

**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**MASTER THESIS**

Jiří Krejčí

**Efficient hyperparameter optimization**

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.

Study programme: Computer Science - Artificial  
Intelligence

Study branch: IUIP

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  
Author's signature

Dedication. It is nice to say thanks to supervisors, friends, family, book authors and food providers.

Title: Efficient hyperparameter optimization

Author: Jiří Krejčí

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Abstracts are an abstract form of art. Use the most precise, shortest sentences that state what problem the thesis addresses, how it is approached, pinpoint the exact result achieved, and describe the applications and significance of the results. Highlight anything novel that was discovered or improved by the thesis. Maximum length is 200 words, but try to fit into 120. Abstracts are often used for deciding if a reviewer will be suitable for the thesis; a well-written abstract thus increases the probability of getting a reviewer who will like the thesis.

Keywords: deep learning hyperparameter optimization (HPO) Bayesian optimization (BO) multi-fidelity

# Contents

<b>Introduction</b>	<b>3</b>
<b>1 Background</b>	<b>4</b>
1.1 Machine learning fundamentals . . . . .	4
1.1.1 Supervised machine learning . . . . .	4
1.1.2 Deep learning . . . . .	5
1.1.3 Training of neural networks . . . . .	8
1.1.4 Regularization . . . . .	9
1.2 Hyperparameter optimization . . . . .	9
1.3 Classic Hyperparameter Search Techniques . . . . .	11
1.4 Bayesian optimization . . . . .	12
1.4.1 Acquisition functions . . . . .	14
1.4.2 Gaussian Process Regression . . . . .	18
1.4.3 Tree-Structured Parzen Estimator . . . . .	23
1.4.4 Random Forest . . . . .	25
<b>2 Multi-fidelity optimization</b>	<b>27</b>
2.1 Early stopping . . . . .	28
2.1.1 Learning curve extrapolation . . . . .	28
2.1.2 Successive Halving . . . . .	29
2.1.3 Hyperband . . . . .	31
2.1.4 Model-based algorithms . . . . .	34
2.2 Subsampling . . . . .	38
2.3 Performance evaluation . . . . .	39
2.3.1 Real benchmarks . . . . .	40
2.3.2 Tabulated benchmarks . . . . .	40
<b>3 Experiment definitions</b>	<b>42</b>
3.1 Benchmarks and datasets . . . . .	42
3.2 Algorithms . . . . .	43
3.3 Evaluation metrics . . . . .	44

3.4	Experimental setup and procedures . . . . .	44
<b>4</b>	<b>Results and discussion</b>	<b>46</b>
4.1	Tabulated benchmarks results . . . . .	46
4.2	Real-world benchmarks results . . . . .	50
4.2.1	CIFAR10 . . . . .	50
4.2.2	SVHN . . . . .	51
4.2.3	PTB-XL . . . . .	51
4.2.4	ChestX-ray14 . . . . .	53
4.3	Discussion . . . . .	53
	<b>Conclusion</b>	<b>55</b>
	<b>Bibliography</b>	<b>56</b>
<b>A</b>	<b>Benchmarks specifications</b>	<b>60</b>
A.1	Tabulated . . . . .	60
A.2	Real-world . . . . .	62
<b>B</b>	<b>Additional figures</b>	<b>65</b>
B.1	Tabulated . . . . .	65
B.2	Real-world . . . . .	71

# Introduction

In this thesis, we are addressing the problem of hyperparameter tuning of deep neural networks. The most common way to tune hyperparameters still to this day is by hand, by trial and error, relying on previous experience and various rules of thumb. It is very time-consuming for people and requires a great deal of expertise to do efficiently and well. With the steady increase in computing power, it is preferable to offload this task to computers. The right algorithm can do the task more thoroughly and without much human effort.

Even though this is an extensively studied problem, there still is no consensus on which method performs best. In this thesis, we want to expand on the knowledge of these algorithms and conduct experiments in the healthcare domain.

The main issue is that the function we optimize is expensive to evaluate and the range of possible input values is vast. One function evaluation, or training the network, can take hours or days. That is why the classical methods for hyperparameter optimization in machine learning such as grid search are not applicable to more complex deep learning architectures. Therefore, more sophisticated methods and algorithms were developed and are still being researched. The most important metric in this field is the efficiency of the search.

Hyperparameter optimization has roots in black-box optimization techniques. The machine learning models act as a black-box system to an extent. In deep learning, there is some potential to exploit the knowledge of the system, but only to a small degree. Many tools and optimization frameworks exist for hyperparameter tuning. Most of them are even open source, such as Optuna [1] or SMAC3 [2]. These tuning tools usually use Bayesian optimization techniques internally. Bayesian optimization is well suited for global optimization of black-box, expensive-to-evaluate functions.

In this thesis, we focus on low-budget search. We research the literature on multi-fidelity hyperparameter tuning and experiment with how to best spend a limited budget. We compare the HPO methods on some machine learning tasks in the domain of health care and others.

# Chapter 1

## Background

### 1.1 Machine learning fundamentals

Our goal is to optimize hyperparameters, but before we introduce the main topic, we need to specify the context – machine learning. We include this chapter to introduce the notation and to give the readers some intuition for the machine learning algorithms and models underlying hyperparameter optimization in this thesis. Readers familiar with machine learning, and with deep learning in particular, can skip this section and start directly with hyperparameter optimization in Section 1.2. First, we will introduce the task of supervised machine learning, how the dataset is commonly split, and the function that is optimized in order to train the model. Then we will briefly introduce deep learning and a couple of frequently used architectures. Finally, we will end with an overview of the optimization algorithm behind neural networks.

#### 1.1.1 Supervised machine learning

Let  $\mathcal{D}$  be a labeled dataset, consisting of examples generated independently from a data-generating distribution. Each example  $(x^{(i)}, y^{(i)})$  consist of a feature vector  $x^{(i)} \in \mathcal{X}$  and its label  $y^{(i)} \in \mathcal{Y}$ . Depending on the labels, we distinguish classification and regression. In classification, the labels are finite-valued, while in regression the labels are real numbers. The goal of supervised learning is to train a model using the examples from the dataset  $\mathcal{D}$  so that it generalizes well to data from the data-generating distribution. Simply put, we want the model to correctly predict the labels on unseen data. We assume the unseen data come from the same data-generating distribution so that the model can use the learned patterns for the task.

Since the goal is for the model to generalize well, we need to estimate the generalization error in the training and model selection process on unseen data.

It is a best practice to split the dataset into  $\mathcal{D}_{train}$ ,  $\mathcal{D}_{val}$ ,  $\mathcal{D}_{test}$ . We use  $\mathcal{D}_{train}$  for the training of the model,  $\mathcal{D}_{val}$  for the estimate of the generalization error during development, and  $\mathcal{D}_{test}$  for the final evaluation. A separate test set is needed for the final evaluation because the validation set has already been used to choose the best model, which means there is a bias in the estimate. As we will see, using a hyperparameter optimization tool introduces yet another optimization layer, meaning we should do a similar split even for the outer optimization of hyperparameters.

We train machine learning algorithms by minimizing the loss function on the training data. Two loss functions are used most commonly. For regression, the mean square error of  $N$  examples is computed as

$$MSE = \frac{1}{N} \sum_{i=1}^N (f(x^{(i)}; \theta) - y^{(i)})^2.$$

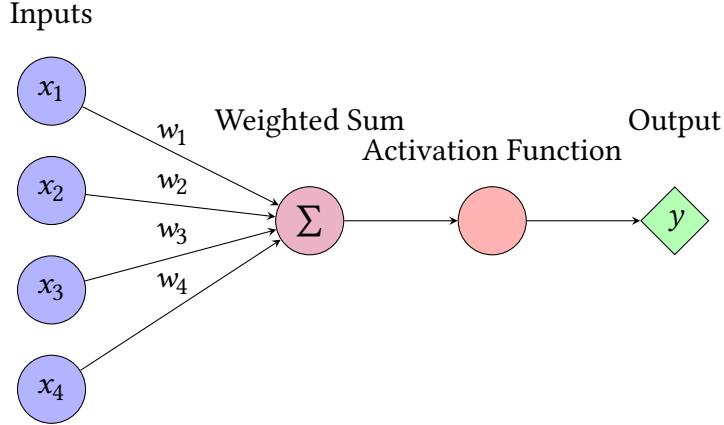
In the case of classification, we use the cross-entropy loss that measures the difference between two probability distributions. Let  $C$  be the number of classification classes and let  $y^{(i)}$  be a distribution over all classes, which will often be just a one-hot encoding of the target class. Let the predictions of the model also form a distribution over the classes. Then the loss is calculated as follows when using mean reduction across examples

$$CE = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_c^{(i)} \log(f(x^{(i)}; \theta)_c).$$

The loss functions depend mostly on the task, not on the specific machine learning model. The training algorithm that optimizes the loss function depends on the model though, so we will revisit optimization later. Even though the training is done exclusively with the loss function, we do not need to use the loss function for the evaluation of the model. Oftentimes a better metric, such as classification accuracy, is used instead. That holds even for hyperparameter optimization; we can optimize hyperparameters for any metric.

### 1.1.2 Deep learning

Deep learning is a subset of machine learning methods based on neural networks. The “deep” in deep learning refers to the use of multiple layers in the network. A neural network describes a computation that models complex patterns and representations in data. The basic unit is called a node and the nodes are arranged in an acyclic graph. Nodes generally represent some operation, while edges can be annotated with parameters for the operation. We call the parameters or



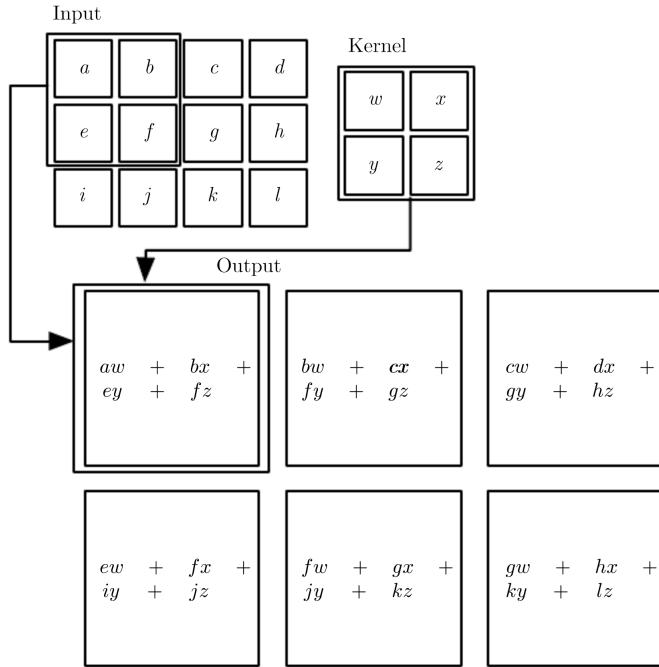
**Figure 1.1** Perceptron neural network

neural network *weights*. A simple example of a neural network is the perceptron (see Figure 1.1). The output value is computed by summing the values of inputs multiplied by the corresponding weights, adding a bias of the node, and passing the weighted sum through an activation function  $a$ :

$$y = a(\sum_j x_j w_j + b). \quad (1.1)$$

We can reuse the computation given by Equation 1.1 as a building block and combine it to form a larger network. If we stack several such blocks, or nodes, on top of each other, we get a *fully connected layer*. It gets its name from the fact that each node in the layer is connected to every input node. If we connect another fully connected layer to the outputs of the first, we get an architecture called a *multilayer perceptron* (MLP). The multilayer perceptron is a good choice for tabular datasets where the features are structured as rows and columns and there are non-linear relationships between the features. The hyperparameters of MLP could be the number of hidden layers, the sizes of the individual layers, or the activation function of the hidden units.

For image data or any other data with some structure, a *convolutional neural network* (CNN) might be a good option. The basic building block of CNNs is a convolution layer. We can imagine a convolution layer as a sliding window, or a tile, that locally weights and sums the data (see Figure 1.2). The parameters of the tile, its size and the weights, are specified by the *kernel*. There can be several kernels of the same size stacked on top of each other, each producing one *channel* of the output. The power of CNNs comes from the fact that they learn the features gradually and combine lower-level features as each kernel processes only a small local neighborhood at a time, into higher-level features as the context window gets wider. Suppose we have two-dimensional data. Formally, the convolution

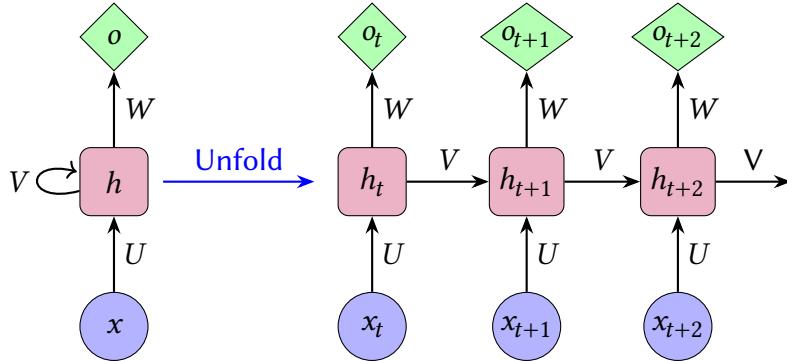


**Figure 1.2** An example of a 2-D convolution. (Source: Figure 9.1 of “Deep Learning” book [3])

layer computes the following function with kernel  $K$  on input  $I$  with  $c$  channels to get an output value at coordinates  $i, j$ :

$$(K * I)_{i,j} = \sum_{m,n,c} I_{i+m,j+n,c} K_{m,n,c}.$$

The *recurrent neural network* (RNN) architecture is designed to handle sequential or time series data. RNNs process the sequence one element at a time. The key is that they also maintain a hidden state, or a memory, that captures information about the sequence from element to element. This is achieved by loops in the graph – the output of an RNN unit (sometimes called a cell) goes back to the same unit as an input with the next element of the sequence (see Figure 1.3). Because of the sequential processing, RNNs can process a sequence of an arbitrary length, reusing all the weights for every step. There are multiple options for the basic computational unit that controls how the input is processed and manages the hidden state. We will use the Long short-term memory (LSTM) unit, which is one of the more refined options. RNNs do not have many special hyperparameters. The most important ones include the number of units and the number of layers. There is also the option to make the units bidirectional, which means the sequence is processed both in a forward and a backward direction.



**Figure 1.3** A single RNN unit. On the left, the RNN unit is compressed and contains a loop connection to itself. On the right, the unit is unfolded, which makes it easier to see how inputs with variable lengths are processed and how the information flows in the network. One of the simplest units could calculate  $h_t = \sigma(Vh_{t-1} + Ux_t + b_h)$  and  $o_t = \sigma(Wh_t + b_o)$ .

### 1.1.3 Training of neural networks

Training of the network can be split into two parts – the forward pass and the backward pass. These two operations are repeated over and over again until some stopping criterion is met. We call one iteration an epoch. In an epoch, the training algorithm iterates through all examples in the training set, usually grouped in batches. In the forward pass, the network produces an output from the input examples and weights by iteratively going through the graph in a topological order. The loss is calculated from the output of the network and true labels. The gradients of the loss function are computed using the backpropagation algorithm, which iterates through the nodes in reverse order and calculates partial derivatives of the loss with respect to the individual weights.

After the gradients are computed, the loss function is minimized using the gradient descent algorithm. Given a loss function  $L(\theta)$  that we want to minimize with respect to the weights, the gradient descent optimizes the function by taking a step in the opposite direction of gradient  $\nabla_\theta L(\theta)$ , which decreases the loss. That is, the weights are updated using a learning rate hyperparameter  $\alpha$ :

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta).$$

For this thesis, it is enough to know there is a gradient descent algorithm optimizing the loss and this algorithm and its more advanced derivatives, such as Adam, comes with some hyperparameters. These usually include a *learning rate*, a *momentum*, or a *decay schedule*, if we want to gradually decrease the learning rate. Later, we will also use the fact that training of neural networks is an iterative

process, and it usually takes at least tens of epochs until the network starts to converge.

### 1.1.4 Regularization

The last concept that is useful to know about for hyperparameter optimization is *regularization*. It is a technique to prevent *overfitting*, which is a situation when the model learns too much from the training set and the validation loss starts increasing because the model learns specific patterns present only in the training dataset. Remember we want the model to generalize well. Regularization is especially important for smaller datasets. There are many regularization techniques and most of them come with some hyperparameter. We call them regularization hyperparameters.

One way to reduce overfitting is to use a *weight decay*, which multiplies the weights by some constant smaller than one and usually close to zero, which we set as a hyperparameter. The intuition behind weight decay is that the network forgets a little each time the weights are updated, so it can learn only general patterns that occur often. Another common regularization technique is the *dropout*. In dropout, some nodes are deactivated during a forward pass with the dropout rate (probability), which is a hyperparameter. The network is forced to work reasonably well with only a subset of nodes, which should also force it to learn general features.

## 1.2 Hyperparameter optimization

Let  $A$  be a machine learning algorithm with hyperparameters  $\lambda_1, \dots, \lambda_n$  with domains  $\Lambda_1, \dots, \Lambda_n$ . Let  $\Lambda = \Lambda_1 \times \dots \times \Lambda_n$  denote its hyperparameter space. Hyperparameters can be continuous, integer-valued, or categorical. Also, we can have conditional hyperparameters. We say that hyperparameter  $\lambda_i$  is *conditional* on another hyperparameter  $\lambda_j$ , if  $\lambda_i$  is only active if hyperparameter  $\lambda_j$  takes values from a given set  $V_i(j) \subset \Lambda_j$

For each hyperparameter configuration or setting  $\lambda \in \Lambda$ , we denote  $A_\lambda$  the learning algorithm  $A$  using this hyperparameter setting. Let  $\mathcal{L}(A_\lambda, \mathcal{D}_{train}, \mathcal{D}_{valid})$  denote the validation loss of algorithm  $A_\lambda$  on data  $\mathcal{D}_{valid}$  when trained on  $\mathcal{D}_{train}$ .

**Definition 1** (Hyperparameter optimization problem). *The hyperparameter optimization problem is to find hyperparameters  $\lambda^*$  that minimize the loss function*

$$\lambda^* = \arg \min_{\lambda} f(\lambda) = \arg \min_{\lambda} \mathcal{L}(A_\lambda, \mathcal{D}_{train}, \mathcal{D}_{valid}).$$

Several properties make the problem hard to solve.

- It is hard to obtain derivatives of the loss function with respect to hyperparameters and we will not use them to find the optimal solution. Such an optimization problem is called a black-box optimization in literature.
- Each function evaluation is expensive. Fully training a single deep neural network can take days.
- Each function evaluation may require a variable amount of time. For example, training larger models (e.g. more artificial neurons) takes more time to train. Therefore, the hyperparameter optimization algorithm should take training time into account.
- Observations are noisy. Repeated training may result in models that vary in performance since it is common to use random initialization of weights. The training process itself may not be deterministic as well. For example, if we use mini-batch shuffling.

On the other hand, we can leverage parallel computation to run multiple trials at the same time. One additional benefit of solving the optimization problem limited to deep neural networks is that we have access to intermediate results, which is not the case for general black-box optimization. This kind of task is sometimes called a grey-box optimization in the literature.

In this thesis, we assume that the general architecture of the neural network is already given and hyperparameters can change only smaller aspects, such as the number of neurons in a layer, or kernel size in a convolutional layer. For literature dealing with the more general problem, please refer to Neural Architecture Search (NAS).

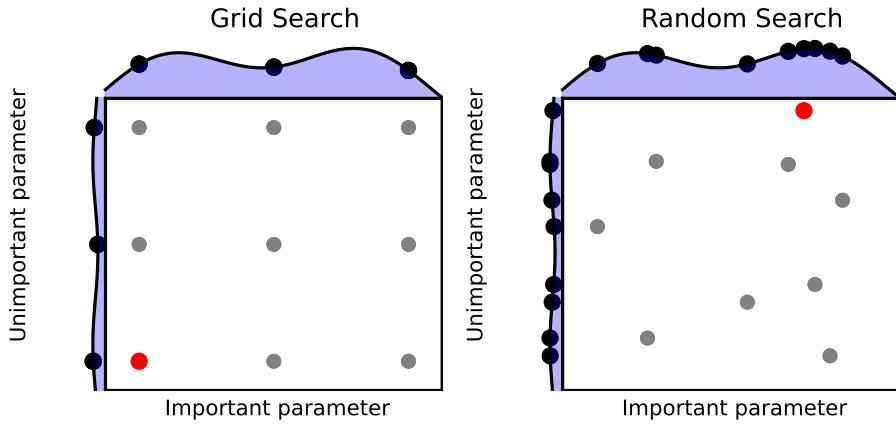
For anyone interested in the process of hyperparameter tuning and how it might be done in practice, we recommend the Deep learning tuning playbook [4]. The authors give valuable insights into practical aspects of hyperparameter tuning that they have collected over more than ten years of working in deep learning. These insights are rarely documented. As the authors state in the text, they could not find any comprehensive attempt to explain how to get good results with deep learning. More importantly for this thesis, it gives us insight into how experts might do hyperparameter tuning. The text reveals that even today, advanced hyperparameter tuning tools are not the ultimate solution to the problem. Instead, they recommend how to use them smartly. They propose that there is still a human expert guiding the search, at least in the first, exploratory, phase.

## 1.3 Classic Hyperparameter Search Techniques

The default approach for many practitioners is manual hyperparameter tuning. We cannot be surprised that people still tune hyperparameters manually. There is no technical overhead or barrier. Also, in the process of hyperparameter tuning, we gain insight into the problem, which might allow us to improve our solution in ways that are not achievable just by hyperparameter tuning. Nevertheless, there are clear limits to manual tuning so let us dive into the algorithmic approaches. The traditional algorithms for hyperparameter optimization are grid search and random search. These algorithms are simple and still widely used.

**Grid Search** Grid search performs an exhaustive search through a manually specified subset of the hyperparameter space. Grid search is best used when the number of hyperparameters is small, or the function evaluation is not that expensive. Its biggest drawback is that the number of configurations to evaluate grows exponentially with the number of hyperparameters. Therefore, it is best to determine which hyperparameters are the most important and limit the search only to this subset. If we did not do this, we would waste a lot of computational resources on hyperparameter combinations, where only the unimportant hyperparameter changes, but the important ones stay the same. We illustrate this situation in Figure 1.4. To mitigate this, we would need a way to determine which hyperparameters are important and need to be tuned together, which is a hard problem itself. On the other hand, grid search is easily parallelizable. That is an enormous advantage since in real-world scenarios, it is not uncommon to have access to a computing cluster.

**Random Search** Random search is often used in the HPO literature as the baseline method for more advanced algorithms. In real-world optimization problems, random search often works better than grid search. Bergstra et al. [5] compared random search to grid search and found that randomly chosen trials are more efficient for hyperparameter optimization than trials on a grid. It is possible to encounter a random search with 2X-budget as a baseline in some research papers. It is just a random search with two times the budget of other methods in comparison. As Li et al. [6] show, 2X-budget random search provides a strong baseline.



**Figure 1.4** A visual comparison of Grid Search versus Random Search. Suppose that we optimize a function with two parameters and the function is the sum of the functions defined for each parameter. The best configuration found after nine trials is highlighted in red. If one of the parameters is not important, grid search wastes resources by repeatedly evaluating similar configurations.

**Quasi-random search** If our budget is low then quasi-random search might be the better option. It works by generating a low-discrepancy sequence. Intuitively, a low-discrepancy sequence covers the whole domain evenly. Therefore, the search space is better covered even with a small number of samples. In the Deep learning tuning playbook, the authors recommend using quasi-random search over grid search and random search for the initial exploration of the hyperparameter space.

## 1.4 Bayesian optimization

So far, we have seen model-less approaches, where each trial is independent. This approach offers some advantages, like parallelization and simplicity of implementation, but it is quite inefficient. Information obtained from previous trials is not used in any way to guide the search. Bayesian optimization methods build and use an internal model of the learning algorithm’s generalization performance. Bayesian optimization is widely covered in literature, we will use the work of Brochu et al. [7] and Frazier [8] for the definitions and the description of the method.

In general, Bayesian optimization is a class of optimization methods focused on optimizing a real-valued objective function

$$\max_{x \in A} f(x).$$

The method got its name from the Bayes' theorem, which is applied by stating that the posterior probability of a model M given observations E is proportional to the likelihood of E given M multiplied by the prior probability of M

$$P(M | E) \propto P(E | M)P(M).$$

The foundations were laid for Bayesian Optimization by Jonas Mockus [9] in the 1970s and 1980s, but it was not until the early 2000s that Bayesian Optimization got applied to machine learning by Carl Edward Rasmussen [10]. Since then, it has become arguably the most prominent advanced technique for hyperparameter optimization, studied by countless groups and being implemented in many popular hyperparameter optimization frameworks.

The basic loop of a Bayesian optimization algorithm is simple as shown in Algorithm 1. First, the internal model is used to suggest the next hyperparameter configuration to try. This query is much cheaper than evaluating the objective function. We can think of the suggestion as the most promising configuration given the previous observations. More precisely, the algorithm maximizes an *acquisition function*  $a$  defined from the *surrogate model*. Then the objective function is evaluated at this configuration and the observed result is added to a database. The surrogate is updated using the database with the new observation and the process is repeated until some stopping condition is triggered, e.g. the algorithm runs out of budget.

---

**Algorithm 1** Bayesian Optimization

---

- 1: **for**  $t = 1, 2, \dots$  **do**
  - 2:     Find the next point  $x_t$  to evaluate:  $x_t = \arg \max_x a(x | \mathcal{D}_{1:t-1})$ .
  - 3:     Sample the objective function:  $y_t = f(x_t) + \epsilon_t$ .
  - 4:     Augment the data:  $\mathcal{D}_{1:t} = \{\mathcal{D}_{1:t-1}, (x_t, y_t)\}$ .
  - 5:     Update the surrogate model given the data  $\mathcal{D}_{1:t}$ .
- 

Before we dive deeper into Bayesian Optimization, let us deal with some practical considerations and properties of the objective function and the feasible set. Since we have defined the hyperparameter optimization problem as minimization of the loss function, we can assume that  $f$  is defined as  $f(\lambda) = -\mathcal{L}(\lambda)$ . Maximizing this function is equivalent to minimizing the original function. We also assume that the objective function is continuous. When we evaluate  $f$ , we observe only  $f(x)$  and no first- or second-order derivatives, which would allow us to use a wider array of optimization methods. We do not know nor assume that  $f$  has any special structure like concavity or linearity. To summarize, we can say that Bayesian Optimization is designed for black-box derivative-free global optimization.

We assume further that all inputs  $x$  are real-valued, which is not the case for all hyperparameters, and we will address this issue later. Finally, we assume the feasible set  $A$  to be a hyper-rectangle  $\{x \in \mathbb{R}^d \mid a_i \leq x_i \leq b_i\}$ , which makes it easy to assess membership.

The two main components of a Bayesian Optimization algorithm are a Bayesian statistical model and an acquisition function. The model approximates the objective function including the uncertainty over its predictions and the acquisition function is used for choosing configurations to evaluate. We will go through both of these components in greater detail. First, we will introduce the commonly used acquisition functions, and then we will show three different models.

### 1.4.1 Acquisition functions

The purpose of an acquisition function is to guide the search. It would be possible to find the predictive mean of the surrogate and sample a configuration that maximizes the mean, but the strength of Bayesian optimization is that it expresses uncertainty. Acquisition functions are designed to take advantage of uncertainty and balance exploration versus exploitation. The acquisition function value might be high if the objective function predicted value is high, but also if the uncertainty of the model is high; the area is not well explored yet but promising. We assume that we have a Bayesian statistical model that for any  $x$  in the domain outputs a prediction  $y \sim \mathcal{N}(\mu(x), \sigma^2(x))$ .

#### Upper Confidence Bound

Probably the simplest acquisition function is the Upper Confidence Bound (UCB). Given a mean  $\mu(x)$  and a variance  $\sigma(x)$ , the UCB is calculated as

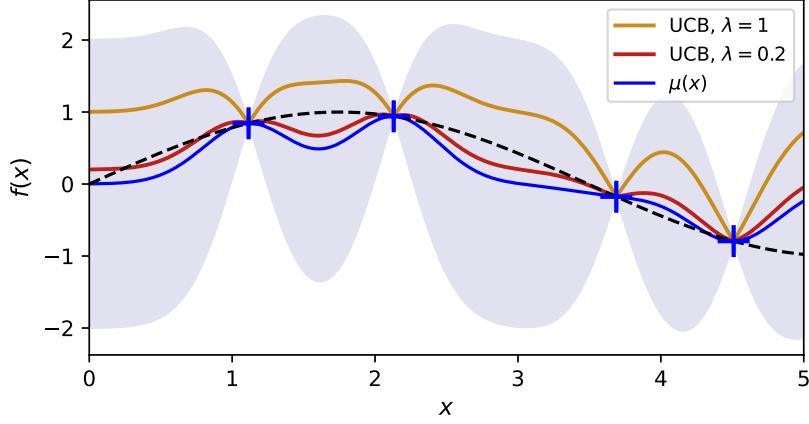
$$\text{UCB}(x) = \mu(x) + \lambda\sigma(x),$$

where  $\lambda$  is an exploration parameter. The smaller the  $\lambda$ , the more the UCB exploits regions that are known to yield good solutions and vice versa. The UCB acquisition function is illustrated in Figure 1.5.

#### Probability of Improvement

Let  $y^* = \max_{i \in [1..n]} f(x_i)$  be the value of the best evaluated point so far after  $n$  iterations and let  $x$  be the next point we consider sampling. We define an improvement random variable as

$$I(x) := \max(f(x) - y^*, 0).$$



**Figure 1.5** An UCB acquisition function with two  $\lambda$  settings demonstrated on a Bayesian model predicting mean  $\mu(x)$ , and variance (the light blue corresponds to two standard deviations from the mean), fitted to the black true function with four noisy observations.

The probability of improvement acquisition function assigns to each candidate  $x$  the probability of  $I(x) > 0$ . We can write

$$PI(x) = P(I(x) > 0) = P(f(x) > y^*) = 1 - P(f(x) \leq y^*).$$

Since we assume that  $f(x)$  is normally distributed, then the PI is calculated from the cumulative density function  $\Phi$  of normal distribution. We remind that for  $X \sim \mathcal{N}(\mu, \sigma^2)$ , we calculate  $P(X \leq x)$  as  $\Phi((x - \mu)/\sigma)$ . We also use the property  $1 - \Phi(x) = \Phi(-x)$ . Therefore, the PI is calculated as follows:

$$PI(x) = 1 - \Phi\left(\frac{y^* - \mu(x)}{\sigma(x)}\right) = \Phi\left(\frac{\mu(x) - y^*}{\sigma(x)}\right).$$

The problem with PI is that it does not factor in the magnitude of improvement. In the standard form, PI cares only about being as certain as possible about improving upon the incumbent solution. This often results in PI suggesting points close to the optimum, i.e. PI is biased towards exploitation. This problem is alleviated by adding a new parameter  $\xi$ , which forces PI to consider only points that are by at least  $\xi$  better than the incumbent:

$$PI(x) = P(I(x) > \xi)$$

### Expected improvement

Expected improvement (EI) [11] enhances PI by considering the magnitude of improvement as well as the probability of improvement. That is achieved by

taking the expected value of  $I(x)$ . Using the assumption that  $I(x)$  is normally distributed, the density of  $I(x)$  for a specific value of improvement,  $I$ , is given by

$$\frac{1}{\sqrt{2\pi}\sigma(x)} \exp\left(-\frac{(\mu(x) - y^* - I)^2}{2\sigma^2(x)}\right).$$

**Is the random variable notation used correctly here for  $I(x)$ ? Is it a problem that we cut off the improvement at 0?**

The expected value is calculated by integrating the density function multiplied by the improvement:

$$EI(x) = \mathbb{E}[I(x)] = \int_{I=0}^{I=\infty} I \frac{1}{\sqrt{2\pi}\sigma(x)} \exp\left(-\frac{(\mu(x) - y^* - I)^2}{2\sigma^2(x)}\right) dI.$$

The analytical solution to this integral can be found in the literature [11] and it is as follows:

$$EI(x) = (\mu(x) - y^*)\Phi(Z) + \sigma(x)\phi(Z),$$

where

$$Z = \frac{\mu(x) - y^*}{\sigma(x)},$$

and  $\phi$  is the density of the normal distribution.

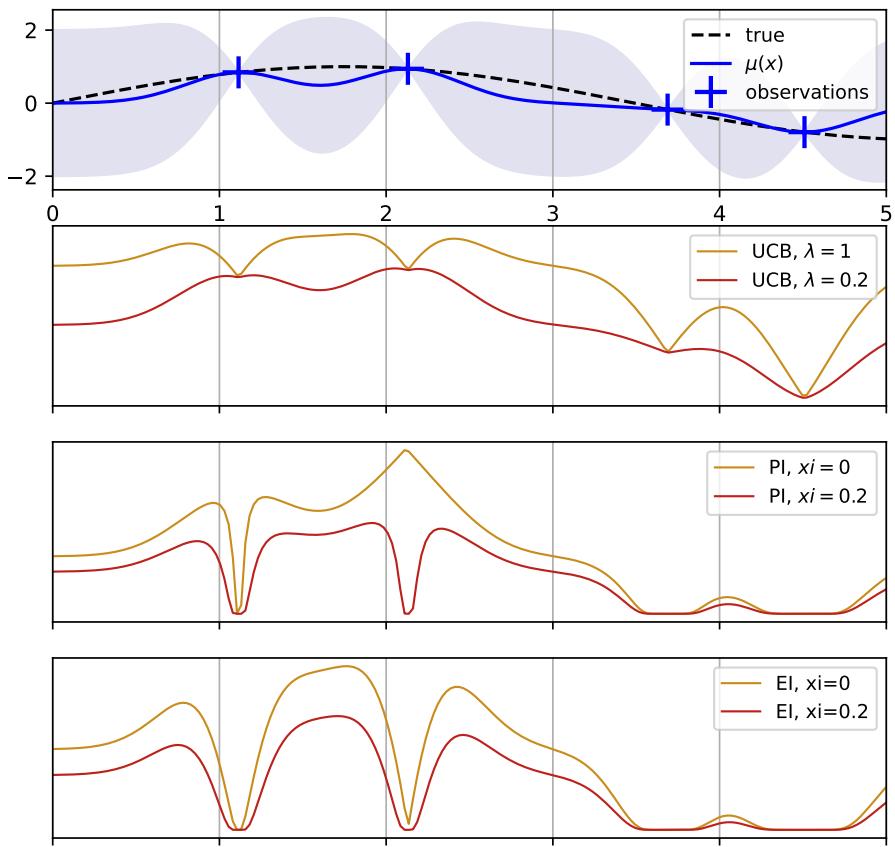
Even the expected improvement can be extended with parameter  $\xi$  as in the PI acquisition function. The role of this parameter stays the same; it enables us to balance exploration and exploitation. The modified acquisition function is

$$EI(x) = (\mu(x) - y^* - \xi)\Phi(Z) + \sigma(x)\phi(Z),$$

where

$$Z = \frac{\mu(x) - y^* - \xi}{\sigma(x)}.$$

Expected improvement is a popular acquisition function that balances the exploration-exploitation trade-off well; it prioritizes exploring regions with a high probability of significantly improving upon the current best solution. We compare all the mentioned acquisition functions in Figure 1.6.



**Figure 1.6** A comparison of different acquisition functions. The top image shows the real target function with a Bayesian model predicting mean and variance (the light blue area corresponds to two standard deviations). The other images show different acquisition functions from top to bottom: upper confidence bound, probability of improvement, and expected improvement.

### 1.4.2 Gaussian Process Regression

The standard model in the Bayesian Optimization literature is the GP regression. An intuitive introduction to the topic was published by Jie Wang [12], and we use similar illustrations to present the topic. The definitions and the notation are taken from the textbook by Rasmussen [10]. There are many other machine learning algorithms for regression, but Gaussian processes offer a unique mix of properties that make them the natural choice. One of the most important properties of GPs is that they quantify the uncertainty, which allows us to incorporate it into our sampling strategy — the areas with the most uncertainty should likely be explored more, or similar heuristic strategy. Another advantage is the possibility of incorporating our prior beliefs about the objective function with the kernel function. As a result, Gaussian processes are remarkably efficient when the amount of data is limited. On the other hand, GPs do not scale well with large datasets, because of their  $\mathcal{O}(n^3)$  complexity. In practice, GPs limit us to the number of samples in the order of hundreds.

#### Gaussian distribution

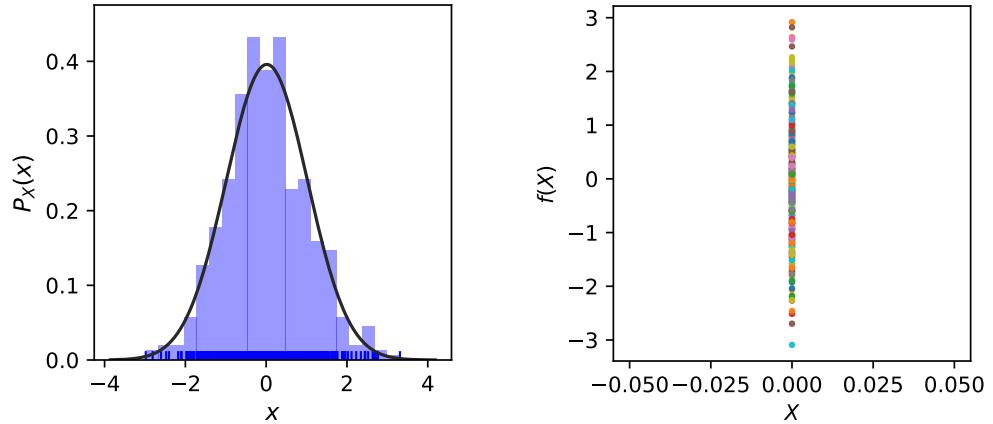
In order to describe the Gaussian process regression, we will start with the basics. A random variable  $X$  is Gaussian or normally distributed with mean  $\mu$  and variance  $\sigma^2$  if its probability density function is:

$$P_X(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

We denote that a random variable is normally distributed as  $X \sim \mathcal{N}(\mu, \sigma^2)$ . For illustration, we have sampled 500 points randomly from a univariate normal distribution into a vector  $x_1 = [x_1^1, x_1^2, \dots, x_1^{500}]$ . In Figure 1.7 we plot a histogram of the points and fit a Gaussian distribution over them. We also plot the points vertically along the Y-axis, which is the first step towards the Gaussian process regression.

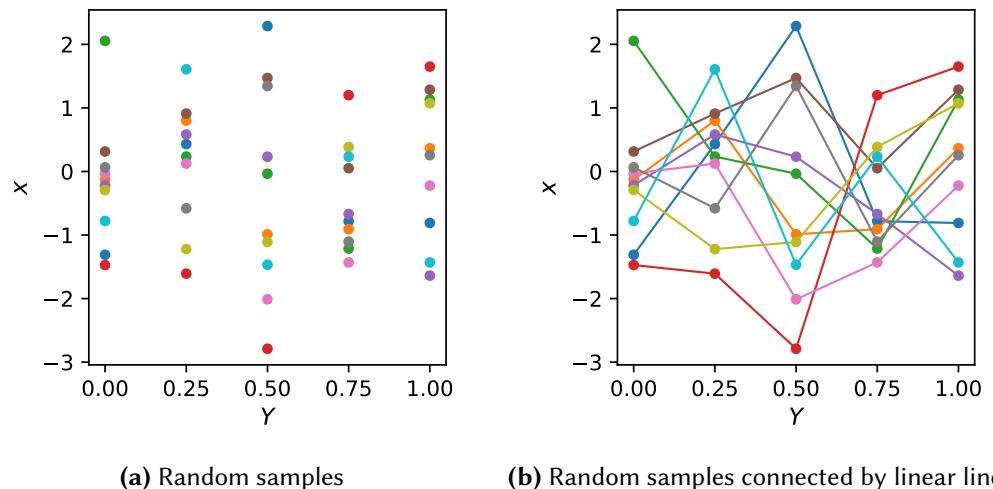
Similarly, we could sample several vectors  $x_1, \dots, x_n$  of points and plot them with a different x-coordinate as shown in Figure 1.8. If we connect the corresponding samples with a line, we could perform a regression task using these lines in the domain  $[0, 1]$ . The issue is that the samples generating the lines are independent, making the predictions of no use. The key assumption for regression is that close points have similar values. Therefore, we have to find a way to introduce a correlation between the samples, preferably based on their distance.

A set of correlated, normally distributed random variables is described by the *multivariate normal distribution* (MVN). We denote MVN as  $\mathbf{X} \sim \mathcal{N}_k(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ , where  $\mathbf{X} = (X_1, X_2, \dots, X_n)^T$  is a random vector,  $\boldsymbol{\mu} = (\mu_1, \mu_2, \dots, \mu_n)^T$  are the means and



(a) Histogram of the samples with the fitted density as a black curve.  
(b) Random samples plotted vertically along the Y-axis with a fixed X coordinate.

**Figure 1.7** A visualization of sampling from a Gaussian distribution. A random variable  $X \sim \mathcal{N}(0, 1)$  is sampled 500 times.



**Figure 1.8**  $N=5$  sampled random vectors, each with  $M=10$  random samples drawn from a normal distribution. Samples from each vector are plotted along the Y-axis with the same X coordinate

$\Sigma \in \mathbb{R}^{k \times k}$  is the covariance matrix. The  $k$  is often omitted as the dimensions are usually clear from the context. The covariance matrix specifies the covariance between each pair of elements of a given random vector, with  $\Sigma_{ij} = \text{cov}[X_i, X_j]$ . The  $\Sigma$  is a symmetric and positive semi-definite matrix with variances on its main diagonal. Finally, the probability density function of  $k$ -dimensional MVN is defined as

$$\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{k/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right).$$

For regression, we are more interested in conditional probability rather than joint probability, as we want to make use of the observed data points. To get the conditional probability, we can partition the random vector  $\mathbf{X}$  into  $\mathbf{X}_1$  and  $\mathbf{X}_2$ . The conditional probability of  $\mathbf{X}_1$  given  $\mathbf{X}_2$  is also an MVN because a multivariate normal distribution is closed under conditioning. We will show the exact formulas in the description of Gaussian process regression.

## Kernels

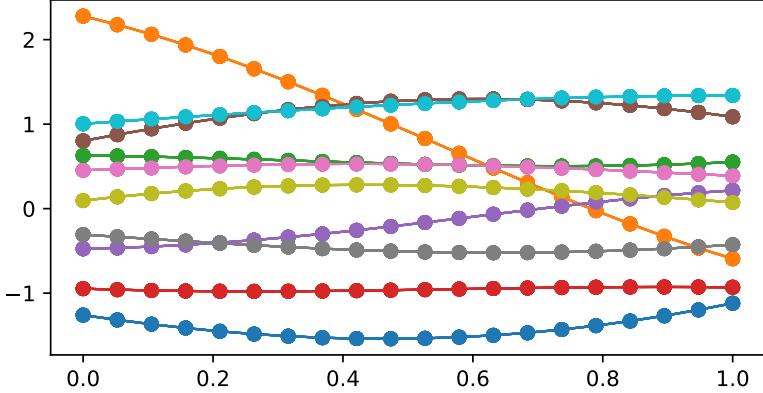
Using the multivariate normal distribution, we can correlate the points of our regression function. Instead of manually specifying the covariance matrix, we will use kernels. A kernel function measures the similarity between a pair of data points. This in turn impacts the smoothness of the regression functions — we want close points to have similar function values, and we use the kernel function to determine the strength of the correlation.

A common choice is the squared exponential kernel, also called the Radial Basis Function (RBF) kernel, defined as

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right).$$

The kernel has a parameter  $\sigma$  called the length scale. It controls how quickly the correlation between two points decays. The functions predicted by the kernel are smooth and infinitely differentiable. We have plotted samples of the twenty-variate normal distribution with RBF kernel as covariance function in Figure 1.9. Note that the samples illustrated in Figure 1.8 can also be viewed as being drawn from MVN distribution, but with an identity covariance function, which highlights the role of a kernel.

We have now covered all the background necessary to get to Gaussian processes. We understand how the MVN is used with kernels to produce correlated predictions that look smoother when connected with lines. As we increase the dimensions of the MVN, the points will get closer and closer together in the domain of interest. For a truly smooth prediction spanning the whole domain, we use MVN distribution with infinite dimensions.



**Figure 1.9** Samples from the 20-VN distribution with RBF covariance.

### Gaussian processes

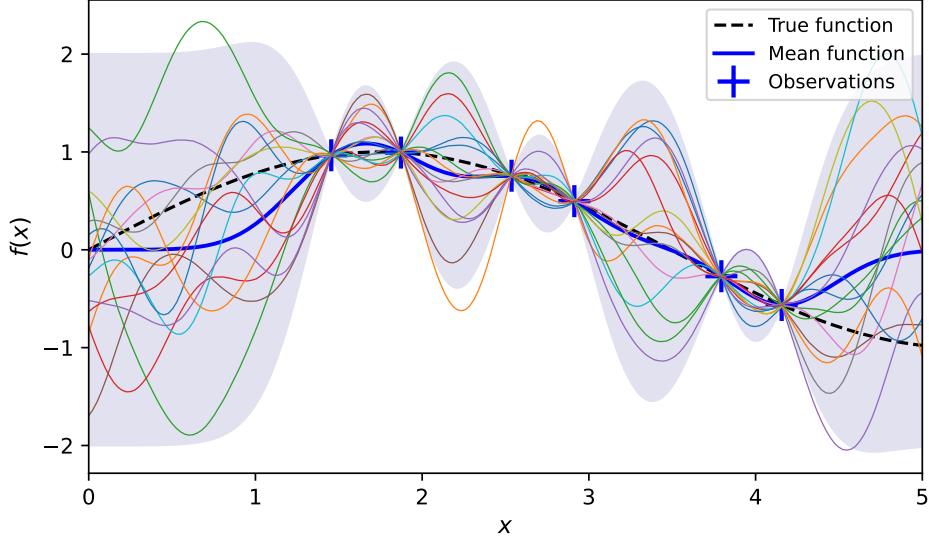
Gaussian processes are a generalization of MVN into infinite dimensions. The kernel specifies permissible functions which make up the prior distribution. The Gaussian process model defines a probability distribution over prior functions that fit the observed data. The concept is best illustrated in an example. In Figure 1.10, we want to perform a regression task over the true (black) function. We are given six noisy observations. We can derive the (blue) mean function by calculating the posterior distribution and averaging over all functions weighted by their posterior probability. For the illustration of the posterior distribution, we can observe the light blue area marking two standard deviations from the mean. Lastly, the colorful functions are randomly sampled from the posterior distribution.

Let us define the Gaussian process formally. Let  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_n]$  represent the observed data points,  $\mathbf{f} = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)]$  the function values,  $\mu = [m(\mathbf{x}_1), \dots, m(\mathbf{x}_n)]$  the mean function, and  $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$  the positive definite kernel function. The mean function represents a prior mean — our initial belief about the average behavior of the function across the input space — and is often just assumed to be 0. Let us consider a regression task. We want to predict function values  $\mathbf{f}(\mathbf{X}_*)$  at new points  $\mathbf{X}_*$  using the Gaussian process regression. The joint distribution of  $\mathbf{f}$  and  $\mathbf{f}_*$  is given by

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathbf{X}) \\ m(\mathbf{X}_*) \end{bmatrix}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{bmatrix}\right),$$

where  $\mathbf{K} = K(\mathbf{X}, \mathbf{X})$ ,  $\mathbf{K}_* = K(\mathbf{X}, \mathbf{X}_*)$ ,  $\mathbf{K}_{**} = K(\mathbf{X}_*, \mathbf{X}_*)$ , and we assume that  $(m(\mathbf{X}), m(\mathbf{X}_*)) = \mathbf{0}$ .

Since we solve the regression task, we are interested in the conditional dis-



**Figure 1.10** An example of Gaussian process regression. The black dashed line is the true target function. We drew six random samples and the function was evaluated at these points with noise (blue crosses). Then we sampled 15 functions from the posterior distribution given the test points. The blue line is the mean function and the light blue area is bounded at two standard deviations from the mean.

tribution  $P(\mathbf{f}_* | \mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ , not the joint distribution. Using the formula for MVN conditional distribution, it can be derived from the joint distribution  $P(\mathbf{f}_*, \mathbf{f} | \mathbf{X}, \mathbf{X}_*)$  as

$$\mathbf{f}_* | \mathbf{f}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}(\mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{f}, \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*) .$$

We often encounter noisy observations in practice, which is often the case for the hyperparameter optimization problem as well. We deal with noisy observations by assuming additive independent and identically distributed Gaussian noise  $\epsilon$  with variance  $\sigma_n^2$ . The noise is incorporated into the prior as  $\text{cov}(y) = \mathbf{K} + \sigma_n^2 \mathbf{I}$ . Then the observations can be expressed as  $y = f(\mathbf{x}) + \epsilon$ . The joint distribution with noise is

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma_n^2 \mathbf{I} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{bmatrix}\right) .$$

Finally, the conditional distribution with noise is derived as

$$\tilde{\mathbf{f}}_* | \mathbf{y}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}(\tilde{\mathbf{f}}_*, \text{cov}(\mathbf{f}_*)) ,$$

where

$$\begin{aligned}\bar{\mathbf{f}}_* &\stackrel{\Delta}{=} \mathbb{E}[\tilde{\mathbf{f}}_* | \mathbf{y}, \mathbf{X}, \mathbf{X}_*] \\ &= \mathbf{K}_*^T [\mathbf{K} + \sigma_n^2 \mathbf{I}]^{-1} \mathbf{y}, \\ \text{cov}(\mathbf{f}_*) &= \mathbf{K}_{**} - \mathbf{K}_*^T [\mathbf{K} + \sigma_n^2 \mathbf{I}]^{-1} \mathbf{K}_*.\end{aligned}$$

Gaussian processes are best suited for optimization over continuous domains of less than 20 dimensions and when the number of function evaluations is very low. GPs tolerate stochastic noise in function evaluations [8].

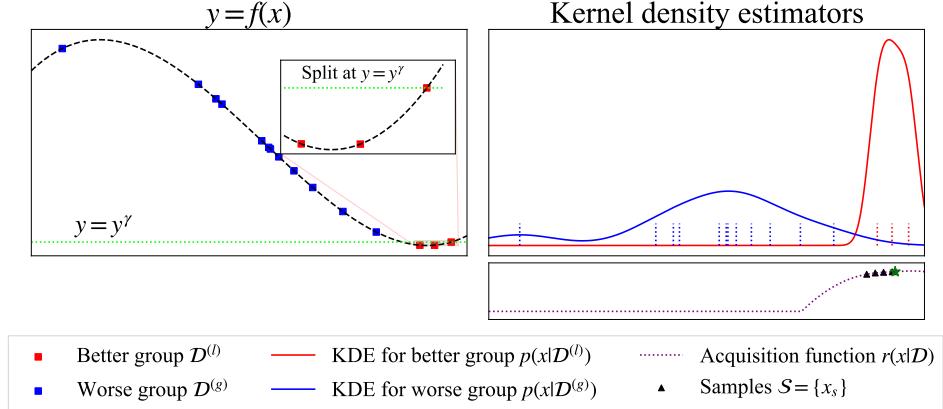
### 1.4.3 Tree-Structured Parzen Estimator

Another popular Bayesian optimization method for hyperparameter optimization is the Tree-structured Parzen Estimator (TPE) introduced by Bergstra et al. [13]. As the name suggests, it is an extension of a Parzen estimator to a tree-structured search space. Therefore, a TPE can handle conditional parameters as well. A Parzen estimator, also known as a kernel density estimator (KDE), is a non-parametric method used to estimate the probability density function of a random variable based on a set of observed data points. Instead of assuming the underlying distribution and fitting it to the data, a Parzen estimator uses a kernel function to determine the shape and influence of each data point on the estimated probability density function.

In a TPE algorithm, two kernel density estimators are used. Keeping the notation from the original paper [13], we want to minimize the objective function  $f(x)$  and the observations  $\mathcal{D}$  are split into the better (lower) group  $\mathcal{D}^{(l)}$  and the worse (greater) group  $\mathcal{D}^{(g)}$  based on their objective value; for illustration see the top left part of Figure 1.11. More precisely, the top-quantile  $y^\gamma$  is computed in each iteration based on the number of observations  $N = |\mathcal{D}|$  and  $y^\gamma$  is the top- $\gamma$ -quantile objective value in the set of observations  $\mathcal{D}$ . Observations with  $y \leq y^\gamma$  are assigned into the  $\mathcal{D}^{(l)}$ , and observations with  $y > y^\gamma$  to the  $\mathcal{D}^{(g)}$ . The subsets are used to model  $p(\mathbf{x} | y, \mathcal{D})$  with the assumption:

$$(\mathbf{x} | y, \mathcal{D}) := \begin{cases} p(\mathbf{x} | \mathcal{D}^{(l)}) & (y \leq y^\gamma) \\ p(\mathbf{x} | \mathcal{D}^{(g)}) & (y > y^\gamma) \end{cases}. \quad (1.2)$$

Now we can show how the kernel density estimations from the equation above are calculated. Let us assume the  $D$  is sorted by  $y_n$  such that  $y_1 \leq y_2 \leq \dots \leq y_N$ . Then the KDEs are estimated as



**Figure 1.11** Example of the TPE algorithm. We are searching for the minimum of the target (black) function. The observations are split into  $D^{(l)}$  and  $D^{(g)}$  by the green line on the objective value of observations. For each group, a kernel density estimation is calculated. The acquisition function is derived from the densities and the algorithm chooses the next point to sample by finding the maximum (Source: Figure 1 from Watanabe [14]).

$$p(\mathbf{x} | \mathcal{D}^{(l)}) = w_0^{(l)} p_0(\mathbf{x}) + \sum_{n=1}^{N^{(l)}} w_n k(\mathbf{x}, \mathbf{x}_n | b^{(l)}), \quad (1.3)$$

$$p(\mathbf{x} | \mathcal{D}^{(g)}) = w_0^{(g)} p_0(\mathbf{x}) + \sum_{n=N^{(l)}+1}^N w_n k(\mathbf{x}, \mathbf{x}_n | b^{(g)}),$$

where  $k$  is a kernel function, the weights  $\{w_n\}_{n=1}^N$  are recomputed every iteration,  $b^{(l)}, b^{(g)} \in \mathbb{R}_+$  are the bandwidth and  $p_0$  is non-informative prior. The KDEs are illustrated in the top right part of Figure 1.11.

The last part of the algorithm yet to be described is the acquisition function. It is calculated from the KDEs using the assumption from Eq. 1.2 as

$$\mathbb{P}(y \leq y^r | \mathbf{x}, \mathcal{D}) \stackrel{\text{rank}}{\simeq} r(\mathbf{x} | \mathcal{D}) := \frac{p(\mathbf{x} | \mathcal{D}^{(l)})}{p(\mathbf{x} | \mathcal{D}^{(g)})},$$

where the  $\stackrel{\text{rank}}{\simeq}$  symbol means the order isomorphic between the left-hand side and the right-hand side. See Figure 1.11 for illustration. The detailed pseudocode is provided in Algorithm 2.

---

**Algorithm 2** Tree-structured Parzen estimator (TPE) (Source: Watanabe [14])

---

$N_{\text{init}}$  (The number of initial configurations),  $N_s$  (The number of candidates to consider in the optimization of the acquisition function),  $\Gamma$  (A function to compute the top quantile  $\gamma$ ),  $W$  (A function to compute weights  $\{w_n\}_{n=0}^{N+1}$ ),  $k$  (A kernel function),  $B$  (A function to compute a bandwidth  $b$  for  $k$ ).

- 1:  $\mathcal{D} \leftarrow \emptyset$
- 2: **for**  $n = 1, 2, \dots, N_{\text{init}}$  **do** ..... *Initialization*
- 3:     Randomly pick  $\mathbf{x}_n$
- 4:      $y_n := f(\mathbf{x}_n) + \epsilon_n$  ..... *Evaluate the (expensive) objective function*
- 5:      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_n, y_n)\}$
- 6: **while** Budget is left **do**
- 7:     Compute  $\gamma \leftarrow \Gamma(N)$  with  $N := |\mathcal{D}|$  ..... *Section ?? (Splitting algorithm)*
- 8:     Split  $\mathcal{D}$  into  $\mathcal{D}^{(l)}$  and  $\mathcal{D}^{(g)}$
- 9:     Compute  $\{w_n\}_{n=0}^{N+1} \leftarrow W(\mathcal{D})$  ..... *See Section ?? (Weighting algorithm)*
- 10:    Compute  $b^{(l)} \leftarrow B(\mathcal{D}^{(l)}), b^{(g)} \leftarrow B(\mathcal{D}^{(g)})$  *Section ?? (Bandwidth selection)*
- 11:    Build  $p(\mathbf{x} | \mathcal{D}^{(l)}), p(\mathbf{x} | \mathcal{D}^{(g)})$  based on Eq. (1.3) *Use  $\{w_n\}_{n=0}^{N+1}$  and  $b^{(l)}, b^{(g)}$*
- 12:    Sample  $\mathcal{S} := \{\mathbf{x}_s\}_{s=1}^{N_s} \sim p(\mathbf{x} | \mathcal{D}^{(l)})$
- 13:    Pick  $\mathbf{x}_{N+1} := \mathbf{x}^* \in \arg \max_{\mathbf{x} \in \mathcal{S}} r(\mathbf{x} | \mathcal{D})$  *The evaluations by the acquisition function*
- 14:     $y_{N+1} := f(\mathbf{x}_{N+1}) + \epsilon_{N+1}$  ..... *Evaluate the (expensive) objective function*
- 15:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{\mathbf{x}_{N+1}, y_{N+1}\}$

---

#### 1.4.4 Random Forest

The last Bayesian optimization surrogate that we are going to introduce is the random forest model. This approach was first introduced by Hutter et al. [15] as SMAC, which stands for Sequential Model-based Algorithm Configuration. It was published as an alternative to the Gaussian processes for the general algorithm configuration problem, approximately at the same time as the TPE. The random forest surrogate is still widely used, mostly in the updated SMAC3 Python package [2].

Random forest [16] is a machine learning method for classification and regression that trains an ensemble of decision trees and predicts the combined value from all the decision trees during inference. In SMAC, random forests are used for regression to directly model the objective function. The advantage of using regression trees is that they perform well on categorical input data, unlike Gaussian processes. The model is trained by randomly sampling  $n$  data points with repetition for each of  $B$  regression trees. At each split point in a tree, only a random subset of features is considered. The default is 5/6 of all features. The

default number of regression trees in the algorithm  $B$  is set to 10. There is one more parameter  $n_{min}$  for the minimal number of data points required to split a node. This parameter is set to  $n_{min} = 10$  by default.

Prediction for a new data point is the empirical mean of individual trees' predictions. A prediction of a single tree is usually the mean of the data points in the leaf that corresponds to the input configuration, but the authors implement an option for a user-defined prediction function as well. The algorithm also calculates the empirical variance from the predictions of the individual trees. To select the next configuration to evaluate, the expected improvement acquisition function is maximized. SMAC solves the maximization problem by a multi-start local search. According to the authors, this method provides better results than random sampling.

# Chapter 2

## Multi-fidelity optimization

Even though Bayesian optimization techniques are more sample-efficient compared to a random search, there is another family of approaches that strive to improve the search efficiency further. Instead of training the network until full convergence to obtain the performance metric, multi-fidelity techniques make use of the assumption that a good enough approximation of the final performance can be obtained much faster — either by stopping early or by training the model on a subset of the training data. This either reduces the computation time or allows the algorithm to evaluate more configurations in the same amount of time. The actual efficiency gain depends on the quality of the intermediate results. The training of neural networks is an inherently noisy process, which makes the problem more challenging. Nevertheless, many approaches successfully exploit multi-fidelity evaluations.

Formally, a new hyperparameter  $\lambda_{fid} \in [0, 1]$  is introduced that allows us to evaluate the function  $f(x, \lambda_{fid})$  using just a  $\lambda_{fid}$  portion of the full budget (e.g. epochs). In most of the literature, the term *budget* is used for specifying the fidelity. Determining the fidelity at which to evaluate the function is the challenging task. Usually, the algorithm has no prior knowledge of the relationship between different fidelities, as well as the reliability of the estimates. That is why most of the multi-fidelity algorithms use some kind of schedule to progressively increase the fidelity hyperparameter.

Finally, we should also note that multi-fidelity techniques are not the universal solution to every hyperparameter optimization problem. If the optimization budget is high, standard random search or Bayesian optimization might perform better, because the partial evaluations can be misleading and good configurations might be discarded too soon. Multi-fidelity techniques usually provide the greatest benefits on a low budget. Often some information is already known before the tuning process. It could be an experience with the same model, or with a similar dataset, that gives us an idea of which hyperparameter configuration

might perform well. We might even have logs of a previous hyperparameter tuning experiment that is similar to the current task. In that case, there is a subfield of hyperparameter optimization called *transfer learning* that deals with the problem of reusing previous experience to speed up the search. Transfer learning methods range from simply suggesting a few configurations that we think could perform well to sophisticated pre-trained Gaussian processes and kernels. Transfer learning methods are beyond the scope of this thesis, and we refer the reader to the literature [17, 18, 19] for further details.

## 2.1 Early stopping

### 2.1.1 Learning curve extrapolation

The simplest way to reduce the runtime of an algorithm is to stop the computation before it finishes. Swersky et al. [20] noticed that human experts have the ability to assess whether the model will eventually be useful early in the training and developed a method to leverage early stopping. They refer to this method as freeze-thaw Bayesian optimization, because it allows for pausing or aborting the training procedure when the model does not seem promising and resuming the training later if needed. This is combined with the Bayesian optimization framework for hyperparameter search and the authors propose a technique for estimating when to pause and resume training. The algorithm uses an information-theoretic criterion to determine which models to thaw.

Freeze-thaw method relies on the assumption that for many models the training loss roughly follows an exponential decay. Swersky et al. [20] developed a new kernel to serve as a prior characterizing the learning curves. This kernel is then used to forecast the final training loss and to provide these estimates to the Bayesian optimization. The kernel was successfully applied to matrix factorization and other problems, but it did not describe the learning curves of deep neural networks well. The same idea was explored by Domhan et al. [21], but with a focus on deep neural networks. They developed a technique for extrapolation of learning curves based on a probabilistic model and used the model to terminate a training run when its performance is most likely going to be worse than the performance of the best model encountered so far. They modeled learning curves using eleven different model families and concluded that even though all of these models capture certain aspects of learning curves, no single model can describe all learning curves by itself. Therefore, they combined the models in a probabilistic framework. Their approach is agnostic to the hyperparameter optimizer and sped up the hyperparameter optimization approximately by a factor of two. Even though the extrapolation of learning curves is a promising approach, it did not

catch up.

### 2.1.2 Successive Halving

Arguably the most influential approach to multi-fidelity optimization is the Successive Halving algorithm proposed by Jameison and Talwalkar [22]. Even though the algorithm is rarely used in its original form, the success of Successive Halving stems from the fact that the algorithm has been used as a basis by many other researchers, presumably for its simplicity and robustness. Successive Halving solves the problem of efficient budget allocation by iteratively increasing the fidelity only for the best candidates. Note that both fidelity and budget denote the same resource in this case.

The algorithm works in rounds, each round has the same fixed budget  $B$  that is uniformly distributed between the candidate solutions. First, it starts with  $n$  randomly sampled configurations and the budget is set as  $B \leftarrow nb_0$ , where  $b_0$  is the minimal budget. For simplicity, we can assume that  $b_0 = 1$ , which means that in the first round, the budget  $B = n$  is uniformly distributed between the  $n$  solutions, so each candidate solution is trained for one epoch. After the intermediate results are obtained for all solutions, the algorithm discards the worst half and the first round ends. Since only half of the solutions are left now and the budget for each round is fixed, each solution is allocated twice the budget in the next round. This process is repeated until a single configuration remains. The pseudocode of the generalized algorithm is provided in the Algorithm 3. The only difference is that the generalized algorithm uses a *reduction factor*  $\eta$  – it keeps only  $1/\eta$  of the best configurations and the budget is increased by the factor of  $\eta$ .

---

**Algorithm 3** Successive Halving

---

**Input:** Search space  $\mathcal{X}$ , number of initial configurations  $n$ , min budget  $b_0$ , max budget  $B$ , reduction factor  $\eta$ .

- 1:  $b \leftarrow b_0$
- 2: Sample  $n$  configurations  $X \subset \mathcal{X}$  at random.
- 3: **while**  $b \leq B$  **do**
- 4:     **for**  $x \in X$  **do**
- 5:         Evaluate  $x$  for a budget of  $b$ .
- 6:      $b \leftarrow \eta b$
- 7:     Select the  $1/\eta$  best performing configurations in  $X$  and discard the rest.

---

To illustrate the efficiency of the algorithm, let us consider a practical example. Suppose we want to optimize the hyperparameters of a network, and we know that it converges within 64 epochs, so we set the maximal budget  $B := 64$ . We

will choose the reduction factor  $\eta := 2$ , the number of initial configurations  $n := 64$ , and the minimal budget  $b_0 := 1$ . There will be seven rounds of the algorithm – with 64, 32, 16, 8, 4, 2, and 1 considered solutions. Each round has a budget of 64, so the total budget spent is 448 epochs. If we have performed a random search and trained the same 64 configurations, a total budget of 4096 would be needed. In this particular case, the search cost was reduced by a factor greater than 9.

The algorithm allocates exponentially more resources to more promising configurations to make the search more efficient. A big advantage of Successive Halving is that it is a general algorithm with almost no assumptions on the optimized function. To work well, it only assumes that there is a rank correlation between different fidelities. That is, the ranking of the solutions does not change dramatically from round to round. Fortunately, this is often the case in practice.

A theoretical drawback is that the algorithm might never converge. For example, if the best configurations perform poorly in the beginning, then the algorithm discards them before they have the opportunity to converge. A practical downside of the algorithm is that we have to choose its hyperparameters. We should have an estimate of the budgets, but more importantly, we have to choose the number of configurations  $n$  in the first round manually. Either we choose large  $n$  if we prefer to train a lot of hyperparameter configurations with a smaller training time, or a small  $n$ , resulting in the exploration of fewer hyperparameter configurations that are allocated a larger budget. This essentially forces us to make the exploration-exploitation trade-off. Finally, the Successive Halving algorithm cannot be parallelized efficiently. After each round, all candidate solutions must be evaluated before a decision can be made and the number of configurations drops each round.

### Asynchronous Successive Halving (ASHA)

Extension of Successive Halving to support massively parallel computing was developed by Li et al. [23]. Their algorithm also improves several other aspects of the original algorithm, making it more practical to use.

ASHA does not have the parameter  $n$  that sets the number of initial configurations. Instead, there are  $n$  workers that run in parallel. It also does not run in distinct rounds, instead, the different fidelity levels are called *rungs*. There are two variants of the algorithm that differ in how the algorithm behaves when a configuration reaches the next rung – *stopping* and *promotion*. Upon reaching the next rung, the stopping variant decides immediately whether to stop the run or let it continue. If the trial is among the top  $1/\eta$  performing trials currently at the rung, it continues. It is stopped otherwise, and a new configuration is sampled at random for the freed worker. Since at least  $\eta$  trials are needed to make the

decision, the default action is to continue.

The promotion variant can pause and resume trials. When a trial reaches the next rung, it is paused there. Whenever a worker becomes available, all rungs are scanned in descending order for a suitable trial to resume. If a paused run that is in the top  $1/\eta$  rung's trials is found, it is promoted and the training continues until the next rung. If no trial to be promoted is found in any of the rungs, a new trial is randomly sampled and starts running.

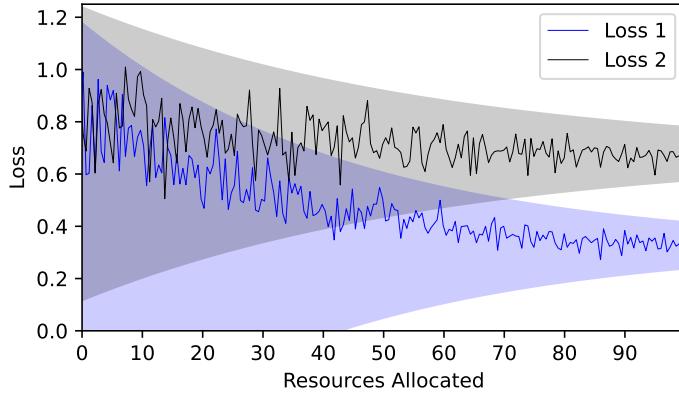
ASHA offers several advantages over Successive Halving. The first is that the number of configurations and rung sizes are not fixed. The algorithm can run for as long as the computational budget allows, adding new configurations as required without any limit. ASHA is also designed for good anytime performance, which is achieved by the promotion and stopping policies. The trials are pushed to finish as soon as possible. As a consequence, the delay until at least one trial is fully finished is reduced, and promising trials do not have to wait for all the other trials to get to the same rung. The anytime performance should be better even if just one worker is available.

### 2.1.3 Hyperband

The Hyperband algorithm, developed by Li et al. [6], is another extension of the Successive Halving algorithm. It is also a pure early-stopping algorithm. The authors suggested extending it with Bayesian optimization but left it for future work. Instead, they focused on the deficiencies of Successive Halving, namely its reliance on well-chosen hyperparameters. That is probably why Hyperband became so popular among researchers as well as in the open-source community in hyperparameter optimization frameworks.

Recall that Successive Halving needed  $n$ , the number of configurations to consider, to be determined beforehand by the user. The authors of the Hyperband paper call it “the  $n$  versus  $B/n$  problem”. Given some finite budget  $B$ , the algorithm has to allocate the resources to  $n$  configurations, allocating  $B/n$  resources on average across the configurations. Therefore, the more configurations the algorithm considers, the less resources per configuration can be allocated on average. Using an example, they illustrate why the optimal choice of  $n$  depends mainly on how hard is it to distinguish similarly performing hyperparameter configurations from each other. Because we do not usually have this information, the optimal setting is not known in advance. We include the example because we think it illustrates a general problem that all hyperparameter optimization algorithms face.

The intermediate losses are noisy, so we have to account for the uncertainty. We bound this uncertainty by the maximum deviation of the intermediate losses from the final loss, which the authors of the paper call the envelope (see Figure 2.1). Ideally, we would wait until the envelopes do not overlap. That is, if the final



**Figure 2.1** An illustration of two loss functions and their envelopes as the shaded area.

losses are  $l_1$  and  $l_2$ , and the width of the envelopes is less than  $l_2 - l_1$ , then the intermediate losses are guaranteed to be less than  $\frac{l_2 - l_1}{2}$  from the final losses. From this example we observe that more resources are needed to distinguish two configurations if the envelopes are wider (the losses are more noisy), or if the terminal losses are closer together. The choice of  $n$  also places an upper bound on the execution time of a single configuration. Therefore, by choosing  $n$  that is too large, there might not be enough time for the best configurations to converge, and we might select a worse hyperparameter configuration as a result.

Hyperband addresses the  $n$  versus  $B/n$  trade-off by considering several possible values of  $n$  for a fixed budget  $B$  and running the Successive Halving algorithm several times with different values of  $n$ . The authors call one run of Successive Halving within Hyperband a *bracket*. Each bracket is designed to use approximately  $B$  total resources. With each value of  $n$  a minimum resource  $r$  is associated that specifies the minimum resource allocated to each configuration. We have called the same parameter  $b_0$  in the description of the Successive Halving algorithm. A larger value of  $n$  corresponds to a smaller  $r$ , which in turn results in more aggressive early-stopping. An additional advantage of resetting the search is that Hyperband is hedging against bad instantiations of the randomly sampled configurations and their initialization.

Now we describe Hyperband in greater detail as presented in Algorithm 4. In order to run Hyperband, we need to specify two parameters. First, the maximum amount of resources that can be allocated to a single configuration  $R$ , and second, the reduction factor  $\eta$  that we have already seen in the Successive Halving algorithm. First, the algorithm computes the number of brackets  $s_{max}$  from the parameters and the budget it spends per bracket  $B$ . After the number of brackets is determined, the outer loop iterates over the brackets (line 1). The iteration goes

from the most aggressive exploratory bracket (the most initial configurations and Successive Halving rounds) to the bracket with just a single Successive Halving iteration, which is equivalent to a random search. For each bracket, the number of initial configurations  $n$  is calculated, as well as the minimal resources spent per configuration  $r$  (line 2), so that the bracket spends approximately  $B$  resources. The configurations are sampled at random on line 3. The inner loop within the bracket runs standard Successive Halving for the number of iterations given by the bracket (line 4). Inside the Successive Halving loop, we explicitly compute the number of configurations considered in  $i$ -th iteration as  $n_i$  (line 5), as well as the budget  $r_i$  that the configurations are trained to (line 6). Then the configurations are evaluated and  $\lfloor n_i/\eta \rfloor$  configurations with the best metric values are kept in  $X$ , while the rest is removed from the set (lines 7–9).

---

**Algorithm 4** Hyperband

---

**Input:** Search space  $\mathcal{X}$ , maximum resource  $R$ , reduction factor  $\eta$ .  
**Initialization:**  $s_{max} = \lfloor \log_\eta(R) \rfloor, B = (s_{max} + 1)R$

- 1: **for**  $s \in \{s_{max}, s_{max} - 1, \dots, 0\}$  **do** ..... *Hyperband brackets*
- 2:      $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil, r = R\eta^{-s}$ .
- 3:     Sample  $n$  configurations  $X \subset \mathcal{X}$  at random. ... *Start of Successive Halving*
- 4:     **for**  $i \in \{0, \dots, s\}$  **do**
- 5:          $n_i = \lfloor n\eta^{-i} \rfloor$
- 6:          $r_i = r\eta^i$
- 7:         **for**  $x \in X$  **do**
- 8:             Evaluate  $x$  for a budget of  $r_i$ .
- 9:         Select the  $\lfloor n_i/\eta \rfloor$  best performing configurations in  $X$ .
- 10: **return** Configuration with the smallest intermediate loss.

---

Since the maximum resource parameter  $R$  to spend on a single configuration is largely specified by the task, the only parameter left is the reduction factor. Even the reduction factor does not give us much freedom for tuning, which is generally seen as an advantage. We can either use the default value  $\eta = 3$ , or we could opt for a little less or a little more aggressive early stopping by setting it to 2, or 4, respectively. It might also be seen as an advantage or as a disadvantage that the number of resources one run of Hyperband spends is largely fixed, except for the little room that the parameters give us. As a consequence, the authors recommend running Hyperband repeatedly if the budget allows it.

In the publication [6], the authors compare the Hyperband algorithm to three Bayesian optimization algorithms (with TPE, Random Forest, and Gaussian Process as a surrogate) on CIFAR-10, rotated MNIST and SVHN datasets. They also include random search and 2x-random search as a baseline. Hyperband

consistently outperformed other algorithms at the beginning of the search. As the search progressed to spending the whole budget, the differences were only small between the methods.

### 2.1.4 Model-based algorithms

#### Hyperband extensions

The drawback of Hyperband is that it does not scale well into larger budgets and random search starts to close the gap. Falkner et al. [24] proposed a new algorithm BOHB to fix this. They combine Hyperband with Bayesian optimization to complement each other. Bayesian optimization needs a few initial trials to gather enough data to fit the surrogate model, so a few iterations of random search are performed at the beginning. This is where the strong low-budget performance of Hyperband is used. On the other hand, a well-fitted Bayesian optimization model provides better suggestions later in the tuning process. The Bayesian optimization surrogate BOHB uses is based on TPE, but instead of a hierarchy of one-dimensional KDEs, the authors decided to use a single multidimensional KDE in order to better handle interaction effects between the hyperparameters. The number of randomly sampled configurations is  $d + 1$  by default, where  $d$  is the number of hyperparameters. BOHB always fits the surrogate using the observations on the highest budget possible, as soon as enough observations become available.

Another algorithm extending Hyperband is the DEHB developed by Awad et al. [25], which uses an evolutionary optimization method instead of Bayesian optimization. More specifically, the DE stands for Differential Evolution. The authors claim that the evolutionary approach provides some benefits over Bayesian optimization, such as better handling of discrete dimensions, better scaling into high dimensions, and conceptual simplicity enabling easy implementation.

DEHB does not run standard Successive Halving. The only time DEHB samples random configurations is in the first step of the first Hyperband iteration. Instead of sampling, DEHB keeps a *subpopulation* at each budget level which serves the purpose of information transfer between brackets — every subsequent bracket reuses the subpopulation updated in the previous bracket. The size of each subpopulation is equal to the maximum number of function evaluations Hyperband allocates for the corresponding budget. Information is not transferred only horizontally at a fixed budget level, but also vertically, to a higher budget. The top-performing configurations are not just promoted, instead, they are collected in a *parent pool* associated with a budget. On the next budget level, configurations are selected from the subpopulation of that budget and then modified similarly to differential evolution — three additional configurations are sampled at random

from the parent pool to generate a new mutant vector. Then a crossover operation is performed with the vector selected from the subpopulation and the mutant vector to create a new configuration. This configuration is evaluated and if it performs better than the old, it replaces the old configuration in the subpopulation. This process is governed by the precomputed Hyperband schedule.

The authors provided a lot of experiments in their paper, comparing DEHB to BOHB, random search, and other optimizers such as SMAC or Bayesian optimization with TPE surrogate. The benchmarks include NAS-Bench-101, NAS-HPO-Bench, or Reinforcement Learning Cartpole environment. DEHB is much more efficient in some benchmarks while performing similarly to BOHB, the next-best HPO optimizer from the experiments, in the rest. DEHB has a strong performance early with a low budget, and the performance does not fall off even for large budgets, which it often does for BOHB.

### Model-based Asynchronous Successive Halving

Even the Asynchronous Successive Halving can be extended to sample configurations from a surrogate model. The biggest challenge it brings is that all the Bayesian optimization models and methods are sequential by default – optimizing the acquisition function gives us just a single (or very similar) configuration until the model is updated. That is because trials that have been started and have not returned any performance metric yet are not taken into account. The following methods take pending evaluations into account and that allows them to get new suggestions from the model asynchronously even when multiple trials are already running.

The first algorithm we introduce is the MOBSTER [26]. MOBSTER uses a single Gaussian process to model  $f(x, b)$ , where  $b$  is the budget. The advantage over fitting a model at the highest budget only as in BOHB is that it models the cross-correlations between fidelities as well. MOBSTER handles pending evaluations by fantasizing. That is, the value of the pending trials is estimated by marginalizing the acquisition function over the predictive distribution of the Gaussian process. In practice, it is cheaper to approximate the value by sampling function values from the Gaussian process and averaging the acquisition function over the samples. The Gaussian process uses the Matérn 5/2 kernel with automatic relevance determination (ARD) and the expected improvement acquisition function. The expected improvement is optimized similarly as in BOHB. It is maximized on the highest budget with enough observations.

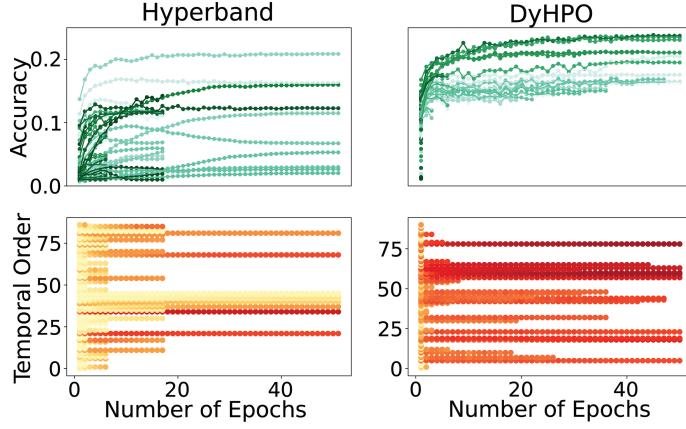
The second algorithm, Hyper-Tune [27], chooses a different approach to tackle the challenges of asynchronous Bayesian optimization. Hyper-Tune utilizes multi-fidelity evaluations by building an independent surrogate model for each fidelity level with enough observations. The acquisition function is then computed from

the whole ensemble using dynamically updated weights to take the reliability of each fidelity level into account. This gives Hyper-Tune the ability to use any Bayesian optimization method without the need to extend it to support multi-fidelity evaluations. The problem of pending evaluations is also solved in an algorithm-agnostic way. Pending evaluations are penalized locally. Hyper-tune is inspired by Hyperband in a way it deals with the “precision vs cost” trade-off. Recall that Hyperband uses brackets for this purpose. Hyper-tune has an adaptive mechanism to select the best bracket to balance this trade-off. It iteratively tries different brackets (starting with  $n_i$  configurations trained for  $r_i$  initial training resources) and based on the measurements from the bracket, the parameters are updated. Hyper-tune uses a measure of precision based on the partial ordering from different resource levels for the updates. As a result, Hyper-tune is a highly adaptable and scalable framework for multi-fidelity optimization.

## DyHPO

One of the most recent multi-fidelity optimization algorithms called DyHPO was developed by Wistuba et al. [28]. The two main improvements of DyHPO over previous approaches are a dynamic allocation of resources and a multi-fidelity acquisition function paired with a deep kernel Gaussian process. Upon reviewing existing multi-fidelity approaches including Hyperband, BOHB, and DEHB, the authors stated a conjecture that these multi-fidelity methods suffer from a major issue. The Hyperband-based algorithms fail when low-budget performance is not a good indicator for the full budget performance. The authors argue as an example that a properly regularized network converges slower in the first few epochs, but typically outperforms a non-regularized network after full convergence. This problem is addressed by a GP kernel capable of capturing the similarity of two hyperparameter configurations even if the configurations are evaluated on different budgets.

The first improvement is the dynamic allocation of resources. Instead of pre-allocating the budget, DyHPO dynamically promotes the most promising configuration to be trained for some additional amount of resources (e.g. one epoch). This is best illustrated with an example, so we include a Figure 2.2 from the original paper. The illustration shows that the dynamic promotion mechanism leads to greater efficiency – DyHPO does not spend so much time on mediocre configurations and gradually increases the budget only for the most promising configurations. The decision to promote a configuration is made by the surrogate model, which uses a multi-fidelity acquisition function. Without going into too much detail, the acquisition function calculates for each configuration the expected improvement gained by evaluating the configuration for one more budget. Therefore, configurations across all budgets compete for the resources at



**Figure 2.2** **Top:** The learning curve for different hyperparameter configurations. The darker the learning curve, the later it was evaluated during the search. **Bottom:** The hyperparameter indices in a temporal order as evaluated during the optimization and their corresponding curves. (Source: Figure 1 from Wistuba et al. [28])

the same time.

DyHPO uses a learnable deep kernel Gaussian process surrogate. Let us consider a hyperparameter configuration  $x$  evaluated up to a budget  $b - 1$ , which produced a learning curve  $\mathbf{Y}_{x,b-1}$ , and some other configuration  $x'$  using the same notation. Standard Gaussian process kernel calculates the similarity of the two configurations as  $k(x, x')$ . The deep kernel first transforms the inputs with a neural network. Let us denote the transformation as  $\varphi$ . Then, the deep kernel in DyHPO calculates the function:  $k(\varphi(x, \mathbf{Y}_{x,b-1}, b), \varphi(x', \mathbf{Y}'_{x',b'-1}, b'))$ . The kernel  $k$  used in the paper is the squared exponential kernel with parameters  $\Theta$ . The neural network uses a linear layer for the configuration  $x$  and the budget, and a one-dimensional convolutional layer for the learning curve followed by a max pooling layer. The outputs of these hidden layers are passed to one more linear layer. If we denote the trainable parameters of the network as  $w$ , the optimal values for  $w$  and  $\Theta$  are found by computing the maximum likelihood estimate, which is optimized by gradient descent.

Dynamic resource allocation comes at a cost. The surrogate model is queried at every step, and new data are generated at every step as well. In order to control the computations of the surrogate model, the model can be updated only every  $i$ -th iteration, or larger steps between budgets can be set. The biggest advantage of DyHPO is the potential to be much more efficient than commonly used algorithms. This was demonstrated on many benchmarks, including the LCBench, NAS-Bench-201 and TaskSet, where DyHPO achieved state-of-the-art results while keeping the overhead at a reasonable level.

## 2.2 Subsampling

The second possibility for reducing fidelity is to use only a fraction of the training set. The intuition behind this approach is that even a small subset of the training data should contain enough information to approximate the performance on the full dataset while training faster. Even though a lot of researchers explored this idea, it did not achieve the same level of widespread usage as early-stopping. To our knowledge, no commonly used hyperparameter optimization framework utilizes subsampling as a means for multi-fidelity evaluations.

One of the first papers to explore the idea was the Multi-task Bayesian optimization (MTBO) by Swersky et al. [29]. The authors focused primarily on the cold start problem of Bayesian optimization methods and suggested a method to use the knowledge from prior experiments on similar datasets, which is called transfer learning in the literature. In order to work efficiently, the algorithm learns the correlation between tasks (datasets). The authors realized that this can be leveraged for faster exploration of the search space even with a single dataset. The authors experimentally verified that using a subset of the training dataset as a cheaper auxiliary task speeds up the search.

Klein et al. [30] developed a Bayesian optimization algorithm FABOLAS – FAst Bayesian Optimization on LArge data Sets. We include the full name to highlight the focus on large datasets. The main idea of the algorithm is to introduce subset size as an additional parameter for the Gaussian process to optimize using the acquisition function information gain per unit cost. The Gaussian process then learns to approximate the correlations between different subset size values, which allows it to efficiently use smaller subsets to accelerate the hyperparameter search. The dataset size is gradually increased throughout the search. The authors performed experiments on the CIFAR10 and SVHN datasets with convolutional neural networks. FABOLAS found a good hyperparameter configuration more than 10 times faster than MTBO and the difference was even larger in comparison to Hyperband. It is worth noting that after a good-performing model was found by all algorithms, the differences in test error were only minor.

A similar approach to BOHB was implemented by G. Zhu and R. Zhu [31]. They combine Successive Halving with progressively increasing the dataset size and the number of training epochs. This way, the algorithm can explore even more configurations early on. The algorithm uses a Bayesian optimization with a surrogate model to suggest configurations for Successive Halving, but the authors do not mention which surrogate model is used. To support the idea of using only a subset of the training data, the authors provide an experiment on the MNIST dataset, where they compared a LeNet trained on a full dataset versus trained only on 10% of the dataset. The results showed a difference of a few percentage points in favor of training on a full dataset. The authors noted that the main difference in

chosen hyperparameters was in regularization hyperparameters. That is expected since training on a smaller dataset should require stronger regularization. Finally, they compared their algorithm to BOHB on CIFAR10 and CIFAR100 datasets. In both cases, the new algorithm outperformed BOHB, especially with fewer resources used.

Some researchers explored the idea of a two-step hyperparameter optimization method — first, optimize hyperparameters on a small subset of data, and then optimize the best-performing models on the full dataset. This approach was recently studied by Yu et al. [32] on a large dataset for aerosol activation emulator, containing almost 20 million examples. Their experiment is interesting because they use random search as an optimization algorithm and focus just on the dataset sizes, trying subsets as small as 0.00025 of the whole dataset (5000 examples). They have found that it does make sense to optimize hyperparameters on a small dataset first and that a lot of good models from the low-fidelity round perform well even on the full dataset. They were able to speed up the search 135 times while using just the simple and parallel random search, albeit on a single and very specific task.

Closely related to the multi-fidelity optimization is the concept of *coresets*. In all the papers above, the subsets were sampled at random. One way of improving the efficiency of the multi-fidelity algorithms might be sampling the subsets cleverly. Such an approach focused on gradient-based machine learning algorithms was proposed by Mirzasoleiman et al. [33]. The main goal of the authors was to develop a method to create a subset of the data, such that the gradient computed on the subset closely estimates the full gradient on the whole dataset. The authors performed experiments on several models and datasets including the CIFAR10 dataset with the ResNet-20 model and showed that their approach increases the efficiency of the training compared to training on the entire dataset, and the performance of the network is better when compared to random subsets of the same size.

## 2.3 Performance evaluation

Evaluating how the algorithms perform and comparing them to each other is a crucial task in an area, where efficiency is among the most important metrics. That is why most of the research papers include some evaluation, and we will summarize the common methodologies and benchmarks used in the literature here.

### 2.3.1 Real benchmarks

Originally, when hyperparameter optimization was being formalized, it was common to choose some datasets and do experiments independently. For example, in the paper Random Search for Hyper-Parameter Optimization [5], the authors chose several variations of the MNIST dataset with their own splits, and a synthetic dataset for the classification of rectangles. In Practical Bayesian Optimization of Machine Learning Algorithms [34], the experiments included the Branin-Hoo function, which was a common benchmark for Bayesian optimization techniques. Another experiment was performed with Support Vector Machines. Finally, they included an experiment on CIFAR-10 with convolutional neural networks, where the Bayesian optimization algorithm was compared to a hand-tuned model. The MNIST and CIFAR-10 datasets continued to be popular among researchers for quite some time. In the Hyperband [6] paper, the authors used MNIST, CIFAR-10, and SVHN datasets still, 6 years after the Random Search paper we have just mentioned. It was not until very recently that the experiments started to get more unified with tabulated benchmarks.

### 2.3.2 Tabulated benchmarks

Tabulated benchmarks have become one of the most common ways to evaluate performance in a standardized manner. A tabulated benchmark is a dataset that contains precomputed results of various models. The biggest advantage of this approach is that evaluating a hyperparameter configuration is as fast as retrieving a value from a table, which is almost instant. This allows researchers to perform experiments with much more repetitions, getting more robust results. Also, since the values are precomputed, the experiments are repeatable and everyone gets the same problem to evaluate their algorithms on, assuming the method is the same.

The disadvantage of tabulated benchmarks is that they contain values only for some finite number of configurations, which cannot cover the search space if some hyperparameters are continuous. Therefore, missing values have to be handled somehow. There are two common approaches. We either limit the search space to values in the dataset or the results for missing values are interpolated using a surrogate model, e.g. a K-Nearest Neighbors classifier, or an MLP classifier. The surrogate model brings variation into the results, though. The carefully chosen search space also means that the optimized number of hyperparameters is usually rather low, and the ranges are also limited. Even then, the number of configurations that have to be evaluated to create a tabular benchmark can be in tens of thousands. Tabular benchmarks are a valuable tool for the development and evaluation of hyperparameter algorithms regardless.

**NAS-Bench-201** The NAS-Bench-201 [35] contains 15625 multi-fidelity configurations of computer vision architectures evaluated on 3 datasets (CIFAR10, CIFAR100, ImageNet-16-120). The NAS stands for neural architecture search, which means that NAS-Bench is not primarily a hyperparameter optimization benchmark. Nevertheless, it is often included in hyperparameter optimization research papers. The search space consists of 6 categorical variables, each having 5 of the same options ( $3 \times 3$  convolution layer, Average Pooling layer, etc.). Through these variables, a cell that is used to form a neural network is optimized.

**LCBench** Another popular tabular benchmark is the LCBench [36]. The LC in the name stands for learning curve, which means that LCBench tracks the performance of configurations throughout the training. It contains training data for 2000 different configurations across different MLP funnel-shaped nets and their hyperparameters. Each configuration is evaluated on 35 datasets over 50 epochs. The task is to optimize 7 hyperparameters, 4 float and 3 integer. The hyperparameters include a learning rate, a weight decay, and a number of layers. The complete list of all hyperparameters and their values is provided in Table A.2 in Appendix A.

**FCNet** The FCNet [37] multi-fidelity benchmark contains 62208 configurations evaluated on 4 datasets. The base architecture is an MLP feed-forward neural network with two fully connected layers followed by a linear output layer. The search space includes 4 architectural choices (number of units and activation function for both layers), and 5 other optimizer and regularization hyperparameters. We list the hyperparameters and their values in Table A.3. The authors chose to discretize the search space and performed an exhaustive evaluation of all 62208 configurations. Each configuration was trained 4 times with full learning curves provided as well.

# Chapter 3

## Experiment definitions

One of the barriers to the widespread adoption of hyperparameter optimization algorithms is the lack of empirical evidence to decide which algorithm to choose. We performed a set of experiments, both on tabulated benchmarks and on real-world problems, in order to provide more evidence and to help practitioners with the decision of choosing a suitable hyperparameter optimization algorithm for the task at hand. In this chapter, we describe how the experiments were performed.

### 3.1 Benchmarks and datasets

We perform experiments on tabular benchmarks LCBench, FCNet, and NAS-Bench-201. Since the NAS-Bench-201 is a neural architecture search benchmark, it is not completely suitable for our experiments. We include it mostly because it is commonly used in the hyperparameter optimization literature. The LCBench contains over 35 datasets, but the neural network architecture is the same for all the datasets and the tasks are similar. Therefore, we selected just a subset of tasks from the LCBench. This helps to keep the results more balanced when aggregating the results. We tried to select the more challenging tasks heuristically. The full list of all used tabular benchmarks can be found in Table A.1.

For the real-world evaluation, we chose several datasets and deep learning models. We compare the algorithms on the well-known image classification CIFAR10 and SVHN datasets. For each dataset, two deep learning models are used — a sequential CNN model, and a residual neural network model. *I should probably describe the architectures used a bit more, here, or in appendix, or just refer to the code?*

Next, we tested the tuning algorithms in a domain that is rarely represented in the hyperparameter optimization literature. We run experiments using the medical dataset PTB-XL. PTB-XL consists of 21837 clinical 12-lead ECGs of 10-

second length annotated by up to two cardiologists with ECG statements. It is a multi-label classification task where the goal is to assign 1 or more of the 5 diagnostic superclasses to the ECG signal. We reviewed the literature [38] to see which models perform well on this dataset and chose an RNN architecture and a CNN architecture xResnet1d.

**Experiment in progress.** Finally, we optimize hyperparameters of a DenseNet model on a ChestX-ray14 dataset. It contains 112200 X-ray images from 30805 unique individuals with fourteen different disease categories. It is a challenging task, which is why we chose the more recent DenseNet architecture instead of a ResNet or simpler architecture. **List the datasets as itemize list instead of paragraphs?**

## 3.2 Algorithms

First, we start with a larger set of algorithms to evaluate on cheap tabulated benchmarks. Then, we select a subset of the algorithms to evaluate on expensive real-world problems. We chose the following algorithms for the comparison:

- **Random search:** Random search serves the purpose of a baseline. We use a random search with a modification. The first configuration is not random, instead, the mid-point rule is used for all values. The mid-point rule chooses a value in the middle of the range, which often works quite well. We use it because in a real-world hyperparameter optimization problem, this is a simple way to have a chance to get a good solution early. In general, this should make the random search as a baseline stronger.
- **Hyperband:** The Hyperband is interesting because BOHB and DEHB use the same scheduling logic. That allows us to directly compare it to the model-based algorithms.
- **ASHA:** ASHA provides a simple asynchronous early-stopping baseline.
- **BOHB:** Popular model-based Hyperband algorithm that uses KDE-based model.
- **DEHB:** So that we have other models than Bayesian optimization.
- **MOBSTER:** Asynchronous model-based baseline.
- **Hyper-Tune:** Should be an improvement over the MOBSTER.
- **DyHPO:** The most recent algorithm. We do not use the original deep kernel version, a normal Gaussian process is used as a surrogate.

For the real-world experiments, we reduced the set of algorithms to Random Search, ASHA, BOHB, HyperTune and DyHPO. We chose to use the stopping variant of ASHA so that there is another alternative to Random Search that does not require checkpointing. MOBSTER and DEHB were removed based on the results of tabulated experiments.

### 3.3 Evaluation metrics

We set the total optimization budget to 20 full function evaluations. For example, all models in LCBench are trained for 50 epochs, so the total budget is 1000 epochs for all LCBench benchmarks. We use 30 repetitions with different seed values for the tabulated benchmarks and 10 for the real-world benchmarks. The HPO algorithms optimize either the validation loss, or metric computed on the validation set (e.g. the accuracy, or the area under the ROC curve). We always use the optimized function for the evaluation. For the aggregation of the results, we convert all metrics to minimization and normalize the values into the range from 0 to 1 for each benchmark, because the metrics differ across datasets and benchmarks. All figures with the normalized metrics can be found in the Appendix B. In the text, we show plots using the original metric. Training loss and validation loss are always tracked, as well as the wall-clock time and the time spent by function evaluation per epoch. In order to obtain the overhead time, we subtract the function evaluation duration from the wall-clock time.

### 3.4 Experimental setup and procedures

We designed the real-world experiments by trial and error. First, a dataset was chosen. Then, we considered which models are suitable for use with the dataset. We often used the literature to choose the model. After the dataset and model were chosen, we started adding hyperparameters to optimize. Optimizer hyperparameters were included in every experiment. Then we often tried to parametrize the neural network so that the capacity can be tweaked with a few high-level hyperparameters. We did not want to do Neural Architecture Search, so we often chose one hyperparameter for the depth of the network and one for the width. Then we usually added regularization hyperparameters, especially for the smaller datasets. Finally, for some tasks, we added hyperparameters to control data augmentation. **Should I include this paragraph?**

After the hyperparameters were chosen, we set the ranges for the individual hyperparameters. We used previous experience to set bounds that we thought could contain a good solution and widened it a bit. Note that we did not carefully

design the hyperparameter spaces to achieve some specific goal (e.g. make the search artificially hard). We wanted the experiments to be close to the real world, albeit on the simpler side. One thing that we did extra was that we checked if the best solutions were not on the boundary of the hyperparameter range. If this was the case, we widened the range. For the capacity hyperparameters, we were limited in the size of the model by the space needed for checkpointing of the models. We set the maximal number of epochs manually for each real-world experiment by observing the learning curves and selecting a slightly larger value than needed for convergence.

We used the Syne-tune [39] Python library to perform the experiments. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. The experiments were run in a Singularity container. We used the PyTorch NGC Container release 23.11. The experiments took over 200 days ([update](#)) of CPU time to compute; larger models were trained on a GPU. Because the experiments were run on a heterogeneous computational grid, we cannot provide further details on the hardware.

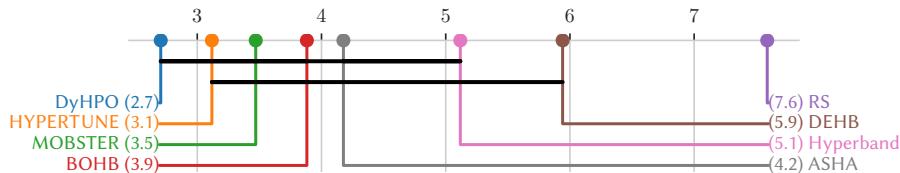
In accordance with the best practices for statistical comparison of algorithms across many datasets by Demšar [40], we use the Friedman test to test the null hypothesis that all algorithms are equivalent. If the Friedman test rejects the null hypothesis, we follow up with the Wilcoxon signed-rank post-hoc test corrected with the Holm method to determine which algorithms perform differently. These tests are recommended because of several properties. They are non-parametric and have almost no assumptions on the data distribution. Since we cannot guarantee normality and homogeneity of variance of the results, it is safer not to use parametric tests like ANOVA. The Wilcoxon test ranks the data internally, so the magnitude of values matters, but only to a limited extent, which is also desired.

# Chapter 4

## Results and discussion

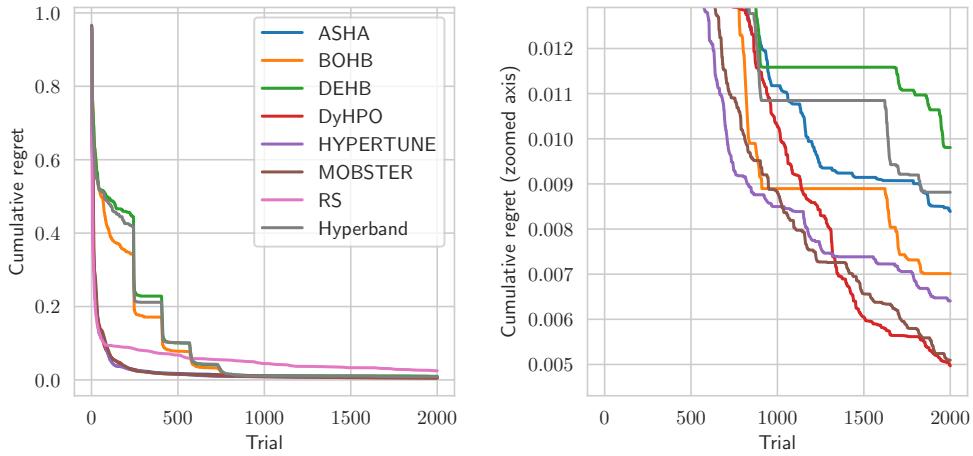
### 4.1 Tabulated benchmarks results

We begin with the aggregated results from all tasks in Figure 4.1. It shows the average ranks of the algorithms across 17 datasets from 3 tabulated benchmarks. The connecting bars signify that there is not a statistically significant difference between the algorithms. From the diagram, we can observe that three distinct groups of algorithms arise. The random search ended up last in almost all tasks except for one, where it was very close, and performed statistically worse than all other algorithms. The situation is not as clear on the other side of the spectrum. Even DyHPO, achieving the best average rank, ended in the worst half of the algorithms on some tasks.



**Figure 4.1** Critical difference diagram from tabular benchmarks at the maximal budget. Average ranks are shown on the x-axis.

For illustration purposes, we present the results of a single benchmark in Figure 4.2. We provide all plots in Appendix B. We chose the FCNet Parkinsons dataset for the presentation because it allows us to show several general traits and performance of the algorithms. It is often the case in our experiments that random search is clearly behind all other algorithms, but it is catching up as the budget increases. We can also notice the fixed schedule of Hyperband. It



**Figure 4.2** The averaged and normalized cumulative performance metric over 30 runs on the FCNet Parkinsons benchmark. On the right, we show the same results but zoomed in.

takes Hyperband approximately 300 trials until it finally starts evaluating the first configuration fully. These delays hurt the early performance of Hyperband. In the case of FCNet with a maximal budget per configuration of 100 epochs, Hyperband should use approximately 25 full evaluations of budget per iteration. Therefore, it does not even finish the first iteration with our budget restriction. We can also notice that in the first bracket, the model-less Hyperband performs similarly to the model-based extensions. DEHB did not perform well in our experiments, it was often outperformed by model-less approaches. The performance of the most advanced methods was comparable. We cannot conclusively say that one is better than the other, as it often depends on the specific problem.

The key focus of this thesis is efficiency. A good way to measure efficiency is to compare the algorithms to the random search and measure how much faster can they find a solution as good as the random search has found after spending the whole budget. We call it the speedup factor and present the results in Table 4.1. The reported speedup factor is averaged over all benchmarks and repetitions. We assume a speedup is greater than one, which was broken for one dataset, where the random search showed the best performance with a small margin. Because we do not have the data beyond the maximal budget, we have set the speedup factor to 1 in such cases. Nevertheless, the speedup factors are significant. Even the lowest reported value from any confidence interval is 2.82 and the values of averages are around 6 for the more advanced algorithms. The confidence intervals are overlapping for all pairs of algorithms, though.

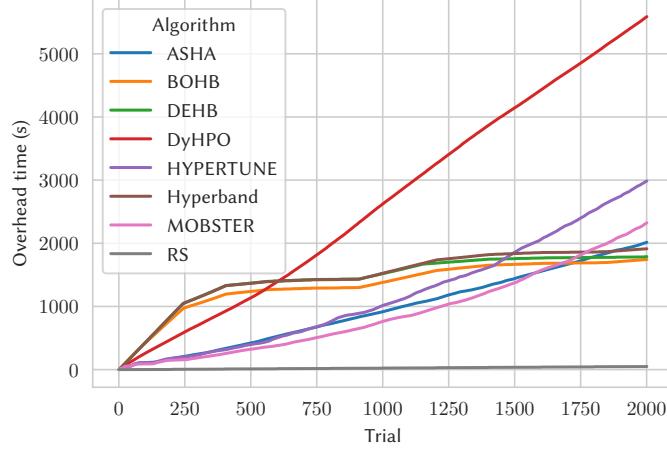
We have seen that in terms of epochs, all advanced methods beat random

algorithm	Avg. speedup	95% CI
ASHA	5.36	[4.52, 6.21]
BOHB	5.39	[4.11, 6.68]
DEHB	4.87	[3.76, 5.98]
DyHPO	7.73	[5.85, 9.61]
HYPERTUNE	5.97	[4.66, 7.28]
Hyperband	4.92	[3.83, 6.00]
MOBSTER	6.14	[4.76, 7.52]

**Table 4.1** This table shows the average speedup factor of the algorithms relative to Random Search across all benchmarks and repeated runs. Speedup factor indicates how much faster an algorithm finds a solution with the same objective function value as Random Search on full budget. A 95% confidence interval is provided for each speedup factor.

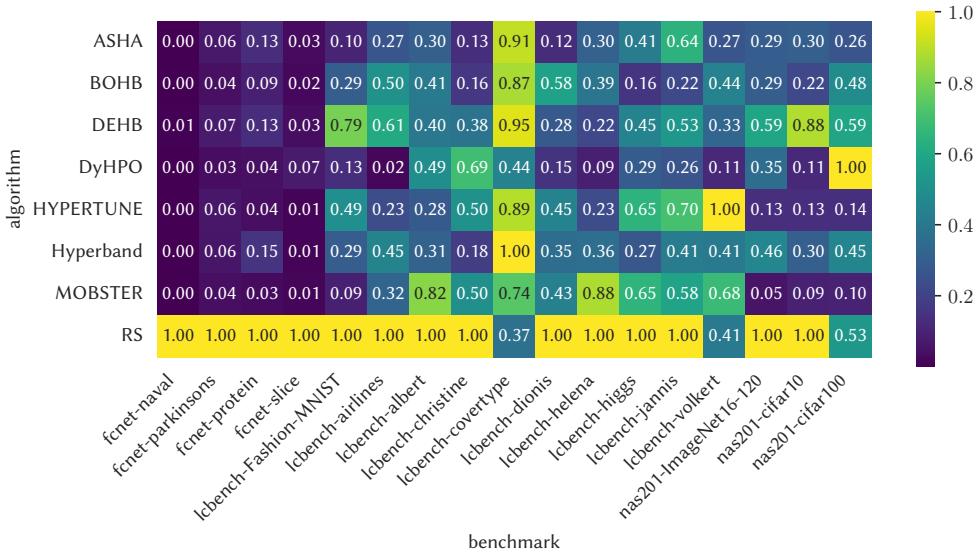
search in efficiency. But that neglects the runtime, which could offset the gains. We examine the overhead in Figure 4.3. There are several things to note first. The overhead depends on the implementation. We compare all algorithms within the same framework on the same benchmark, but we have noticed there is an overhead of approximately 2 seconds when configurations change, be it for a new configuration or resumed configuration. DyHPO suffers from this kind of overhead the most because the used implementation pauses training after 2 epochs at most, even if the same configuration is immediately resumed again. That is why we see the steepest and almost linear line for DyHPO. The methods based on Hyperband have the highest overhead in the beginning because they sample a new configuration in each trial the first 250 trials. From the comparison of ASHA to HyperTune or MOBSTER we can see that the overhead of the Bayesian surrogate model is small in this case. We can notice that the slope of the overhead changes more for the model-based algorithms than for ASHA. **I could re-run this benchmark for more trials to see how it behaves. It would be nice to measure the overhead of the model directly, but I'm not sure if I could do that.**

In order to estimate the reliability and consistency of the methods, we provide the heatmap of variances in Figure 4.4. The variances are normalized across benchmarks because we are interested in a relative comparison of the algorithms, not in the absolute values. We can see that random search has the largest variance in almost all benchmarks. Interestingly, it has the smallest variance in the lcbench-covertype benchmark. **Why?** The other algorithms seem to be similar in terms of variance, but there are some differences. It seems that DyHPO is the most consistent on most tasks, except for the nas201-cifar100, where its relative variance

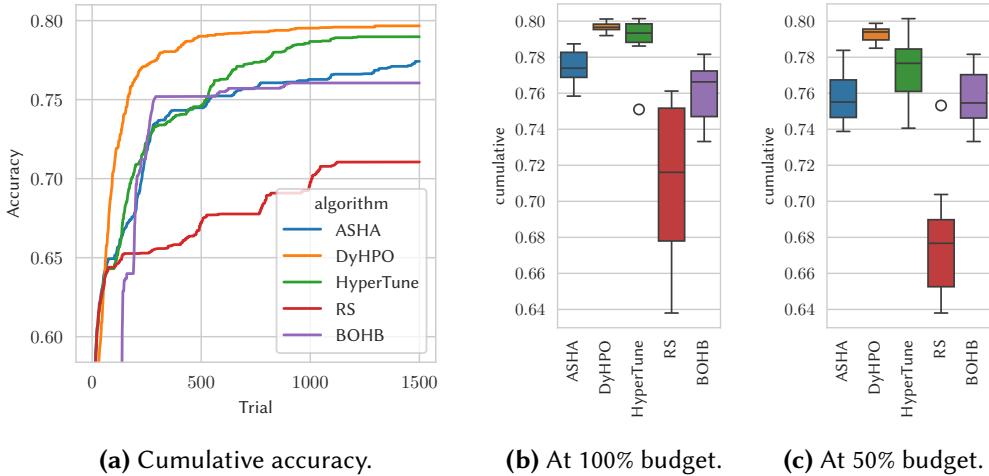


**Figure 4.3** Overhead on FCNet Naval benchmark.

is the worst. Let us also highlight that ASHA is as consistent as most model-based algorithms on the majority of tasks, if not better. **But I should probably analyze variance AND quality of solution together. It is not good to be consistently bad.**



**Figure 4.4** The heatmap of variances of the best performance across the 30 repetitions. The variances are normalized for each benchmark.



**Figure 4.5** Results of the experiment on CIFAR10 with simple CNN.

## 4.2 Real-world benchmarks results

First draft.

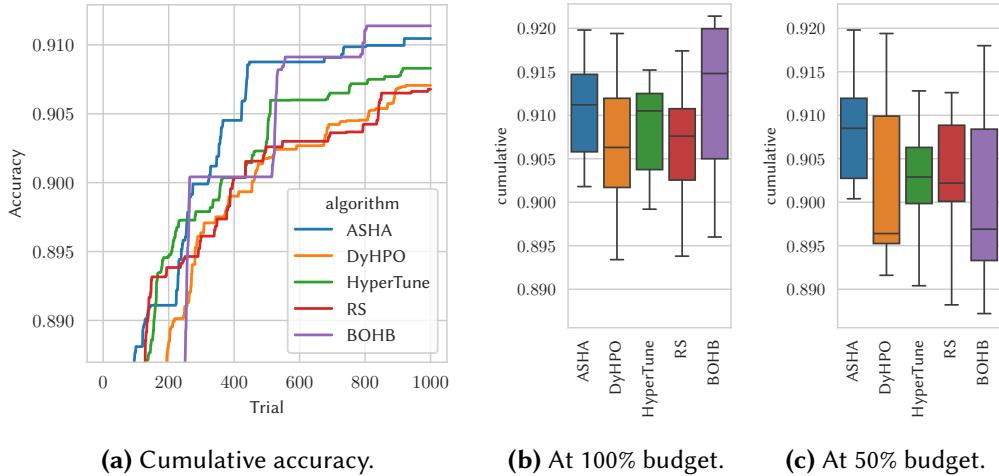
### 4.2.1 CIFAR10

#### CNN

The optimization problem consisted of eight hyperparameters, four real-valued, three integer-valued, and one binary. We optimized the optimizer hyperparameters, regularization hyperparameters, and also the size of the model with the number of layers and a multiplier for the number of channels. All hyperparameters and their domains are summarized in the Table A.5 located in Appendix A.

For training, we used the AdamW optimizer. Learning rate and the final value of cosine decay  $\eta_{min}$  were optimized. The neural network architecture consists of sequential convolutional layers, optionally with batch normalization. The network is trained for up to 75 epochs. Therefore, the total budget is equal to 1500 epochs. We compare the algorithms in classification accuracy on the validation set. We use the trial number on the x-axis.

Plot from another experiment, I'm not sure if it's not too much plot and rather provide a table with results. The following plots are evaluated after 350 training epochs because that is where the difference is the largest; so that we can test if the results will be statistically significant, or if we need more repetitions.



**Figure 4.6** Resnet-like network trained on CIFAR10 dataset for 20 full evaluations.

## Residual CNN

### 4.2.2 SVHN

I can just repeat the experiments done for CIFAR, or maybe alter them a little. But a completely different dataset and model combination would probably be better since SVHN will not give us much new insight.

### Simple CNN

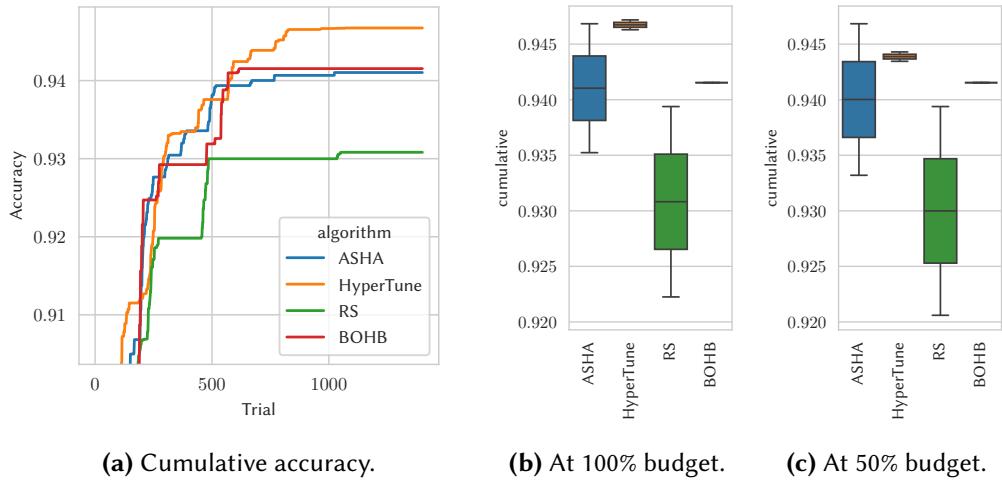
### Residual CNN

### 4.2.3 PTB-XL

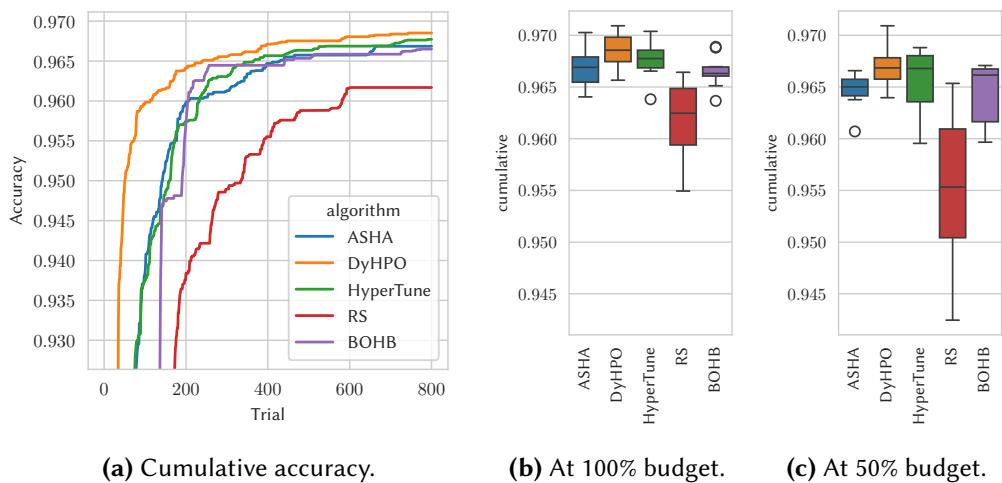
There are some specifics to consider with the PTB-XL dataset. With a little over 21000 training examples, it is a smaller dataset. Therefore, regularization could be important. The training examples are 10-second records of ECG signal at 100Hz sample rate. For training and inference, we use random segments of length 256. Consequently, the model is fed different parts of the signal at each epoch, which prevents overfitting and introduces noise into the training process.

### RNN

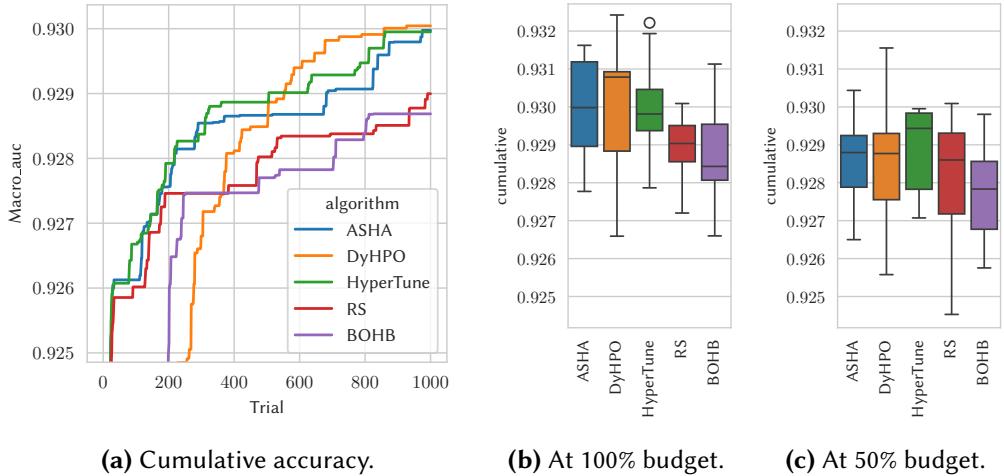
For the RNN model, we wanted to simulate a situation where more hyperparameters are optimized, especially since the tabular benchmarks are all limited in the size of the search space. In total, the search space consists of 11 hyperparameters, the full table with the optimized hyperparameters is in Table A.9. We



**Figure 4.7** Results of the experiment on SVHN with simple CNN.



**Figure 4.8** Resnet-like network trained on SVHN dataset for 20 full evaluations.



**Figure 4.9** RNN network trained on the PTB-XL dataset for 20 full evaluations.

optimize RNN-specific hyperparameters such as the RNN cell type and whether the network should be bidirectional. Hyperparameters that control the size of the network, the number of layers and the number of hidden units per layer, are optimized as well. There are the optimizer parameters as well, we have included two options for the learning rate decay as well. Finally, we optimize the regularization hyperparameters such as dropout and weight decay.

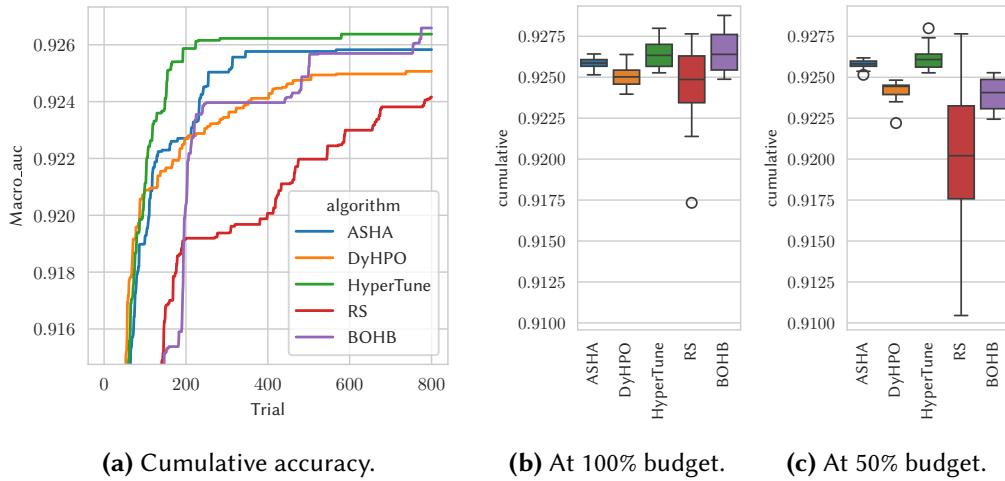
### xResnet1d

With the xResnet1d architecture, we wanted to simulate a scenario when the architecture is chosen already and the practitioner decides between just a few variants of the model. In this case, the algorithm has to choose between xresnet1d18, xresnet1d50, and the largest xresnet1d101. There is also the original f-number hyperparameter, which refers to the f-number used in the PTB-XL challenge paper. [Explain more](#). Then there are just optimizer hyperparameters and regularization hyperparameters. The full table with the optimized hyperparameters is in Table A.10.

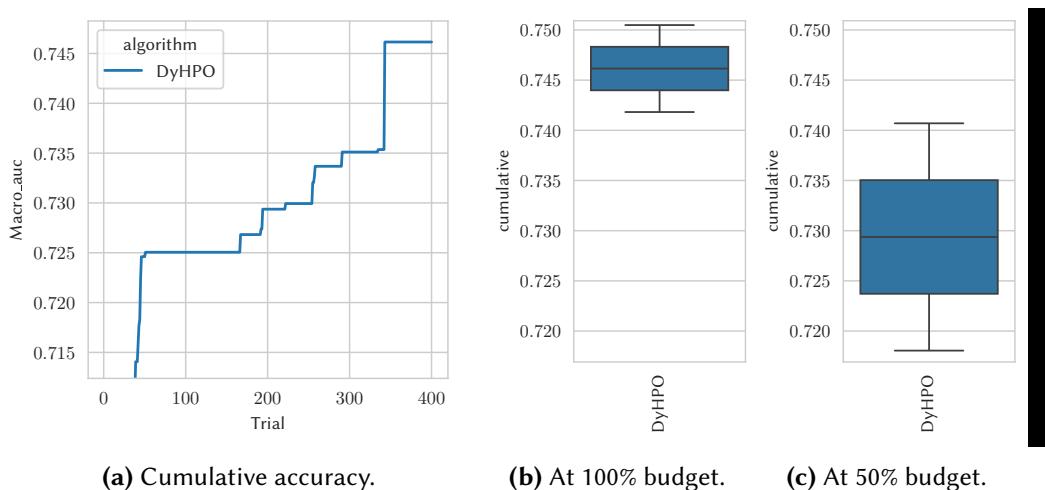
#### 4.2.4 ChestX-ray14

##### DenseNet

### 4.3 Discussion



**Figure 4.10** xResnet1d network trained on the PTB-XL dataset for 20 full evaluations.



**Figure 4.11** DenseNet trained on the ChestX-ray14 dataset for 20 full evaluations.

# **Conclusion**

# Bibliography

- [1] Takuya Akiba et al. “Optuna: A Next-Generation Hyperparameter Optimization Framework”. In: *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 2623–2631.
- [2] Marius Lindauer et al. “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 23.54 (2022), pp. 1–9. URL: <http://jmlr.org/papers/v23/21-0888.html>.
- [3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [4] Varun Godbole et al. *Deep Learning Tuning Playbook*. Version 1.0. 2023. URL: [http://github.com/google-research/tuning\\_playbook](http://github.com/google-research/tuning_playbook).
- [5] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization.” In: *Journal of machine learning research* 13.2 (2012).
- [6] Lisha Li et al. “Hyperband: A novel bandit-based approach to hyperparameter optimization”. In: *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52.
- [7] Eric Brochu, Vlad M Cora, and Nando De Freitas. “A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1012.2599* (2010).
- [8] Peter I Frazier. “A tutorial on Bayesian optimization”. In: *arXiv preprint arXiv:1807.02811* (2018).
- [9] Jonas Mockus. “On Bayesian methods for seeking the extremum”. In: *Proceedings of the IFIP Technical Conference*. 1974, pp. 400–404.
- [10] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006.

- [11] Donald R Jones, Matthias Schonlau, and William J Welch. “Efficient global optimization of expensive black-box functions”. In: *Journal of Global optimization* 13 (1998), pp. 455–492.
- [12] Jie Wang. “An intuitive tutorial to Gaussian processes regression”. In: *Computing in Science & Engineering* (2023).
- [13] James Bergstra et al. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems* 24 (2011).
- [14] Shuhei Watanabe. “Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance”. In: *arXiv preprint arXiv:2304.11127* (2023).
- [15] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Sequential model-based optimization for general algorithm configuration (extended version)”. In: *Technical Report TR-2010-10, University of British Columbia, Computer Science, Tech. Rep.* (2010).
- [16] Leo Breiman. “Random forests”. In: *Machine learning* 45 (2001), pp. 5–32.
- [17] Rémi Bardenet et al. “Collaborative hyperparameter tuning”. In: *International conference on machine learning*. PMLR. 2013, pp. 199–207.
- [18] Dani Yogatama and Gideon Mann. “Efficient transfer learning method for automatic hyperparameter tuning”. In: *Artificial intelligence and statistics*. PMLR. 2014, pp. 1077–1085.
- [19] Valerio Perrone et al. “Scalable hyperparameter transfer learning”. In: *Advances in neural information processing systems* 31 (2018).
- [20] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. “Freeze-thaw Bayesian optimization”. In: *arXiv preprint arXiv:1406.3896* (2014).
- [21] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves”. In: *Twenty-fourth international joint conference on artificial intelligence*. 2015.
- [22] Kevin Jamieson and Ameet Talwalkar. “Non-stochastic Best Arm Identification and Hyperparameter Optimization”. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, Sept. 2016, pp. 240–248. URL: <https://proceedings.mlr.press/v51/jamieson16.html>.
- [23] Liam Li et al. “A system for massively parallel hyperparameter tuning”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 230–246.

- [24] Stefan Falkner, Aaron Klein, and Frank Hutter. “BOHB: Robust and efficient hyperparameter optimization at scale”. In: *International conference on machine learning*. PMLR. 2018, pp. 1437–1446.
- [25] Noor Awad, Neeratyoy Mallik, and Frank Hutter. “Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization”. In: *arXiv preprint arXiv:2105.09821* (2021).
- [26] Aaron Klein et al. “Model-based asynchronous hyperparameter and neural architecture search”. In: *arXiv preprint arXiv:2003.10865* (2020).
- [27] Yang Li et al. “Hyper-tune: Towards efficient hyper-parameter tuning at scale”. In: *arXiv preprint arXiv:2201.06834* (2022).
- [28] Martin Wistuba, Arlind Kadra, and Josif Grabocka. “Supervising the multi-fidelity race of hyperparameter configurations”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 13470–13484.
- [29] Kevin Swersky, Jasper Snoek, and Ryan P Adams. “Multi-task bayesian optimization”. In: *Advances in neural information processing systems* 26 (2013).
- [30] Aaron Klein et al. “Fast bayesian optimization of machine learning hyperparameters on large datasets”. In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 528–536.
- [31] Guanghui Zhu and Ruancheng Zhu. “Accelerating hyperparameter optimization of deep neural network via progressive multi-fidelity evaluation”. In: *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part I* 24. Springer. 2020, pp. 752–763.
- [32] Sungduk Yu et al. “Two-step hyperparameter optimization method: Accelerating hyperparameter search by using a fraction of a training dataset”. In: *Artificial Intelligence for the Earth Systems* 3.1 (2024), e230013.
- [33] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. “coresets for data-efficient training of machine learning models”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 6950–6960.
- [34] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical bayesian optimization of machine learning algorithms”. In: *Advances in neural information processing systems* 25 (2012).
- [35] Xuanyi Dong and Yi Yang. “NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search”. In: *International Conference on Learning Representations (ICLR)*. 2020. url: <https://openreview.net/forum?id=HJxyZkBKDr>.

- [36] Lucas Zimmer, Marius Lindauer, and Frank Hutter. “Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3079–3090.
- [37] Aaron Klein and Frank Hutter. “Tabular benchmarks for joint architecture and hyperparameter optimization”. In: *arXiv preprint arXiv:1905.04970* (2019).
- [38] Nils Strodthoff et al. “Deep learning for ECG analysis: Benchmarks and insights from PTB-XL”. In: *IEEE journal of biomedical and health informatics* 25.5 (2020), pp. 1519–1528.
- [39] David Salinas et al. “Syne Tune: A Library for Large Scale Hyperparameter Tuning and Reproducible Research”. In: *International Conference on Automated Machine Learning, AutoML 2022*. 2022. URL: <https://proceedings.mlr.press/v188/salinas22a.html>.
- [40] Janez Demšar. “Statistical comparisons of classifiers over multiple data sets”. In: *The Journal of Machine learning research* 7 (2006), pp. 1–30.

# Appendix A

## Benchmarks specifications

### A.1 Tabulated

Hyperparameter	Values
Batch size	$\{16, \dots, 512\}$
Learning rate	$[1e-4, 1e-1]$
Momentum	$[0.1, 0.99]$
Weight decay	$[1e-5, 1e-1]$
Layers	$\{1, 2, 3, 4, 5\}$
Max units/layer	$\{64, \dots, 1024\}$
Dropout	$[0.0, 1.0]$

**Table A.2** LCBench hyperparameter values

Hyperparameter	Values
Learning rate	$\{0.0005, 0.001, 0.005, 0.01, 0.05, 0.1\}$
Batch size	$\{8, 16, 32, 64\}$
LR schedule	$\{\text{cosine}, \text{fix}\}$
Activation L1	$\{\text{relu}, \text{tanh}\}$
Activation L2	$\{\text{relu}, \text{tanh}\}$
L1 size	$\{8, 16, 32, 64, 128, 256, 512\}$
L2 size	$\{8, 16, 32, 64, 128, 256, 512\}$
Dropout L1	$\{0.0, 0.3, 0.6\}$
Dropout L2	$\{0.0, 0.3, 0.6\}$

**Table A.3** FCNet hyperparameter values

Benchmark	Epochs	Budget	Avg runtime (min)
fcnet-naval	100	2000	26
fcnet-parkinsons	100	2000	14
fcnet-protein	100	2000	92
fcnet-slice	100	2000	173
lcbench-Fashion-MNIST	50	1000	303
lcbench-airlines	50	1000	487
lcbench-albert	50	1000	593
lcbench-christine	50	1000	651
lcbench-covertype	50	1000	643
lcbench-dionis	50	1000	1423
lcbench-helena	50	1000	104
lcbench-higgs	50	1000	103
lcbench-jannis	50	1000	88
lcbench-volkert	50	1000	85
nas201-ImageNet16-120	200	4000	3788
nas201-cifar10	200	4000	1295
nas201-cifar100	200	4000	1307

**Table A.1** Summary of the tabular benchmarks.

## A.2 Real-world

Benchmark	Epochs	Budget	Avg runtime (min)
cifar10 residual	50	1000	561
cifar10 simple	75	1500	487
ptbxl rnn	50	1000	290
ptbxl xResNet1d	40	800	133
svhn residual	40	800	458
svhn simple	70	1400	1159

**Table A.4** Summary of the real-world benchmarks.

Hyperparameter	Values
Learning rate	[1e-4, 1e-1]
$\eta_{min}$	[0.0, 1.0]
FC neurons	{8, ..., 128}
Channels multiplier	{1, ..., 16}
Conv layers	{1, 2, 3, 4}
Dropout	[0.0, 0.8]
Label smoothing	[0.0, 0.3]
Batch norm	{True, False}

**Table A.5** CIFAR-10 simple CNN HPO optimization search space.

Hyperparameter	Values
Learning rate	[1e-5, 1e-1]
$\eta_{min}$	[0.0, 1.0]
FC neurons	{8, ..., 128}
Channels multiplier	{1, 2, 3, 4}
Depth	{1, ..., 5}
Weight decay	[1e-6, 1e-1]
Label smoothing	[0.0, 0.4]

**Table A.6** CIFAR-10 residual HPO optimization search space.

Hyperparameter	Values
Learning rate	[ $1e-4$ , $1e-1$ ]
$\eta_{min}$	[0.0, 1.0]
FC neurons	{8, ..., 128}
Channels multiplier	{1, ..., 16}
Conv layers	{1, 2, 3, 4}
Dropout	[0.0, 0.8]
Label smoothing	[0.0, 0.3]
Batch norm	{True, False}

**Table A.7** SVHN simple CNN HPO optimization search space.

Hyperparameter	Values
Learning rate	[ $1e-5$ , $1e-1$ ]
$\eta_{min}$	[0.0, 1.0]
Momentum	[0.0, 0.99]
FC neurons	{8, ..., 128}
Depth	{1, ..., 5}
Channels multiplier	{1, 2, 3, 4}
Weight decay	[ $1e-6$ , $1e-1$ ]
Label smoothing	[0.0, 0.4]
Rotation	{0, ..., 30}
Translation factor	[0.0, 0.3]
Scale factor	[0.0, 0.3]
Scale offset	[0.0, 0.2]
Sharpness factor	[0.0, 2.0]

**Table A.8** SVHN residual CNN HPO optimization search space.

Hyperparameter	Values
Optimizer	{AdamW, RMSprop}
Learning rate	[1e-4, 1e-1]
Weight decay	[0.0, 0.2]
Decay	{cosine, ReduceLROnPlateau}
$\eta_{min}$	[1e-5, 0.99]
RNN type	{LSTM, GRU}
Bidirectional	{True, False}
RNN hidden	{64, ..., 512}
RNN layers	{1, 2, 3}
RNN dropout	[0.0, 0.6]
Dropout	[0.0, 0.6]

**Table A.9** RNN PTB-XL.

Hyperparameter	Values
Model size	{xresnet1d18, xresnet1d50, xresnet1d101}
Learning rate	[1e-4, 1e-1]
Weight decay	[0.0, 0.2]
$\eta_{min}$	[1e-5, 0.99]
Model dropout	[0.0, 0.6]
FC dropout	[0.0, 0.6]
Original $f$ number	{True, False}

**Table A.10** xResnet1d PTB-XL.

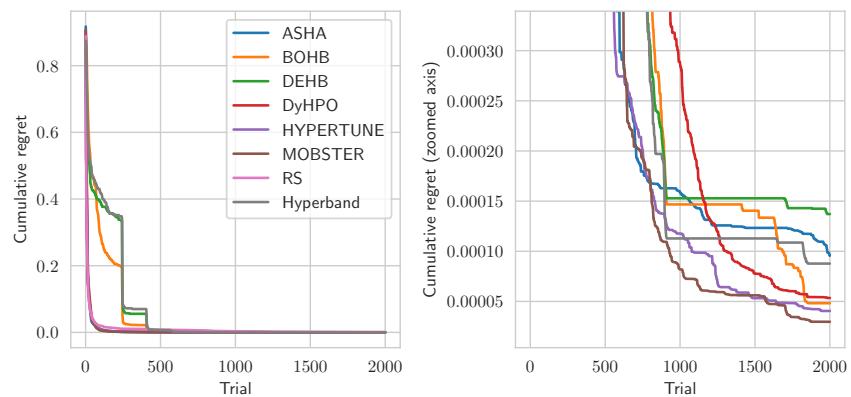
Hyperparameter	Values
Learning rate	[1e-5, 1e-1]
$\eta_{min}$	[1e-5, 0.99]
Weight decay	[1e-6, 1e-1]
Rotation	{0, ..., 30}
Translation factor	[0.0, 0.3]
Resize crop factor	[0.7, 1.0]

**Table A.11** ChestX DenseNet.

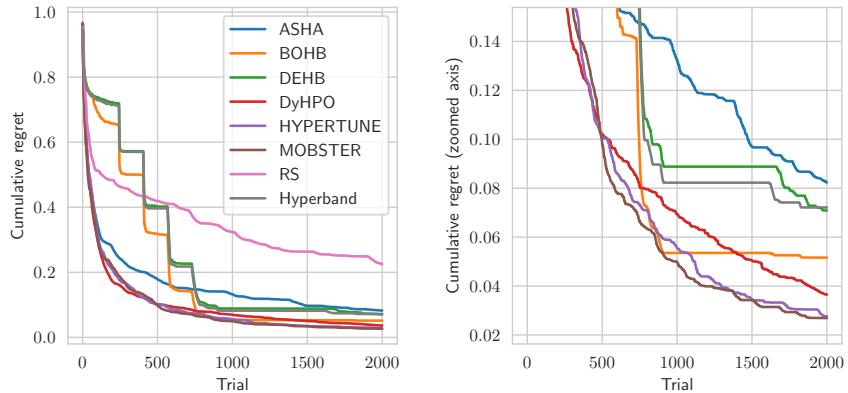
## Appendix B

## Additional figures

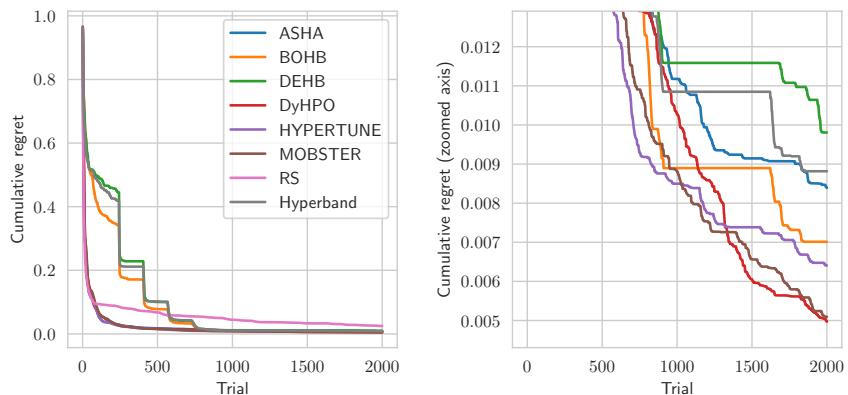
### B.1 Tabulated



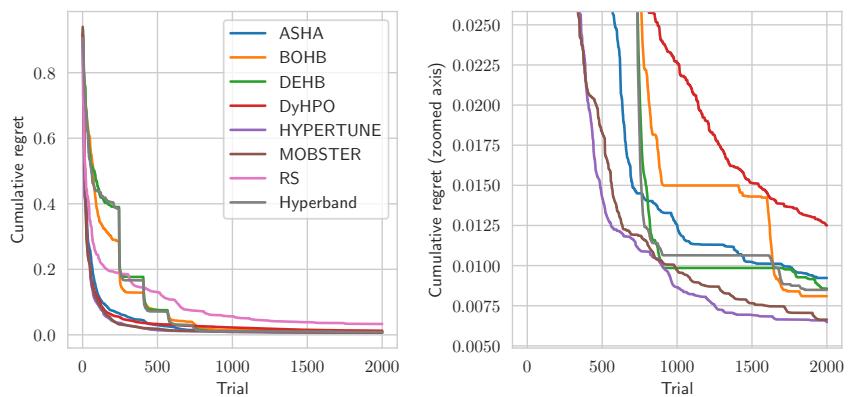
**Figure B.1** FCNet-naval



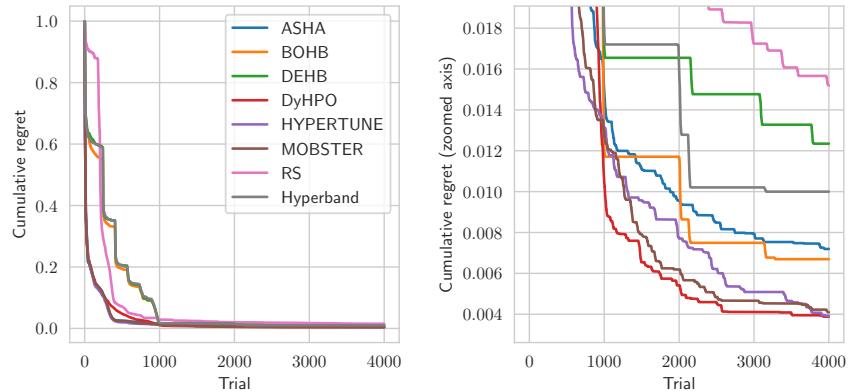
**Figure B.2** FCNet-protein



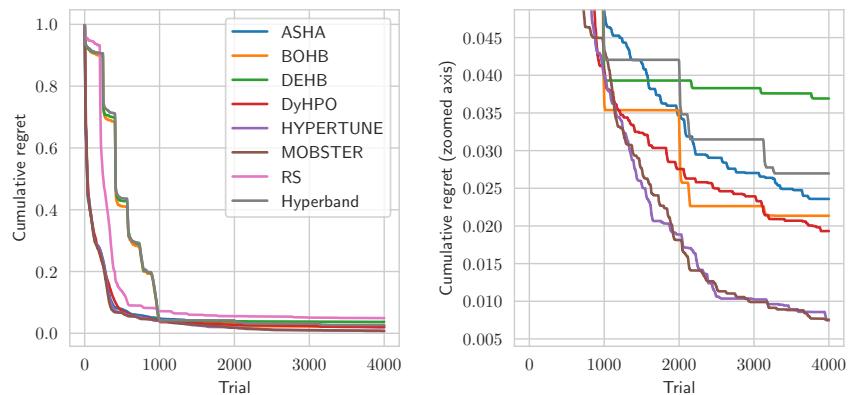
**Figure B.3** FCNet-parkinsons



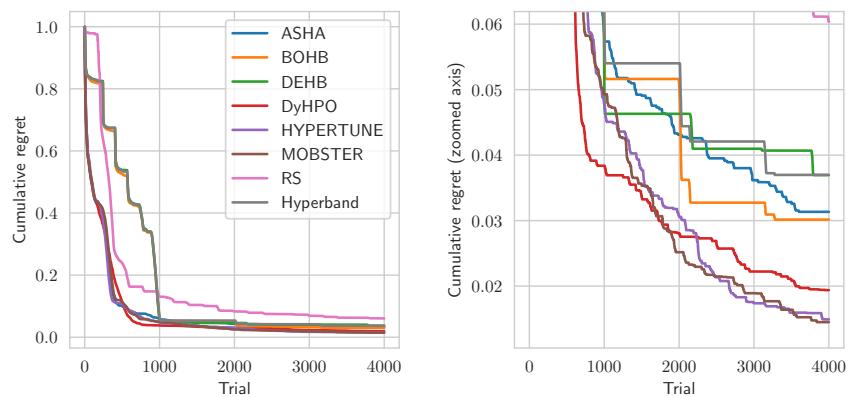
**Figure B.4** FCNet-slice



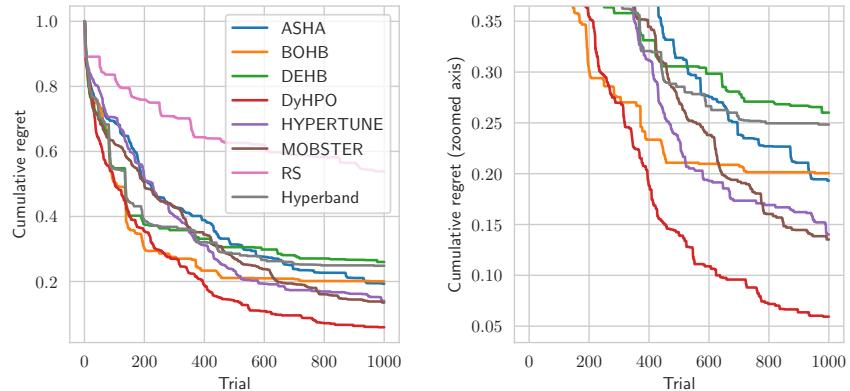
**Figure B.5** nas201-cifar10



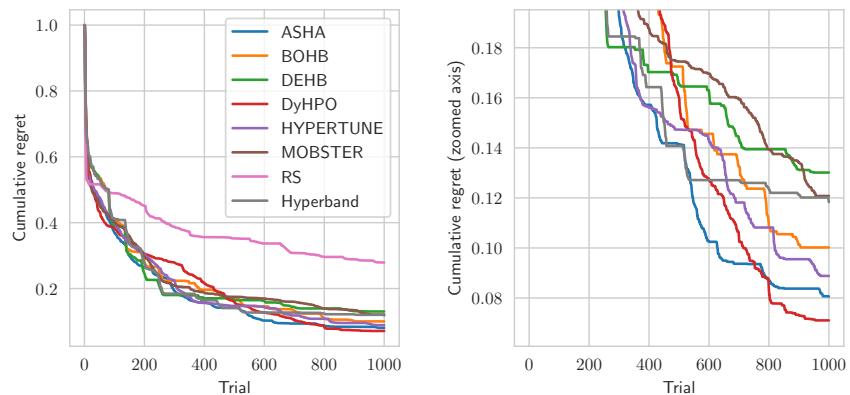
**Figure B.6** nas201-cifar100



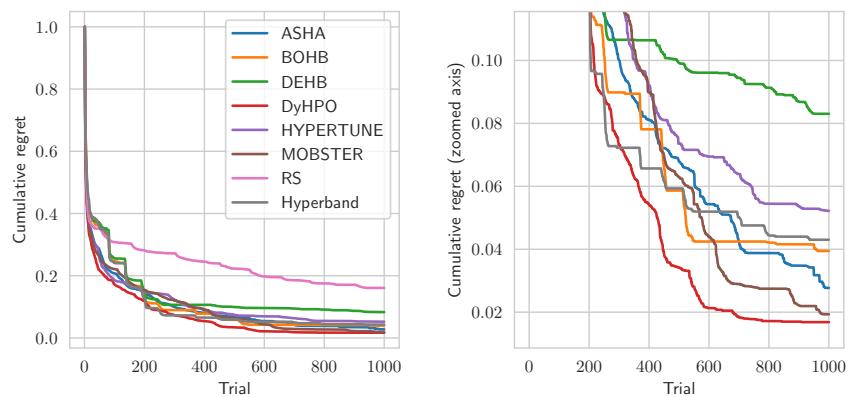
**Figure B.7** nas201-ImageNet16-120



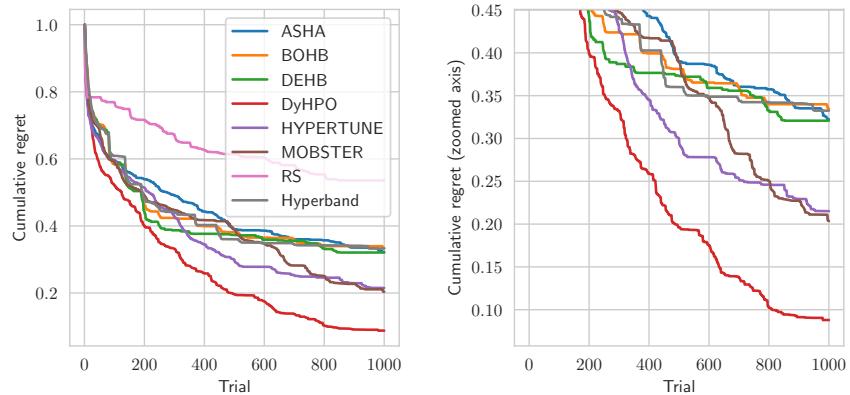
**Figure B.8** LC-Bench-airlines



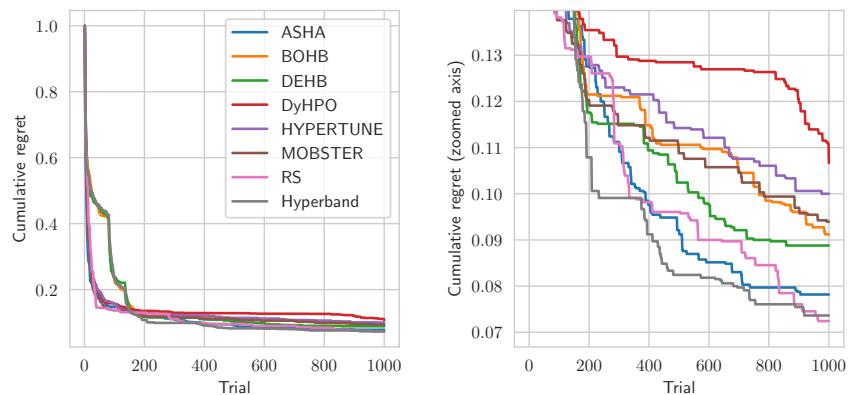
**Figure B.9** LC-Bench-albert



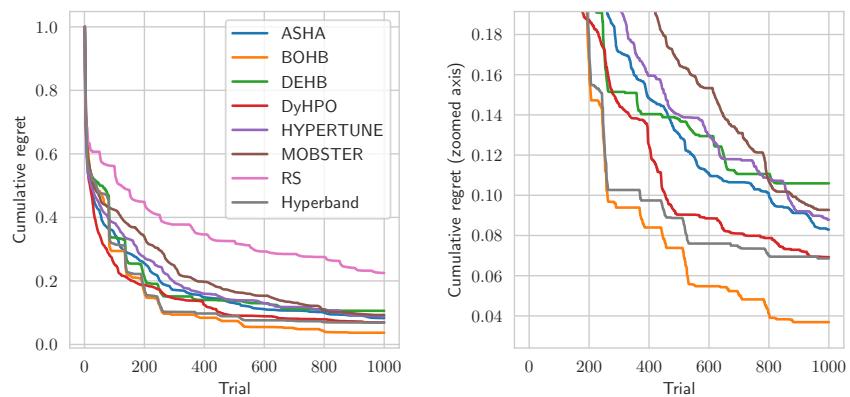
**Figure B.10** LC-Bench-Fashion-MNIST



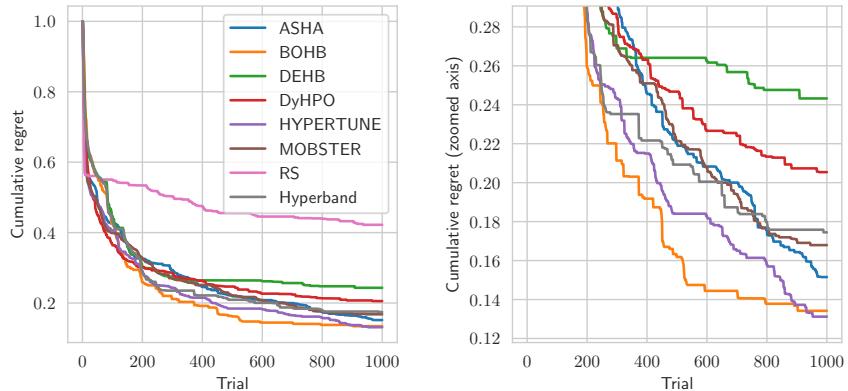
**Figure B.11** LCbench-covertype



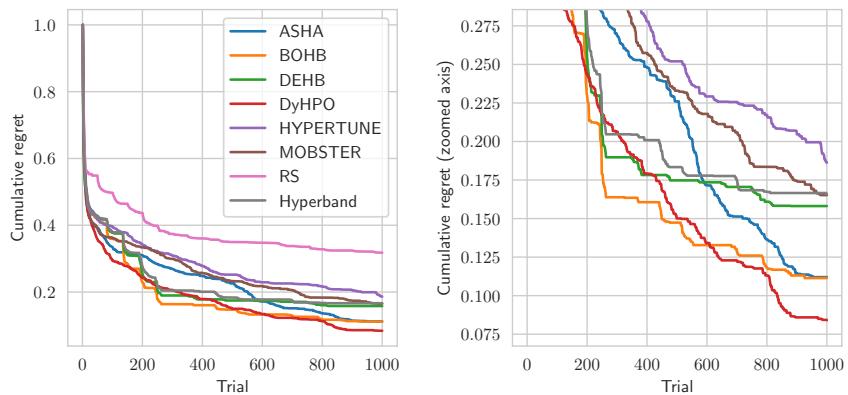
**Figure B.12** LCbench-christine



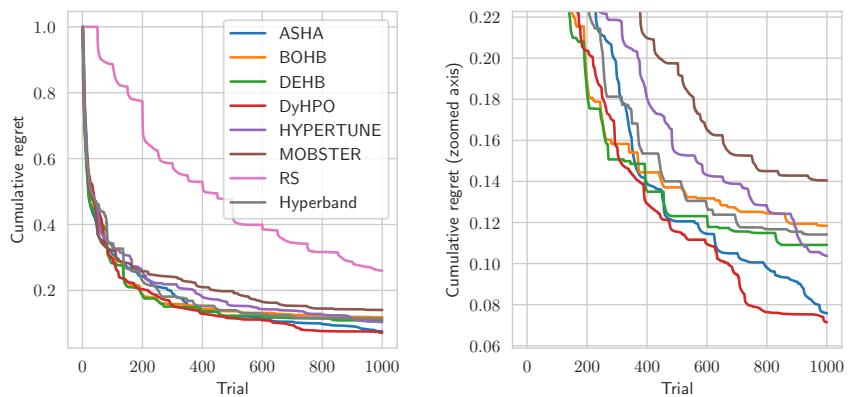
**Figure B.13** LCbench-higgs



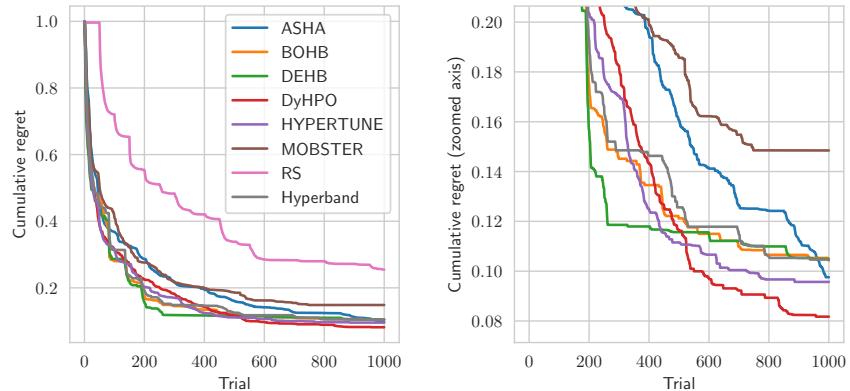
**Figure B.14** LCbench-jannis



**Figure B.15** LCbench-volkert

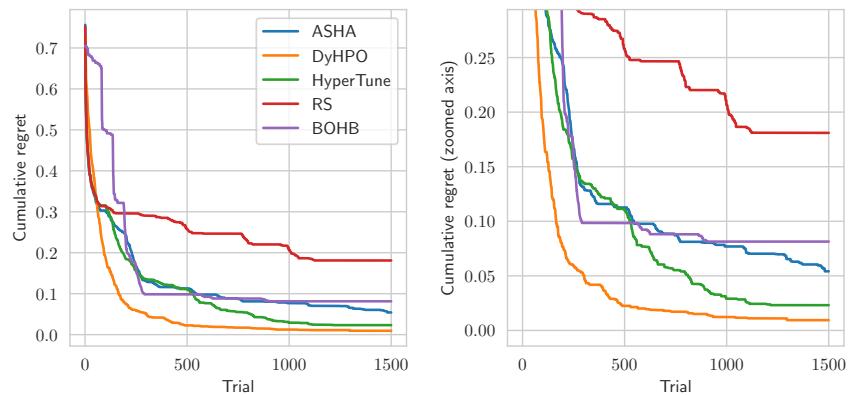


**Figure B.16** LCbench-dionis

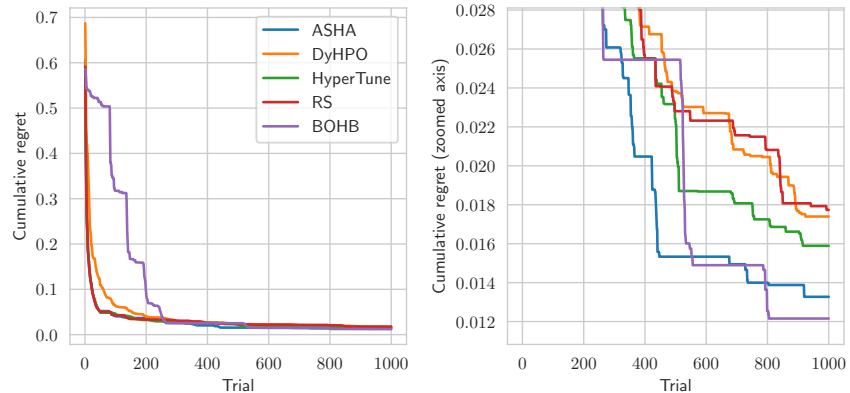


**Figure B.17** LCbench-helena

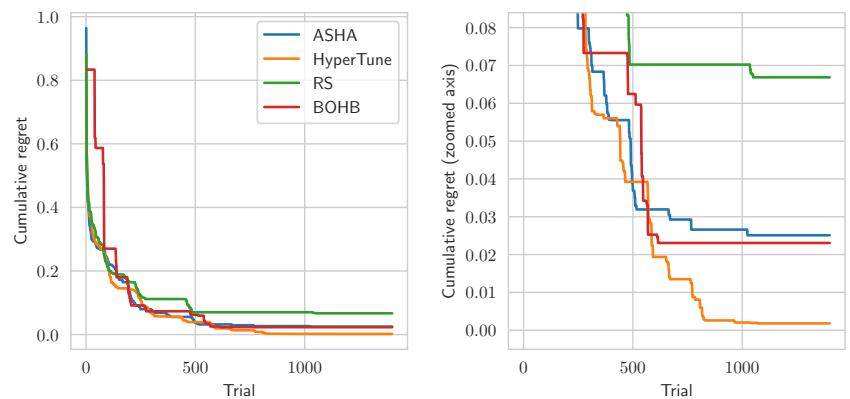
## B.2 Real-world



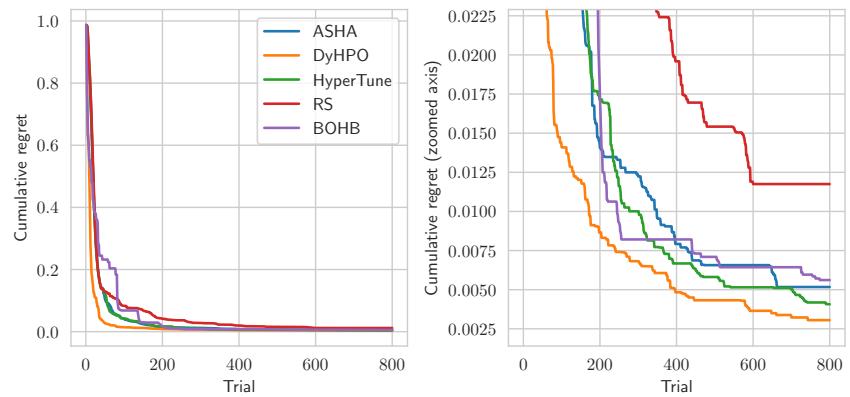
**Figure B.18** CNN network trained on CIFAR10 dataset for 20 full evaluations.



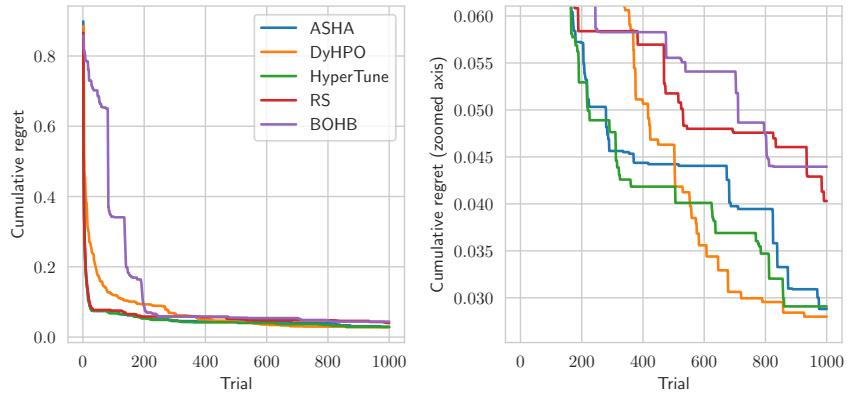
**Figure B.19** Resnet-like network trained on CIFAR10 dataset for 20 full evaluations.



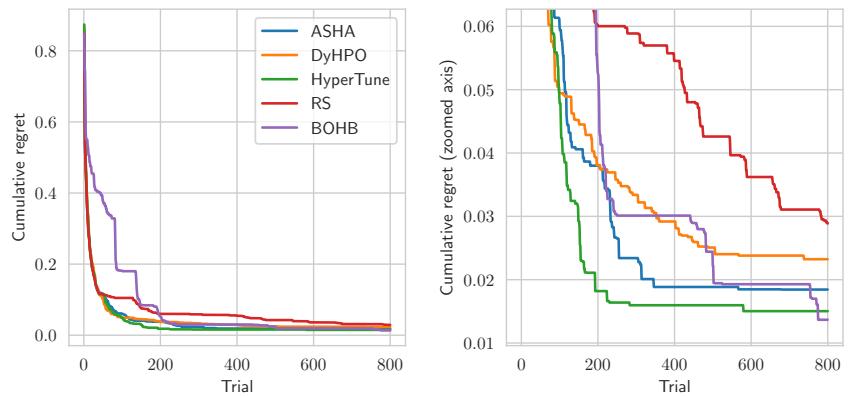
**Figure B.20** CNN network trained on SVHN dataset for 20 full evaluations.



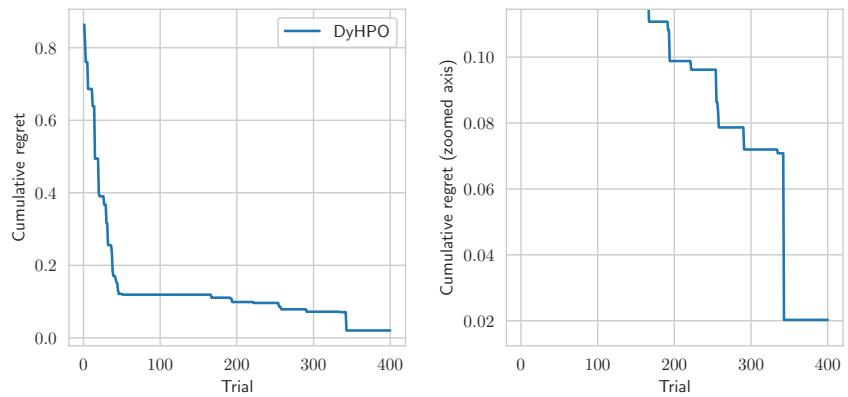
**Figure B.21** Resnet-like network trained on SVHN dataset for 20 full evaluations.



**Figure B.22** RNN network trained on the PTB-XL dataset for 20 full evaluations.



**Figure B.23** xResnet1d network trained on the PTB-XL dataset for 20 full evaluations.



**Figure B.24** DenseNet trained on the ChestX dataset for 20 full evaluations.