FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Model Performance Approximation in Hyper-parameter Optimization
Student:	Bc. Markéta Jůzlová
Supervisor:	Ing. Tomáš Borovička
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

Hyper-parameter optimization methods try to find, in an effective way, an optimal configuration of a machine learning algorithm. However, a large amount of models need to be trained to select a configuration that maximizes predictive performance. Despite the growth of computational power, training a model is still an expensive operation. One of the approaches to deal with the high cost of training a model is to approximate performance of a given configuration without actually training the model.

1) Review and theoretically describe state of the art approaches for hyper-parameter optimization; focus on methods that approximate configuration and model performance.
2) Use or implement at least three of the reviewed methods and experimentally compare their performance on various data sets. Avoid implementing anew those methods that can be easily taken over from available implementations.
3) Propose a direction for further improvement of reviewed hyper-parameter optimization methods.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 6, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Model Performance Approximation in Hyper-parameter Optimization

Bc. Markéta Jůzlová

Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Borovička

May 8, 2018

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Tomáš Borovička for the mentorship of this thesis and Datamole, s.r.o. for providing computational resources. Also, I would like to thank my friends who provided advice and emotional support during writing this thesis. Finally, I would like to thank my family for all the support they gave me during my whole studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 8, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Jůzlová, Markéta. *Model Performance Approximation in Hyper-parameter Optimization.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem automatické optimalizace hyper-parametrů je najít nastavení hyper-parameterů učícího algorithm bez lidské pomoci. Protože k vyhodnocení jednoho nastavení je potřeba natrénovat daný model, optimalizační metody které se snaží redukovat počet vyhodnocení jsou třeba. Užitečná technika jsou takzvané náhradní modely, které aproximují přesnost modelu s danou konfigurací.

Tato práce zkoumá některé postupy optimalizace hyper-parameterů. Mezi popsané metody patří dvě tradiční methody: mřížková optimalizace a náhodná optimalizace, a dvě nejpokročilejší metody: sekvenční optimalizace založená na náhradním modelu (Bayesovská optimalizace) a Hyperband. Dále je popsáno několik náhradních modelů, které mohou být použity ke zlepšení optimalizace. Efektivita optimalizace a přesnost náhradních modelů je porovnána na dvou datasetech s různým stupněm obtížnosti a algoritmu dopředných umělých neuronových sítí. Výsledky ukazují, že Hyperband dosahuje nejlepších výsledků na obouch datasetech. Analýza výsledků také potvrzuje, že náhradní modely směřují hledání do slibných oblastí a tím urychlují optimalizaci.

Klíčová slova optimalizace hyper-parametrů, náhradní modely, approximace přesnosti modelů, optimalizace s náhradním modelem, Bayesovská optimalizace, Hyperband

Abstract

Automatic hyper-parameter optimization aims to tune hyper-parameters of machine learning algorithms without human effort. Due to necessity to learn a model to evaluate a configuration, optimization methods that avoid excessive amount of evaluations are desired for the task. A useful technique is to employ a surrogate model which approximates performance of the trained model with given configuration.

This thesis reviews some of the approaches that are being used for the hyper-parameter optimization. The described methods include two traditional methods: grid search and random search as a baseline, and two state-of-the-art techniques: sequential model-based optimization (Bayesian optimization) and Hyperband. Several surrogate models that can be used to improve the optimization are described. The performance of the methods and the surrogate models is compared using two datasets of different complexity and a feed-forward artificial neural network as the machine learning algorithm. On both tasks, Hyperband outperforms the other methods. The analysis also confirms that the surrogate models positively bias the search to promising regions and, thus, speed up the optimization.

Keywords hyper-parameter optimization, surrogate modeling, performance approximation, model-based optimization, Bayesian optimization, Hyperband

Contents

List of Figures

List of Tables

Introduction

The purpose of machine learning algorithms is finding a general solutions for complex problems using just a set of sample observations of the problem with known solutions. This is achieved by learning a model which captures information from the observations and uses them to approximate the problem. Variety of machine learning algorithms were proposed in recent years. These algorithms have proven to handle a large number of complex problems like object recognition in the image or prediction of market prices. However, flexibility often comes with a price. In order to be able to train different models for different tasks, a machine learning algorithm has some free parameters that need to be set before training the model. These parameters have often significant impact on algorithm behavior and in consequence on the performance of the trained model. We refer to those parameters as hyper-parameters and the process of identifying their optimal values as hyper-parameter optimization.

To determine whether the learning algorithm with given hyper-parameter configuration produces a good model, we need to actually train the model using this algorithm. Training a model is typically an expensive operation which requires a lot of computational resources. Therefore, we want to avoid training excessive amount of models. This prompts an interest in advanced and efficient hyper-parameter optimization methods.

Traditionally, hyper-parameter optimization is performed by a human expert. The expert selects hyper-parameter configuration based on his knowledge of the problem, the machine learning algorithm and largely on his experience. The expert trains the model using the algorithm with the selected configuration. If the performance of the trained model is not sufficient, the expert has to select a new configuration and train the model again with this configuration. This procedure is repeated until the configuration with sufficient performance is found.

Manual search is usually quite an efficient way for hyper-parameter optimization. However, it requires manual effort of the human expert. It can be very difficult for a non-expert user to find a good hyper-parameter configu-

ration. This limits the application of machine learning algorithms to a small group of experts which is not desirable. The idea behind machine learning is that the machine learns from the experience provided in the form of data and does not rely on expert knowledge and intuition.

Methods for automatic hyper-parameter optimization aim to select a hyper-parameter configuration without human assistance. Many optimization methods were proposed, ranging from the simpler methods like grid search and random search to more sophisticated methods that use information gathered during previous training to select next tested configuration.

One of the ways how to improve the optimization is to build a model, called surrogate model, that approximates the performance of the original model depending on its hyper-parameters. The surrogate model can then be used to guide the optimization and reduce the number of trained models.

There are many surrogate models for hyper-parameter optimization. The models differ in the type of returned value (point estimate or distribution over possible values), construction method, time and memory requirements, a number of hyper-parameter configurations which is needed to approximate the performance well and many other properties.

The aim of this work is to compare different methods for hyper-parameter optimization with a focus on methods which use model performance approximation. Moreover, different models for performance approximation will be described and compared.

The work is organized as follows. In theoretical part (Chapter 1), hyper-parameter optimization problem is formally described in Section 1.1. Section 1.2 examines the methods for hyper-parameter optimization. The models for performance approximation are described in Section 1.3. Section 1.4 covers other aspects which need to be considered before optimizing the hyper-parameters. Experimental part in Chapter 2 focuses on methods comparison. Section 2.1 contains experiments structure and specification. Short description of the implementation is given in 2.2. Results of the experiments and their analysis can be found in Section 2.3. Chapter 3 contains discussion over the results along with proposal to improvement. We conclude by summarization of the work and achieved results.

Chapter 1

Theoretical part

1.1 Hyper-parameter optimization

The aim of machine learning task Π is to use the set of observations Π_{train} and machine learning algorithm $\mathcal{A}$ to train a model which minimizes expected loss.

The loss is defined as real-valued function $\mathcal{L}$ that quantifies the model performance on given task for observation π. This loss function is constructed to reflect a regret of inaccurate prediction. For example, if we use the trained model to predict house price, the loss might be the difference between predicted price and real price for which the house was sold. We wish to minimize the difference between the prices for all houses that can appear on the market, i.e. the expected loss for all possible observations. We use a machine learning algorithm to train such a model.

The machine learning algorithm $\mathcal{A}$ maps training set of the observations Π_{train} to the model. The algorithm behavior is parameterized by the set of d hyper-parameters from hyper-parameter space $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_d$. The hyper-parameter configuration $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_d \end{bmatrix}^T$ can significantly influence the trained model and in consequence the expected loss. Example of learning algorithm can be an artificial neural network. The hyper-parameters of this algorithm are the number of hidden units, type of activation functions, or a learning rate.

The hyper-parameter value $x_i \in \mathcal{X}_i$ can be of any type. It can be continuous (e.g. learning rate for neural network), discrete (number of layers) or categorical (activation function). In some cases, its relevance is conditioned by the value of another hyper-parameter (number of units in the second layer is conditioned by the existence of the second layer).

The objective of hyper-parameter optimization is to find hyper-parameter configuration $\mathbf{x}_*$ that minimizes the expected loss $\mathcal{L}$ for task Π and algorithm

$\mathcal{A}$. Formally we want to find $\mathbf{x}_*$ such as

$$\mathbf{x}_* = \arg\min_{\mathbf{x}\in\mathcal{X}} \mathbb{E}_{\pi\sim\mathcal{D}_\pi} \mathcal{L}(\pi; \mathcal{A}_\mathbf{x}(\Pi_{train}))$$

where $\mathcal{D}_\pi$ is distribution from which the observations are sampled. This distribution is given by nature of the problem and it is usually unknown.

To solve this problem we define a cost function $f : \mathcal{X} \to \mathbb{R}$

$$f(\mathbf{x}) = \mathbb{E}_{\pi\sim\mathcal{D}_\pi} \mathcal{L}(\pi; \mathcal{A}_\mathbf{x}(\Pi_{train})).$$

The hyper-parameter optimization problem can be now written in a form

$$\mathbf{x}_* = \arg\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})$$

Function f is a black-box function. An explicit formula is not known and we do not have access to its derivation. We can only evaluate its value at given point $\mathbf{x}$.

In order to evaluate f, the model must be trained first. Therefore, it is usually an expensive operation and we want to minimize the number of evaluations. Moreover, the evaluation is noisy, because we cannot evaluate the expectation over unknown distribution $\mathcal{D}_\pi$. Instead, we access a limited set of observations Π_{valid} called validation set and approximate expected loss and our cost function with mean over this validation set:

$$y = \frac{1}{|\Pi_{valid}|} \sum_{\pi\in\Pi_{valid}} \mathcal{L}(\pi; \mathcal{A}_\mathbf{x}(\Pi_{train})) \approx f(\mathbf{x}).$$

The value of y is called validation loss.

For hyper-parameter optimization, this corresponds with cost function evaluation given by:

$$y = f(\mathbf{x}) + \epsilon.$$

Noise value ϵ is assumed to come from Gaussian distribution with zero mean and variance σ, $\epsilon \sim \mathcal{N}(0, \sigma)$. Moreover, the noise for one evaluation is assumed to be independent on other evalations.

We can now treat the hyper-parameter optimization problem as a typical global optimization problem of f with properties described above.

Note that we seek for minimum of the function. However, we could easily to transform our minimization problem to maximization problem by tranforming the cost function (for example we could use negative value of cost).

Hyper-parameters vs. model paramaters The core of the learning procedure often lies in optimizing parameters of a family of functions to fit the training data Π_{train} via learning algorithm $\mathcal{A}_\mathbf{x}$. For example, we can take linear functions defined as $l(\pi) = a_0 + \pi_1 a_1 + \ldots + \pi_m a_m$, where π_i

are individual components of π, and optimize the parameters $a_0, a_1, \ldots, a_m$ using $\mathcal{A}_{\mathbf{x}}$. These parameters of the model are called *model parameters* [1]. It is important distinguish them from *hyper-parameters*. The former refers to the parameters whose optimization is a part of the learning procedure, the latter has to be set before the learning. Thus, the learning represents the "inner loop" optimization meanwhile the hyper-parameter optimization is the "outer loop" optimization [2]. The objective of this work is hyper-parameter optimization, thus the outer loop.

Related terms In the literature, we can encounter interchangeable terms such as *hyper-parameter optimization* and *algorithm configuration selection* [3]. Similarly to hyper-parameter optimization, algorithm configuration selection refers to selecting values for free parameters of an algorithm. However, algorithm configuration is used for an arbitrary algorithm, meanwhile, the term hyper-parameter optimization is established for parameters of the machine learning algorithm. Another closely related term is *algorithm selection*, also referred to as *model selection*. The term refers to the selection of a machine learning algorithm $\mathcal{A}$ from the set of algorithms. Hyper-parameter optimization and algorithm selection are often denoted and solved together due to their similarity and close relationship [1]. Actually, the algorithm itself can be considered as categorical hyper-parameter in join space algorithms and their hyper-parameters. Moreover, the same or similar methods are used to solved both tasks. The focus of this work is only on hyper-parameter optimization.

1.1.1 Notation

The literature for hyper-parameter optimization is missing any established notation convention. The notation introduced above will be used in the rest of the text unless stated otherwise. The symbols which will be used in most of the text are stated here and their list is provided in Appendix A. Other notation specific for individual topics will be defined in relevant sections.

The hyper-parameters are indexed with natural numbers from 1 to d. For hyper-parameter i we define a space $\mathcal{X}_i$, a value x_i, $x_i \in X_i$, and a set of values as X_i, $X_i \subset \mathcal{X}_i$.

The entire hyper-paramater space is denoted as $\mathcal{X}$, $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_d$. For a hyper-paramater setting (configuration) we will use vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix},$$

$\mathbf{x} \in \mathcal{X}$.

An index set with n hyper-parameter configurations is referred to as $\mathrm{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and $n = |\mathrm{X}|$. In some cases it will be useful to have X in a matrix form

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,n} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \ldots & x_{d,n} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \ldots & \mathbf{x}_n \end{bmatrix}$$

where $x_{i,j}$ is the value of hyper-parameter i in configuration j.

The value of the cost function for the hyper-parameter configuration $\mathbf{x}$ is denoted by $f(\mathbf{x})$. For the indexed configuration $\mathbf{x}_i$ we can shortly write $f_i = f(\mathbf{x}_i)$. Function values for n indexed configurations will be jointly referred to as a vector

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}.$$

Similarly, we will write

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

for vector of noisy evaluations of f for n configurations. Evaluation y_i corresponds with the configuration $\mathbf{x}_i$ and therefore with the cost function value f_i.

We will refer to the pair of configuration and its evaluation as $(\mathbf{x}, y)$. An indexed set of these pairs forms a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n} = (\mathbf{X}, \mathbf{y})$.

1.2 Hyper-parameter optimization methods

A typical hyper-parameter optimization algorithm selects a subset of configurations $\mathrm{X} \subset \mathcal{X}$, evaluates a cost function f for all configurations from X, and returns the best configuration $\mathbf{x}_{best} \in \mathrm{X}$.

In order to select the best configuration for a given set, we must define what is best configuration from given set. A natural choice is a configuration with the smallest observed value of the cost function f. In other words, it is the configuration with the smallest validation loss.

Selecting the subset of configurations that will be evaluated (X) is more difficult task. We aim to minimize the size of X, to avoid too many evaluations. Moreover, we want to minimize the validation loss of the best configuration $\mathbf{x}_{best} \in \mathrm{X}$ so that it was as close to the optimal loss as possible.

Various methods have been proposed in the literature. Probably, the most widespread method in practice is manual search and manual search combined with simple optimization method as grid search, coordinate descent, etc. [4]. The popularity of manual search is given mainly by the absence of technical restrictions and a good trade-off between accuracy of the resulting model and the number of trained models. Moreover, manual search provides the user with some degree of insight to f. However, it is difficult to reproduce its results which makes it difficult to apply, particularly, by users with limited knowledge of the model and training algorithm. Automated hyper-parameter optimization methods aim to overcome the difficulties in the application of the search. The description of four such methods follows.

1.2.1 Grid search

Grid search is one of the most commonly used approaches for automatic hyper-parameter optimization. The idea is to select a set of values for each hyper-parameter and then test every possible combination of these values. More formally, if we select the set of values X_i for each hyper-parameter $i = 1, \ldots, d$, we get the set of configurations $\mathrm{X} = X_1 \times X_2 \times \ldots \times X_d$. We train the model with all the configurations $\mathbf{x} \in \mathrm{X}$ and then select the best.

The choice of X_i depends on the space of values and meaning of the hyper-parameter i. Usually, we take the entire set $\mathcal{X}_i$ for categorical hyper-parameters and sample $\mathcal{X}_i$ uniformly or exponentially (geometrically) for numerical hyper-parameters.

The size of the selected configuration set is $|\mathrm{X}| = \prod_{i=1}^{d} |X_i|$ which makes grid search to suffer from the curse of dimensionality. For example, if we select 5 distinct values for each hyper-parameter then for 1 hyper-parameter we test 5 configurations, for 2 hyper-parameters 25 configurations and for 5 it is 3125 distinct configurations. With higher number of hyper-parameters, the training of such amount of models can be computationally unfeasible.

If we look at the cost function f from the perspective of each input dimension, f might vary more for some dimensions than for the others. We say that f has *low effective dimensionality*. The low effective dimensionality means that f is more sensitive to values of some hyper-parameters and we should use more unique values of these hyper-parameters to identify good ones. However, the grid will cover only a finite number ($|X_i|$) of distinct values for each hyper-parameter. Figure 1.1 shows the projection of a 2-dimensional function with low effective dimensionality to each dimension. The left square demonstrates a projection of the grid sampling. We can see that due to insufficient sampling in the important dimension we miss significant changes in the

function values. The solution could be to increase the size of X_i for important hyper-parameters and reduce it for the others. Unfortunately, the importance of each hyper-parameter is typically not well-known in advance. Even for the same algorithm and hyper-parameters, different hyper-parameters are important for different tasks Π [2].

Despite the negatives, grid search is a widely used technique mainly due to its conceptual and implementation simplicity and easy parallelization (see Section 1.4.2). It is a reliable choice for problems with low dimensionality (1-d, 2-d) [2].

1.2.2 Random search

Random search generates configurations randomly and independently from the each other. The hyper-parameter configuration $\mathbf{x}$ is taken as a random vector with pre-defined distribution $\mathcal{D}_{\mathbf{x}}$, and the individual configurations $\mathbf{x}_j, j = 1, \ldots, |\mathrm{X}|$ are i.i.d. samples drawn from $\mathcal{D}_{\mathbf{x}}$.

The implementation of random search allows updating a set of tested configurations X during the algorithm's runtime. We can add or remove the configurations from X based on the current results, changes in the computational resources (e.g. node in the distributed system fails), etc.

The right panel in Figure 1.1 shows how random search deals with low effective dimensionality. In constrast with grid search, random search does not limit the number of unique values for individual hyper-parameters. Therefore, variations in function values depending on the effective dimension might be better covered by the sampling.

Despite the conceptual simplicity, random search is a strong method for hyper-parameter tunnig. The empirical results show superiority of random search over grid search [2]. It is a widely used method and often serves as a baseline for other hyper-parameter optimization methods.

1.2.3 Sequential model-based optimization

Sequential model-based optimization (SMBO) selects tested configurations adaptively based upon a performance of already evaluated configurations.

We assume that the models teached by the algorithms with similar configurations will have similar performance. Therefore, we can use results of previous configurations to build a surrogate model which predicts the performance of other configurations. When deciding which configuration will be evaluated next, we examine the predicted performance and amount of information the evaluation will give us. Naturally, we aim to select a configuration with good predicted performance. Furthermore, we want to get a maximum of new information to improve the accuracy of the surrogate model and in consequence to support the next decision with more accurate prediction.

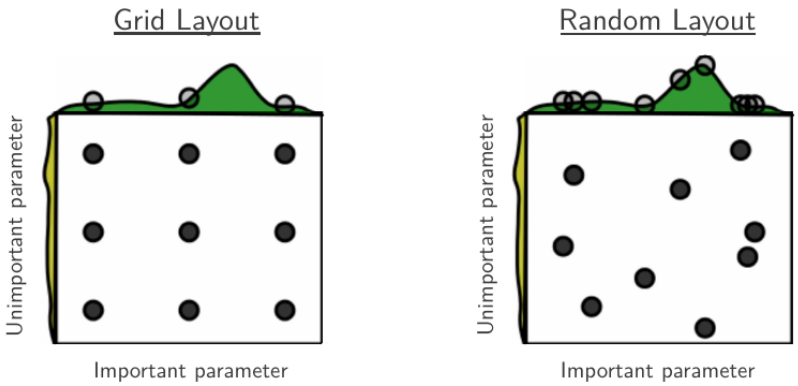


Figure 1.1: Function $f(x, y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above and on the left side of each square we can see a projection of f to each input dimension. Grid sampling in the left figure covers only 3 distinctive values for each dimension and does not capture f well. On the other hand, random sampling in the right panel takes several distinctive values for each dimension. It is not important that values in both dimension differ, because we can write $f(x_1, y_1) \approx g(x_1)$ and $f(x_1, y_2) \approx g(x_1)$, thus the value of y does not have a great impact on f [2].

1.2.3.1 General framework

SMBO is built around a regression model $\mathcal{M}$, called *surrogate model*, that approximates the optimized function. The model $\mathcal{M}$ accepts the parameters of a function f as an input and outputs prediction of function value. In hyper-parameter optimization, the function parameters are hyper-parameters and $\mathcal{M}$ approximates the performance of the trained model depending on the hyper-parameters of the machine learning algorithm. The training dataset D which is used to learn $\mathcal{M}$ is formed by pairs $(\mathbf{x}, y)$ of already evaluated configurations.

The surrogate model $\mathcal{M}$ helps us decide which configuration to evaluate, i.e. which configuration should be added to X. Specifically, we select the configuration that maximizes the so-called *acquisition function* $a(\mathbf{x}; \mathcal{M})$. The acquisition function takes surrogate model $\mathcal{M}$ and configuration $\mathbf{x}$ as an input and returns utility of $\mathbf{x}$ for the next evaluation. Typically, a is constructed to balance exploration and exploitation [1]. Thus $\mathbf{x}$ with a great value of the acquisition function does not have to necessary have a good predicted value of f, however, its evaluation can give us a lot of new information.

[1]The terms 'exploration' and 'exploation' refer to competing approaches which occur in global optimization. Exploration gathers more information from the entire space (explores the space), meanwhile exploitation looks closer to given region (focuses on the promising area). Typically, we need to find a good trade-off between these approaches to find the global optimum.

The pseudocode of SMBO is given in Algorithm 1. At first, the initial dataset D_0 is created, i.e. some configurations are sampled and evaluated. There are several methods how to select initial configurations. Their short overview is given in Section 1.4.1. The main optimization loop starts in Line 2. The algorithm iterates over four steps: building a surrogate model, maximization of the acquisition function, evaluation of the cost function and adding observed results to the dataset. The addition of pair $(\mathbf{x}_k, y_k)$ to dataset D_{k-1} in last step results in a new dataset D_k and in consequence the neccessity to build a new model $\mathcal{M}_{k+1}$. Figure 1.2 shows an illustration of the optimization loop in SMBO for three iterations. The algorithm terminates when a stopping condition is met. We can limit the number of iterations, the time of the optimization or we can stop the optimization loop if the performance of the trained model is sufficient. Algorithm 1 terminates the loop after n iterations. The configurations $\mathbf{x}_j,\ j = 1, ..., n$ form the set of selected configurations $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. In the final step in accordance with our general hyper-parameter optimization framework, we select the best configuration from X.

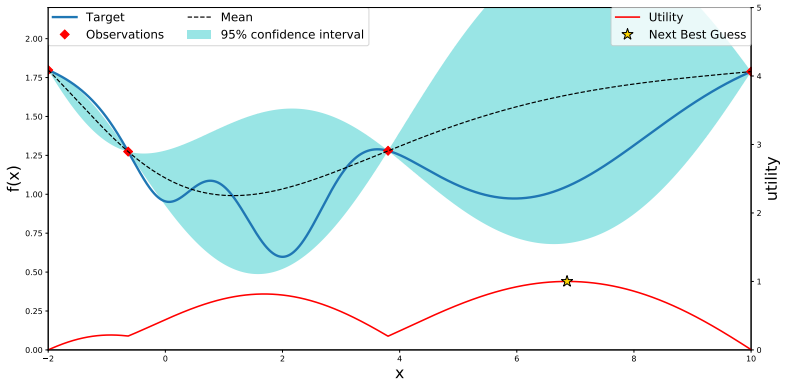
Algorithm 1 Sequential model-based optimization

1: Initialize dataset D_0
2: **for** $k = 1, 2, ..., n$ **do**
3: Build model $\mathcal{M}_k$ using dataset D_{k-1}.
4: Find $\mathbf{x}_k = \arg\max_{\mathbf{x}\in\mathcal{X}} a(\mathbf{x}; \mathcal{M}_k)$.
5: Evaluate the cost function at a point $\mathbf{x}_k$.
6: Add $\mathbf{x}_k$ and its observed cost y_k to dataset $D_k = D_{k-1} \cup \{(\mathbf{x}_k, y_k)\}$.
7: **end for**
8: Return configuration $\mathbf{x}_{best}$ from D_n with the lowest best cost.

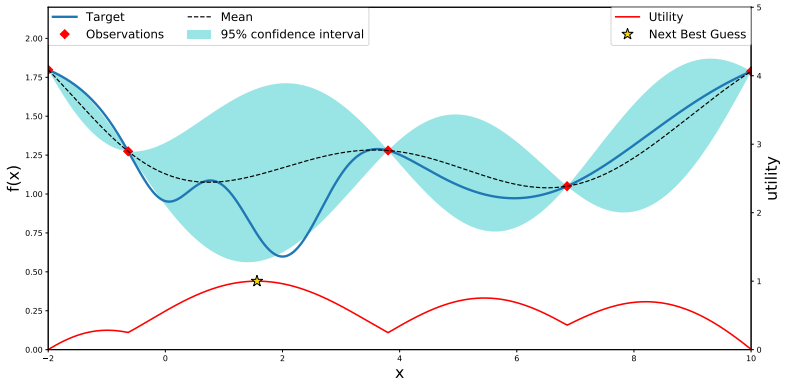
1.2.3.2 Surrogate model

A surrogate model $\mathcal{M}$ is a regression model learned using an algorithm $\mathcal{A}'$ (different from the algorithm $\mathcal{A}$ whose hyper-parameters we optimize) to approximate cost function f.

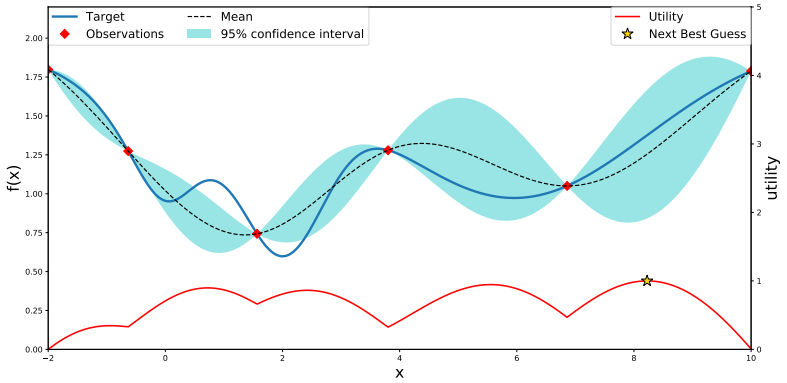
The data points used for training are already evaluated configurations. In each iteration, a new data point is added to the dataset and the model has to be trained again. Thus, when selecting the surrogate model $\mathcal{M}$ and the algorithm $\mathcal{A}'$ we need to consider the time overhead caused by the surrogate training. Nevertheless, as long as the time required to learn $\mathcal{M}$ is negligible in comparison to one evaluation of f, we can justify selection of a more expensive and more accurate $\mathcal{M}$. Moreover, the number of training points equals to the number of evaluated configurations which means we have tens, maximum of hundreds of points. The learning using such a small dataset is not an expensive operation for most of the training algorithms.



(a) k = 4



(b) k = 5



(c) k = 6

Figure 1.2: Illustration of sequential model-based optimization over three iterations. The objective is to minimize a target function f with one input parameter x. SMBO builds a surrogate model that approximates the target function based on the observations. The plot shows distribution over f, visualized by mean value and 95% confidence interval, predicted using a probabilistic model of the target function. Utility (acquisition) function returns a utility of each point for evaluation. The utility balance exploitation, given by the mean value, and exploration, given by the uncertainty of the prediction. The point with the maximal utility is evaluated. In the next iteration, the surrogate model is built again to fit a new observation.

Most research is focus on probability models which return a distribution over the cost function value. The majority of these models use the Bayes approach to get the predictive posterior distribution given the evaluated configurations, $p(f|\mathbf{x}, \mathrm{D})$ ($p(x)$ represents a density of the distribution over random variable x). SMBO with such a model is called *Bayesian optimization*. Another type of a surrogate model is a model which returns only point prediction. Such model does not capture the uncertainty of the prediction, thus it is less informative. Sometimes, we are forced to make a point prediction even for probability models. If this occurs, we have to decide which value to take. The mean value is commonly used for model point prediction in SMBO.

Some models used as surrogate models are described in Section 1.3. In this section, we mention the surrogate models proposed for hyper-parameter optimization using SMBO.

The surrogate model has a great impact on performance of SMBO. Many models was proposed as surrogate model. A popular choice are Gaussian processes. The posterior distribution for Gaussian processes is Gaussian which results in calculations that are often analytically tractable. Moreover, Gaussian processes are flexible and can be customized to include knowledge from different tasks [5], or to use partial observations [6, 7]. Sequential model-based algorithm configuration (SMAC) [3] is SMBO which uses random forests as the surrogate. Bergstra *et al.* [8] proposed tree-structured parzen estimator (TPE). A tree-structured model used to get the posterior. Both models are designed specially for the algorithm configuration selection and they belong to the most used surrogate models for hyper-parameter optimization [1, 9, 10]. Neural network was suggested as the surrogate model in [11, 12]. Radial basis functions (RBFs) and support vector regression (SVR) are used as non-probabilistic models for surrogate modeling. Recently RBFs were used for hyper-parameter optimization in [10] and [13].

1.2.3.3 Acquisition functions

The acquisition function a determines the policy for selecting the next tested configuration. It measures the utility of evaluation for each configuration based on the surrogate model. The configuration with the largest utility is then evaluated.

An intuitive utility measure is the point prediction of the cost value. However, taking the configuration with the lowest predicted cost leads to convergence to the local optimum, while it leaves large areas unexplored [14]. Figure 1.3 shows an example of this approach. The area where the global minimum lies stays unexplored. There is no pressure to explore it due to the inaccurate model which predicts the high cost in the area.

The measure of utility based only on the predicted cost is a pure exploitation approach. We need to incorporate exploration to ensure that no promising area will be overlooked in our search. This is where the probability

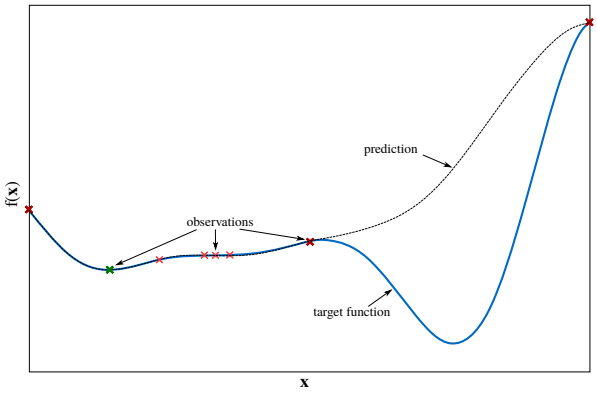


Figure 1.3: SMBO with acquisition function that assigns the highest utility to points with lowest estimated cost value. We start with an evaluation of three highlighted points: lower bound for $\mathbf{x}$ (the left point), upper bound (the right point) and in the middle of the interval. SMBO builds a model and evaluates other points. Points for evaluation are selected to have minimal predicted cost. The optimization ends with a green cross in a local minimum. The area where the global minimum lies stays unexplored due to an inaccurate prediction.

surrogates benefit from the distribution over the cost function value. The distribution naturally provides us with information about the uncertainty of the prediction. If some area remains unsampled, the uncertainty in this area will be high. Therefore, if we assign high utility to the points with high uncertainty, the area where these points lie will be explored. For non-probabilistic surrogates, we need to force exploration in a different way. See [10] for an example.

There is a rich literature on acquisition functions that aim to balance exploitation and exploration. Most of them focus on models which return the (Gaussian) distribution over f. Three of them which are often used for hyper-parameter optimization are described here and their behavior is illustrated in Figure 1.4. Probability of improvement and expected improvement belong to the so-called improvement-based policies. Lower confidence bound acquisition function is so-called optimistic policy. There are other approaches such as information-based acquisition functions [15] or acquisition function that incorporates other costs such as time of the training [5]. Some authors propose to take a portfolio of functions each of which provides a candidate for the evaluation. We then choose among these candidates based on a meta-criterion [16, 17].

Probability of improvement (PI) Probability of improvement measures a probability that a function value at a point $\mathbf{x}$ will be lower than a given

threshold τ:

$$a_{PI}(\mathbf{x}; \mathcal{M}) = \mathbb{E}_f \mathbb{1}(f < \tau) = p_{\mathcal{M}}(f < \tau),$$

where $\mathbb{1}$ is an indicator function of $f < \tau$. The distribution $p_{\mathcal{M}}(f < \tau)$ is given by the surrogate model $\mathcal{M}$. It comes from the posterior distribution of f at a point $\mathbf{x}$. A common distribution is Gaussian distribution for which PI holds:

$$a_{PI}(\mathbf{x}; \mathcal{M}) = \Phi\Big(\frac{\tau - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\Big),$$

where Φ is the cumulative distribution function of the standard Gaussian distribution. Functions $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})^2$ are mean and variance of the predictive posterior distribution.

There are various heuristics for the threshold value τ. However, in practice, it is hard to set τ to result in a good exploitation and exploration trade-off. Wrong choice of τ leads to aggressive exploitation and thus poor performance [15].

Expected improvement (EI) Expected improvement (EI), as well as PI, measures the probability of improvement, however, EI takes the amount of improvement into account. The acquisition function is defined as:

$$a_{EI}(\mathbf{x}; \mathcal{M}) = \mathbb{E}_f[(\tau - f)\mathbb{1}(f < \tau)] = \int_{-\infty}^{\infty} \max(\tau - f, 0) p_{\mathcal{M}}(f) df.$$

For the Gaussian distribution the formula is analytically tractable as:

$$a_{EI}(\mathbf{x}; \mathcal{M}) = (\tau - \mu(\mathbf{x}))\Phi\Big(\frac{\tau - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\Big) + \sigma(\mathbf{x})\phi\Big(\frac{\tau - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\Big),$$

where Φ is the cumulative distribution function of the standard Gaussian distribution and ϕ is the standard Gaussian density function (see [18] for derivation). An analytical formula can be derived for other distributions, as for example in [19].

As with PI, we need to set the threshold τ, however, EI is not overly sensitive to τ. In practise, a reasonable choice is to set τ adaptively to the current best cost function observation $\tau = y_{best}$.

Lower confidence bound (LCB) Lower confidence bound (or upper confidence bound for maximization problems) assumes that the function holds the lower bound value. Therefore, in the face of uncertainty is optimistic and assumes the best case scenario. The value is calculated as

$$a_{LCB}(\mathbf{x}; \mathcal{M}) = \mu(\mathbf{x}) - \kappa\sigma(\mathbf{x})$$

where the parameter $\kappa \geq 0$ is set by the user.

As with the other described functions, κ should be tuned to balance exploitation and exploration.

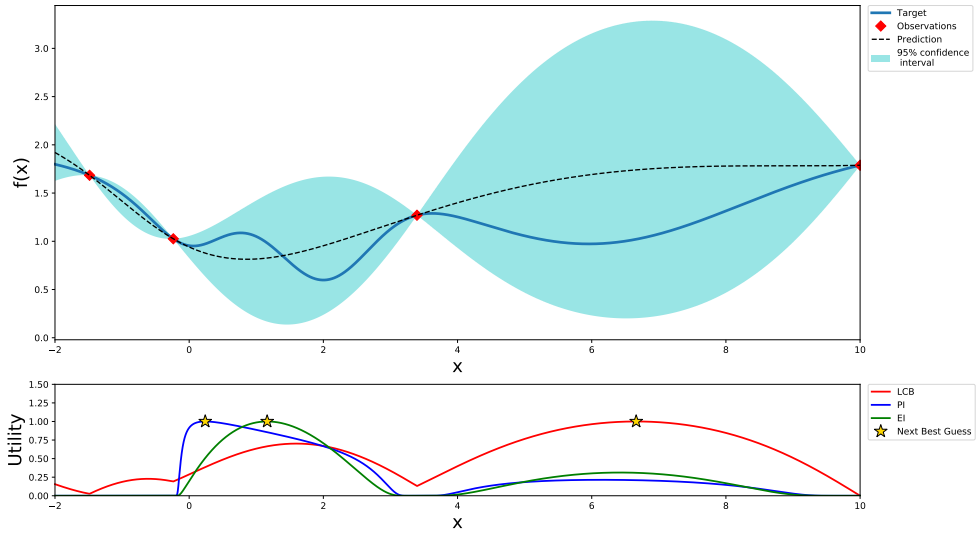


Figure 1.4: Illustration of three acquisition functions: lower confidence bound (LCB), probability of improvement (PI) and expected improvement (EI). Each acquisition function measure the utility differently and thus finds a different point for the evaluation.

1.2.3.4 Maximizing acquisition function

The objective of Step 4 in SMBO algorithm (1) is to find $\mathbf{x}$ which maximizes the acquisition function. However, the acquisition function is often multi-modal and maximization is not a trivial task. An exhaustive grid search or Latin hypercube search can be used for the task. In contrast with the optimized cost function, the acquisition function is cheap to evaluate thus we can afford many evaluations. Nevertheless, more efficient techniques were proposed to use in practice. In [2], the covariance matrix adaptation evolution strategy (CMA-ES) is used to optimize the acquisition function for continuous hyper-parameters and the estimation of distribution (EDA) for discrete hyper-parameters. Hutter *et al.* [3] combine a multi-start local search and a random sampling. Many other methods were suggested. See [15] for overview.

1.2.4 Hyperband

As with other hyper-parameter optimization methods, Hyperband selects a set of the configurations and returns the one with the lowest observed cost. However, unlike the other mentioned methods, Hyperband does not fully evaluate all selected configurations. Instead, each configuration get allocated a number of resources and is evaluated using only these resources resulting in partial observations. A partial observation means that the model is not fully trained using all available training data, thus the model might have poor performance.

The core of the Hyperband lies in Successive halving algorithm [20]. Successive halving takes a set of configurations X and a budget of resources B (e.g. time budget) as input. The algorithm evaluates each configuration from X using a given part of the budget and discards poorly-performing configurations. The remaining ones get allocated more resources from the budget for further training. We iteratively alternate between training and dropping until some models are fully trained.

The budget B is divided equally between iterations and all configurations receives an equal number of resources in the iteration. At the end of each iteration, a half of the configurations is discarded resulting in $B/|\mathrm{X}|$ resources on average for a configuration from initial set.

The budget is usually fixed (e.g. we have limited amount of time). However, we need to choose the number of tested configurations n, $n = |\mathrm{X}|$. We can either (a) consider a large number of configurations with a small number of resources or (b) consider fewer configurations with more resources. It is not usually known a priori which option will be better for a concrete task. If poorly-performing configurations can be distinguished with fewer resources, we should take a large n. On the other hand, for problems where the separation of good and poor configurations is difficult, we need to allocate more resources to the individual configurations.

Hyperband aims to resolve the configurations/resources problem by searching over different values of n. Thus, it consists of Successive halving run ("inner loop"), called bracket, and searching over different n ("outer loop"). In addition, Hyperband allows determining how many configurations will be dropped in each iteration which is a feature missing in the original design of Successive halving.

The algorithm has two parameters. A maximal number of resources R that can be allocated to one configuration (e.g. maximal training time), and a factor η that controls the proportion of configurations dropped in each iteration of Successive halving. Pseudocode of the algorithm is depicted in 2.

At first, we calculate the number of tested settings in outer loop, s_{max}, and the budget size B for each bracket (Line 2). We start the outer loop with the largest n corresponding with the greatest exploration, i.e. we test a large number of configurations with the small number of resources. The parameter r calculated in Line 4 is a minimal number of resources allocated to a configuration in the bracket.

We generate the set of configurations X_s with size n and evaluated them with Successive halving. Poorly-performing configurations are removed and only n/η configurations with the smallest observed cost are kept. In the next iteration, the remaining configurations are evaluated with more $(r\eta)$ resources. We proceed with the next iteration evaluating $n\eta^{-k}$ best configurations with $r\eta^k$ resources.

The outer loop sets different n and r in each iteration. The last loop $(s = 0)$ generates $s_{max} + 1$ configurations each of which is evaluated using all

Algorithm 2 Hyperband

Input R, η
Set $s_{max} = \lfloor \log_\eta(R) \rfloor$ and budget $B = (s_{max} + 1)R$
for $s = s_{max}, s_{max} - 1, \ldots, 0$ **do**
    Calculate $n = \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$ and $r = R\eta^{-s}$
    Generate n configurations X_{s0}
    for $k = 0, \ldots, s$ **do** ▷ Sucessive halving (bracket)
        Set number of resources for iteration $r_k = r\eta^k$
        Evaluate configurations in X_{sk} with r_k resources and get partial performance observations
        Set $n_{k+1} = \lfloor n\eta^{-k} \rfloor$
        Drop $n_k - n_{k+1}$ worst performing configurations from X_{sk}
    end for
end for
Return configuration $\mathbf{x}_{best}$ with best performance

resources R resulting in the classical random search.

The inputs R and η are user-defined. The maximal number of resources is usually naturally limited and this limitation is used to set the value of R. The setting of factor η is less straightforward. Great η results in an aggressive elimination of configurations and fewer iterations in both loops. The authors of Hyperband [9] stress that Hyperband is not very sensitive to the choice of η. The recommended value in practice is 3 or 4.

1.2.4.1 Resource types

As mentioned before, Hyperband allocates resources from the pre-defined budget for each configuration. The more resources, the more accurate the observation will be, and the more computational power will be used for getting the observation. There are various types of resources that can be used for hyper-parameter optimization.

Time A run of an algorithm is often bounded by time. Thus, a fixed time budget is a natural resource for the evaluation. It is particularly preferred when the configurations differ in evaluation time. The time limit ensures that each configuration will run for the same amount of time [2]. Moreover, time is understandable and easy to work with for any user. However, in order to use time as a resource, the algorithm has to give meaningfull performance information when interupting during the run which is not always possible.

[2] We assume that time limit is set as a fraction of time required for full evaluation so that none of the partial evaluations finishes before this limit is reached. Therefore, all resources all used without any rebalancing.

Iteration Many learning algorithms work in an iterative manner. The learned model is improved using repeated learning loops. Such a loop can be used as a resource. Epochs in neural network learning procedure are a simple example. Training only a limited number of epochs gives us partial information about the algorithm performance.

Dataset subsampling The learning algorithm uses a training set of observations Π_{train} for the learning. The size of the training set has a great impact on the algorithm runtime and performance. The larger the amount of data, the more time we need, and the more accurate the learned model will be. Thus a fixed-size subset of training set can be taken as the resource unit. The limit for maximal resources R is the whole training set.

Feature subsampling Another possibility is feature subsampling. Each observation from Π_{train} is described using pre-defined features. Selecting a subset of the features reduces the amount of data and thus speeds up the evaluation. However, the missing information from other features will probably decrease the performance. The subset of features with a fixed size is another candidate for resource unit. The whole feature set forms a natural bound for R.

1.2.4.2 Configuration generation

Hyperband generates a set of configurations in each iteration and passes this set to Successive halving which evaluates it. The exact policy for the configuration generation is not determined. A simple approach is to sample values for each configuration independently from a prior distribution $D_{\mathbf{x}}$, i.e. we use the same policy as random search. More sophisticated methods which aim to cover the configuration space more evenly are quasi-random methods like Sobol sequence or Latin hypercube.

A different strategy is to incorporate information from the previous brackets and favour promising configurations. The idea is similar as in SMBO. A model which approximates the cost function is created and based on this model, we can select configurations that are more likely to perform well. To use all available information, the model should be able to work with partial observations. This model-based approach is proposed in [21]. The authors use Bayesian neural network for performance modeling. Candidate configurations are generated uniformly and the ones with the lowest predicted cost form the set for Successive halving. However, the model-based extension of Hyperband is not limited to Bayesian neural networks and described generation policy.

1.2.5 Other methods

Besides described methods, other approaches can be used for hyper-parameter optimization. Two families which include many algorithms are gradient-based optimization and evolutionary methods.

Gradient-based optimization Gradient-based optimization methods use a gradient of the optimized function to obtain a search direction. The gradient of a function is a vector which shows a direction of the greatest rate of increase of the function from a given point. Thus, it gives us an idea about the function behavior which can be use to choose a next point for evaluation. One of the simplest strategies is to follow the gradient using small steps. The function is thus evaluated in a sequence of points with increasing function value until it convergates to a local maximum where the search stops.

An analytical expression to calculate the gradient of the cost function is typically not available. However, the usual approach is to to approximate it. A popular method is an automatic reverse-mode differentiation, a numerical method which approximates the gradient from a training procedure [22]. Most of the methods for the gradient approximation have similar time requirements as the evaluation of f. Thus, a gradient approximation is an expensive operation. However, less expensive methods were proposed to compute the gradient by reversing the dynamics of stochastic gradient descent with momentum [23], or to approximate the gradient using a Hessian of the loss function with respect to model parameters [24].

Evolutionary methods Evolutionary methods are a well-known group of algorithms for global optimization. They are based upon an imitation of biological evolution. They work with so-called individuals which represent input points. Each individual has assigned a score calculated using a fitness function which determines a quality of the point. The goal is to breed individuals with higher quality. The individuals form a population. A typical evolutionary algorithm begins with an initial population and then alternates between three steps: selection, crossover, and mutation. In selection, individuals are selected based on their score, the higher the score, the greater probability of selection. Crossover mixes the selected individuals and mutation randomly changes an individual.

There are various evolutionary methods. They differ in a representation of the individual, implementation of the basic operators, and the existence of other operators. The evolutionary methods proposed for hyper-parameter optimization with best results are CMA-ES ([25, 26]), and the particle swarm optimization (PSO) ([27, 28]).

1.3 Surrogate models

This section contains introduction to a problem of approximation of the algorithm's performance and its usage for hyper-parameter optimization.

The role of the performance approximation was already briefly mentioned in sections 1.2.3.2 and 1.2.4.2. However, the performance modeling is a general concept and its usage is not limited to a specific method nor optimization.

1.3.1 Introduction

Section 1.2 contains desriptions of several hyper-parameter optimization methods. As one might notice, some of them use a model which helps them decide which point to evaluate. The model approximates a cost function behavior. However, in contrast to the real function, it is cheap to evaluate. Therefore, it can be queried many times without a significant overhead compared to one evaluation of the cost function. Such model is called a *surrogate model*, or simply *surrogate* (also *response surface model*, or *meta-model*).

Due to the black-box nature of the cost function, the surrogate is built using data-driven process. The cost function is evaluated at points $\mathbf{X}$, the values $\mathbf{y}$ are observed, and these data are used to learn the surrogate. The surrogate learning is similar to the learning of the original model whose learning algorithm's hyper-parameters are optimized. That is, a model with free parameters is taken and its parameters are adjusted using a machine learning algorithm. However, the domains of the original and the surrogate models are different. The original model models Π, the surrogate takes hyper-parameters as inputs and predicts the performance of the original model. Thus, the models and the training algorithms might differ. The surrogate predicts real values. Such a model is called a *regression model* or simply *regressor*.

There exist various models that can be used as a surrogate. The right choice is undermined by the characteristics of the modeled function. Our cost function was described as a black-box function about which a limited amount of information is available. However, often we assume some properties a priory. This information should be incorporated into a surrogate model selection.

The surrogate model can return a point prediction $\hat{f}(\mathbf{x}; \mathrm{D})$ or a predictive distribution over possible values of f, $p(f|\mathbf{x}, \mathrm{D})$. In case that only a point prediction is required, a point have to be selected based on $p(f|\mathbf{x}, \mathrm{D})$. A typical choice is a mean value $\mu(\mathbf{x})$, however, any value, such as a γ-quantile, can be considered.

A popular choice of models that return a predictive distribution are models constructed using the Bayesian approach. Consider a variable A. As a model for A, we choose a distribution (e.g. Gaussian distribution) with free parameters θ (e.g. mean and variance). The goal is to establish a distribution over values of the parameters θ. The Bayesian approach assumes that we have a prior model for θ, $p(\theta)$, and we can take observations of A and use them

to update the model. Based on the observations, a model of the data called likelihood is formed, $p(A|\theta)$. Using Bayes' theorem we get posterior

$$p(\theta|A) = \frac{p(A|\theta)p(\theta)}{p(A)}.$$

The denominator $p(A)$ is a normalizing constant which is not usually considered. Moreover, it is often computationally intractable. Thus, we model θ as

$$p(\theta|A) \propto p(A|\theta)p(\theta). \tag{1.1}$$

The posterior includes information from the prior model and observations which make it more accurate than both inputs on their own.

1.3.1.1 Hyper-parameter space

In Section 1.1, hyper-parameters and hyper-parameter optimization problem are defined. We said that hyper-parameters can have values of various types and that they can be related. However, up until now, the hyper-parameters were treated equally, and no assumption was made about their values, types, or relations among themselves. Described optimization methods optimize a general function with no assumption about its properties including properties of its input space. Though, some methods (random search, Hyperband) require a generation of hyper-parameter configurations from a pre-defined distribution $\mathcal{D}_{\mathbf{x}}$. However, the distribution and the generation method are independent of the optimization method. Surrogate models are more sensitive to input values, therefore, more detail analysis will be given here.

A hyper-parameter can be either *numerical* or *categorical*. Numerical hyper-parameters might be further divided into *continuous* and *discrete*. Typically, a continuous hyper-parameter have a value from a (closed) real interval, a discrete numerical hyper-parameter from a subset of integers, and a categorical hyper-parameter from a set defined by enumeration.

As we saw, it may be convenient to consider the values of hyper-parameters as random variables x_i, $i = 1, \ldots, d$, and define their joint distribution $\mathcal{D}_{\mathbf{x}}$, also defined using a probability density function or a probability mass function, $p(\mathbf{x}) = p(x_1, x_2, \ldots, x_d)$. The initial joint distribution is formed by the user based upon a prior knowledge of the hyper-parameters. The simpliest case is to define a marginal distribution over each hyper-parameter x_i, $x_i \sim \mathcal{D}_{x_i}$, and assume that the hyper-parameters are independent, that is $p(\mathbf{x}) = p(x_1)p(x_2)\ldots p(x_d)$. This can be applied, for example, if we optimize a learning rate and a number of units of a neural network which might be considered independent. However, as we mentioned before, other hyper-parameters such as a number of units in the second layer and a number of layers are closely related, i.e. the number of units in the second layer is conditioned by the existence of the second layer.

Generally, the hyper-parameter x_i can be conditioned by the value of x_{pi}. We will refer to x_{pi} as a parent and x_i as a child, and describe x_i with respect to its parent. Thus, instead of the marginal distribution $p(x_i)$ we define a conditional distribution $p(x_i|x_{pi})$. Typically, as in our example, the child is only relevant, also called active, if the parent has a specific value or values. An inactive child is not included in the configuration and the joint distribution $p(\mathbf{x})$. Therefore, the hyper-parameter space can be viewed as an oriented graph. The nodes represent the hyper-parameters, and the links represent the conditionality. Theoretically, the graph might have any structure. However, we restrict ourselves to tree-structured spaces as they are sufficiently accurate and simple to work with.

Regarding the used distributions, a uniform or log-uniform are common for continuous hyper-parameters, a discrete uniform for discrete numerical hyper-parameters, and a vector of equal probabilities for categorical hyper-parameters. However, there is no restriction on the distribution.

Some surrogate models handle various input types and complex structures natively. Others have to be adjusted to the complex space, or relax on it and treat it as all hyper-parameters are always active.

1.3.2 Models

This section contains a description of several models that can be used as surrogates for hyper-parameter optimization.

1.3.2.1 Gaussian processes

A Gaussian process (GP) is a stochastic process that models a distribution over function values for all input points.

There are two ways how to construct a prediction with a Gaussian process. We can use a prior distribution for parameters of a general linear model or we can define a prior distribution over functions. The prior distributions can then be updated using observed data to get a posterior. Both approaches lead to identical results. Their description follows. A detailed description of GP, its properties and derivation of formulas used below is given in [29].

A general linear model of a function f is

$$f(\mathbf{x}) = \boldsymbol{\Phi}(\mathbf{x})^{\mathrm{T}}\mathbf{w},$$

where $\mathbf{x}$ is the input vector and $\mathbf{w}$ are parameters of the model, often referred to as weights. The function $\boldsymbol{\Phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \ldots, \phi_J(\mathbf{x}))$ represents a projection of the input space into J dimensional space using a given set of J basis functions $\phi_j, j = 1, \ldots, J$. The projection increases expressiveness of the model. For notation simplicity, for the rest of the section, we will write $f = f(\mathbf{x})$, $\phi_j = \phi_j(\mathbf{x})$, $\boldsymbol{\Phi} = \boldsymbol{\Phi}(\mathbf{x})$, and $\Phi = \boldsymbol{\Phi}(\mathbf{X})$ for the projection of a matrix with multiple inputs.

The parameters $\mathbf{w}$ are modeled using a Bayesian approach. A Gaussian distribution is taken as their prior distribution

$$\mathbf{w} \sim \mathcal{N}(\mu, \Sigma).$$

where μ is mean vector and Σ is covariance matrix. For notation simplicity we will assume zero mean. However, the relations defined below apply even for non-zero mean. In practice, the mean can incorporate an expert's knowledge about the modeled function.

As mentioned before, the observations y contain a normally distributed noise $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$. A model for observations $p(y|\mathbf{x}, \mathbf{w})$ is derived from distribution of the noise and the linear model resulting in a Gaussian distribution with mean $f = \Phi^{\mathrm{T}}\mathbf{w}$ and variance σ_n^2. For multiple observations we have

$$\mathbf{y}|\mathbf{X}, \mathbf{w} \sim \mathcal{N}(\Phi^{\mathrm{T}}\mathbf{w}, \sigma_n^2\mathbf{I}).$$

The parameters $\mathbf{w}$ can be marginalized out and the prediction for $\mathbf{y}$ is calculated as

$$p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{X}, \mathbf{w})p(\mathbf{w})d\mathbf{w} \sim \mathcal{N}(0, \Phi^{\mathrm{T}}\Sigma\Phi + \sigma_n^2\mathbf{I}). \tag{1.2}$$

The prior for $\mathbf{w}$ and the likelihood form the posterior for $\mathbf{w}$ as in 1.1. The posterior distribution is again Gaussian

$$\mathbf{w}|\mathbf{X}, \mathbf{y} \sim \mathcal{N}(\frac{1}{\sigma_n^2}\mathbf{A}^{-1}\Phi\mathbf{y}, \mathbf{A}^{-1}),$$

where $\mathbf{A} = \sigma_n^{-2}\Phi\Phi^{\mathrm{T}} + \Sigma^{-1}$.

The prediction for f is now gained using the model and the posterior for parameters of the model. For a new point $\mathbf{x}$ we obtain distribution over the cost function value

$$\begin{aligned} p(f|\mathbf{x}, \mathbf{X}, \mathbf{y}) &= \int p(f|\mathbf{x}, \mathbf{w})p(\mathbf{w}|\mathbf{y}, \mathbf{X})d\mathbf{w} \\ &\sim \mathcal{N}(\frac{1}{\sigma_n^2}\Phi^{\mathrm{T}}\mathbf{A}^{-1}\Phi\mathbf{y}, \Phi^{\mathrm{T}}\mathbf{A}^{-1}\Phi). \end{aligned} \tag{1.3}$$

By rewriting the mean and the variance for the predictive distribution in 1.3, we derive formulas where basis functions appear only in the form $\Phi^{\mathrm{T}}\Sigma\Phi'$. By defining a kernel function $k(\mathbf{x}, \mathbf{x}') = \Phi^{\mathrm{T}}\Sigma\Phi'$ for two points $\mathbf{x}$ and $\mathbf{x}'$, we get the distribution as

$$f|\mathbf{x}, \mathbf{y}, \mathbf{X} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})), \tag{1.4}$$

where

$$\begin{aligned} \mu(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x})(\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{y} \\ \sigma(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}(\mathbf{x})^{\mathrm{T}}(\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{k}(\mathbf{x}). \end{aligned}$$

Symbols $\mathbf{k}(\mathbf{x})$ and $\mathbf{K}$ denote the vector of values $\mathbf{k}(\mathbf{x})_i = k(\mathbf{x}, \mathbf{x}_i)$ and the matrix of values $\mathbf{K}_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$ respectively, where $\mathbf{x}_i$ and $\mathbf{x}_j$ $i, j = 1, \ldots, n$ are points from the training dataset. The model now incorporates the prior knowledge and the information gathered from the observations of the cost function.

The derivation taken above results in Gaussian distribution over the function value. Alternatively, we can construct a process which defines a distribution over the function values and updates it based on the observed values. Short description of this approach follows.

A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution [29]. For Gaussian process distribution we write

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')).$$

The distribution is fully defined by its mean function $m(\mathbf{x})$ and a positive definite covariance function $k(\mathbf{x}, \mathbf{x}')$. The mean function defines offset. It is often set to zero (and further in this work, the zero mean function will be assumed). The kernel function specifies covariance between a pair of variables. The covariance of a pair $f(\mathbf{x})$ and $f(\mathbf{x}')$ is defined as a function of the inputs, i.e. as relation between a pair of hyper-parameter configurations $\mathbf{x}$ and $\mathbf{x}'$.

The definition of GP implies that a finite set of function values has Gaussian distribution. Therefore, for training observations $\mathbf{f}$ in locations $\mathbf{X}$ we have joint distribution

$$\mathbf{f}|\mathbf{X} \sim \mathcal{N}(0, \mathbf{K}).$$

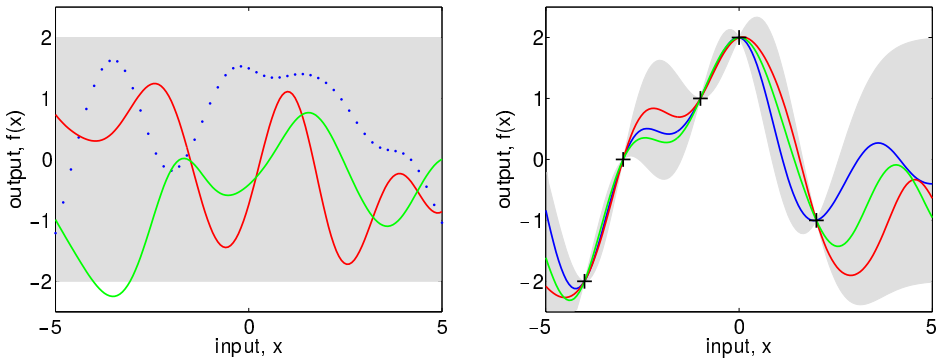
For noisy observations, the noise is added to covariance function $k(\mathbf{x}, \mathbf{x}')+\sigma_n\delta$, where δ is 1 if $\mathbf{x} = \mathbf{x}'$ and 0 otherwise, which gives us a model

$$\mathbf{y}|\mathbf{X} \sim \mathcal{N}(0, \mathbf{K} + \sigma_n^2\mathbf{I}).$$

Setting covariance as product of basis functions and covariance matrix Σ as defined above will result in the same formula as in 1.2. The model is constructed using only our prior knowledge which is included in the mean and the kernel function just like the prior knowledge about $\mathbf{w}$ is included in the prior model in 1.2.

We are interested in a model which will also incorporate the information from observations. The joint distribution of these observations and a new point $\mathbf{x}$ is

$$\begin{bmatrix} \mathbf{y}|\mathbf{X} \\ f|\mathbf{x} \end{bmatrix} \sim \mathcal{N}\Big(\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma_n^2\mathbf{I} & \mathbf{k}(\mathbf{x}) \\ \mathbf{k}(\mathbf{x})^{\mathrm{T}} & k(\mathbf{x}, \mathbf{x}) \end{bmatrix}\Big).$$



(a) prior (b) posterior

Figure 1.5: Panel 1.5a shows three functions drawn at random from a GP prior. The dots indicate actually generated values of f. The lines representing the two other functions have been drawn by joining a large number of generated values of f. Panel 1.5b shows three random functions drawn from the posterior, i.e. the prior conditioned on the five noise free observations indicated. In both plots the shaded area represents the pointwise mean plus and minus two times the standard deviation for each input value (corresponding to the 95% confidence region), for the prior and posterior respectively [29].

To get prediction for $\mathbf{x}$ with model that incorporates the observations $\mathbf{y}$, we condition the joint distribution and get

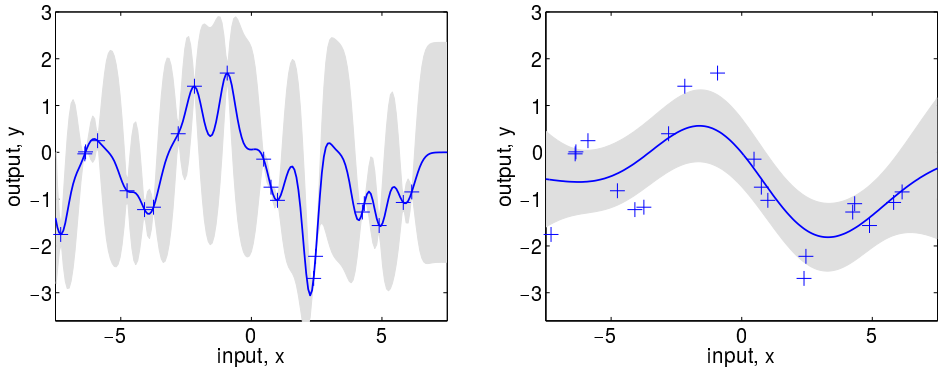
$$f|\mathbf{x}, \mathbf{y}, \mathbf{X} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$$

with $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ same as in 1.3. Therefore, identical results were reached with both approaches.

To better understand a distribution over functions, see Figure 1.5. Functions of one parameter x from prior distribution are shown in Panel 1.5a. Three samples were observed and used to define posterior distribution in 1.5b. The figure shows how the distribution changed. Only functions that fit the data are generated by the posterior distribution. Other functions are rejected.

Covariance of the Gaussian process As mentioned above, the covariance function is one of the components which defines a Gaussian process. The covariance describes properties of the model as smoothness, noisiness, periodicity, etc. It is determined by a kernel function and a noise variance. The noise is taken as normally distributed with variance σ_n^2 and it is independent for each variable. The kernel function is more complex. A number of different kernels was designed. Close attention should be paid to the selection of the right one.

Matérn kernels [29] are a popular choice for hyper-parameter optimization. They are parameteric functions parameterized by a smoothness parameter



(a) $(l, \sigma_f, \sigma_n) = (0.3, 1.08, 0.00005)$ (b) $(l, \sigma_f, \sigma_n) = (3.0, 1.16, 0.89)$

Figure 1.6: Panel 1.6a data is generated from a GP with covariance parameters $(l, \sigma_f, \sigma_n) = (0.3, 1.08, 0.00005)$, as shown by the + symbols. Using Gaussian process prediction with these hyper-parameters we obtain a 95% confidence region for the underlying function f (shown in grey). Panel 1.6b again shows the 95% confidence region, but this time for covariance parameter values $(3.0, 1.16, 0.89)$ [29].

$\nu > 0$. A special case is squared exponential kernel with $\nu \to \infty$ which is given by

$$k_{se}(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp(-\frac{1}{2l^2}\|\mathbf{x} - \mathbf{x}'\|^2)$$

where $\|\cdot\|$ is the L^2 vector norm. The parameter l is a characteristic lenght-scale. It determines "closeness" of the points, in other words, how much the points will infuence each other. The parameter σ_f^2 is a signal variance. It says how much the function changes. If it is large, the function has a high frequency. Different values of parameters can be used depending on the problem instance. Figure 1.6 shows examples of the GPs with different parameters of the squared exponential kernel and the noise variance. Other popular choices for smoothness parameter are $\nu = 3/2$ and $\nu = 5/2$ which corresponds with less smooth function which is often more realistic.

Other kernels were suggested directly for hyper-parameter optimization often to address a specific problem like building in a knowledge from other problem instances [5], or to include partial information gained during the optimization [7].

Gaussian processes are not primarily suitable for other than real-valued input spaces because standard kernels are defined for real-vectors. To use a Gaussian process with different input types, special kernels need to be used. This applies on conditional spaces as well [15].

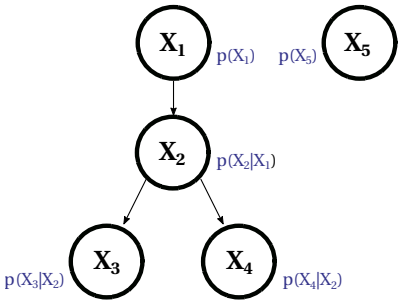


Figure 1.7: Tree-structure parzen estimator. Each node represents one variable and holds the associated distribution. The variables x_1 and x_5 are independent and always active. The variable x_2 is conditioned by value for variable x_1. The variables x_3 and x_4 are conditioned by x_2. Joint probability of a point $\mathbf{x}$ with all variables active is given by $p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_2)p(x_4|x_2)p(x_5)$.

1.3.2.2 Tree-structured parzen estimator

A tree-structured parzen estimator (TPE) is a model proposed as a surrogate for hyper-parameter optimization by Bergstra *et al.* [8]. The probability $p(f|\mathbf{x})$ is modeled from separate models for $p(y)$ and $p(\mathbf{x}|y)$ using Bayes' theorem

$$p(f|\mathbf{x}) = \frac{p(y)p(\mathbf{x}|y)}{p(\mathbf{x})}.$$

The model for $p(\mathbf{x}|y)$ is a graph-structured process constructed based upon a hyper-parameter space structure and distributions. This allows the TPE to naturally handle various types of input variables including conditional variables. The construction of the graph is as follows.

- For each input variable x_i, create a node x_i.
- For each input variable x_i with parent x_{pi}, add link from node x_{pi} to node x_i.
- Add associated distribution to each node. The associated distribution will be $p(x_i)$ for the node x_i without the parent node, and $p(x_i|x_{pi})$ for the node x_i with the parent x_{pi}.

We are restricted on tree-structured spaces (Section 1.3.1.1), therefore, the graph will also have a tree structure. The model captures a decomposition of the joint distribution over all variables. Ilustration of a constructed model is in Figure 1.7.

After collecting data D the model is updated by transforming the prior process. The transformation replaces the prior input distributions in nodes

with distributions derived from observations. Two separate densities $l(\mathbf{x})$ and $g(\mathbf{x})$ are used to define $p(\mathbf{x}|y, \mathrm{D})$ so that

$$p(\mathbf{x}|y, \mathrm{D}) = \begin{cases} l(\mathbf{x}) & \text{if } y < \tau \\ g(\mathbf{x}) & \text{if } y \geq \tau. \end{cases}$$

The threshold value τ is chosen as a γ-quantile of observed values $\mathbf{y}$, i.e. $p(y < \tau) = \gamma$, where no specific distribution of $p(y)$ is required [8]. The separation results in two models which are formed using different parts of the dataset. The points with y smaller than τ form $l(\mathbf{x})$, the rest of the points form $g(\mathbf{x})$. Both models have the same structure as the prior model, however, instead of prior hyper-parameter distributions, mixture distributions are calculated from observations. Specifically, we construct the density $l(x_j)$ in the node x_j as follows.

The mixture distribution represented with a density function $l(x_j)$ for finite set of probability density functions $p_1(x_j), p_2(x_j), \ldots, p_k(x_j)$ with weights $w_{1,j}, w_{2,j}, \ldots, w_{k,j}$, where $\sum_{i=1}^{k} w_{i,j} = 1$, is defined as

$$l(x_j) = \sum_{i=1}^{k} w_{i,j} p_i(x_j).$$

In our case, individual densities are formed using i-th point from indexed set of points S_l, where S_l contains all training points with active variable j and observed cost smaller than τ, and $k = |\mathrm{S}_l|$. The weigths are all equal and the densities $p_i(x_j)$ come from the same parametric family. The parameters of the distribution $p_i(x_j)$ are given by the value $x_{i,j}$. Bergstra *et al.* [8] suggested following replacements for density $l(x_j)$.

- For continuous variables with original uniform distribution $\mathcal{U}(a, b)$, a sum of truncated Gaussian distributions is used. The mean for each distribution $p_i(x_j)$ is $x_{i,j}$ and the variance is set to greater of the distances to the left and right neighbor for each point $(\mathbf{x}_i, y_i) \in \mathrm{S}_l$.
- For discrete variables with a prior vector of N probabilities p_i, vector's new elements are proportional to $Np_i + C_i$, where C_i is a number of occurences of a value i in S_l.

The same method is used for all variables and $g(\mathbf{x})$.

1.3.2.3 Random forests

A random forest regression model is an ensemble of tree-structured regressors each of which returns a point prediction for a given input point. Averaging these predictions, we get a point prediction of the target function with the forest. An uncertainty of the prediction is calculated as a sample variance

of regressors predictions. A Gaussian distribution with this variance and the average value as a mean is taken as a predictive distribution.

To construct a random forest with B regressors, called regression trees, we first need to create a training dataset D_k for each tree, $k = 1, \ldots, B$. The dataset for k-th tree consists from n points sampled randomly with repetition from the original dataset D. Each tree is then built independently on other trees.

A learning algorithm for a tree $h_\theta(\mathrm{D})$ requires a loss function $\mathcal{L}$. Shortly, we will write loss for point $(\mathbf{x}, y)$ as $\mathcal{L}(y, \hat{f}(\mathbf{x})) = \mathcal{L}((\mathbf{x}, y); h_\theta(\mathrm{D}))$, where $\hat{f}(\mathbf{x})$ is a point prediction of the trained model. An example of a loss function is a mean squared error (MSE) defined for set of n points as

$$\mathcal{L}_{MSE}(\mathbf{y}, \hat{f}(\mathbf{X})) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{f}(\mathbf{x}_i))^2.$$

The learing algorithm constructs a tree by recursivly splitting given training dataset. Each split is represented by an internal node. The split in a node is found as follows.

1. Select a random subset $\mathcal{F}$ of $\lceil dp \rceil$ input variables, where p is a user-defined parameter and d is a number of input variables.

2. For each variable $x \in \mathcal{F}$, find a split of the data in the node which mimimize loss.

3. Select variable $x^* \in \mathcal{F}$ whose split has minimal loss. This variable will be associated with the node and all input data will be passed on to the following nodes according to the value which minimized the loss.

A split of data is realized as a division into two separate subset S_1, S_2. The prediction for point $\mathbf{x} \in \mathrm{S}_1$ is an average of the observed values from S_1, similarly for $\mathbf{x} \in \mathrm{S}_2$ is an average of observed values in S_2. Ordinal parameters are split based upon one value, depending whether the input point's value is smaller or greater. For categorical parameters, a set of values is defined and one data subset contains points with values from the set, and the other contains points that have different values.

A new dataset S_1 is passed to one of the following nodes, the other dataset is passed to the other following node. The following nodes perform the same procedure with given datasets. Thus, the input dataset is divided until a node with number of training points smaller than a user-defined parameter n_{min} is reached. The data is not split further and the node is marked as a leaf. Each leaf is assigned a constant value equal to the average of leaf's training points observations.

The prediction for a point $\mathbf{x}$ is given by traversing the tree based on node's variables. When a leaf node is reached, the leaf's constant is taken as a prediction $\hat{f}(\mathbf{x})$.

For random forest prediction, all regression trees yield a prediction $\hat{f}_k(\mathbf{x})$. Random forest prediction is then an average of $\hat{f}_k(\mathbf{x})$ for each tree:

$$\mu(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B}\hat{f}_k(\mathbf{x})$$

Moreover, we can calculate a variance estimate as

$$\sigma^2(\mathbf{x}) = \frac{1}{B-1}\sum_{k=1}^{B}(\hat{f}_k(\mathbf{x}) - \mu(\mathbf{x}))^2.$$

The $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ are interpreted as mean and variance of $p(f|\mathbf{x}, \mathrm{D})$. The distribution is taken as Gaussian, thus the model is

$$f|\mathbf{x}, \mathrm{D} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$$

As one might notice, the random forests can work with various input types. Moreover, the trees can learn to ignore inactive variables so the forest can handle conditional spaces.

1.3.2.4 Artificial neural networks

Artificial neural networks are a well-known group of machine learning algorithms that can be used for many tasks including surrogate modeling.

An artificial neural network consists of many simple units, called neurons, which are organized into interconnected layers. A connections has a direction and so-called weight which express a strenght of the connection. An output of the source neuron is multiplied by the weight of the connection and the product is used as an input for the target neuron. The target neuron takes inputs from all input connections, sums them, and applies a function, called activation function, on the sum. The output of the activation function is then the output of the neuron. The network's training algorithm optimizes the weights of the connections so that the neurons in the last layer give a desired output.

Neural networks can be used for surrogate modeling in various ways. Snoek *et al.* [12] use a neural network which maps an input point to a vector of D parameters, denoted as $\phi(\mathbf{x}) = \left[\phi_1(\mathbf{x}), \ldots, \phi_D(\mathbf{x})\right]^{\mathrm{T}}$. The weights for the mapping are set via a standard neural network training procedure. The resulting setting can be viewed as a maximum a posteriori estimate of the parameters [3]. For the purpose of training, there is the last linear layer which is removed after termination of the learning. The layer is replaced by a Bayesian linear model (which takes $\phi(\mathbf{x})$ as an input) which estimates mean and variance of

[3] Maximum a posterior (MAP) estimate of parameter θ is value that maximize probability of θ given the data, i.e. $\theta_{MAP} = \arg\max_\theta p(\theta|\mathrm{D})$.

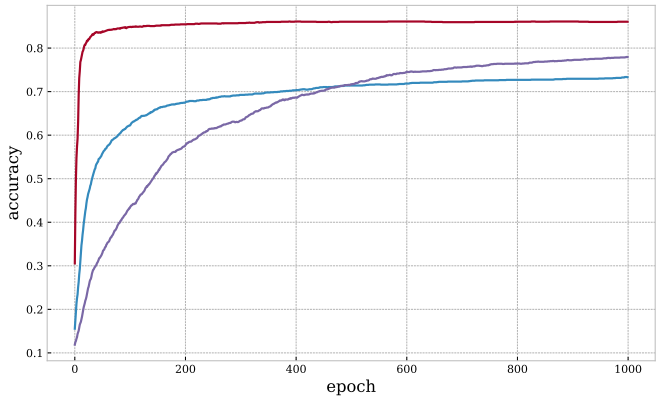


Figure 1.8: A learning curve for different neural networks

the cost function in a similar way to Gaussian processes. Therefore, we get a prediction which estimates the expected cost and captures an uncertainty of the prediction. The advantage of the model, called Deep networks for global optimization (DNGO), over Gaussian processes is its smaller complexity given a number of training points. The estimates of the mean and the variance of GP in 1.4 require an inversion of $n \times n$ matrix, thus, the complexity increases cubically with the number of training samples. In contrast, the Bayesian linear model in DNGO requires inversion of $D \times D$ matrix, where D is size of the last layer of the network which is not dependent on the number of training points [12].

Klein *et al.* [21] proposed a model for iterative machine learning algorithms where a loss function can be observed in different time points. Specifically, suppose that for a full evaluation of f, we take an observation y in time T. An iterative algorithm allows us to observe f in time $t \in (0, T]$ resulting in a partial observation $y^{(t)}$. This observation will be even more noisy than y, however, it still gives us some information. Moreover, for machine learning algorithms, the validation loss improves over time which can be described by so-called learning curve. An illustration of a learning curve is in Figure 1.8.

The proposed model aims to extrapolate a learning curve for given configuration which allows to predict the loss at an arbitrary point in time. A prediction for an asymptotic value of the loss and a weighted sum of D parametrized basis functions are used to estimate the mean loss for an arbitrary point in time

$$\mu(\mathbf{x}) = \mu_\infty(\mathbf{x}) + \sum_{i=1}^{D} w_i(\mathbf{x})\phi_i(\mathbf{x})$$

where $\mu_\infty(\mathbf{x})$ is the predicted asymptotic loss, $\phi_i(\mathbf{x})$ is the i-th basis func-

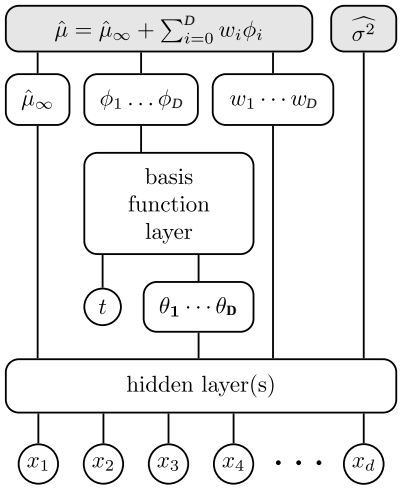


Figure 1.9: Architecture for the learning curve prediction using a Bayesian neural network. The network learns simultaneously μ_∞, parameters of basis functions $\Theta = (\theta_1, \dots, \theta_D)$, weights $\mathbf{w} = (w_1, \dots, w_D)$ and variance estimate σ^2. The prediction in time t is weighted sum of the basis functions with learned parameters and μ_∞ [21].

tion and $w_i(\mathbf{x})$ is its weight. The values for $\mu_\infty(\mathbf{x})$, $w_i(\mathbf{x})$ and $\phi_i(\mathbf{x})$, where $i = 1, \dots, D$, are free parameters. These parameters are set with a Bayesian neural network learned using observations of f including partial observations. Moreover, the network also estimates a variance of the prediction. Thus, we get a probability model which includes partial information. A graphical illustration of the model's architecture is in Figure 1.9. For a detailed description of the model, see [21].

Neural networks work with numerical inputs. To handle categorical input variables, the values have to be encoded so that a new set of input variables is created. Moreover, Snoek *et al.* [12] succesfully used their network-based model on a complex space with conditional variables.

1.4 Other aspects

This section covers some other aspects that need to be considered when performing hyper-parameter optimization. These aspects can have negative impact on the optimization or they can improve it. Moreover, they largely depend on the problem being solved, amount of information we have, and available computational resources.

1.4.1 Initialization

By the initialization of the hyper-parameter optimization, we understand generation and evaluation of a set of configurations before the optimization algorithm is employed. We can generate these initial configurations to explore the space or to improve the optimization by testing promising configurations.

The exploration is related mainly to model-based algorithms as SMBO or model-based Hyperband. An initial dataset which covers the important parts of the space can be convenient when building the first surrogate models. We can use a fixed pre-defined set, a random set or a quasi-random set of configurations to form this initial dataset.

The initialization can be used to test some promising configurations. Such configurations are usually the ones that performed well on similar tasks which we already solved. The group of techniques that deals with searching similarities between the tasks and algorithms performance is called meta-learning. Feurer *et al.* [30] use meta-learning to initialize SMBO. Their framework generates initial configurations based on a similarity measure of features extracted from a dataset referred to as meta-features. That is, the meta-features from a dataset for the task Π are compared with the meta-features extracted from datasets of previous tasks. The configurations that performed well on the datasets with similar meta-features are then used for the initialization.

1.4.2 Parallelization

A machine learning task is often solved on a system with multiple computational nodes. In such case, we want to effectively parallelize the calculation in order to use the all available resources. For hyper-parameter optimization, this means that we want to parallelize optimization algorithm to use all available nodes at once.

The most expensive operation is the cost function evaluation. Therefore, trivial parallelization schema assigns a configuration to each node. The node then evaluates given configuration independently from the other nodes, i.e. it trains its own model.

The mechanism of assignment of a configuration to a free node depends on the optimization method. Grid search knows the set of tested configurations in advance, thus, a configuration from this set is given to the node. Random search can generate a configuration at any time, therefore, when a node is free, a new configuration is generated and assigned to it. Hyperband has two options for parallelization. We can parallelize the outer loop, that is each bracket will run on its own node. However, this is possible only for a static generation method where each bracket is independent of results of the others. The second option is to parallelize the inner loop. A drawback of this method is that the number of function evaluations decreases exponentially, thus, a more sophisticated job priority queue must be managed in later stages [9].

The last described optimization method is sequential model-based optimization. How the name suggests, SMBO is a sequential method. The emphasis is placed on gathering all information before a decision about a next tested configuration is made. Thus in a parallel run, we should wait until all currently running configurations finish before choosing a new one. This approach would result into the waste of resources. To overcome this, we can relax on gathering all information, ignore currently running configurations and use only finished evaluations from all nodes as data for the surrogate. Theoretically, when maximize acquisition function, the same global optimum should be found and evaluted for all free nodes until a new obseravation is added. However, the optimization usually finds only local optimum, thus, we may count on stochasticity of the optimization that it will result in different local optimas [8]. Different approach is to use so-called constant layer, i.e. to assign a constant value of y to all currently running configurations and add them to dataset. The acquisition function will not suggest these configurations because they will appear as already evaluated. Nevertheless, ignoring the running configurations can lead to a less effective search. Thus, more effective parallelization schemas were proposed, see [15] for their review.

1.4.3 Optimization of multiple tasks

Up until now, we considered optimization of hyper-parameters for only one task, that is we solved a specific problem such as digit recognition. However, we often have multiple different tasks (e.g. several different problems such as digit recognition, animal classification, etc.) which can be related. Thus, knowledge from one task can give us information about another task. Typically, we have a set of already solved tasks and we want to incorporate knowledge about their hyper-parameters to the optimization.

As already mentioned, one approach is to use meta-learning for the initialization. Promising configurations are generated based on a similarity of the datasets and their results.

A different approach is to create a joint surrogate model for multiple tasks. This approach was considered mainly for Gaussian processes where covariance function can be defined to measure both a similarity between a pair of points and a similarity between tasks. Formally, for a pair of tasks Π and Π', and two configurations $\mathbf{x}$ and $\mathbf{x}'$, we define a kernel as a product of two separate kernels

$$k((\mathbf{x}, \Pi), (\mathbf{x}', \Pi')) = k_{\mathcal{X}}(\mathbf{x}, \mathbf{x}')k_{\Pi}(\Pi, \Pi').$$

The kernel $k_{\mathcal{X}}$ measures relationship between configurations and k_{Π} relationship between tasks. Their product then can be used for a multi-task model [5]. Other methods were also considered, see [15] for their review.

Chapter 2

Experimental part

2.1 Experiments design

In this section, performed experiments will be described. The aim of the experiments is to compare different hyper-parameter optimization methods in terms of convergence and selected configurations on various problems. Results and their analysis is given in the section 2.3.

2.1.1 Hyper-parameter optimization problems

A hyper-parameter optimization problem aims to find best hyper-parameters of a machine learning algorithm to train the model for given data. The dataset and the machine learning algorithm used in the experiments are described in the following subsection. Hyper-parameters of the machine learning algorithms that are optimized are described in the next subsection.

2.1.1.1 Datasets

Two datasets are used in the experiments: MNIST and its variation MRBI. MNIST dataset is choosen due to its simplicity which allows deep analysis of the hyper-parameter methods behavior. The MRBI dataset is more difficult thus more challenging for hyper-parameter optimization. Both datasets come from the same domain and are related which allows us to compare the optimization method behavior on problems with various degrees of complexity.

MNIST The MNIST digit recognition dataset is well-known image dataset, often used for testing of various machine learning techniques. The dataset consists of 28×28 pixels, gray-scale images of handwritten digits ranging from 0 to 9. A digit in each image is written in white, and it is size-normalized and centered on a black background. The aim of the classification task is

to recognize the digit in a given image, i.e. to classify an image to classes $0 - 9$ [31].

The MNIST dataset is usually divided into a training set of size 50000 images and a test set with 10000 images. However, in order to speed up the experiments, a partition adopted from [4] is used in this work, that is 10000 images is used for training, 2000 for validation and 50000 for testing.

MNIST rotate background (MRBI) The MRBI dataset is a variation on MNIST constructed by Larochelle *et al.* [4]. A digit in each MNIST image is rotated by an angle chosen randomly between 0 and 2π. Moreover, the black background is replaced by a random, 28×28 pixels, black and white image. The classification task remains the same, to assign correct digit to given image. However, the variations performed with the images change the task's behavior and increase its complexity.

The training, validation and testing dataset partition is the same as for the MNIST, that is a training set consists of 10000 images, a validation set from 2000 and a test set from 50000 images.

2.1.1.2 Machine learning algorithms

An artificial neural network is used as machine learning algorithm which trains models.

Artificial neural networks are a group of algorithms that were successively used for wide range of tasks. However, they come with a large number of hyper-parameters that need to be tuned. The hyper-parameter setting is a non-trivial task due to the complexity of the hyper-parameter space and the fact that the training can be time-consuming – a training of one network often takes hours or days depending on the network's structure and the task being solved.

A neural network consists of structured network model and optimization algorithm. The network's hyper-parameter space is formed by a mixture of optimization algorithm's parameters and parameters of the model.

In this work, a feed-forward neural network with stochastic gradient descent (SGD) optimization is used for both datasets. In order to simplify the task and make it computationally feasible, only a subset of hyper-parameters is selected for hyper-parameter optimization. The subset covers real, integer and categorical hyper-parameters. For MRBI dataset some hyper-parameters are conditioned by others.

The classification of the MNIST dataset is not very difficult, thus a network with only one hidden layer is trained. The space of optimized hyper-parameters consists of five variables:

- An initial learning rate for SGD.

- A learning rate decay for SGD which decreases the learning rate in each epoch [4]. A learning rate lr for an iteration t is derived from the initial rate lr_0 and the learning rate decay d as $lr_t = \frac{lr_0}{1+dt}$.
- A number of units in the hidden layer.
- An activation function for the units in the hidden layer.
- A weight distribution from which are initial weights generated.

The types, values and prior distributions for the hyper-parameters are given in Table 2.1. The network is trained for 1000 epochs. An early-stopping criterion can be used to stop the training if the network's performance is not improving.

The MRBI dataset is more complex, thus a network can have from one up to three hidden layers each of which comes with hyper-parameters: a number of units, an activation function in the units, a weight initialization distribution. The values of these parameters are conditioned by the existence of the layer. Each hidden layer is followed by another layer with probability 0.5, that is the second layer exists with probability 0.5 and the third layer with probability 0.25. In order to prevent overfitting of larger networks, a technique called dropout is employed [32]. The dropout introduces a new hyper-parameter to each hidden layer and the input layer: a dropout rate. The optimized hyper-parameters for SGD are the same as for MNIST: a learning rate and a learning rate decay. The hyper-parameter space is thus formed by 17 hyper-parameters some of them conditioned by others. Their values are specified in Table 2.1. The network is trained for 2000 epochs but can be stopped if the performance is not improving.

The output layers of the networks for both tasks has 10 units corresponding to digit $0-9$. As activation function softmax is used in all units. The networks are trained to yield a probability of each digit being in the image. Categorical cross-entropy loss function is used to train the network for these probabilities.

2.1.2 Compared hyper-parameter optimization methods

In theoretical part (Chapter 1), several hyper-parameter optimization methods are described including surrogate models that can be used to guide the optimization. In the experiments, five combinations of the methods and surrogates are compared:

[4] An epoch of a neural network is a training unit in which all training data are processed, i.e. all training data are used to adjust network's model parameters. A typical neural network runs iteratively a large number of these epochs and thus sees the training data several times. After each epoch, the model can be used for prediction. However, its performance does not have to be sufficient, therefore, the training continues with other epochs until the performance is sufficient or a maximal number of epochs is reached.

[5] Only for the MRBI dataset.

Hyper-parameter	Type	Values
Learning rate	Real	$\log\mathcal{U}(0.001, 10)$
Learning rate decay	Real	$\log\mathcal{U}(10^{-5}, 10^{-3})$
Dropout rate [5]	Real	$\mathcal{U}(0, 0.6)$
Number of units in a layer	Integer	$\log\mathcal{U}(18, 1024)$
Activation function in a layer	Categorical	$\mathcal{U}(\{\tanh, \text{sigmoid}\})$
Weight inicialization for a layer	Categorical	$\mathcal{U}(\{\mathcal{U}(-1, 1), \mathcal{N}(0, 1)\})$
Next layer exists [5]	Categorical	$\mathcal{U}(\{\text{True}, \text{False}\})$

Table 2.1: The optimized hyper-parameters, their types, values, and prior distributions. The notation $\mathcal{U}(a, b)$, where $a < b$, denotes continuous or discrete uniform distribution over values from a to b. For categorical hyper-parameters $\mathcal{U}(\text{S})$ denote that all values from set S have equal probability. The notation $\log\mathcal{U}(a, b)$ is used for log-uniform distribution, i.e. values are generated in logarithm domain between $\log a$ and $\log b$, where $0 < a < b$. For integer variables the generated values in logarithm domain are transformed back and rounded to the nearest integer.

- random search
- sequential model-based optimization with surrogates
  - Gaussian processes
  - random forests
  - tree-structured parzen estimator
- Hyperband

The random search and SMBO iteratively generate and evaluate 10 configurations for the MNIST experiment and 30 configurations for the MBRI experiment. The number of configurations are determined based on the number of optimized hyper-parameters, available computational resources, and empirical tests. Hyperband does not use the same iterations as random search and SMBO. It does not fully evaluate all generated configurations, thus for the same budget of resources, a larger number of configurations can be generated. The number of generated configurations is given by its resource parameters r, R, and η factor. Therefore, the values of these parameters are chosen to give Hyperband similar computational resources as to the random search and SMBO.

The experiments focus strictly on the optimization methods comparison. No technique described in Section 1.4 as meta-learning or multi-task is used and the optimization run sequentially for all methods.

2.1.2.1 Methods setting

This section describes settings of individual hyper-parameter optimization methods.

Random search Random search has no other setting except a number of generated configurations and prior distribution.

SMBO SMBO and each surrogate model have several parameters. Expected improvement is used as acquisition function for all surrogates along with random sampling for its optimization. In each iteration, 10000 random configurations are sampled and the one with the highest EI is used for training.

Matérn kernel with smoothness parameter $\nu = 5/2$ is used for Gaussian processes as suggested in [33]. The length-scale for each hyper-parameter and noise level are optimized to fit the data. The Matérn kernel is defined for real variables, therefore, the non-real hyper-parameters have to be transformed. Integer hyper-parameters are treated as real in the surrogate and they are rounded to nearest integer before using for training the network. Categorical hyper-parameters are encoded using one-hot encoding. The kernel is not designed to handle conditional variables, thus, all variables are considered always active.

The random forest model constructs 100 trees. A minimal number of points in a leaf is three and all hyper-parameters are considered to be split upon in each node corresponding with p equal one. These values are taken from implementation in [34] as well-founded. As for the forest size, as Breiman [35] points out, random forests do not overfit and their performance increase with a number of trees. The only drawback is that with increasing size of the forest, the computational complexity increases. However, for the performed experiment, the time for training 100 trees is insignificant in comparison to the time required by one cost function evaluation. The dataset for the forest is small, thus, the number of leaf points is set to small number.

Tree-structured parzen estimator implementation is taken as suggested by Bergstra *et al.* [8].

Hyperband As mentioned before, the neural network algorithm runs in iterations called epochs. These epochs are used as a resource for Hyperband. However, twenty epochs are set as one resource unit instead of one epoch in order to reduce the number of generated configurations. Moreover, a typical network takes several epochs to change significantly its performance, therefore, using more than one epoch as a resource is well-founded. A maximal number of resources R is thus set to 50 for the MNIST and 100 for the MRBI dataset. The last parameter η is set to four and three for MNIST and MRBI, respectively. The values for η are chosen based on a recommendation in [9] and so that the Hyperband trained in total a similar number of epochs as other methods for all

trained networks together. The configurations for Hyperband are generated randomly from a given prior distribution.

2.1.3 Evaluation

To quantify a neural network model's performance, classification accuracy is used. The accuracy is defined as proportion of correctly classified input points

$$\mathrm{acc}(\Pi) = \frac{1}{|\Pi|} \sum_{(\pi,c)\in\Pi} \mathbb{1}(\hat{c} = c)$$

where π is input vector, c is its class, and $\hat{c}$ is class predicted by the model. The accuracy can be transformed to loss function, called error rate or simply error, by substraction from one

$$\mathcal{L}(\Pi) = \mathrm{error}(\Pi) = 1 - \mathrm{acc}(\Pi).$$

which measures proportion of missclassified inputs. As mentioned above, the model outputs probabilities of individual digits. We then take a digit with maximal probability as prediction $\hat{c}$.

The error rate is calculated for training, validation and testing dataset separately. For hyper-parameter optimization, the error on the validation set is used as a cost function. The optimization then recommends a model with a minimal validation error. The models founded by hyper-parameter optimization are compared using the error rate on the test data.

All compared methods include a random element, thus, with different random generator settings, the optimization can have different results. In order to reduce the randomness, each method is run ten times with different random seed for the generator.

2.2 Implementation

The experiments are written in Python language version 3.6[6]. Python is a high-level open-source language often used for machine learning tasks due to its simplicity, which allows quick prototyping, and availability of many scientific packages. The experiments use some of these packages:

- SciPy [36] – a package for numerical operations.
- NumPy [36] – a package for working with N-dimensional arrays, linear algebra operations and random number capabilities.
- Pandas [36] – a package for handling structured data.

[6] `https://docs.python.org/3/index.html`

- scikit-learn [36] – a package for machine learning algorithms.
- Matplotlib [36]– a plotting package.
- Seaborn [37] – a package for statistical visualization built on top of Matplotlib.
- Plotly [38] – a plotting library which provides interactive graphs.
- TensorFlow [39] – a package for numerical computations and machine learning modeling.
- Keras [40] – high-level API for neural network modeling. The model in Keras can be built using several different packages which implements the neural network functionality.

A full list of dependencies is specified on attached CD.

The experiments were prototyped in Jupyter Notebooks[7] – a web application which allows run interactive code, visualizations, and text writing.

The neural networks for both tasks are implemented using Keras with TensorFlow backend for the network functionality. Keras allows a simple definition of the network and an interface for its training and prediction. It also allows saving and loading the neural network's model. The further is used for optimization with Hyperband where partially trained models are saved and loaded if the training should continues.

Two packages which implement functionality for hyper-parameter optimization with described methods are used – Scikit-Optimize [34] and Hyperopt [41]. Scikit-Optimize implements sequential model-based optimization with various surrogate models. Its implementation of SMBO with Gaussian processes and random forests is used in this work. Hyperopt is a package which implements random search and SMBO with tree-structured parzen estimator both of which are used in the experiments. My own implementation of Hyperband is used in the experiments.

2.3 Results

In this section, results of the experiments are described. Each experiment is analyzed in separate subsection. Subsection 2.3.3 contains summary of the results. A discussion of results follows in Chapter 3.

Note, that in all figures and tables below, hyper-parameters with prior log-uniform distribution are visualized with a logarithm scale.

[7] `http://jupyter.org/`

	validation error (%)			test error (%)		
method	mean	median	std	mean	median	std
RANDOM	7.14	6.90	1.19	8.44	8.48	1.38
GP	8.27	7.95	2.30	9.07	8.58	2.42
TPE	7.94	7.97	1.08	9.05	8.99	1.10
RF	7.35	7.58	1.37	8.39	8.11	1.61
HYPERBAND	**6.25**	**6.43**	**0.78**	**7.46**	**7.38**	**0.96**

Table 2.2: The error rate in percentages for the MNIST dataset for configurations found by individual hyper-parameter optimization methods

2.3.1 MNIST

In the first experiment, a configuration of a neural network for the MNIST dataset is seeked. The optimization methods are compared in terms of found configurations and ways how they reach the solution, meaning what configurations and regions of hyper-parameter space the methods focus on.

Table 2.2 and figures 2.1 and 2.2 show validation error (cost function) and test error for configurations found by each method. Note that the common approach is to compare methods using a mean error, however, the mean may be biased by distant values. In that sense, the median error is more representative and thus is used in plots below. The table and figures contain the mean and the median (the solid line marks the median, the dashed line represents the mean) as well. A standard deviation is given to represent sensitivity to initialization. As we can see, Hyperband found the best configurations. Moreover, it has a low variance. There are two reasons for this. Hyperband generates a larger number of configurations than other methods, thus, the hyper-parameter space is explored better resulting in less dependence on random generator setting. Furthermore, as the results show, the poorly performing configurations are easily distinguished, thus Hyperband drops them in initial stages leaving more resources for the promising configurations. The performance of other methods is slightly worse. We may notice, that SMBO with Gaussian processes has a high variance. The cause is a small number of evaluations which results in the dependence of the model's accuracy on the initial sampling. As we will see in the experiment for the MRBI dataset, once enough data points are given to sufficiently represent the hyper-parameter space, the variance decreases. In general, there are small differences between methods which suggest that the task is easy to solve.

To see a progress of the optimization, a convergence is shown in Figure 2.3. The plot uses logarithm scale on the y-axis to emphasize the differences. Note the due to early stopping used for the networks, the optimization times differs

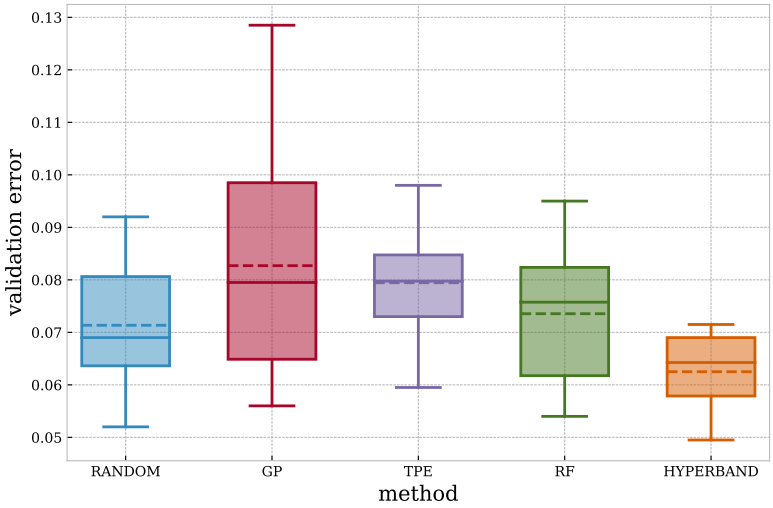


Figure 2.1: The validation error of the best found configurations for the MNIST dataset.

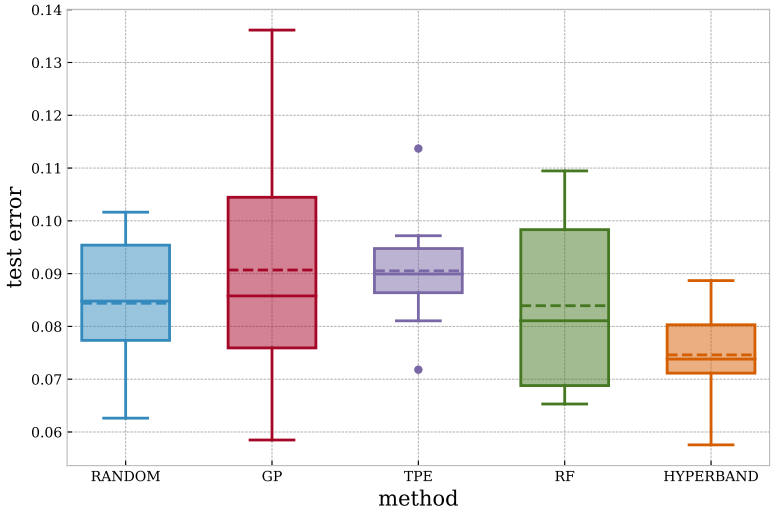


Figure 2.2: The test error of the best found configurations for the MNIST dataset.

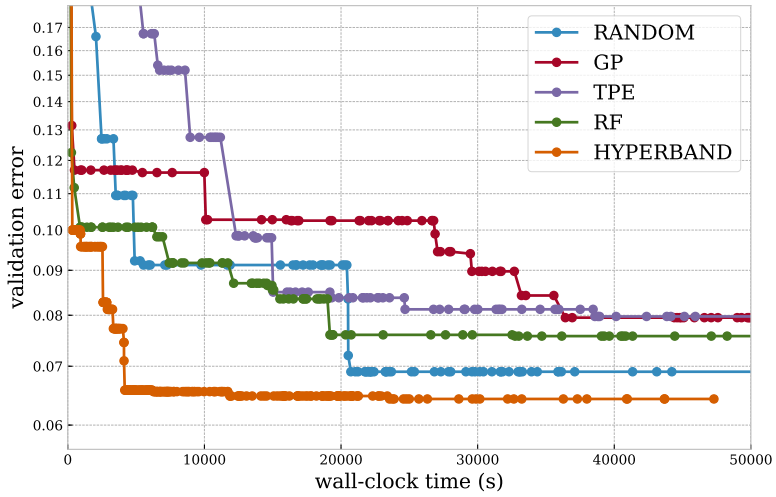


Figure 2.3: MNIST convergence.

depending on which configurations is generated in particular run. However, in average, the methods require similar time. As we can see, Hyperband found a good solution fast and then did not improve it. We will see below, that the bracket that find the best configurations is usually the most exploratory one. In the Hyperband's implementation this bracket runs as the first one, thus, gives the results at the beginning of the optimization. The convergence of the remaining methods is similar to one another.

In order to analyze configurations generated by individual methods, a five percent of the best performing configurations are explored. The values of the top configurations are visualized by a parallel plot in Figure 2.4. The color corresponds with the method which found each configuration. As we can see, the top configurations cover a large part of the space and none of the methods seems to focus on a specific area.

As for the values of these top configurations, a learning rate has narrow range suggesting that it is the most important hyper-parameter. To confirm this, automatic relevance determination (ARD) using Gaussian processes with Matérn kernel with smoothness parameter $\nu = 5/2$ was performed. The Matérn kernel belongs to a group of kernels that allows determining the importance of each input variable based on the optimized model parameters. Specifically, the training with Matérn kernel optimizes parameter *length-scale* for each input variable. This length-scale then determines the influence of the variable. The lower it is, the more relevant the variable is. For ARD of hyper-parameters the data sampled by random search were used. The training was repeated 50 times with 80% random points from 100 configurations. Table 2.3 shows median of inverse lenght-scale, i.e. the relevance measure. From the results, we can see that the learning rate is the most important parameter

hyper-parameter	1/lenght-scale
learning rate	**3.77**
learning rate decay	0.63
number of hidden units	**2.18**
activation	0.91
weights initialization	0.47

Table 2.3: The relevance of each hyper-parameter as determined by Gaussian processes with Matérn kernel for the MNIST dataset.

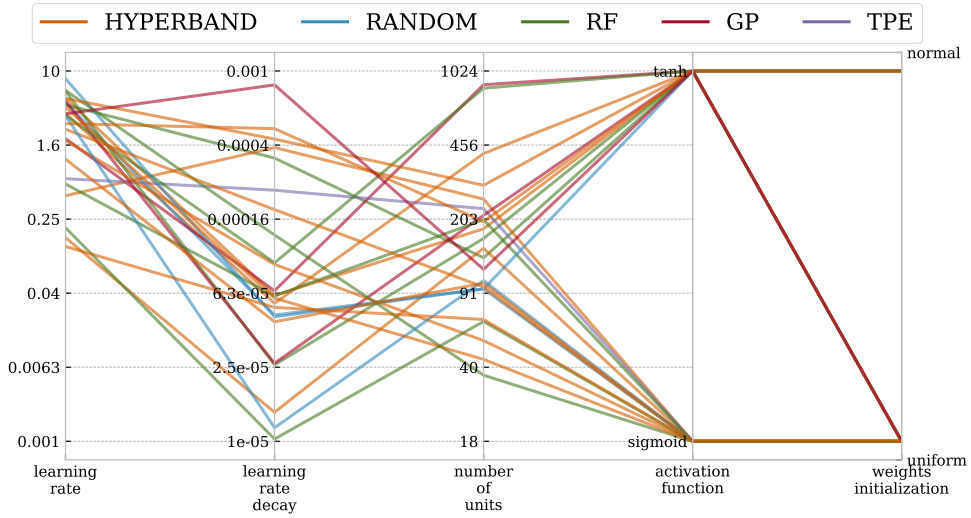


Figure 2.4: The best found configurations for the MNIST dataset.

(1/lenght-scale = 3.77), the second most relevant parameter is the number of hidden units (2.18). The other parameters have inversed length-scale smaller than one meaning that they are not considered relevant. The most relevant hyper-parameters have the highest impact on optimization results, thus, the analysis focus on them.

The behavior of individual methods is illustrated by pair plots for numerical hyper-parameters and validation error. Each pair plot contains all configurations generated by all runs. The red color in each pair plot marks the top configurations.

2.3.1.1 SMBO with Gaussian processes

The Gaussian process model is very accurate in prediction and uncertainty estimate. Figure 2.5 shows a pair plot of generated configurations by this

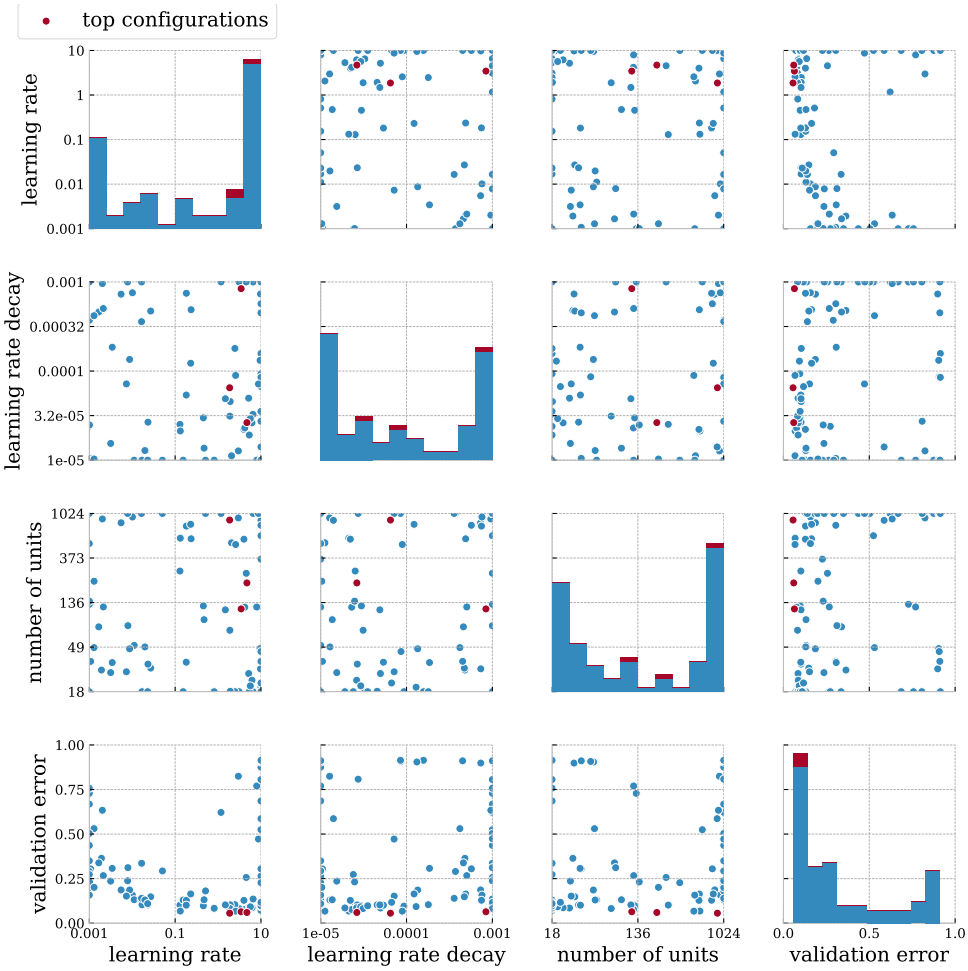
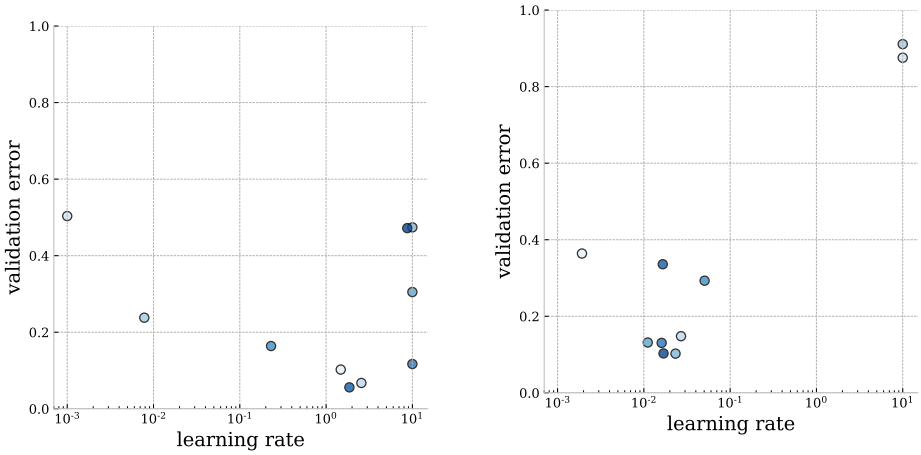


Figure 2.5: Sampled values by SMBO with the Gaussian process model for the MNIST dataset.

method. As we can see from diagonal histograms for hyper-parameters, the method often samples border values of the configuration space. Even the scatter plots for pairs of hyper-parameters contains a large number of points on the edges. This is caused by the the uncertainty estimate which forces the optimizer to explore the space when a small number of data points is given. The sparsity of the dataset results in regions with high uncertainty and, in consequence, in high acquisition function for these regions. The points with boundary values are usually the furthest from already sampled points, thus they have the highest uncertainty.

As already said, the Gaussian processes-based optimization has high variance which is caused by the initial exploration. If the configurations sampled at the beginning are not representative, they can bias the search to regions



(a) Sampling that leads to good results

(b) Sampling that bias the search to sub-optimal space

Figure 2.6: Sampling by SMBO with the Gaussian process model for two different intitializations.

with suboptimal configurations. The optimization is given relatively little time, thus the model does not have enough time to correct this. In the left bottom panel in Figure 2.5 we can notice points that have high learning rate and high validation error. These points can confuse the model to think that high learning rate is not desirable, even though the top configurations have high learning rate as illustrated by the red points. Figure 2.6 illustrates this for two optimization runs with different intitialization. The figure shows the learning rate and the validation error in individual iterations (visualized by color, the darker the color, the higher the iteration number). In the left panel, a run where good configurations are sampled at the beginning is shown. This sampling results in a focus on the region with high learning rate, thus, the optimization yields a good configuration. On the other hand, the second panel shows a run where two configurations with high learning rate and validation error are sampled at the beginning (the two top right points). The search then focuses on low values of learning rate and does not find good configuration.

2.3.1.2 SMBO with random forests

Random forests are not so accurate in uncertainty estimates as Gaussian processes. Therefore, the borders are not sampled the same way as with the Gaussian process model (see Figure C.1). The optimization results have lower variance than Gaussian processes-based optimization. Similarly to Gaussian processes, unsuitable initial configurations can bias the search towards suboptimal regions. However, the majority of the poorly performing configurations with high learning rate is placed on the borders of the hyper-parameter space

which is not sampled so often by SMBO with random forests, thus the variance is smaller.

A little more samples were generated in the region with high learning rate which contains the top configurations. However, the changes are inconclusive, probably due to lack of time given to optimization.

2.3.1.3 SMBO with tree-structured parzen estimator

The first thing to notice in pair plot for the TPE in Figure 2.7 is that the model often generates correlated values for different hyper-parameters (see for example panel for the learning rate and the number of hidden units in the first column and the third row). It is even more clear from Figure 2.8 which shows sampled values for one run where this correlation appeared between the learning rate and the number of hidden units. To find the cause, we have to look at the way how TPE is built. Each evaluated configuration changes the prior distribution. Specifically, for uniform (or log-uniform) distribution, the prior distribution is replaced with truncated Gaussian mixture model with means of individual Gaussian distributions in sampled values and variances set to a greater of the distances to the left and right neighbor for each input point. Therefore, if values for two independent variables with uniform prior distribution lie in the same region of the distribution, the change in their distributions will be the same up to scaling. Thus, their shapes will be the same and they will produce correlated values. This is the reason of the poor results and low variance of TPE – many runs encounter into this problem. Especially, it is a problem when the two correlated hyper-parameters are the two most important ones, the learning rate and the number of hidden units, which however have different optimal values (lies in different regions of the prior distribution). To prevent this, the optimization should be initialized with more than one random point. The more initial random points, the lower probability that the posterior distributions will have the same shape.

2.3.1.4 Hyperband

The Hyperband's setting for MNIST results in 3 brackets (Successive Halving runs) with a different trade-off between a number of generated configurations and resources given to each configuration. The first bracket generated 16 configurations, the second 6 and the third 3 configurations with median validation error of the best configurations 6.48%, 8.03%, and 8.58%, respectively. As we can see, the best bracket is the one which generates a large number o configurations and assigns them a small number of resources (epochs). Thus we see, that the good configurations are easy to distinguish even after only a few epochs.

A pair plot in Figure 2.9 shows all configurations that finished the evaluation, that is they used a maximal number of resources or were stopped

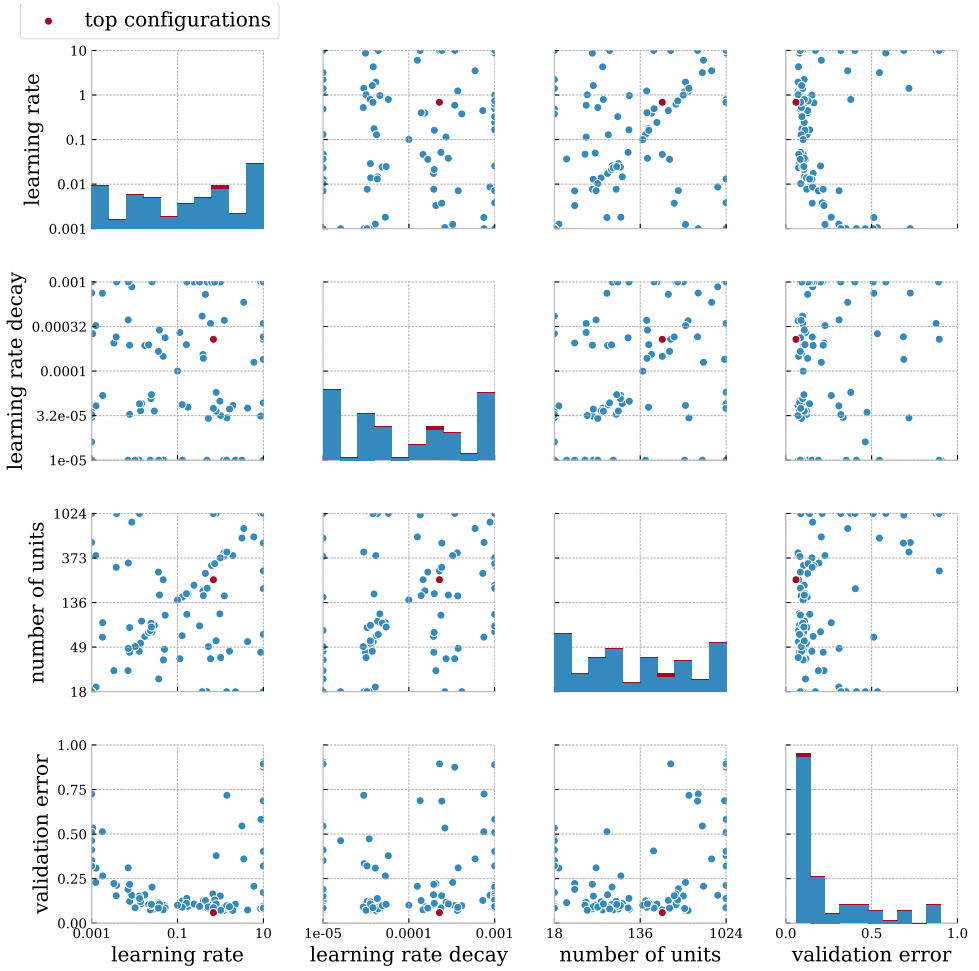


Figure 2.7: Sampled values by SMBO with tree-structured parzen estimator for the MNIST dataset.

prematurely because the performance did not increase. In panels in the left column, we can notice a focus on large learning rate which corresponds to values for the top configurations. Thus, Hyperband keep the best configurations for a full evaluation. It is important to note that the learning rate has a great impact on the convergence rate. This might be partly the reason for the concentration on the large learning rate and the smaller number of units (we can notice that in the panel for learning rate and number of hidden units in the left column on the third row). These parameters influence the convergence of the network (the higher the learning rate the faster the convergance, and the smaller network the faster the training). The fact that high learning rate turned out to be optimal could be one of the causes why the first bracket is good. It prefers networks that learn fast. If the lower values were better, the

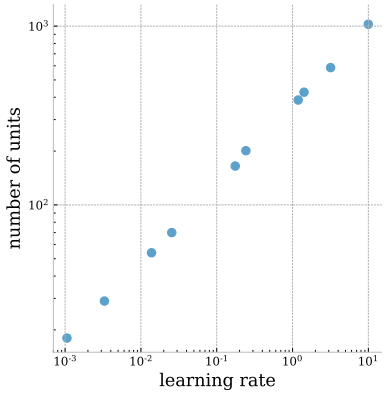


Figure 2.8: The correlation of sampled values for the learning rate and the validation error SMBO with tree-structured parzen estimator.

good configurations could be discarded prematurely by this bracket.

2.3.2 MRBI

The second experiment optimizes a neural network for the MRBI dataset. In Table 2.4 we can see the validation and the test error of the best-found configurations (box plots are given in Appendix C). As in case of MNIST, Hyperband found the best configurations. SMBO with different models yielded similar performance. However, as we can see in contrast to MNIST, Gaussian processes have low variance and, on the other hand, random forests have high variance. Moreover, this task proved to be more difficult which results in the worst performance of random search. The search space is larger and the prior distribution forces random search to explore it uniformly. On the other hand, as we will see, the SMBO methods change the distribution to sample the promising regions more often and Hyperband assign more resources to the promising configurations.

The convergence for all methods is shown in Figure 2.10. Hyperband again quickly finds a good configuration. Random search and SMBO are comparable at the beginning of the optimization. However, around 15th iteration, SMBO starts to outperform random search. The convergence for different surrogates is similar, however as we will see, there are differences in the optimizer's behavior depending on the selected model.

As with the MNIST experiment, in order to explore the methods, the best configurations are explored. However, a larger number of configurations is generated for MRBI, thus, only three percent of the best configurations are selected. This results in 42 top configurations. From these configurations, 37 have only one layer suggesting that one-layer networks are the most suitable for the task. Table 2.5 shows the percentage of configurations with different numbers of layers generated by random search and SMBO. According to prior

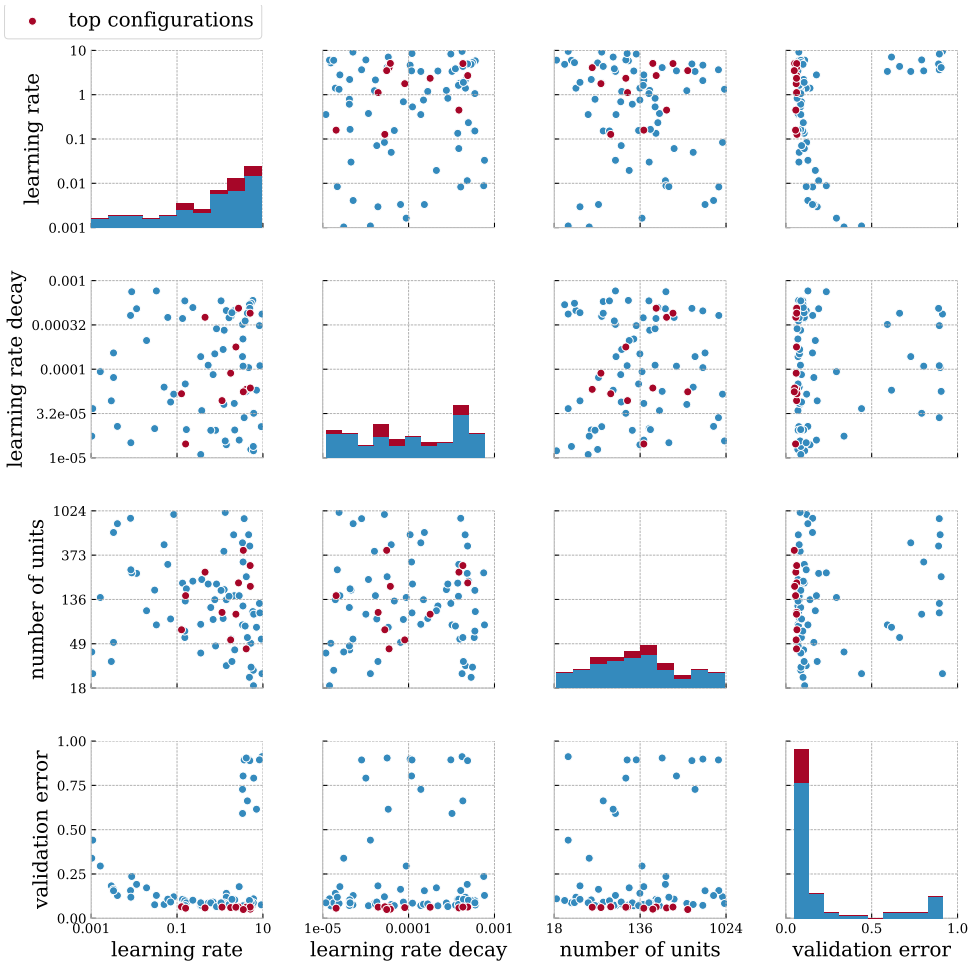


Figure 2.9: Sampled values by the Hyperband for the MNIST dataset

	validation error (%)			test error (%)		
method	mean	median	std	mean	median	std
RANDOM	62.62	62.88	1.52	63.57	63.50	2.04
GP	60.65	60.90	1.53	61.65	61.15	**1.67**
TPE	61.62	60.90	1.95	63.07	62.18	2.44
RF	61.35	60.70	2.86	62.57	61.29	2.92
HYPERBAND	**60.15**	**60.05**	**1.42**	**61.16**	**60.90**	2.15

Table 2.4: The error rate in percentages for the MRBI dataset for configurations found by individual hyper-parameter optimization methods

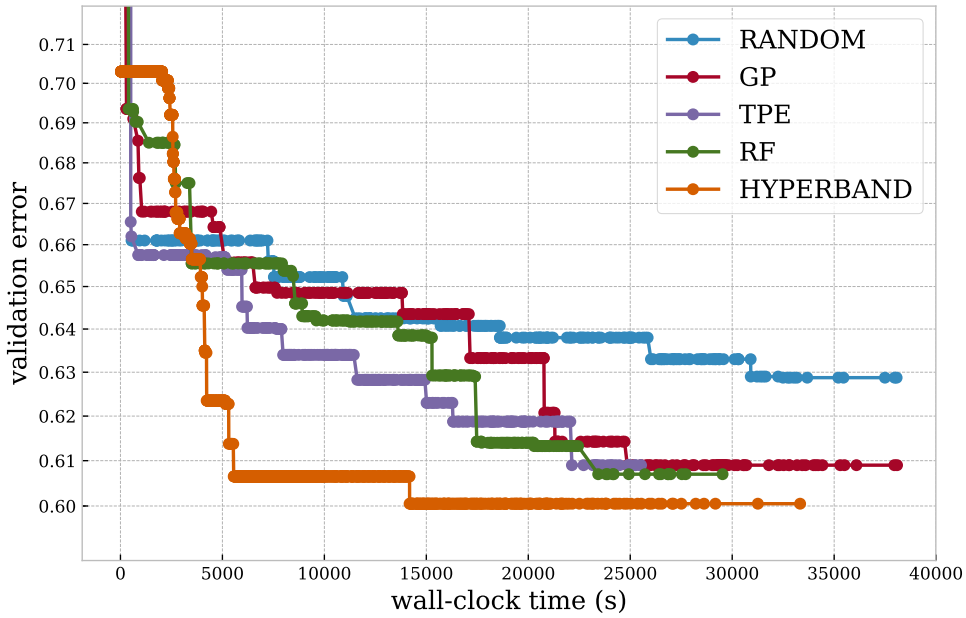


Figure 2.10: MRBI convergence.

50% of configurations should have one layer, and 25% two and 25% three layers. The random search indeed follows the prior. However, model-based methods alters the distribution to focus on one-layer networks. This is even more obvious for the last 10 iterations where the model changes the distribution the most. The Hyperband's configurations are shown in Table 2.6. The table contains only configurations that finished the evaluation. As we can see, for all brackets the numbers follow the prior, however, this is largely caused by the later brackets which behave as random search. For the first three most aggressive brackets, we can see that one-layer networks receive more attention.

2.3.2.1 SMBO

Figure 2.11 shows histograms for sampled values of the learning rate for optimization with SMBO. Each panel shows sampled values for the first (left column) and the last (right column) ten evaluations using individual models (rows). The red lines marks values for the top configurations, thus, the regions which the optimization should focus on. As we can see, the frequency of different values changes during the search. All models bias the search to regions where the best performing configurations are found. However, the exact shape of the histogram is dependent on the model.

Similarly, as in the MNIST experiment, the Gaussian process surrogate forces the optimizer to explore the space at first focusing on the border values (the top left panel). However, for the MRBI experiment the optimization

method	number of layers	all iterations (%)	last 10 iterations (%)
RANDOM	1	51	54
	2	22	18
	3	26	28
GP	1	65	76
	2	16	10
	3	19	14
TPE	1	71	74
	2	16	17
	3	13	9
RF	1	68	77
	2	15	13
	3	17	10

Table 2.5: The percentage of configurations with 1, 2, and 3 layers generated by random search and SMBO for MRBI dataset

method	number of layers	all brackets (%)	first 3 brackets (%)
HYPERBAND	1	55	65
	2	25	22
	3	20	13

Table 2.6: The percentage of configurations with 1, 2, and 3 layers that finished evalution with Hyperband for MRBI dataset

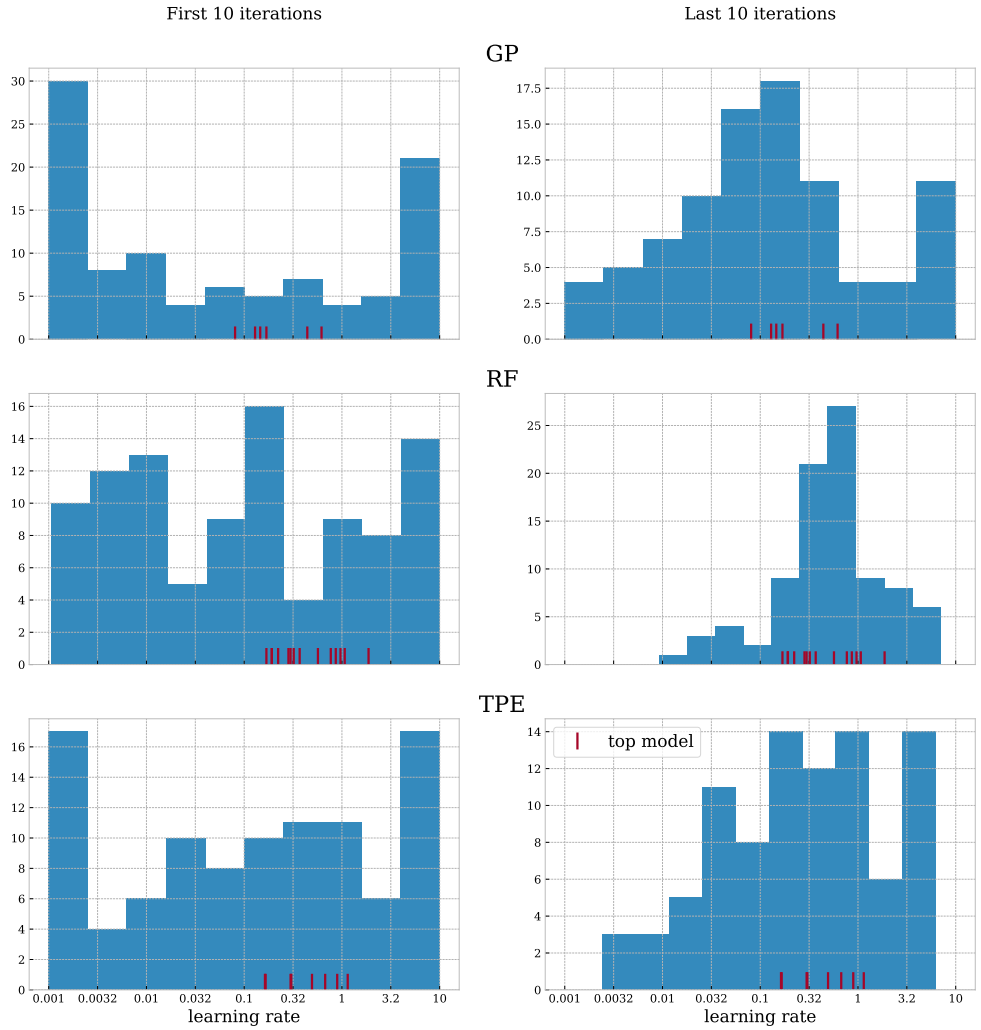


Figure 2.11: Sampled values for learning rate by SMBO with different models for the MRBI dataset.

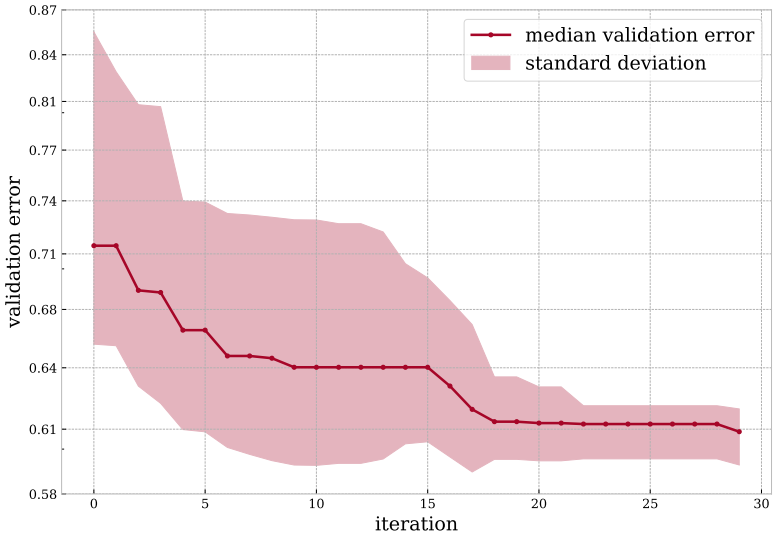
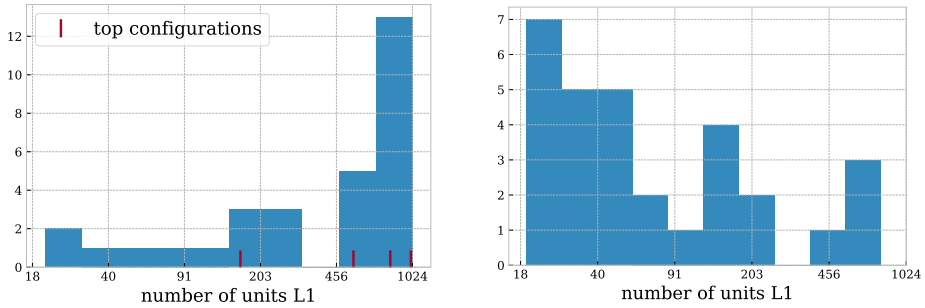


Figure 2.12: The convergence of SMBO with Gaussian processes for the the MRBI dataset.

receives more iterations. Thus, after several iterations, the uncertainty decreases and the optimization starts to focus on the promising region (the top right panel). Thanks to the initial exploration phase, Gaussian processes have a low variance. The initial exploration gives us global information which is not dependent on initialization. A price we pay for this information is high error and variance at the beginning as shown Figure 2.12. The figure contains the median error for the best model from individual runs (the line) and the region around it with the width corresponding with the value of standard deviation. This width is large at the beginning, however, in later iterations, it decreases. Note that thanks to the fact that one-layer networks reported the best performance, Gaussian processes can consider conditional hyper-parameters irrelevant which simplifies the optimization.

Random forests do not explore the space as Gaussian processes. The model does not sample the borders (left panel in the second row of Figure 2.11). Moreover, it often starts to focus on the specific area from the beginning which results in fast convergence to local optima making the optimization more sensitive to initialization. Scatter plots in Figure 2.13 illustrate this for two different runs with the number of hidden units in the first layer (denoted as L1). In Panel 2.13a, the search focuses on a larger number of hidden units which is indeed the area where the best models are found. However, for a different run, shown in the second panel (2.13b), the main focus is on a smaller number of units. Moreover, if we would look at the sampled values depending on iteration for the second run (Appendix C.4), we might notice that the configurations with the larger number of hidden units are generated



(a) Run with good results

(b) Run with poor results

Figure 2.13: Sampling by SMBO with the random forest model for two runs with different intitializations.

in later iterations. The first iterations focus only on small networks. However, the fact that the search starts to generate higher values in the last iterations suggest that it would escape from the local optima if given more time. The higher exploitation causes the higher variance of the results due to its higher sensitivity on the initialization.

SMBO with TPE is initialized with one random point which sometimes results in generation of configurations with correlated values for some hyperparameters. However, in most cases the values are not completely correlated and after several iterations, the distributions are different enough to generate different values. Example of such case can be found in Appendix C.

In Figure 2.11, TPE corresponds with the last line. As we can see, SMBO with TPE samples the borders similarly to Gaussian processes suggesting that the uncertainty estimate is more accurate than random forest estimate. However, it starts to focus on a specific area earlier and on the other hand, it is less focused on optimal values in the last iterations.

2.3.2.2 Hyperband

The Hyperband finds best results in the first bracket as shown in Figure 2.14 suggesting that good configurations are easily distinguished. The box plot shows validation error of the best configurations found by individual brackets. Clearly, the first bracket, denoted according to the number of generated configurations as $n = 81$, has the lowest error. With decreasing number of configurations (and increasing number of resources for each of it) the error increases. Partly, this can be caused by the fact that 1-layer networks, which learn quickly, turned out to be the best. If the optimal networks had more layers, other brackets might be better.

As for the configurations that finished the evaluation, they are similar to the ones that SMBO concentrates on. Histogram of sampled values for the learning rate is shown in Figure 2.15. Each panel shows finished evaluations

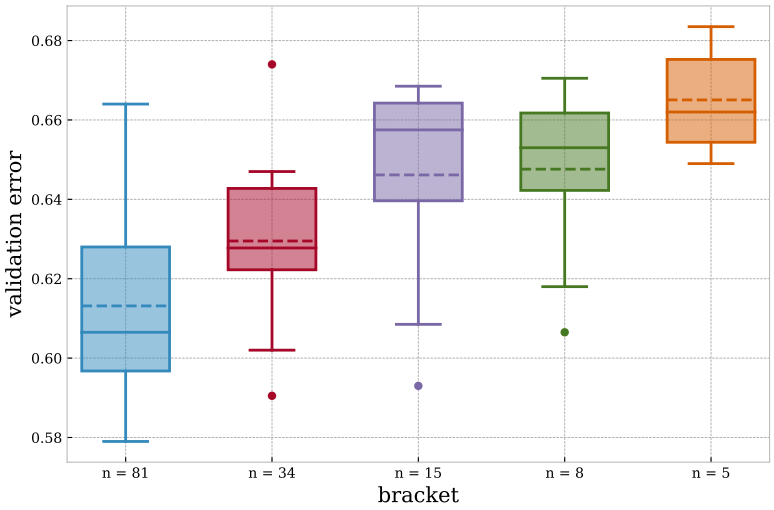


Figure 2.14: The validation error of configurations that finished evalution in different brackets of the Hyperband for the MRBI dataset.

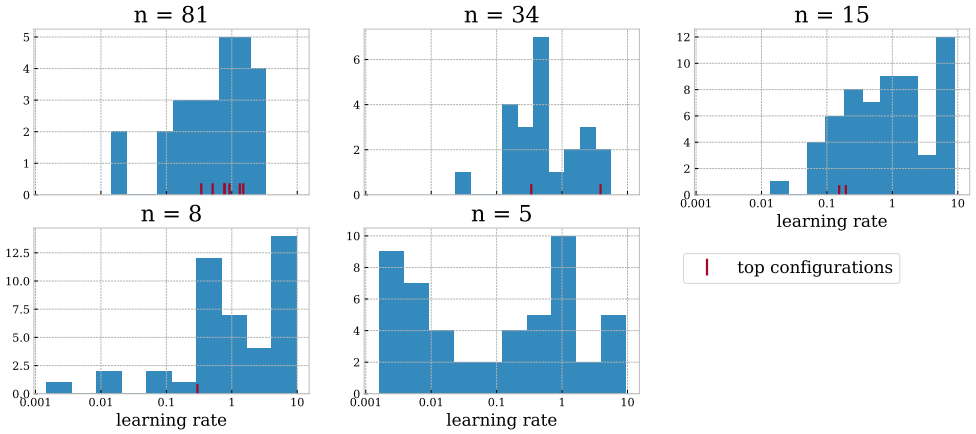


Figure 2.15: Sampled values for the learning rate by the Hyperband for MRBI dataset.

for one bracket. As we can see, only the last bracket ($n = 5$) which finishes all generated configuration does not concentrate on the good values.

2.3.3 Summary

In the previous section, five hyper-parameter optimization methods are compared on two datasets. In both experiments, Hyperband outperforms other methods yielding the smallest validation and test error. Moreover, the variance of the results from different optimization runs is low for Hyperband. The

reason behind it is an easy separation of the poorly performing configurations from the good ones. Hyperband recognizes these poor configurations, discards them, and assigns more resources to the promising configurations.

Different surrogate models in SMBO yields a similar performance. However, there are differences in variance and overall behavior of SMBO with different surrogates.

Gaussian processes force the optimizer to explore the space which results in high variance for a small number of evaluations and, on the other hand, a low variance for later iterations when optimized space is sufficiently sampled. Random forest-based optimization is less exploratory. Due to inaccurate uncertainty estimate from random forests, the optimization focus on a local area. This often results in high performance due to the larger proportion of time spent in the promising region. However, in some cases, the optimization get stuck in a suboptimal region. The tree-structured parzen estimator can cause sampling of correlated values for different input variables. Thus, if the well-performing configurations do not have these hyper-parameters correlated, the optimization ends with a high validation error. However, as the experiment with MRBI shows, SMBO with TPE explores the space in a similar way to Gaussian processes.

Random search is used as a baseline. The experiments shows that for a simple problem, random search is comparable to other methods. However, for more complicated problem, it yields worse performance due to wasting of resources on poorly-performing configurations.

Chapter 3

Discussion

In the previous chapter, five hyper-parameter optimization methods are used to optimize hyper-parameters of a feed-forward neural network. Overall summary of the results is given in 2.3.3. However, there are some points worth to discuss in more detail.

The first thing to note is that both used datasets come from image domain which is only one of the domains where machine learning methods are being used. Similarly, feed-forward networks represent only a small fraction of neural networks and machine learning algorithms in general. For example, so-called convolutional neural networks yield state-of-the-art performance on many image datasets, such as the ones used in this work. Feed-forward networks were choosen as good representant to evaluate how the optimization works, however there is space to experiment with further architectures and datasets.

On both presented problems, Hyperband outperforms other methods. However, as analysis shows, both of these tasks have a rather convenient optimal setting for Hyperband – the good configurations corresponds with smaller networks (1-layer network for MRBI) and higher learning rate. Both of these facts cause that the learning converges quickly making the optimal configurations easily distinguish from the poor ones at the beginning of the training. It is a question, whether Hyperband would perform so well if a more difficult problem in terms of model complexity and slower learning was presented. Similarly, Gaussian process model performs well on given tasks with mixed numerical, categorical and conditional hyper-parameters even though it is not designed for them. However, the numerical hyper-parameters turned out to be the most relevant.

In the theoretical part, another group of surrogate models is described – artificial neural networks. The SMBO in experiments does not use this surrogates. It might be interesting to compare its behavior against the other models. For example the DNGO model should be able to handle conditional variables and have similar properties as Gaussian processes. Comparation

on both low dimensional problems with independent variables and high dimensional mixed space could show whether DNGO would behave similar or outperformed GPs. In general, neural networks might learn interesting features without complicated specification of kernels as in case of GPs.

As mention, the two techniques that proven to be useful in terms of being resource efficient are a partial evaluation of the configurations which provides an insight into configuration's performance for a small price and altering the input distribution using a surrogate model. These two approaches are not in conflict, thus, further research on combination both seems to be promising. Some steps were already taken in this direction. As mentioned in theoretical part, Klein *et al.* [21] use a neural network model which attemps to model a learning curve for the training and use this model for Hyperband configuration generation. Recently, Falkner *et al.* [42] combined TPE model with Hyperband.

Perhaps the simplest approach might be to provide SMBO several partially evaluated configurations as initial points. This could help to initialize the optimization. How the experiments showed, the performance of the models at the beginning of the optimization is dependent on initial sampling. Typically, it takes several steps before SMBO starts to give good results not influenced by initial configuration. In setting where one function evaluation may take hours or days, we would like to reduce the number of these initial steps. A promising direction seems to be meta-learning – using results from previous tasks to warm-start the optimization. As described in section 1.4 already proposed methods include selecting a list of configuration based on results from previous datasets [30], or building a surrogate model that incorporates information from previous tasks [5, 43, 44].

Conclusion

This work deals with a problem of finding a hyper-parameter setting for machine learning algorithms. The problem is formulated as an optimization problem which seeks a minimum of a loss function depending on hyper-parameters of the training algorithm. The complexity of hyper-parameter space and cost of one evaluation makes this optimization non-trivial task.

Four methods that aim to automatically optimize given function are reviewed in theoretical part – grid search, random search, sequential-model based optimization and Hyperband. Some of them use a surrogate model that approximates the optimized function. This model improves the optimization by directing the search to promising regions of optimized space. Several such models – Gaussian processes, random forests, tree-structured parzed estimator, and neural networks – are described as well.

Practical part focuses on methods comparison. Five combinations of the described methods and models are evaluated and their behavior is analyzed on two datasets with various degrees of complexity.

The results shows that Hyperband outperformed other methods on both tasks owing its success to a large number of generated configurations and discarding the poorly performing ones. Sequential-model based optimization is another compared method. Three surrogate models are used. The analysis shows that the models transforms the distribution from which the configurations are sampled resulting in concentration on configurations that are likely to perform well. The main drawback is a higher variance of the optimization results which is caused by the bias of some runs to suboptimal regions. The problem arise especially if only a few function evaluations are allowed. Discussed solutions includs partial evaluation similar to Hyperband evaluations and meta-learning which can bias the search based on information from similar tasks. Random search has proven to perform well on a simple problem, however, for the more complex problem, the optimization wastes resources on poorly-performing configurations.

Machine learning methods are a tool for gaining useful information from

data. With the development of complex models, the hyper-parameter setting is a crucial part of the usage of these models on concrete problems. This work confirmed, that automatic hyper-parameter optimization methods are a useful tool for tuning this setting. Nevertheless, there is still a large space for improvement of these methods, such as speeding up the initial phase of the search, to yield state-of-the-art performance in reasonable time.

Bibliography

[1] Thornton, C.; Hutter, F.; et al. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. *CoRR*, volume abs/1208.3, 2012: pp. 847–855, ISSN 1613-0073, doi:10.1145/2487575.2487629. Available from: http://arxiv.org/abs/1208.3719

[2] Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, volume 13, no. Feb, 2012: pp. 281–305, ISSN 1532-4435, doi:10.1.1.306.4385. Available from: http://www.jmlr.org/papers/v13/bergstra12a.html

[3] Hutter, F.; Hoos, H. H.; et al. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, 2011, ISSN 0302-9743, pp. 507–523, doi:10.1007/978-3-642-25566-3_40. Available from: https://www.cs.ubc.ca/{~}hutter/papers/10-TR-SMAC.pdf

[4] Larochelle, H.; Erhan, D.; et al. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine Learning*, ACM, 2007, ISBN 9781595937933, pp. 473–480, doi:10.1145/1273496.1273556. Available from: https://www.iro.umontreal.ca/~lisa/twiki/pub/Public/DeepVsShallowComparisonICML2007/icml-2007-camera-ready.pdf

[5] Swersky, K.; Snoek, J.; et al. Multi-task Bayesian optimization. In *Advances in Neural Information Processing Systems 26*, Curran Associates, Inc., 2013, pp. 2004–2012. Available from: http://papers.nips.cc/paper/5086-multi-task-bayesian-optimization.pdf

[6] Klein, A.; Falkner, S.; et al. Fast Bayesian optimization of machine Learning hyperparameters on large datasets. *CoRR*, volume abs/1605.07079, 2016, ISSN 1938-7228. Available from: https://arxiv.org/abs/1605.07079

[7] Swersky, K.; Snoek, J.; et al. Freeze-thaw Bayesian optimization. *CoRR*, 2014: pp. 1–12, ISSN 1049-5258. Available from: http://arxiv.org/abs/1406.3896

[8] Bergstra, J. S.; Bardenet, R.; et al. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, Curran Associates, Inc., 2011, ISBN 9781618395993, pp. 2546–2554. Available from: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[9] Li, L.; Jamieson, K.; et al. Hyperband: A novel bandit-based approach to hyperparameter optimization. *CoRR*, volume abs/1603.06560, 2016. Available from: http://arxiv.org/abs/1603.06560

[10] Ilievski, I.; Akhtar, T.; et al. Hyperparameter optimization of deep neural networks using non-probabilistic RBF Surrogate Model. *CoRR*, volume abs/1607.08316, 2016. Available from: http://arxiv.org/abs/1607.08316

[11] Springenberg, J. T.; Klein, A.; et al. Bayesian optimization with robust Bayesian neural networks. In *Advances in Neural Information Processing Systems 29*, Curran Associates, Inc., 2016, pp. 4134–4142. Available from: http://papers.nips.cc/paper/6117-bayesian-optimization-with-robust-bayesian-neural-networks.pdf

[12] Snoek, J.; Rippel, O.; et al. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, 2015, ISBN 9781510810587, ISSN 1938-7228, pp. 2171—-2180. Available from: https://arxiv.org/abs/1502.05700

[13] Diaz, G. I.; Fokoue, A.; et al. An effective algorithm for hyperparameter optimization of neural networks. *CoRR*, volume abs/1705.08520, 2017. Available from: http://arxiv.org/abs/1705.08520

[14] Forrester, A. I.; Keane, A. J. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, volume 45, no. 1-3, 2009: pp. 50–79, ISSN 0376-0421, doi:10.1016/j.paerosci.2008.11.001.

[15] Shahriari, B.; Swersky, K.; et al. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, volume 104, no. 1, jan 2016: pp. 148–175, ISSN 0018-9219, doi:10.1109/JPROC.2015.2494218. Available from: http://ieeexplore.ieee.org/document/7352306/

[16] Brochu, E.; Hoffman, M. D.; et al. Hedging strategies for Bayesian optimization. *CoRR*, volume abs/1009.5419, 2010. Available from: http://arxiv.org/abs/1009.5419

[17] Shahriari, B.; Wang, Z.; et al. An entropy search portfolio for Bayesian optimization. *CoRR*, 2015. Available from: https://arxiv.org/abs/1406.4625

[18] Brochu, E.; Cora, V. M.; et al. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement Learning. *CoRR*, volume abs/1012.2599, 2010. Available from: http://arxiv.org/abs/1012.2599

[19] Gardner, J. R.; Matt Kusner, W. J.; et al. Bayesian optimization with inequality constraints. In *International Conference on Machine Learning*, volume 32, 2014, pp. 937–945. Available from: http://proceedings.mlr.press/v32/gardner14.pdf

[20] Jamieson, K. G.; Talwalkar, A. Non-stochastic best arm identification and hyperparameter optimization. *CoRR*, volume abs/1502.07943, 2015. Available from: http://arxiv.org/abs/1502.07943

[21] Aaron Klein, J. T. S. F. H., Stefan Falkner. Learning curve prediction with Bayesian neural networks. 2017.

[22] Bartholomew-Biggs, M.; Brown, S.; et al. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, volume 124, no. 1, 2000: pp. 171–190, ISSN 0377-0427, doi:10.1016/S0377-0427(00)00422-2, numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations. Available from: http://www.sciencedirect.com/science/article/pii/S0377042700004222

[23] Maclaurin, D.; Duvenaud, D.; et al. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, volume 37, 2015, pp. 2113–2122. Available from: http://arxiv.org/abs/1502.03492

[24] Pedregosa, F. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, 2016, pp. 737–746. Available from: http://proceedings.mlr.press/v48/pedregosa16.pdf

[25] Hansen, N. The CMA Evolution Strategy: A Tutorial. *CoRR*, volume abs/1604.00772, 2016. Available from: http://arxiv.org/abs/1604.00772

[26] Loshchilov, I.; Hutter, F. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, volume abs/1604.07269, 2016. Available from: http://arxiv.org/abs/1604.07269

[27] Kennedy, J. *Particle swarm optimization*. Boston, MA: Springer US, 2010, pp. 760–766, doi:10.1007/978-0-387-30164-8_630.

[28] Guo, X.; Yang, J.; et al. A novel LS-SVMs hyper-parameter selection based on particle swarm optimization. *Neurocomputing*, volume 71, no. 16-18, oct 2008: pp. 3211–3215, ISSN 0925-2312, doi:10.1016/J.NEUCOM.2008.04.027. Available from: https://www.sciencedirect.com/science/article/pii/S0925231208002932

[29] Rasmussen, C. E.; Williams, C. K. I. *Gaussian processes for machine learning (adaptive computation and machine learning)*. The MIT Press, 2005, ISBN 026218253X.

[30] Feurer, M.; Springenberg, J. T.; et al. Initializing Bayesian hyperparameter optimization via meta-learning. In *AAAI Conference on Artificial Intelligence Initializing*, AAAI Press, 2015, ISBN 0262511290, pp. 1128–1135.

[31] Lecun, Y.; Bottou, L.; et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, volume 86, no. 11, Nov 1998: pp. 2278–2324, ISSN 0018-9219, doi:10.1109/5.726791.

[32] Srivastava, N.; Hinton, G.; et al. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, volume 15, 2014: pp. 1929–1958. Available from: http://jmlr.org/papers/v15/srivastava14a.html

[33] Snoek, J.; Larochelle, H.; et al. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 2951–2959. Available from: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[34] Head, T.; MechCoder; et al. scikit-optimize/scikit-optimize: v0.5.2. mar 2018, doi:10.5281/zenodo.1207017. Available from: https://doi.org/10.5281/zenodo.1207017

[35] Breiman, L. Random Forests. *Machine Learning*, volume 45, 2001: pp. 5–32, ISSN 1573-0565, doi:10.1023/A:1010933404324. Available from: https://doi.org/10.1023/A:1010933404324

[36] Jones, E.; Oliphant, T.; et al. SciPy: Open source scientific tools for Python. 2001–, [Online; accessed 2018-04-20]. Available from: http://www.scipy.org/

[37] Waskom, M.; Botvinnik, O.; et al. seaborn: v0.5.0 (November 2014). nov 2014, doi:10.5281/ZENODO.12710. Available from: https://zenodo.org/record/12710

[38] Inc., P. T. Collaborative data science. 2015. Available from: https://plot.ly

[39] Abadi, M.; Agarwal, A.; et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015, software available from tensorflow.org. Available from: https://www.tensorflow.org/

[40] Chollet, F.; et al. Keras. https://keras.io, 2015.

[41] Bergstra, J.; Yamins, D.; et al. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, volume 28, Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 115–123. Available from: http://proceedings.mlr.press/v28/bergstra13.html

[42] Wang, J.; Xu, J.; et al. Combination of Hyperband and Bayesian optimization for hyperparameter optimization in deep learning. *CoRR*, volume abs/1801.01596, 2018. Available from: http://arxiv.org/abs/1801.01596

[43] Feurer, M.; Letham, B.; et al. Scalable meta-learning for Bayesian optimization. *CoRR*, volume abs/1802.02219. Available from: https://arxiv.org/abs/1802.02219

[44] Wistuba, M.; Schilling, N.; et al. Hyperparameter optimization machines. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2016, ISBN 9781509052066, pp. 41–50, doi:10.1109/DSAA.2016.12.

Appendix A

Notation

$\mathcal{A}$	machine learning algorithm
$\mathcal{A}_{\mathbf{x}}$	machine learning algorithm with configuration $\mathbf{x}$
d	dimension of hyper-parameter space
D	training set of configurations $\{(\mathbf{x}_i, y_i) \vert i = 1, \ldots, n\}$
$\mathcal{D}_{\Pi}$	distribution of task Π
$\mathcal{D}_{\mathbf{x}}$	joint distribution of hyper-parameters
$\mathbb{E}$	expected value for given random variable
$f(\mathbf{x}), f$	cost function (expected loss) for configuration $\mathbf{x}$
$f(\mathbf{x}_i), f_i$	cost function (expected loss) for training configuration $\mathbf{x}_i$
$\mathbf{f}$	expected loss for all training configurations $\mathbf{f}^T = \begin{bmatrix} f_1 & f_2 & \ldots f_n \end{bmatrix}$
$\hat{f}(\mathbf{x})$	point prediction for $f(\mathbf{x})$
$\mathbb{1}$	indicator function
$\mathcal{L}$	loss function
$\mathcal{L}(\pi; \mathcal{A}_{\mathbf{x}}(\Pi_{train}))$, $\mathcal{L}(\pi)$	loss for sample π and algorithm $\mathcal{A}$ with configuration $\mathbf{x}$ trained on set of samples Π_{train}
$\mathcal{M}$	surrogate model
n	number of training data points (configurations)
$\mathcal{N}(\mu, \Sigma)$	Gaussian (normal) distribution with mean μ and covariance matrix Σ
$p(y\vert x), y\vert x$	conditional random variable y given x and its probability (density)
$\pi, (\pi, c)$	sample (instance) of Π, for classification task c is the class label of π

$\mathit{\Pi}$	machine learning task
Π	data sampled from $\mathit{\Pi}$
$\mathbb{R}$	real numbers
$\mathcal{U}(a, b)$	uniform distribution on interval (a, b)
x_i	i-th hyper-parameter
$\mathbf{x}$	data point (configuration)
$\mathbf{x}_i$	i-th training point (configuration)
X	an index set of hyper-parameter configurations
$\mathbf{X}$	$d \times n$ matrix of training points (configurations) $\{\mathbf{x}_i\}_{i=1}^{n}$
X_i	set of values for hyper-parameter i
$\mathcal{X}$	hyper-parameter space
y_i	observed value of f for training configuration x_i
$\mathbf{y}$	observed value of f for all training configurations

Appendix B

Acronyms

ARD Automatic relevance determination

CMA-ES Covariance matrix adaptation evolution strategy

DNGO Deep networks for global optimization

GP Gaussian process

EDA Estimation of distribution

EI Expected improvement

LCB Lower confidence bound

MSE Mean squared error

MRBI MNIST rotate background

PI Probability of improvement

PSO Particle swarm optimization

RANDOM Random search

RBF Radial basis function

RF Random forest

SGD Stochastic gradient descent

SMBO Sequential model-based optimization

SVR Support vector regression

TPE Tree-structured parzen estimator

Appendix C

Experiments Results

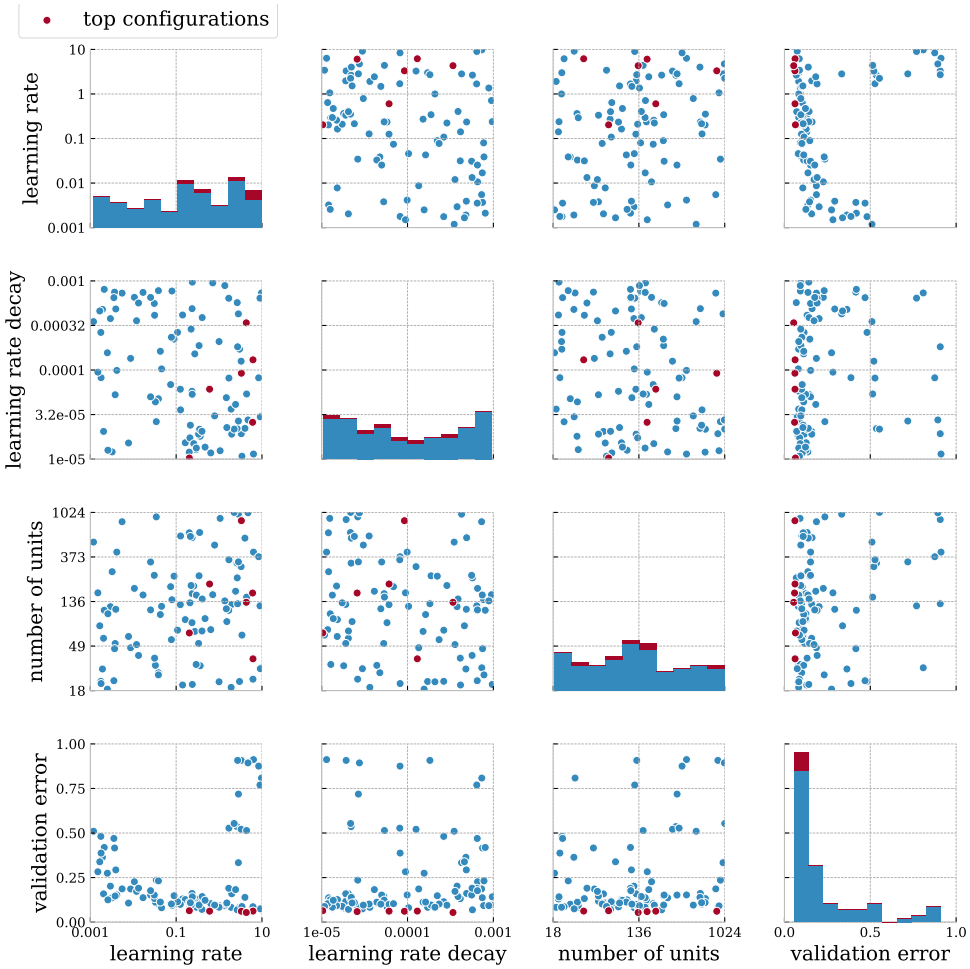


Figure C.1: Sampled values by SMBO with the random forest model for the MNIST dataset.

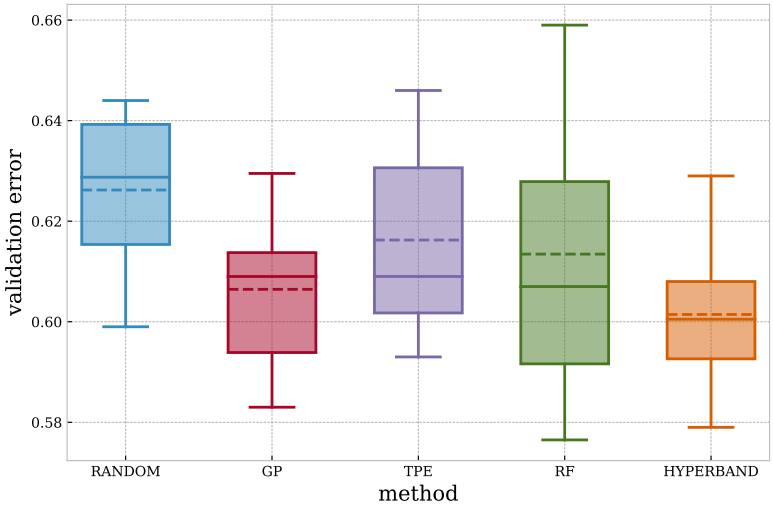


Figure C.2: The validation error of the best found configurations for the MRBI dataset.

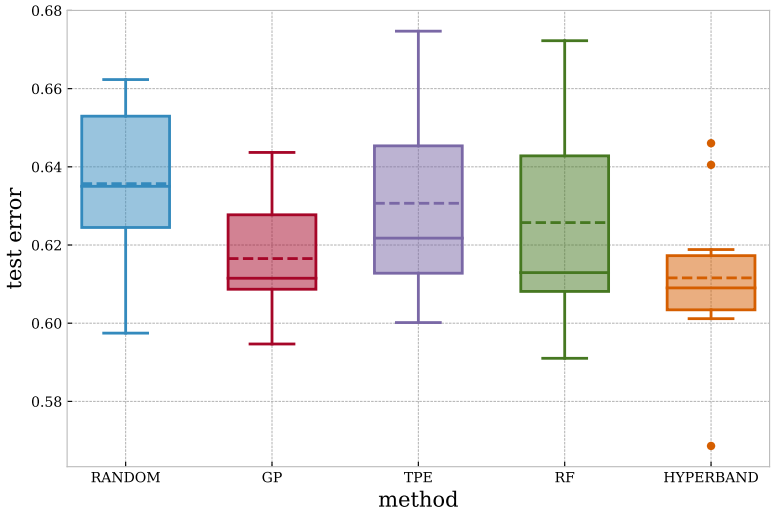
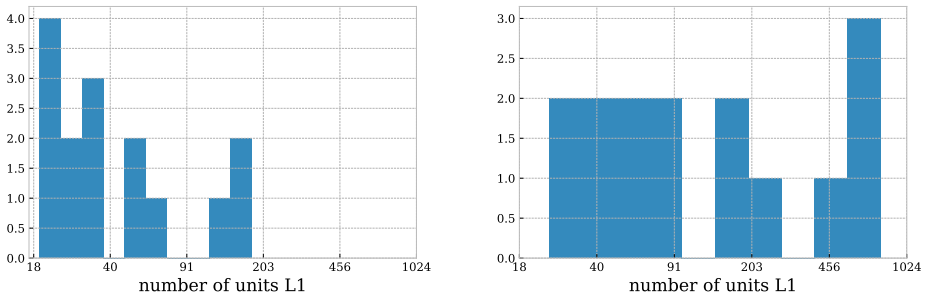


Figure C.3: The test error of the best found configurations for the MRBI dataset.



(a) First 15 iterations

(b) Last 15 iterations

Figure C.4: Sampling by SMBO with the random forest model which focused on values with suboptimal cost separated to first and last iterations.

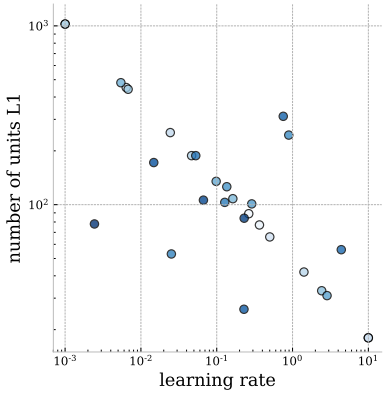


Figure C.5: The correlation of sampled values for learning rate and validation error SMBO with tree-structured parzen for MRBI.

Appendix D

Contents of CD

- `README.adoc` ..... the file with CD contents description
- `data` ..... the dataset files directory
- `docs` ..... the thesis text directory
  - `figures` ..... the directory with figures presented in text
  - `tables` ..... the directory with tables used in text
  - `sections` ..... the directory with source code for individual sections
  - `DP_Juzlova_Marketa_2018.tex` the LaTeX source code files of the thesis
  - `DP_Juzlova_Marketa_2018.pdf` .... the Diploma thesis in PDF format
- `environment.yml` .. the conda environment file with Python dependencies
- `notebooks` ..... the Jupyter notebooks with Python source code
  - `visualization` ..... the figures source code
  - `modeling` ..... the model and optimization source codes
  - `evaluation` ..... the evaluation source codes
  - `reports` ..... the the results evaluation
- `results` ..... the CSV files with the experiments results
  - `mlp_on_mnist` ..... the results for the MNIST dataset
  - `mlp_on_mrbi` ..... the results for the MRBI dataset
- `hyperparameter_optimization` .. the scripts with Python source code for the experiments