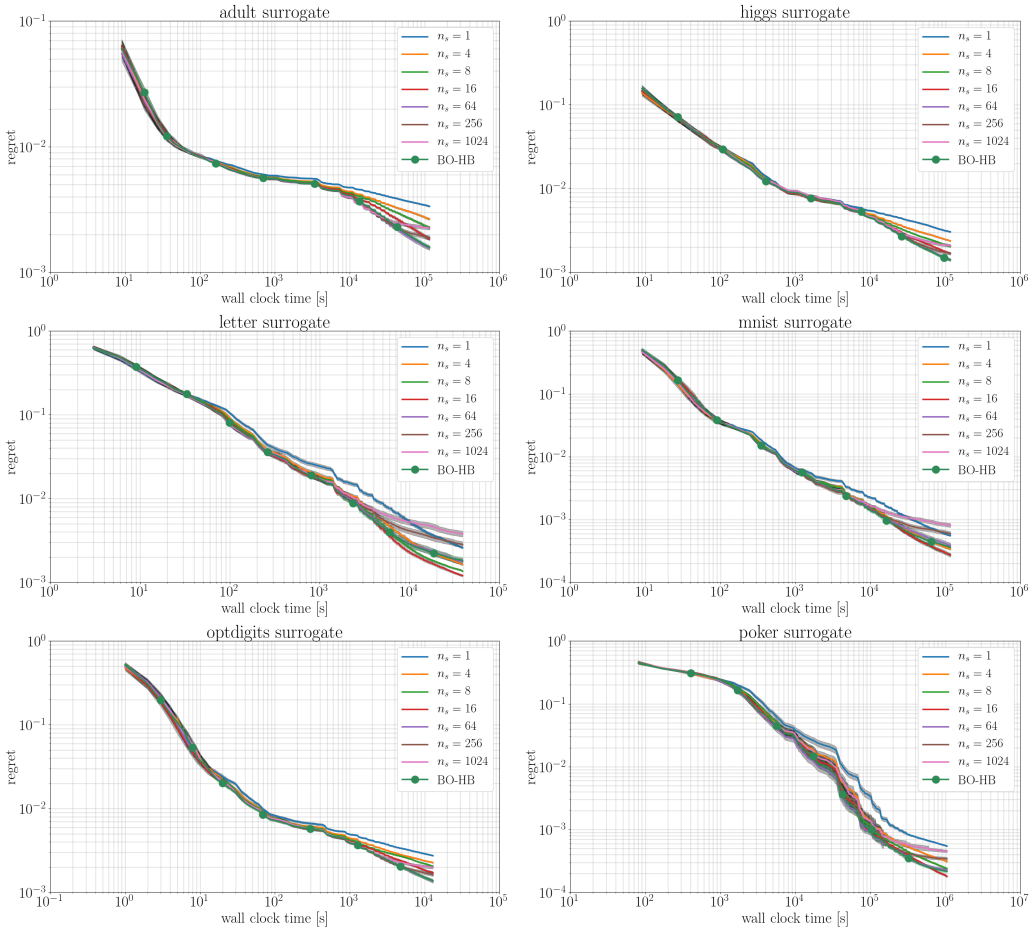


Figure C.4.: Performance on the surrogates for all six datasets for different number of samples

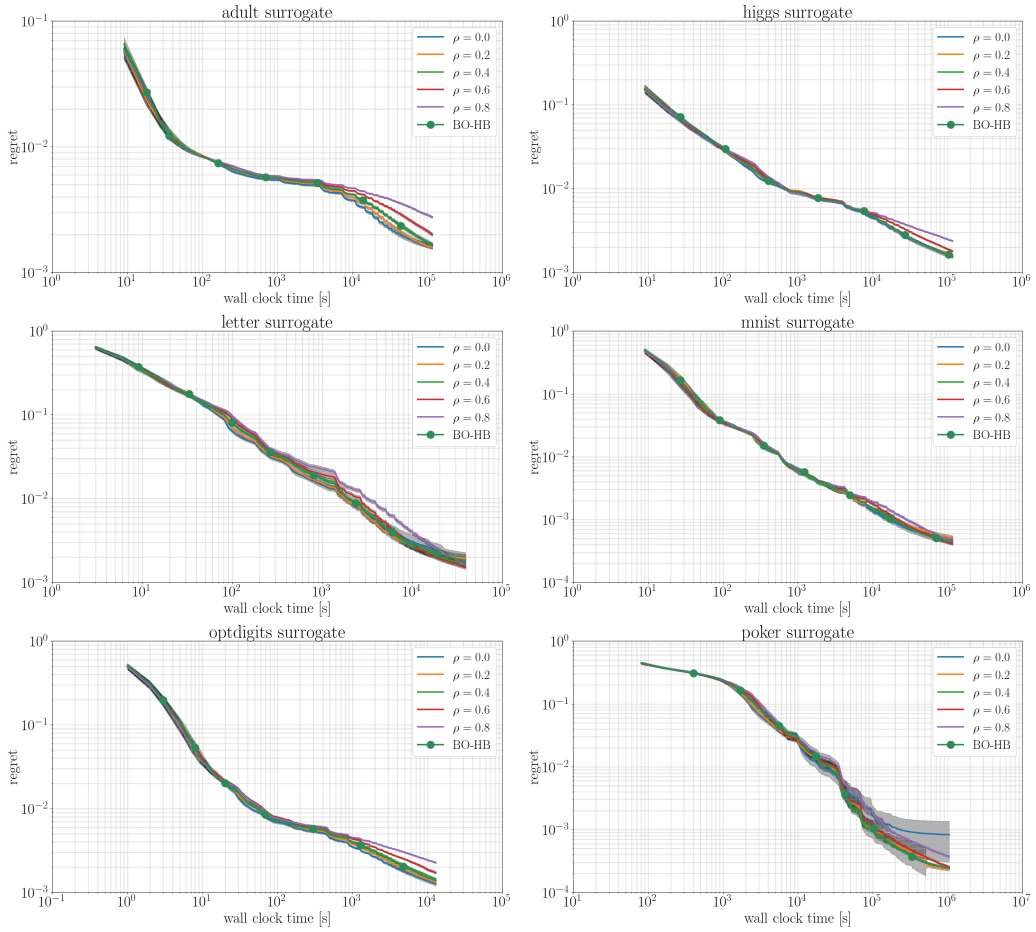


Figure C.5.: Performance on the surrogates for all six datasets for different random fractions

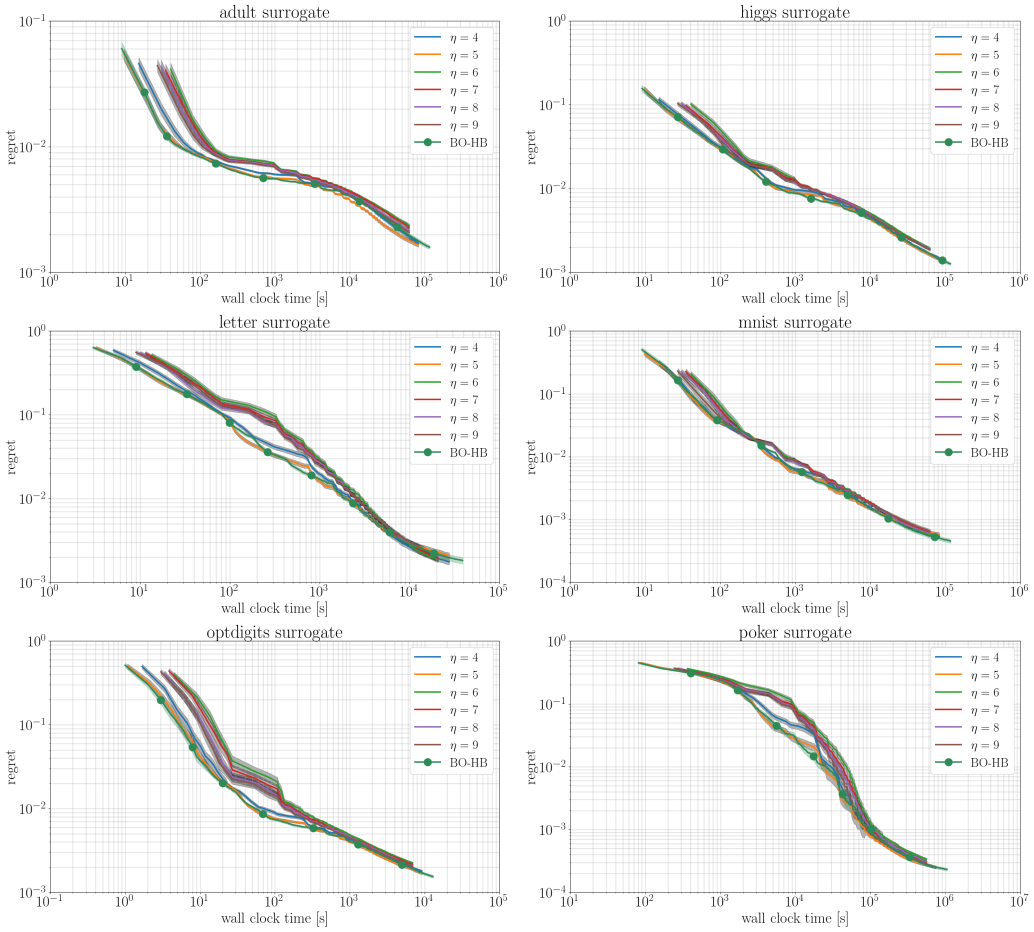


Figure C.6.: Performance on the surrogates for all six datasets for different values of η.

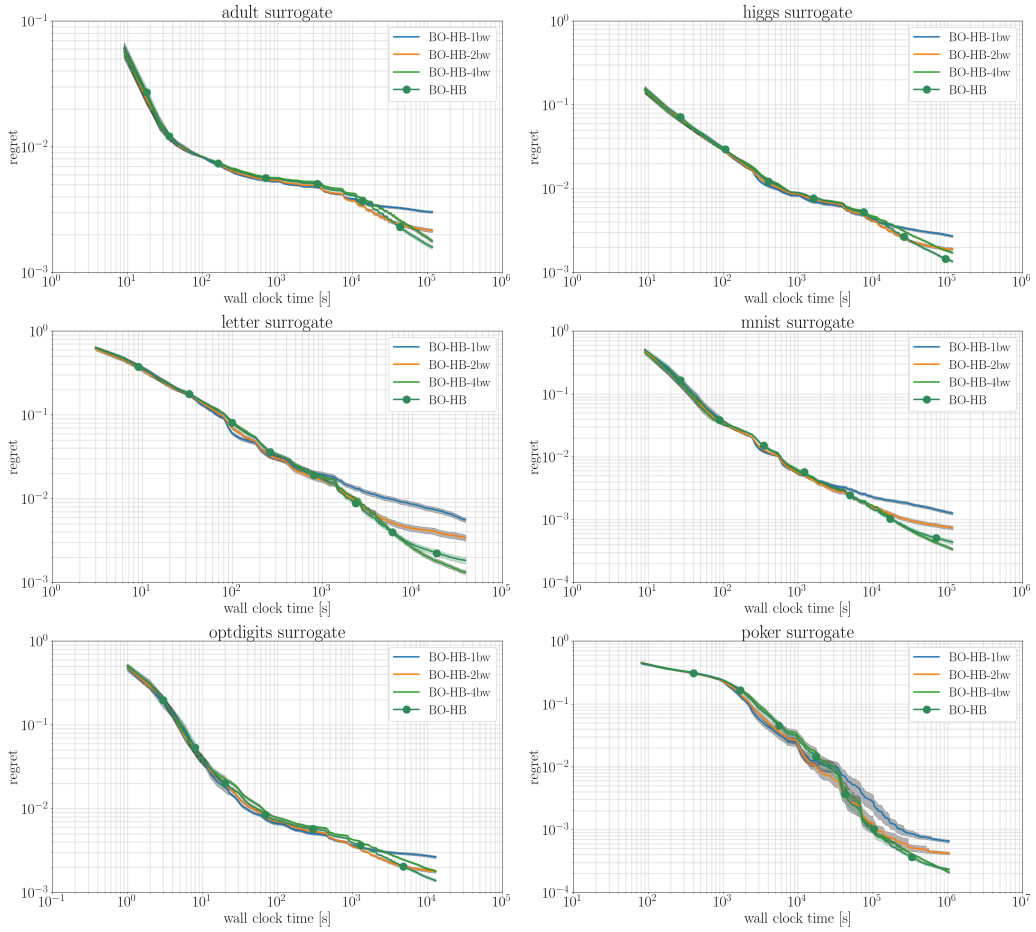


Figure C.7.: Performance on the surrogates for all six datasets for different bandwidth factors.

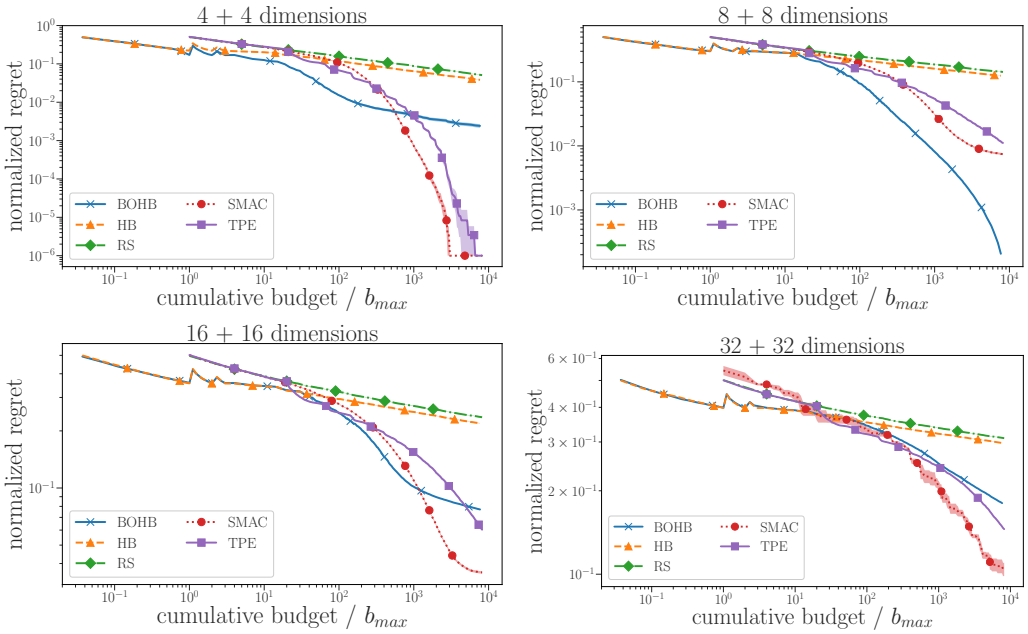


Figure C.8.: Mean performance of BOHB, HB, TPE, SMAC and RS on the mixed domain counting ones function with different dimensions. As uncertainties, we show the standard error of the mean based on 512 runs.

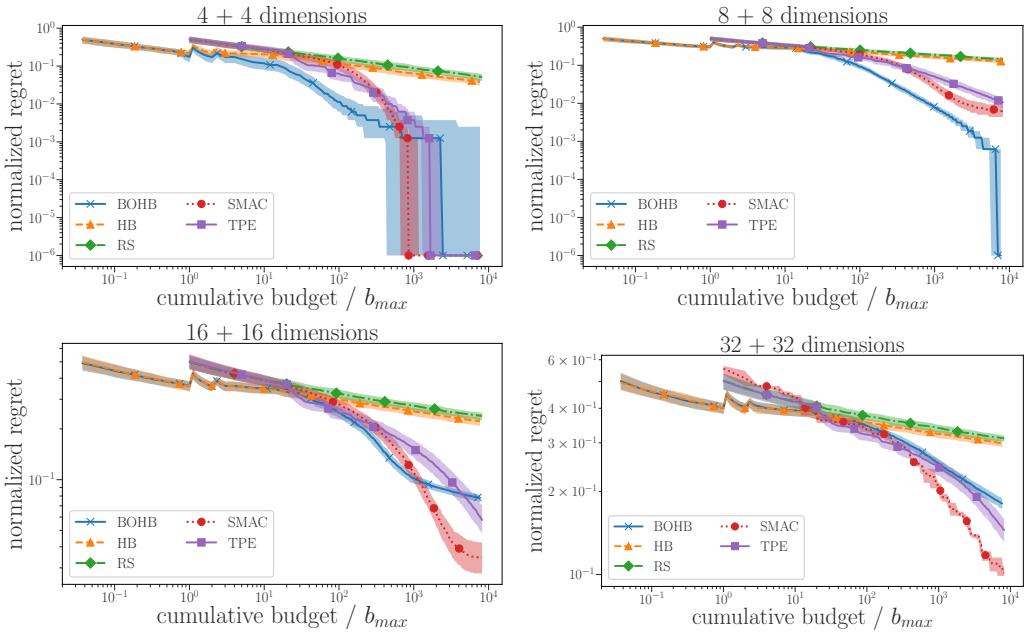


Figure C.9.: Median performance of BOHB, HB, TPE, SMAC and RS on the mixed domain counting ones function with different dimensions. As uncertanties we show the interquartile range based on 512 runs.

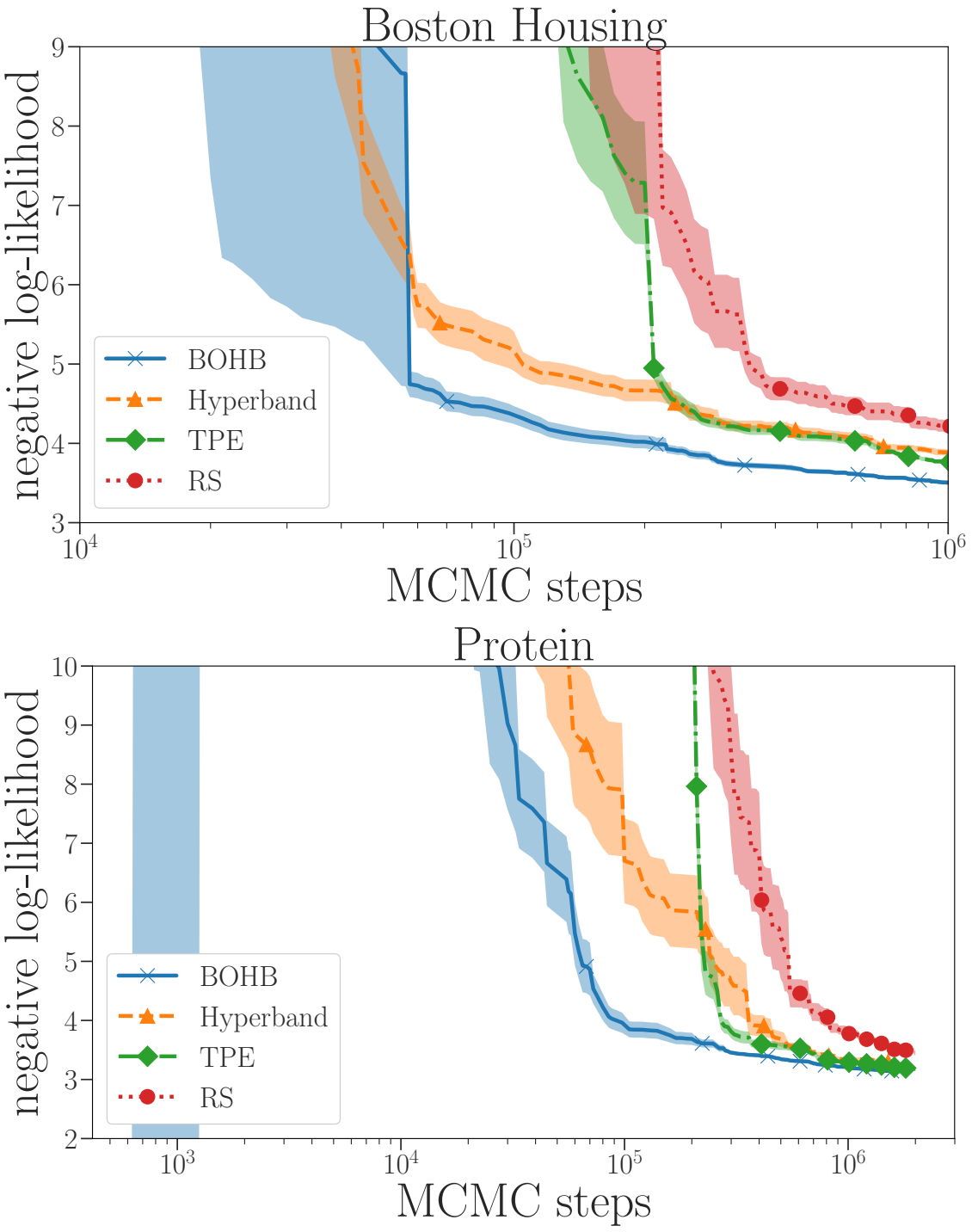


Figure C.10.: Mean performance of TPE, RS, HB and BOHB for optimizing the 5 hyperparameters of a Bayesian neural network on two different UCI datasets. As uncertainties, we show the standard error of the mean based on 50 runs.

D. Supplementary Material to Chapter 7

D.1. Dataset Statistics

We now show the empirical cumulative distribution (ECDF) of all four datasets for: the mean squared error for training, validation and test (Figure D.1), the number of parameters (Figure D.2), the measured wall-clock time for training (Figure D.3) and the noise, defined as standard deviation between the individual trials of each configuration (Figure D.4). Note that we computed the ECDF of the mean squared error and the runtime based on the average over the four trials.

Figure D.5 shows the Spearman rank correlation between the performance of a hyperparameter configuration after training for the final budget of 100 epochs and its performance after training for the corresponding number of epochs on the x-axis. We also show the correlation if only the top $1\%, 10\%, 25\%$ and 50% configurations are taken into account.

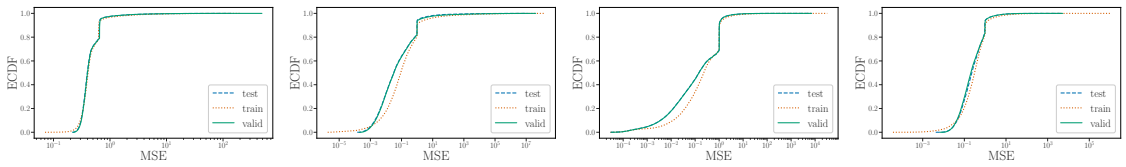


Figure D.1.: The empirical cumulative distribution (ECDF) of the average train/valid/test error after 100 epochs of training, computed on HPO-Bench-Protein (left), HPO-Bench-Slice (left middle), HPO-Bench-Naval (right middle) and HPO-Bench-Parkinson (right).

D.2. Hyperparameter Importance

Figure D.6, D.7, D.8 and D.9 show the importance values based on the fANOVA tool for the top 1% , top 10% and all configurations as well as the most important pair-wise plots for HPO-Bench-Naval, HPO-Bench-Parkinson, HPO-Bench-Protein and HPO-Bench-Slice, respectively. Table D.1, D.2, D.3 and D.4 show the local

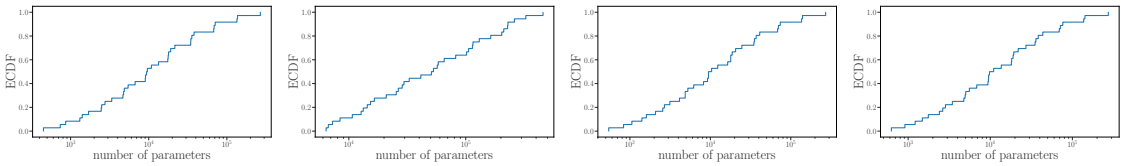


Figure D.2.: The empirical cumulative distribution (ECDF) of the number of parameters, computed on HPO-Bench-Protein (left), HPO-Bench-Slice (left middle), HPO-Bench-Naval (right middle) and HPO-Bench-Parkinson (right).

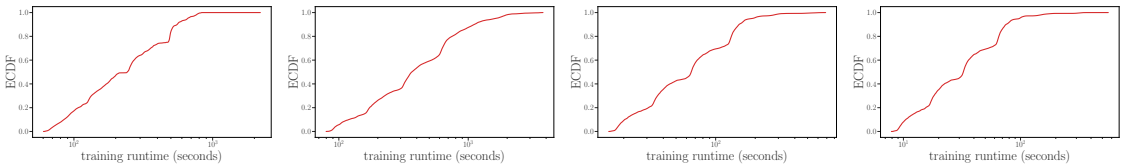


Figure D.3.: The empirical cumulative distribution (ECDF) of the training runtime, computed on HPO-Bench-Protein (left), HPO-Bench-Slice (left middle), HPO-Bench-Naval (right middle) and HPO-Bench-Parkinson (right).

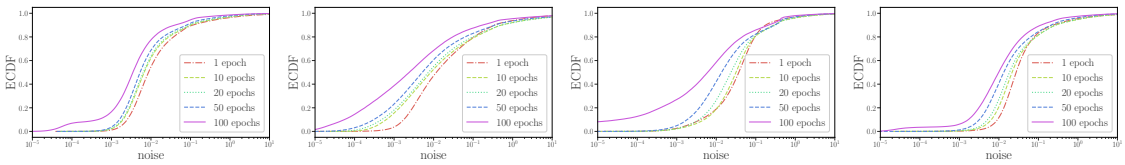


Figure D.4.: The empirical cumulative distribution (ECDF) of the noise across the 4 repeated training processes for each configuration, computed on HPO-Bench-Protein (left), HPO-Bench-Slice (left middle), HPO-Bench-Naval (right middle) and HPO-Bench-Parkinson (right).

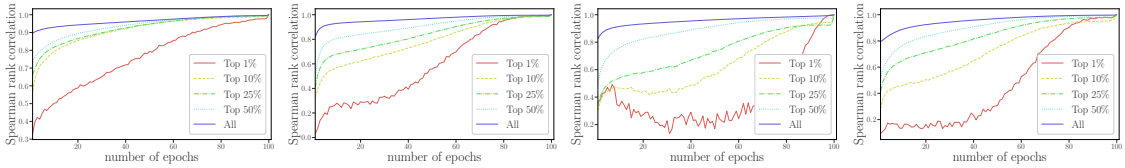


Figure D.5.: The Spearman rank correlation between different number of epochs for the HPO-Bench-Protein (left), HPO-Bench-Slice (left middle), HPO-Bench-Naval (right middle) and HPO-Bench-Parkinson (right) when we consider all configurations or only the top 1%, 10%, 20%, and 50% of all configurations based on their test error.

neighbourhood for HPO-Bench-Naval, HPO-Bench-Parkinson, HPO-Bench-Protein

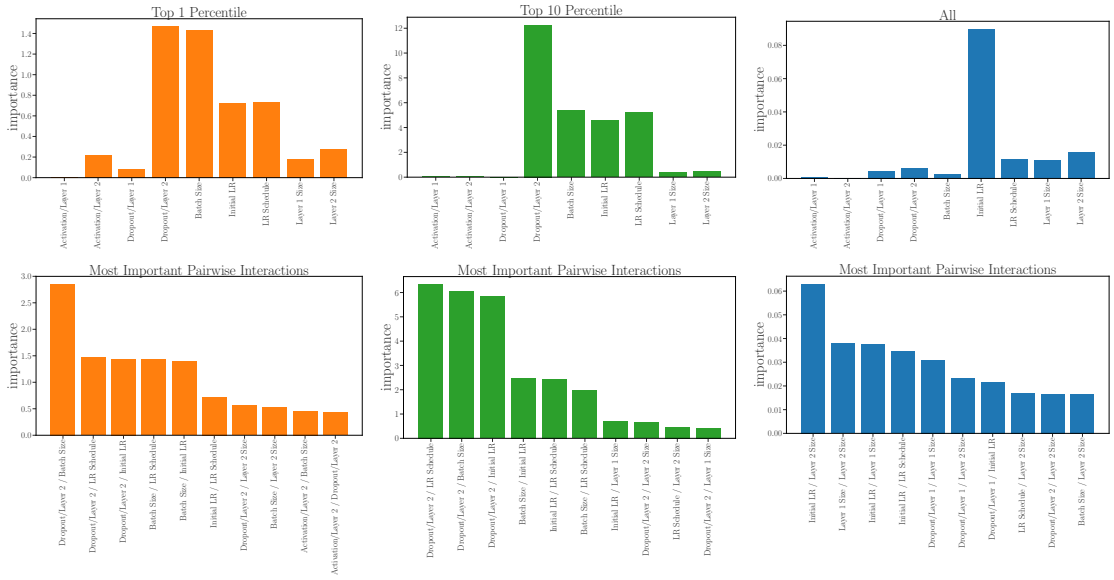


Figure D.6.: HPOBench-Naval. Top row: Importance of the different hyperparameter based on the fANOVA for: (left) only the top 1% ; (middle) top 10% ; (right) all configurations. Bottom row: most important hyperparameter pairs with (left) only the top 1% ; (middle) top 10% ; (right) all configurations.

and HPO-Bench-Slice.

D.3. Comparison HPOBench

We now present a more detail discussion on how we set the metaparameters of the individual optimizer for our comparison. Code to reproduce the experiments is available at `https://github.com/automl/nas_benchmarks`.

Random Search (RS): We sample hyperparameter configurations from a uniform distribution over all possible hyperparameter configurations.

Hyperband: We set the $\eta = 3$ which means that in each successive halving step only a third of the configurations are promoted to the next step. The minimum budget is set to 4 epochs and the maximum budget to 100 epochs of training.

BOHB: As for Hyperband we set $\eta = 3$ and keep the same minimum and maximum budgets. The minimum possible bandwidth for the KDE is set to 0.3 to prevent that the probability mass collapses to a single value. The bandwidth factor is set to 3, the number of samples to optimize the acquisition function is 64, and the fraction of random configurations is set to $1/3$ which are the default values for BOHB.

TPE: We used all predefined metaparameter values from the Hyperopt package.

SMAC: We set the maximum number of allowed function evaluations per configuration to 4. The number of trees for the random forest was set to 10 and the fraction of

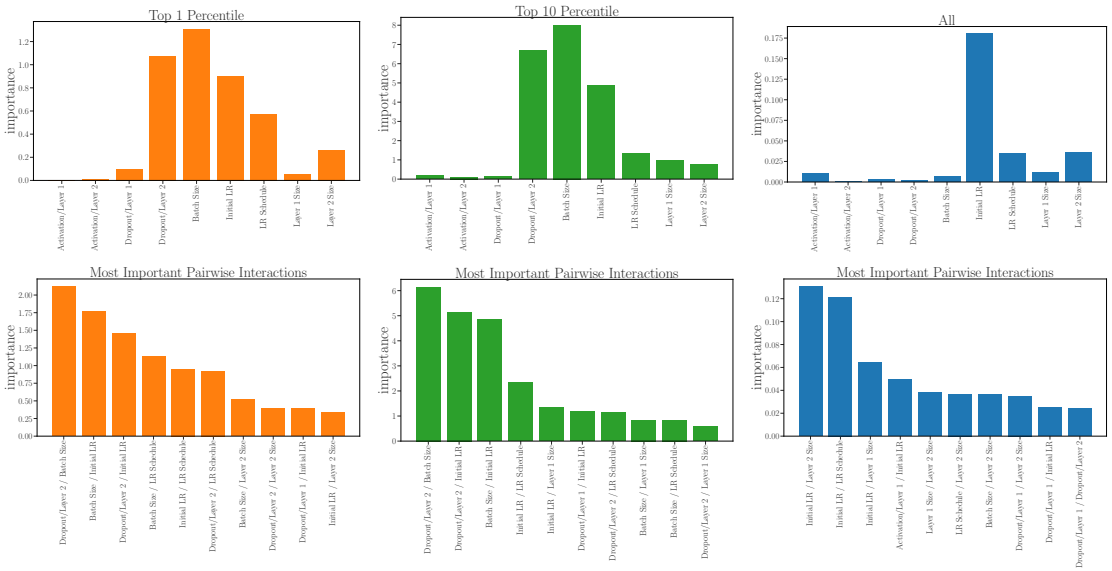


Figure D.7.: HPOBench-Parkinson. Top row: Importance of the different hyperparameter based on the fANOVA for: (left) only the top 1% ; (middle) top 10% ; (right) all configurations. Bottom row: most important hyperparameter pairs with (left) only the top 1% ; (middle) top 10% ; (right) all configurations.

random configurations was set to $1/3$ which are also the default values in the SMAC3 package.

Regularized Evolution (RE): To mutate architectures, we first sample uniformly at random a hyperparameter and then sample a new value from the set of all possible values except the current one. RE has two main hyperparameters, the population size and the tournament size, which we set to 100 and 10, respectively.

Reinforcement Learning (RL): Starting from a uniform distribution over the values of each hyperparameter, we used REINFORCE to optimize the probability values directly (see also Ying et al. (2019). After performing a grid search, we set the learning rate for REINFORCE to 0.1 and used an exponential moving average as baseline for the reward function with a momentum of 0.9.

Bohamiann: We used a 3 layer fully connected neural network with 50 units and tanh activation functions in each layer. We set the step length for the adaptive SGHMC sampler (Springenberg et al., 2016) to 0.01 and the batch size to 8. Starting from a chain length of 20000 steps, the number of burn-in steps was linearly increased by a factor of 10 times the number of observed function values. To optimize the acquisition function, we used a simple local search method, that, starting from a random configuration, evaluates the one-step neighborhood and then jumps to the neighbor with the highest acquisition value until it either reaches the maximum number of steps or converges to a local optimum.

Figure D.10 shows the comparison and the robustness of all considered hyperparameter optimization methods on the four tabular benchmarks. We performed 500

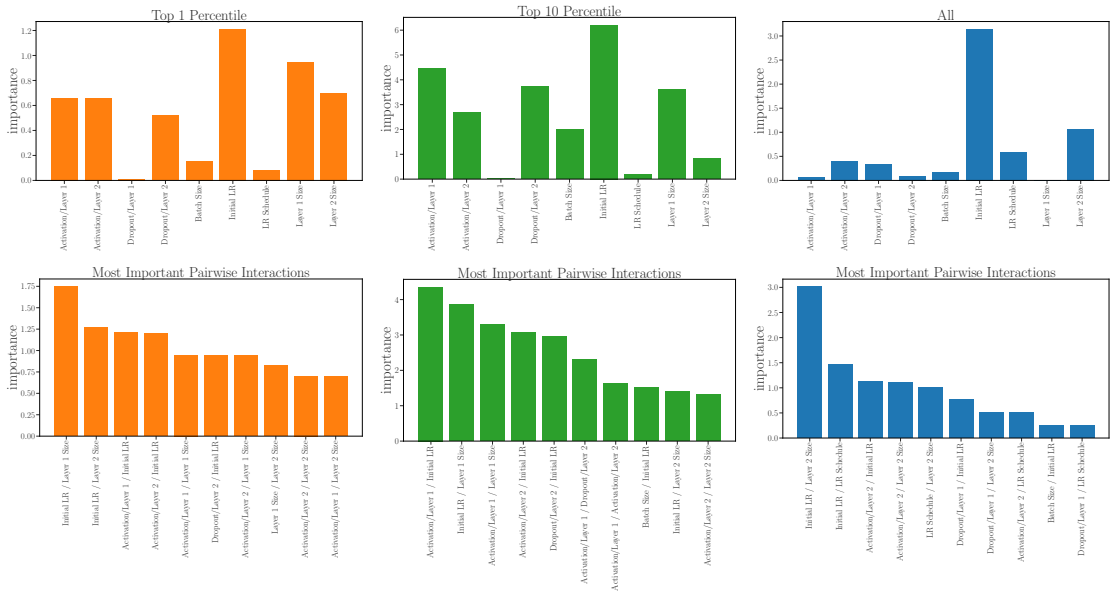


Figure D.8.: HPOBench-Protein. Top row: Importance of the different hyperparameter based on the fANOVA for: (left) only the top 1% ; (middle) top 10% ; (right) all configurations. Bottom row: most important hyperparameter pairs with (left) only the top 1% ; (middle) top 10% ; (right) all configurations.

independent runs for each method and report the mean and the standard error of the mean across all runs. For a detailed analysis of the results see the main text.

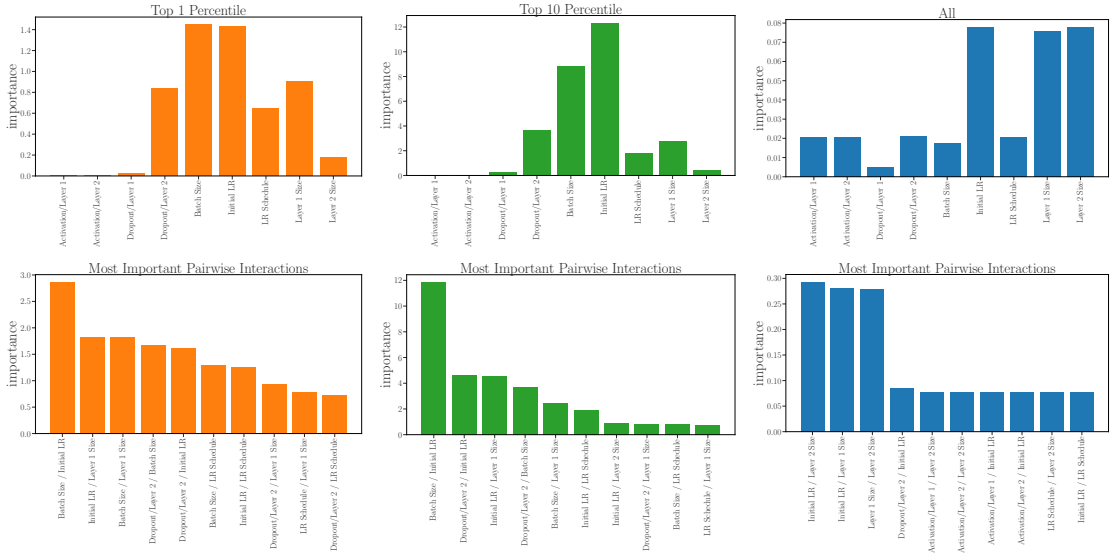


Figure D.9.: HPOBench-Slice. Top row: Importance of the different hyperparameter based on the fANOVA for: (left) only the top 1% ; (middle) top 10% ; (right) all configurations. Bottom row: most important hyperparameter pairs with (left) only the top 1% ; (middle) top 10% ; (right) all configurations.

Table D.1.: HPO-Bench-Naval: Performance change if single hyperparameters of the incumbent (average test error 0.000029) are flipped.

Hyperparameter	Change	Test Error	Relative Change
Layer 1 Size	128 → 256	0.0000	0.1331
Initial LR	0.0005 → 0.001	0.0000	0.1751
Layer 1 Size	128 → 64	0.0000	0.2196
Layer 2 Size	512 → 256	0.0000	0.4929
Batch Size	8 → 16	0.0000	0.5048
Activation/Layer 1	*tanh* → *relu*	0.0000	0.6933
Activation/Layer 2	*relu* → *tanh*	0.0002	4.9685
Dropout/Layer 2	0.0 → 0.3	0.0004	11.1872
Dropout/Layer 1	0.0 → 0.3	0.0010	34.3490
LR Schedule	*cosine* → *const*	0.0063	217.0092

Table D.2.: HPO-Bench-Parkinson: Performance change if single hyperparameters of the incumbent (average test error 0.004239) are flipped.

Hyperparameter	Change	Test Error	Relative Change
Layer 1 Size	512 → 256	0.0051	0.2142
Layer 2 Size	512 → 256	0.0054	0.2740
Batch Size	16 → 32	0.0059	0.3962
Dropout/Layer 1	0.0 → 0.3	0.0081	0.9012
Batch Size	16 → 8	0.0085	1.0068
Activation/Layer 1	*tanh* → *relu*	0.0106	1.5100
Initial LR	0.005 → 0.001	0.0111	1.6268
Activation/Layer 2	*tanh* → *relu*	0.0178	3.1980
Initial LR	0.005 → 0.01	0.0189	3.4530
Dropout/Layer 2	0.0 → 0.3	0.0216	4.0912
LR Schedule	*cosine* → *const*	0.1407	32.1805

Table D.3.: HPO-Bench-Protein: Performance change if single hyperparameters of the incumbent (average test error 0.2153) are flipped.

Hyperparameter	Change	Test Error	Relative Change
Batch Size	8 → 16	0.2163	0.0042
Initial LR	0.0005 → 0.001	0.2169	0.0072
Layer 2 Size	512 → 256	0.2203	0.0231
Layer 1 Size	512 → 256	0.2216	0.0288
Dropout/Layer 2	0.3 → 0.6	0.2257	0.0478
LR Schedule	*cosine* → *const*	0.2269	0.0534
Dropout/Layer 2	0.3 → 0.0	0.2280	0.0587
Dropout/Layer 1	0.0 → 0.3	0.2307	0.0711
Activation/Layer 2	*relu* → *tanh*	0.2875	0.3351
Activation/Layer 1	*relu* → *tanh*	0.3012	0.3987

Table D.4.: HPO-Bench-Slice: Performance change if single hyperparameters of the incumbent (average test error 0.000144) are flipped.

Hyperparameter	Change	Test Error	Relative Change
Layer 1 Size	512 → 256	0.0002	0.0831
Batch Size	32 → 64	0.0002	0.2014
Dropout/Layer 1	0.0 → 0.3	0.0002	0.2514
Layer 2 Size	512 → 256	0.0002	0.2535
Batch Size	32 → 16	0.0002	0.3087
Activation/Layer 1	*relu* → *tanh*	0.0002	0.3383
Activation/Layer 2	*tanh* → *relu*	0.0002	0.6326
Initial LR	0.0005 → 0.001	0.0003	0.7668
Dropout/Layer 2	0.0 → 0.3	0.0006	3.3757
LR Schedule	*cosine* → *const*	0.0007	4.0016

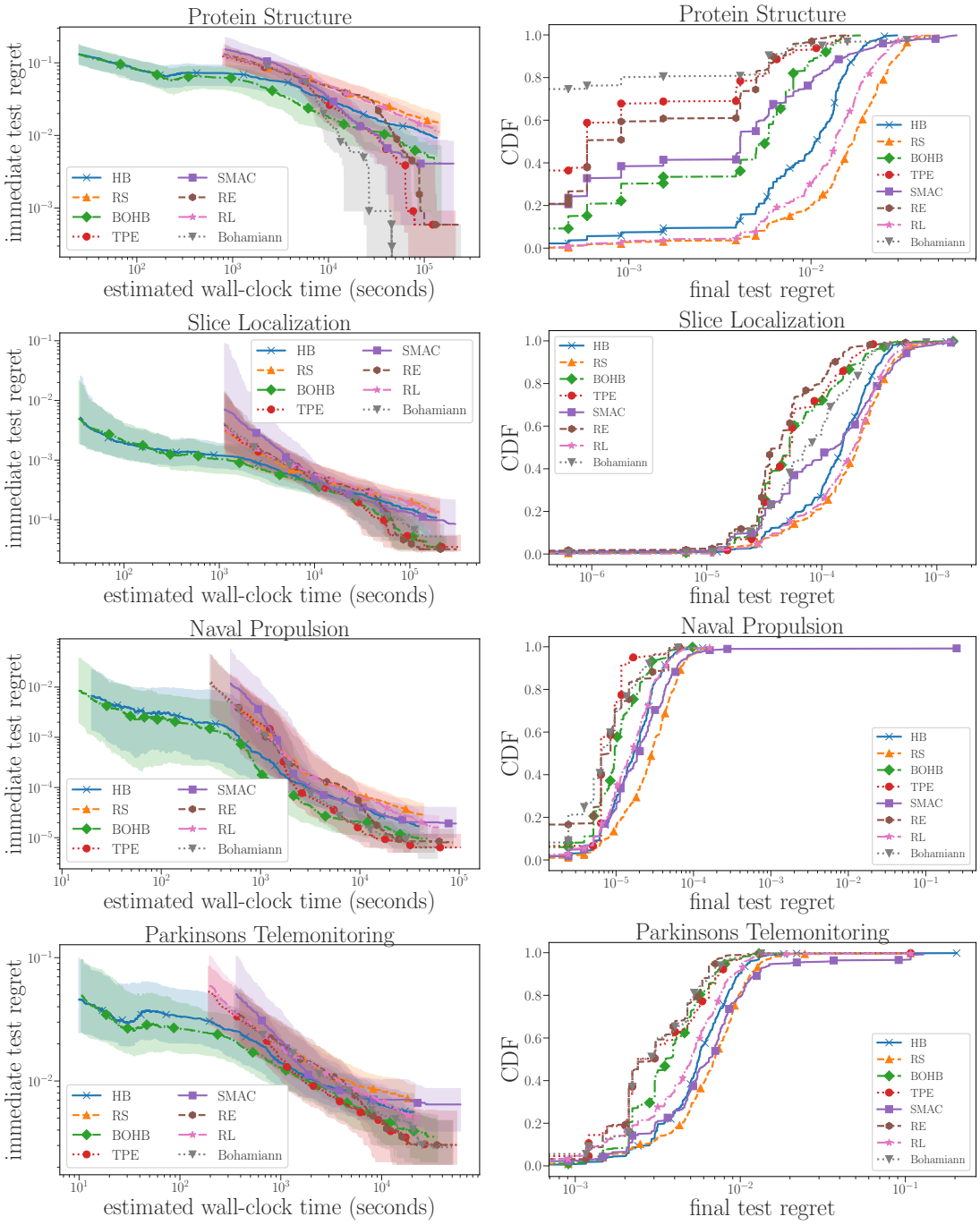


Figure D.10.: Left column: Comparison of various HPO methods on all the datasets. For each method, we plot the median and the 25th and 75th quantiles of the test regret of the incumbent (determined based on the validation performance) across 500 independent runs. Right column: The empirical cumulative distribution of the final regret over all runs after 10^5 seconds for HPO-Bench-Protein, 3×10^5 seconds for HPO-Bench-Slice, 6×10^4 for HPO-Bench-Naval and 3×10^4 seconds for HPO-Bench-Parkinson.

E. Supplementary Material to Chapter 8

E.1. Meta Benchmarks

In Table E.1 we list all OpenML dataset that we used to generate the Meta-SVM and Meta-FCNet benchmarks and in Table E.2 the UCI datasets that we used for the Meta-XGBoost benchmark. The ranges of the hyperparameters for all benchmarks are given in Table E.3. Figure E.1 shows the empirical cumulative distribution over the observed target values based on the Sobol grid for all tasks.

Name	OpenML Task ID	number of features	number of datapoints
kr-vs-kp	3	37	3196
covertype	2118	55	110393
letter	236	17	20000
higgs	75101	29	98050
optdigits	258	65	5620
electricity	336	9	45312
magic telescope	75112	12	19020
nomao	146595	119	34465
gas-drift	146590	129	13910
mfeat-pixel	250	241	2000
car	251	7	1728
churn	167079	101	1212
dna	167202	181	3186
vehicle small	283	19	846
vehicle	75191	101	98528
MNIST	3573	785	50000

Table E.1.: OpenML dataset we used for the FC-Net and SVM classification benchmarks

Name	number of features	number of datapoints
boston housing	13	506
concrete	9	1030
parkinsons telemonitoring	26	5875
combined cycle power plant	4	9568
energy	8	768
naval propulsion	16	11934
protein structure	9	45730
yacht-hydrodynamics	7	308
winequality-red	12	4898
slice localization	386	53500

Table E.2.: UCI regression dataset we used for the XGBoost benchmark. All dataset can be found at https://archive.ics.uci.edu/ml/datasets.html

	Name	Range	log scale
SVM	C	$[e^{-10}, e^{10}]$	✓
	γ	$[e^{-10}, e^{10}]$	✓
FC-Net	learning rate	$[10^{-6}, 10^{-1}]$	✓
	batch size	$[8, 128]$	✓
	units layer 1	$[16, 512]$	✓
	units layer 2	$[16, 512]$	✓
	drop. rate l1	$[0.0, 0.99]$	-
	drop. rate l2	$[0.0, 0.99]$	-
XGBoost	learning rate	$[10^{-6}, 10^{-1}]$	✓
	gamma	$[0, 2]$	-
	L1 regularization	$[10^{-5}, 10^{3}]$	✓
	L2 regularization	$[10^{-5}, 10^{3}]$	✓
	number of estimators	$[10, 500]$	-
	subsampling	$[0.1, 1]$	-
	max. depth	$[1, 15]$	-
	min. child weight	$[0, 20]$	-

Table E.3.: Hyper-parameter configuration space of the support vector machine (SVM), fully connected neural network (FC-Net) and the gradient tree boosting (XGBoost) benchmark.

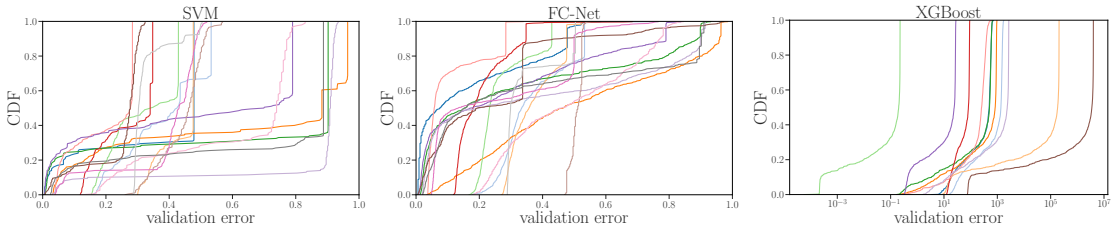


Figure E.1.: The empirical cumulative distribution plots of all observed target values for all tasks.

E.2. Comparison Random Search vs. Bayesian Optimization on XGBoost

For completeness we show in Figure E.2 the comparison of random search (RS) and Bayesian optimization with Gaussian processes (BO-GP) on several UCI regression datasets. Out of the 10 datasets, GP-BO perform better than RS on 10, worse on one, and ties on 2 and hence performs overall better than RS which is inline with the results obtained from out meta-model. However, if we would look only on the first three datasets: Boston-Housing, PowerPlant and Concrete it would be much harder to draw strong conclusions.

E.3. Details about the Forrester Benchmark

Figure E.3 shows the original 9 tasks (left), their representation on the latent space of the model (middle) and an example of 10 new generated task (right), that resemble the original ones.

E.4. Samples for the Meta-SVM Benchmark

In Figure E.4 and Figure E.4 we show additional randomly sampled function sampled with and without noise. One can see that, while the general characteristics of the original objective function, i. e. bowl shaped around the lower right corner, remains, the local structure changes across samples.

E.5. Comparison of HPO Methods

We now described the specific detail of each optimizer in turn.

Random search (RS) Bergstra and Bengio (2012) We defined a uniform distribution over the input space and in each iteration randomly sampled a datapoint from this distribution.

Differential Evolution (DE) (Storn and Price, 1997) maintains a population of data points and generates new candidate points by mutation random points from the population. We defined the probability for mutation and crossover to be 0.5. The population size was 10 and we sampled new candidate points based on the 'rand/1/bin' strategy.

Tree Parzen Estimator (TPE) (Bergstra et al., 2011) is a Bayesian optimization method that uses kernel density estimators (KDE) to model the probability of 'good' points in the input space that achieve a function value that is lower than a certain value and 'bad' points that achieve a function value than a certain value. TPE computes the acquisition as the ration between the likelihood of the two KDE which is equivalent to expected improvement. We used the default provided by the hyperopt (`https://github.com/hyperopt/hyperopt`) package.

SMAC (Hutter et al., 2011) is also a Bayesian optimization methods that uses random forests to model the objective function and stochastic local search to optimize the acquisition function. We followed the default of SMAC and set the number of trees for the random forest to 10.

CMA-ES (Hansen, 2006) is an evolutionary strategy that models a population as a multivariate normal distribution. We used the open source pycma package (`https://github.com/CMA-ES/pycma`). We set the initial standard deviation to 0.6.

Gaussian Process based Bayesian optimization (BO-GP) as described by Snoek et al. (2012). We used expected improvement as acquisition function and an adapted random search strategy, which given a maximum number of allowed points $N = 500$ samples first 70% uniformly at random and the rest from a Gaussian with a fixed variance around the best observed point. While other methods such as gradient ascent techniques or continuous global optimization methods could also be used, we found this to work faster and more robustly. We marginalized the acquisition function over the Gaussian process hyperparameters (Snoek et al., 2012) and used the emcee package (`http://dfm.io/emcee/current/`) to sample hyperparameter configuration from the marginal log-likelihood. We used a Matern 52 kernel for the Gaussian process.

BOHAMIANN (Springenberg et al., 2016) uses a Bayesian neural network inside Bayesian optimization where the weights are sampled based on stochastic gradient Hamiltonian Monte-Carlo (Chen et al., 2014). We use a step length of 1E-2 for the MCMC sampler and increased the number of burnin step by a factor of 100 times the number of observed data points. In each iteration we sampled 100 weight vectors over 10000 MCMC steps. We used the same random search method to optimize the acquisition function as for BO-GP.

All methods started from a uniformly sampled point and we estimated the incumbent after each function evaluation as the point with the lowest observed function value.

In Figure E.6 and Table E.4 we show the aggregated results based on the runtime and the ranking for all methods on all three benchmarks. We also show in Figure E.6 the p-values of the Mann-Whitney U test between all methods. For a detailed analysis of the results see Section 5.3 in the main paper.

Benchmark	RS	DE	TPE	SMAC	BOHAMIANN	CMAES	BO-GP
Meta-SVM (noiseless)	52.19	74.37	79.64	73.77	90.33	73.69	**98.88**
Meta-SVM (noise)	56.64	77.29	76.44	78.56	**89.80**	76.27	88.70
Meta-FCNet (noiseless)	45.71	77.99	78.73	72.71	82.50	56.31	**84.71**
Meta-FCNet (noise)	33.66	49.88	46.84	43.09	**57.28**	37.41	56.04
Meta-XGBoost (noiseless)	41.59	80.35	71.02	84.95	94.01	77.17	**94.69**
Meta-XGBoost (noise)	41.71	80.05	71.05	85.34	94.23	77.15	**94.87**
Meta-SVM (noiseless)	5.89	4.47	4.50	4.64	2.75	4.52	**1.22**
Meta-SVM (noise)	5.72	4.13	4.42	4.11	**2.62**	4.17	2.84
Meta-FCNet (noiseless)	5.67	3.70	3.72	4.09	2.90	5.14	**2.79**
Meta-FCNet (noise)	4.92	3.74	3.95	4.26	**3.21**	4.66	3.27
Meta-XGBoost (noiseless)	6.15	4.11	4.95	3.78	2.40	4.57	**2.03**
Meta-XGBoost (noise)	6.15	4.12	4.96	3.76	2.39	4.58	**2.02**

Table E.4.: *Top*: Each element of the table show the averaged runtime after 100 function evaluations for each method-benchmark pair. *Bottom*: Same but for the ranking of the methods.

E.6. Details of the Meta-Model

The neural network architecture for our meta-model consisted of 3 fully connected layers with 500 units each and tanh activation functions. The step length for the MCMC sampler was set to $1E-2$ and we used the first 50000 steps as burn-in. For the probabilistic encoder, we used Bayesian GP-LVM[1](Titsias and Lawrence, 2010) with a Matern52 kernel to learn a $Q = 5$ dimensional latent space for the task description.

[1]We used the implementation from GPy (2012)

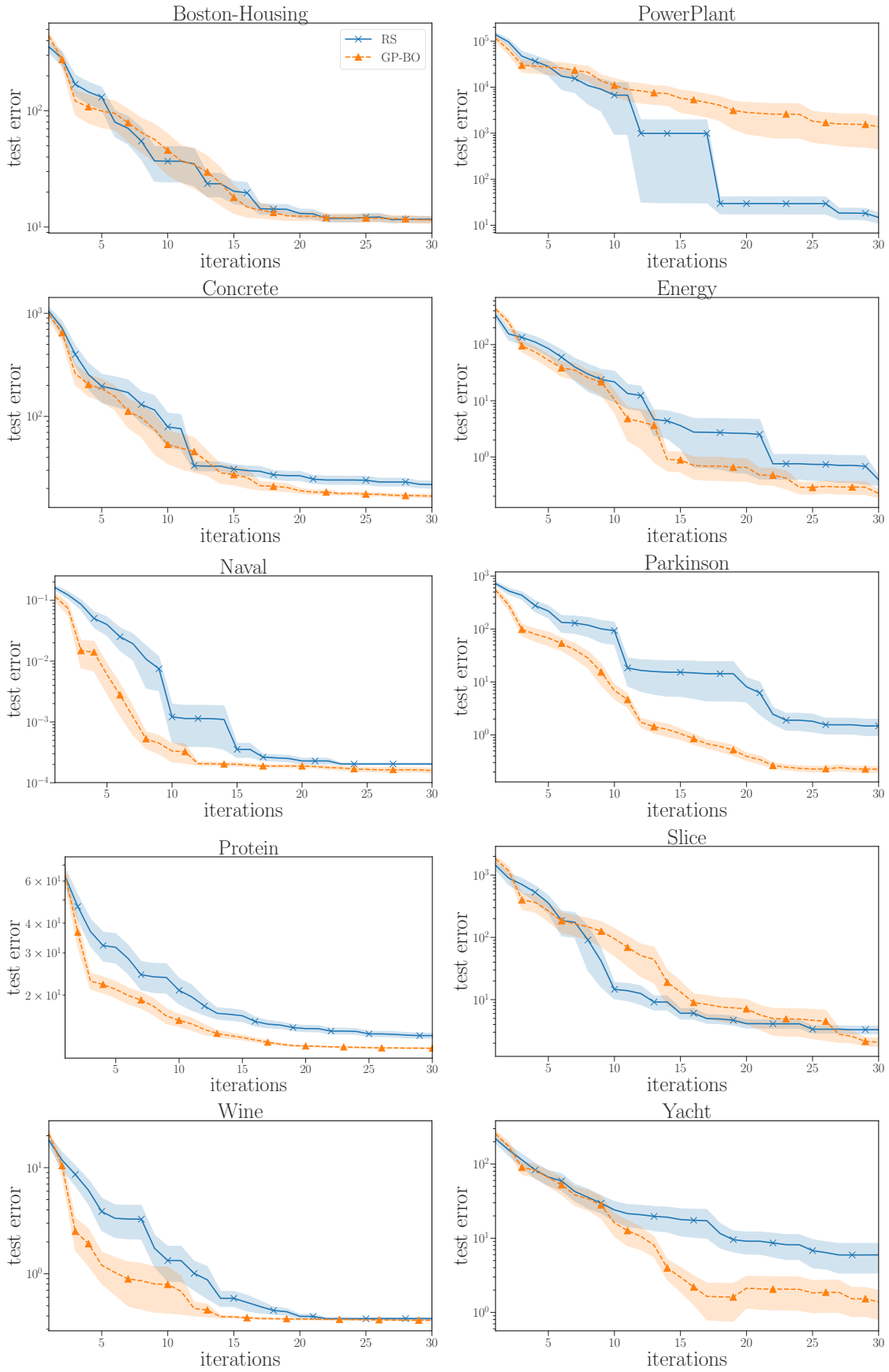


Figure E.2.: Comparisons Bayesian optimization with Gaussian processes (GP-BO) and random search (RS) for optimizing the hyperparameters of XGBoost.

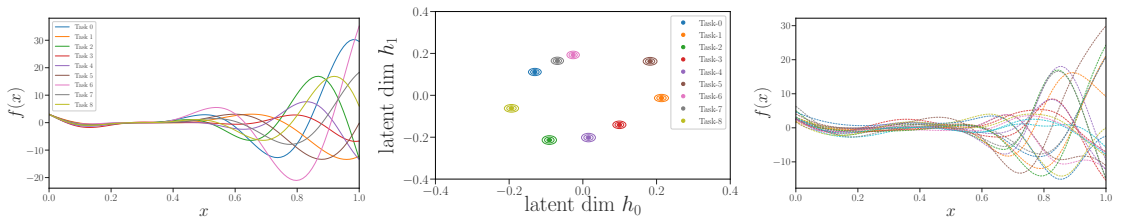


Figure E.3.: Visualizing the concept of our meta-model on the one-dimensional Forrester function. *Left:* 9 different tasks (solid lines) coming from the same distribution. *Middle:* We use a probabilistic encoder to learn a two-dimensional latent space for the task embedding. *Right:* Given our encoder and the multi-task model we can generate new task (dashed lines) that, based on the collected data, resemble the original tasks.

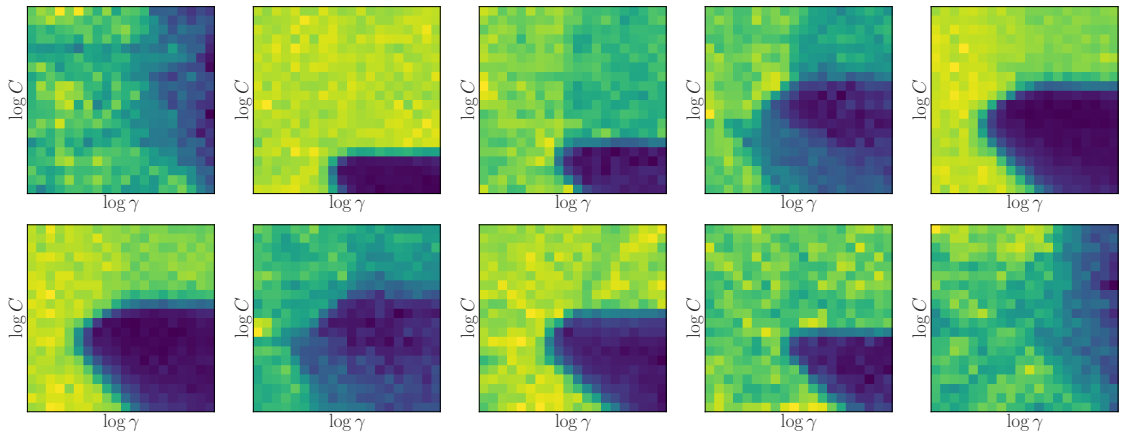


Figure E.4.: Noisy samples from our meta-model for the SVM benchmark

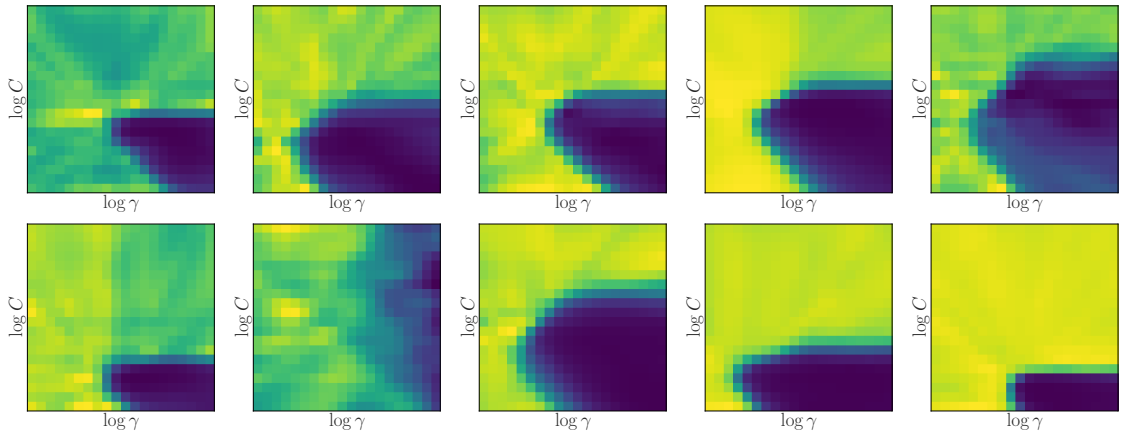


Figure E.5.: Noiseless samples from our meta-model for the SVM benchmark

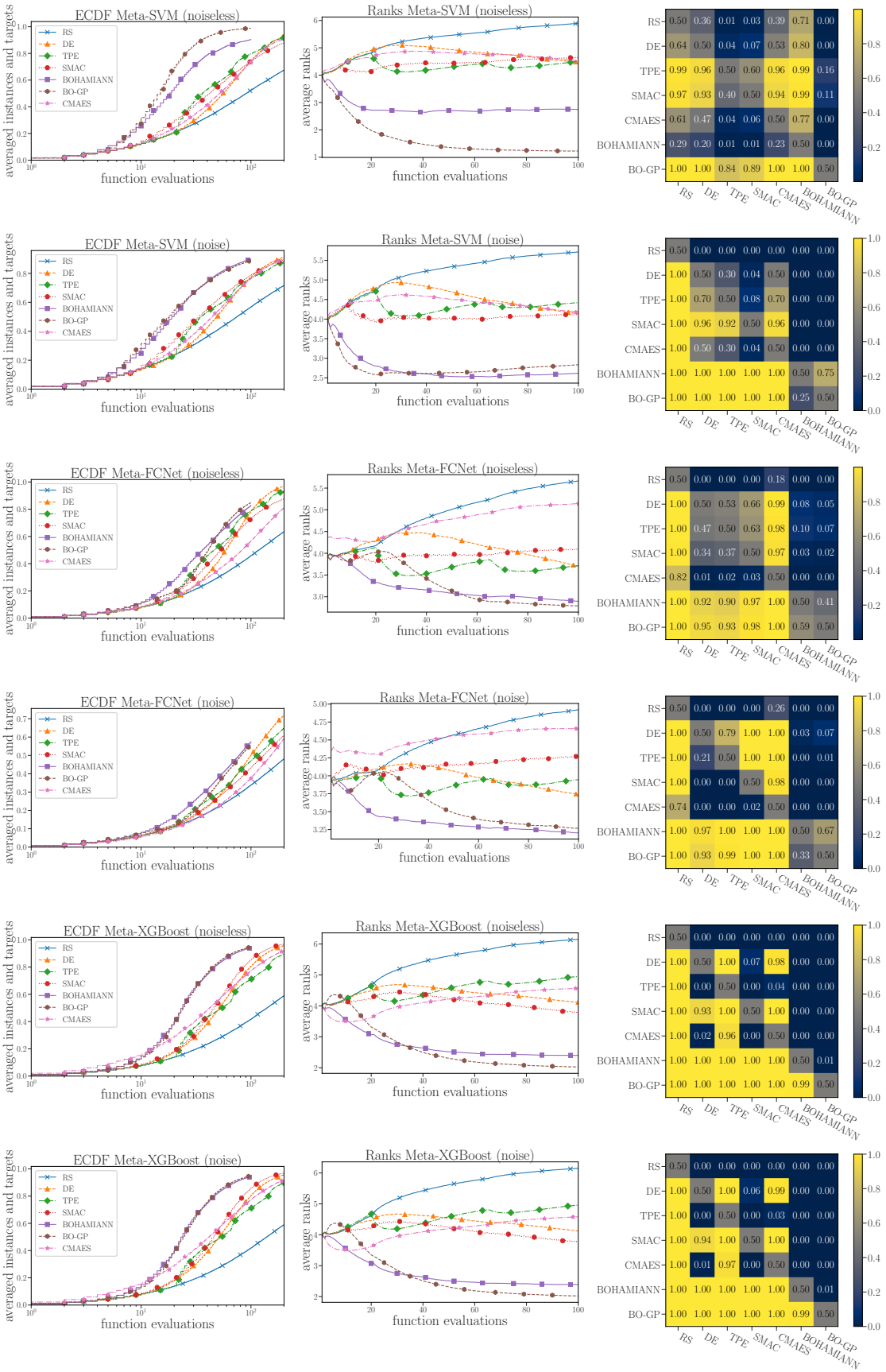


Figure E.6.: Comparison of various different methods on all three HPO problems. From above to below 2-dimensional support vector machine, 6-dimensional feed-forward neural network and 8-dimensional gradient boosting. The two columns on the left show the ECDF and ranking for the noiseless version of each HPO problem (same for the noisy version).

Bibliography

Ahn, S., Korattikara, A., and Welling, M. (2012). Bayesian posterior sampling via stochastic gradient Fisher scoring. In *Proceedings of the 29th International Conference on Machine Learning (ICML'12)*.

Almeida, L. B., Langlois, T., Amaral, J. D., and Plakhov, A. (1999). *Parameter Adaptation in Stochastic Optimization*, chapter 6. Publications of the Newton Institute. Cambridge University Press.

Baldi, R., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*.

Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013). Collaborative hyperparameter tuning. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*.

Baydin, A. G., Cornish, R., Rubio, D. M., Schmidt, M., and Wood, F. (2018). Online learning rate adaptation with hypergradient descent. In *International Conference on Learning Representations (ICLR'18)*.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Advances in Neural Information Processing Systems (NIPS'11)*.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*.

Bertrand, H., Ardon, R., Perrot, M., and Bloch, I. (2017). Hyperparameter optimization of deep neural networks: Combining hyperband with Bayesian model selection. In *Proceedings of Conférence sur l'Apprentissage Automatique (CAP'17)*.

Biedenkapp, A., Lindauer, M., Eggensperger, K., Fawcett, C., Hoos, H., and Hutter, F. (2017). Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the Conference on Artificial Intelligence (AAAI'17)*.

Blackard, J. A. and Dean, D. J. (1999). Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Bottou, L. (2012). Stochastic gradient tricks. In *Neural Networks, Tricks of the Trade, Reloaded*. Springer.

Breimann, L. (2001). Random forests. *Machine Learning*.

Brochu, E., Cora, V., and de Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv:1012.2599 [cs.LG]*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv:1606.01540 [cs.LG]*.

Calandra, R., Gopalan, N., Seyfarth, A., Peters, J., and Deisenroth, M. (2014). Bayesian gait optimization for bipedal locomotion. In *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*.

Cattral, R., Oppacher, F., and Deugo, D. (2002). Evolutionary data mining with automatic rule generalization. *Recent Advances in Computers, Computing and Communications*.

Chen, C., Carlson, D. E., Gan, Z., Li, C., and Carin, L. (2016). Bridging the gap between stochastic gradient MCMC and stochastic optimization. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS'16)*.

Chen, T., Fox, E., and Guestrin, C. (2014). Stochastic gradient Hamiltonian Monte Carlo. In *Proceedings of the 31th International Conference on Machine Learning, (ICML'14)*.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

Chen, Y., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Lillicrap, T. P., Botvinick, M., and de Freitas, N. (2017). Learning to learn without gradient descent by gradient descent. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Chen, Y., Huang, A., Wang, Z., Antonoglou, I., Schrittwieser, J., Silver, D., and de Freitas, N. (2018). Bayesian optimization in AlphaGo. *arXiv:1812.06855 [cs.LG]*.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Conn, A. R., Scheinberg, K., and Vicente, L. N. (2009). *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics.

Coraddu, A., Oneto, L., Ghio, A., Savio, S., Anguita, D., and Figari, M. (2014). Machine learning approaches for improving condition based maintenance of naval propulsion plants. *Journal of Engineering for the Maritime Environment*.

Cunningham, J., Hennig, P., and Lacoste-Julien, S. (2011). Gaussian probabilities and expectation propagation. *arXiv:1111.6832 [stat.ML]*.

Dai, Z., Álvarez, M. A., and Lawrence, N. (2017). Efficient modeling of latent information in supervised learning using gaussian processes. In *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NIPS'17)*.

Depeweg, S., Hernández-Lobato, J., Doshi-Velez, F., and Udluft, S. (2018). Decomposition of uncertainty in Bayesian deep learning for efficient and risk-sensitive learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*.

DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *arXiv:1708.04552 [cs.CV]*.

Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*.

Duane, S., Kennedy, A. D., Pendleton, B. J., and Roweth, D. (1987). Hybrid monte carlo. *Physics Letters B*.

Duchi, J. C., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., and Leyton-Brown, K. (2013). Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS Workshop on Bayesian Optimization (BayesOpt'13)*.

Eggensperger, K., Hutter, F., Hoos, H., and Leyton-Brown, K. (2015). Efficient benchmarking of hyperparameter optimizers via surrogates. In *Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'15)*.

Elsken, T., Metzen, J. H., and Hutter, F. (2018). Neural architecture search: A survey. *arXiv:1808.05377 [stat.ML]*.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations (ICLR'19)*.

Falkner, S., Klein, A., and Hutter, F. (2017). Combining hyperband and Bayesian optimization. In *NIPS Workshop on Bayesian Optimization (BayesOpt'17)*.

Falkner, S., Klein, A., and Hutter, F. (2018a). BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*.

Falkner, S., Klein, A., and Hutter, F. (2018b). Practical hyperparameter optimization for deep learning. In *ICLR 2018 Workshop Track*.

Feurer, M. and Hutter, F. (2018). Hyperparameter optimization. In *Automatic Machine Learning: Methods, Systems, Challenges*. Springer.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems (NIPS'15)*.

Feurer, M., Springenberg, T., and Hutter, F. (2015b). Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'15)*.

Foreman-Mackey, D., Hogg, D. W., Lang, D., and Goodman, J. (2013). emcee : The MCMC Hammer. *Publications of the Astronomical Society of the Pacific*.

Franceschi, L., Donini, M., Frasconi, P., and Pontil, M. (2017). Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Frazier, P. I. (2018). A tutorial on bayesian optimization. *arXiv:1807.02811 [stat.ML]*.

Frey, P. W. and Slate, D. J. (1991). Letter recognition using holland-style adaptive classifiers. *Machine Learning*.

Fusi, N., Sheth, R., and Elibol, M. (2018). Probabilistic matrix factorization for automated machine learning. In *Proceedings of the 31th International Conference on Advances in Neural Information Processing Systems (NIPS'18)*.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML'16)*.

Gal, Y., Islam, R., and Ghahramani, Z. (2017). Deep Bayesian active learning with image data. *arXiv:1703.02910 [cs.LG]*.

Garnett, R., Osborne, M. A., and Roberts, S. J. (2010). Bayesian optimization for sensor set selection. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*.

Gastaldi, X. (2017). Shake-shake regularization. *arXiv:1705.07485 [cs.LG]*.

Girolami, M. and Calderhead, B. (2011). Riemann manifold langevin and hamiltonian monte carlo methods. *Journal of the Royal Statistical Society Series B - Statistical Methodology*.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. (2018). Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*.

González, J., Longworth, J., James, D. C., and Lawrence, N. D. (2014). Bayesian optimization for synthetic gene design. In *NIPS Workshop on Bayesian Optimization (BayesOpt'14)*.

GPy (since 2012). GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`.

Graf, F., Kriegel, H. P., Schubert, M., Pölsterl, S., and Cavallaro, A. (2011). 2d image registration in ct images using radial image descriptors. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI'11)*.

Graves, A. (2011). Practical variational inference for neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*.

Gutmann, M. U. and Corander, J. (2016). Bayesian optimization for likelihood-free inference of simulator-based statistical models. *Journal of Machine Learning Research*.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. (2009). The WEKA data mining software: An update. *SIGKDD Explorations*.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*. Springer Berlin Heidelberg.

Hansen, N., Auger, A., Brockhoff, D., Tusar, D., and Tusar, T. (2016a). COCO: performance assessment. *arXiv:1605.03560 [cs.NE]*.

Hansen, N., Auger, A., Mersmann, O., Tušar, T., and Brockhoff, D. (2016b). COCO: A platform for comparing continuous optimizers in a black-box setting. *arXiv:1603.08785 [cs.AI]*.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Proceedings of the 32nd Conference on Artificial Intelligence (AAAI-18)*.

Hennig, P. and Schuler, C. (2012). Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*.

Hernández-Lobato, J. and Adams, R. (2015). Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Hernández-Lobato, J., Hoffman, M., and Ghahramani, Z. (2014). Predictive entropy search for efficient global optimization of black-box functions. In *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS'14)*.

Hernández-Lobato, J. M., Requeima, J., Pyzer-Knapp, E. O., and Aspuru-Guzik, A. (2017). Parallel and distributed thompson sampling for large-scale accelerated exploration of chemical space. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Hinton, G. E. and van Camp, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory (COLT'93)*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*.

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., and Weinberger, K. Q. (2017). Snapshot ensembles: Train 1, get M for free. In *International Conference on Learning Representations (ICLR'17)*.

Hutter, F., Hoos, H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*.

Hutter, F., Hoos, H., and Leyton-Brown, K. (2014a). An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31th International Conference on Machine Learning (ICML'14)*.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*.

Hutter, F., Kotthoff, L., and Vanschoren, J., editors (2018). *Automatic Machine Learning: Methods, Systems, Challenges*. Springer.

Hutter, F., Xu, L., Hoos, H., and Leyton-Brown, K. (2014b). Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*.

Ilg, E., Cicek, O., Galesso, S., Klein, A., Makansi, O., Hutter, F., and Brox, T. (2018). Uncertainty estimates for optical flow with multi-hypotheses networks. In *The European Conference on Computer Vision (ECCV'18)*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. (2017). Population based training of neural networks. *arXiv:1711.09846 [cs.LG]*.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS'16)*.

Jones, D., Schonlau, M., and Welch, W. (1998). Efficient global optimization of expensive black box functions. *Journal of Global Optimization*.

Jones, D. R. (2001). A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*.

Kandasamy, K., Dasarathy, G., Oliva, J., Schneider, J., and Poczos, B. (2016). Gaussian process optimisation with multi-fidelity evaluations. In *Proceedings of*

the 29th International Conference on Advances in Neural Information Processing Systems (NIPS'16).

Kandasamy, K., Dasarathy, G., Schneider, J., and Póczos, B. (2017). Multi-fidelity bayesian optimisation with continuous approximations. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17).*

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. (2018). Neural architecture search with Bayesian optimisation and optimal transport. In *Proceedings of the 31th International Conference on Advances in Neural Information Processing Systems (NIPS'18).*

Kendall, A. and Gal, Y. (2017). What uncertainties do we need in Bayesian deep learning for computer vision? In *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NIPS'17).*

Khan, M., Nielsen, D., Tangkaratt, V., Lin, W., Gal, Y., and Srivastava, A. (2018). Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18).*

Kingma, D. and Welling, M. (2014). Auto-encoding variational bayes. In *International Conference on Learning Representations (ICLR'14).*

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR'15).*

Kingma, D. P., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. In *Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems (NIPS'15).*

Kiureghian, A. D. and Ditlevsen, O. (2009). Aleatory or epistemic? does it matter? *Structural Safety.*

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017a). Fast Bayesian hyperparameter optimization on large datasets. In *Electronic Journal of Statistics.*

Klein, A., Falkner, S., Mansur, N., and Hutter, F. (2017b). Robo: A flexible and robust bayesian optimization framework in python. In *NIPS Workshop on Bayesian Optimization (BayesOpt'17).*

Klein, A., Falkner, S., Springenberg, J. T., and Hutter, F. (2017c). Learning curve prediction with Bayesian neural networks. In *International Conference on Learning Representations (ICLR'17).*

Klein, A. and Hutter, F. (2019). Tabular benchmarks for joint architecture and hyperparameter optimization. *arXiv:1905.04970 [cs.LG].*

Kohavi, R. (1996). Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96).*

Komer, B., Bergstra, J., and Eliasmith, C. (2014). Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML 2014 AutoML Workshop.*

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS'12)*.

Krueger, T., Panknin, D., and Braun, M. (2015). Fast cross-validation via sequential testing. *Journal of Machine Learning Research*.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. In *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NIPS'17)*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (2001). Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*. IEEE Press.

Lévesque, J., Gagné, C., and Sabourin, R. (2016). Bayesian hyperparameter optimization for ensemble learning. In *Proceedings of the 32nd Conference on Uncertainty in Artificial Intelligence (UAI'16*.

Li, C., Chen, C., Carlson, D. E., and Carin, L. (2016). Preconditioned stochastic gradient langevin dynamics for deep neural networks. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'16)*.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *International Conference on Learning Representations (ICLR'17)*.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, K., Hardt, M., Recht, B., and Talwalkar, A. (2018). Massively parallel hyperparameter tuning. *arXiv:1810.05934 [cs.LG]*.

Lichman, M. (2013). UCI machine learning repository.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018). Progressive neural architecture search. In *The European Conference on Computer Vision (ECCV'18)*.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR'19)*.

Loshchilov, I. and Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. In *ICLR 2016 Workshop Track*.

Loshchilov, I. and Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations (ICLR'17)*.

Ma, Y., Chen, T., and Fox, E. B. (2015). A complete recipe for stochastic gradient MCMC. In *Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems (NIPS'15)*.

MacKay, D. J. C. and Neal, R. M. (1994). Automatic relevance detection for neural networks. Technical report, University of Cambridge.

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Mandt, S., Hoffman, M. D., and Blei, D. M. (2017). Stochastic gradient descent as approximate bayesian inference. *Journal of Machine Learning Research*.

Marchant, R. and Ramos, F. (2012). Bayesian optimisation for intelligent environmental monitoring. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Matérn, B. (1960). Spatial variation. *Meddelanden fran Statens Skogsforskningsinstitut*.

Melis, G., Dyer, C., and Blunsom, P. (2018). On the state of the art of evaluation in neural language models. In *International Conference on Learning Representations (ICLR'18)*.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH'10)*.

Minka, T. P. (2001). Expectation propagation for approximate Bayesian inference. In *Proceedings of the 30th conference on Uncertainty in Artificial Intelligence (UAI'01)*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*.

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of Bayesian methods for seeking the extremum. *Towards Global Optimization*.

Moré, J. J. and Wild, S. M. (2009). Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*.

Murphy, K. P. (2013). *Machine learning : a probabilistic perspective*. MIT Press.

Nagapetyan, T., Duncan, A. B., Hasenclever, L., Vollmer, S. J., Szpruch, L., and Zygalakis, K. (2017). The true cost of stochastic gradient langevin dynamics. *arXiv:1706.02692 [stat.ME]*.

Neal, R. M. (1996). Bayesian learning for neural networks. *PhD thesis, University of Toronto*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*.

Nickson, T., Osborne, M. A., Reece, S., and Roberts, S. (2014). Automated machine learning on big data using stochastic algorithm tuning. *arXiv:1407.7969 [stat.ML]*.

Osband, I., Blundell, C., Pritzel, A., and Roy, B. V. (2016). Deep exploration via bootstrapped dqn. In *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NIPS'16)*.

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

Perrone, V., Jenatton, R., Seeger, M., and Archambeau, C. (2018). Scalable hyperparameter transfer learning. In *Proceedings of the 31th International Conference on Advances in Neural Information Processing Systems (NIPS'18)*.

Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. (2018). Efficient neural architecture search via parameters sharing. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*.

Poloczek, M., Wang, J., and Frazier, P. (2016). Warm starting bayesian optimization. In *Proceedings of the 2016 Winter Simulation Conference*.

Poloczek, M., Wang, J., and Frazier, P. (2017). Multi-information source optimization. In *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NIPS'17)*.

Rana, P. S. (2013). Physicochemical properties of protein tertiary structure data set.

Rasmussen, C. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. The MIT Press.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Conference on Artificial Intelligence (AAAI'19)*.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Riquelme, C., Tucker, G., and Snoek, J. (2018). Deep Bayesian bandits showdown. In *International Conference on Learning Representations (ICLR'18)*.

Robbins, H. (1956). An empirical bayes approach to statistics. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability*.

Schaarschmidt, M., Kuhnle, A., and Fricke, K. (2017). Tensorforce: A tensorflow library for applied reinforcement learning.

Schaul, T., Zhang, S., and LeCun, Y. (2013). No More Pesky Learning Rates. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv:1707.06347 [cs.LG]*.

Seabold, S. and Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*.

Shahriari, B., Swersky, K., Wang, Z., Adams, R., and de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*.

Siebert, J. P. (1987). *Vehicle Recognition Using Rule Based Methods*. Turing Institute.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS'12)*.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, and Adams, R. (2015). Scalable Bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Sobol, I. M. (1967). Distribution of points in a cube and approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*.

Sobol, I. M. (1993). Sensitivity estimates for nonlinear mathematical models. *Mathematical Modeling and Computational Experiment*.

Sollich, P. (2001). Gaussian process regression with mismatched models. In *Proceedings of the 14th International Conference on Advances in Neural Information Processing Systems (NIPS'01)*.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. (2016). Bayesian optimization with robust bayesian neural networks. In *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NIPS'16)*.

Srinivas, N., Krause, A., Kakade, S., and Seeger, M. (2010). Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*.

Storn, R. and Price, K. (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*.

Sun-Hosoya, L., Guyon, I., and Sebag, M. (2018). Lessons learned from the AutoML challenge. In *Conférence sur l'Apprentissage Automatique 2018*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS'14)*.

Swersky, K., Snoek, J., and Adams, R. (2013). Multi-task Bayesian optimization. In *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems (NIPS'13)*.

Swersky, K., Snoek, J., and Adams, R. (2014). Freeze-thaw Bayesian optimization. *arXiv:1406.3896 [stat.ML]*.

Thornton, C., Hutter, F., Hoos, H., and Leyton-Brown, K. (2013). Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*.

Tieleman, T. and Hinton, G. (2012). RmsProp: Divide the gradient by a running average of its recent magnitude.

Titsias, M. and Lawrence, N. (2010). Bayesian Gaussian process latent variable model. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS'10)*.

Tsanas, A., Little, M. A., McSharry, P. E., and Ramig, L. O. (2010). Accurate telemonitoring of parkinson's disease progression by noninvasive speech tests. *IEEE Transactions on Biomedical Engineering*.

van Rijn, J. and Hutter, F. (2018). Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*.

Vanschoren, J. (2018). Meta-learning. In *AutoML: Methods, Sytems, Challenges*. Springer.

Vanschoren, J., van Rijn, J., Bischl, B., and Torgo, L. (2014). OpenML: Networked science in machine learning. *SIGKDD Explorations*.

Villemonteix, J., Vazquez, E., and Walter, E. (2008). An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*.

Volpp, M., Fröhlich, L., Doerr, A., Hutter, F., and Daniel, C. (2019). Meta-learning acquisition functions for bayesian optimization. *arXiv:1904.02642 [stat.ML]*.

Wang, J., Xu, J., and Wang, X. (2018). Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning. *arXiv:1801.01596 [cs.CV]*.

Welling, M. and Teh, Y. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*.

Williams, B., Santner, T., and Notz, W. (2000). Sequential design of computer experiments to minimize integrated response functions. *Statistica Sinica*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*.

Wu, A., Nowozin, S., Meeds, E., Turner, R. E., Hernández-Lobato, J. M., and Gaunt, A. L. (2019). Deterministic variational inference for robust bayesian neural networks. In *International Conference on Learning Representations (ICLR'19)*.

Ying, C., Klein, A., Real, E., Christiansen, E., Murphy, K., and Hutter, F. (2019). NAS-Bench-101: Towards reproducible neural architecture search. *arXiv:1902.09635 [cs.LG]*.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR'17)*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*.