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

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

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 and Literature Review

1.1 Machine learning fundamentals

Our goal is to optimize hyperparameters. Before we introduce the main topic, we need to specify the context. We are interested in supervised machine learning methods. 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 what we want to predict, 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.

We train machine learning algorithms by minimizing the loss function on the training data. Two loss functions are most commonly used. 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

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

For neural networks, 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 model parameters, or weights, the gradient descent optimizes the function by repeatedly calculating the gradient $\nabla_\theta L(\theta)$ and performing an update to the weights in the opposite direction:

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

Since the goal is for the model to generalize well, we need to estimate the generalization error during training. That is why we estimate it on data not used during training. We also want to verify the performance of the final model after we choose one. Not to introduce a bias to the estimate, this should also be done on unseen data. That is why it is a best practice to split the dataset into $\mathscr{D}_{train}$, $\mathscr{D}_{val}$, $\mathscr{D}_{test}$. 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.

A neural network describes a computation. The basic unit is called a node and the nodes are arranged in an acyclic graph. Training of the network can be split into two parts — the forward pass and the backward pass. In the forward pass, the network produces an output from the input examples by iteratively going through the graph in a topological order. Then the loss is calculated. The gradients 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.

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 $\boldsymbol{\Lambda} = \Lambda_1 \times \dots \times \Lambda_n$ denote its hyperparameter space. Hyperparameters can be continuous, integer-valued, or categorical. Also, we can have conditional hyperparameters. We say that hyperparameter λ_i is *conditional* on another hyperparameter λ_j, if λ_i is only active if hyperparameter λ_j takes values from a given set $V_i(j) \subset \Lambda_j$

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

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

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

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.

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 [3]. 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

Before we get to algorithmic approaches, let us consider 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 automated 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 power on hyperparameter combinations, where only the unimportant hyperparameter changes, but the important ones stay the same. But how do we determine which hyperparameters are important and we need to tune them together? 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. [4] 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. [5] show, 2X-budget random search provides a strong baseline.

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. [6] and Frazier [7] 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

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

The foundations were laid for Bayesian Optimization by Jonas Mockus [8] 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 [9]. 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.

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 $f(\lambda) = -\mathscr{L}(\lambda)$. Maximizing this function is equivalent to minimizing the original function. We

Algorithm 1 Bayesian Optimization

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

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 $\{x \in \mathbb{R} \mid a_i \leq x_i \leq b_i\}$, 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 $y \sim \mathcal{N}(\mu(x), \sigma^2(x))$.

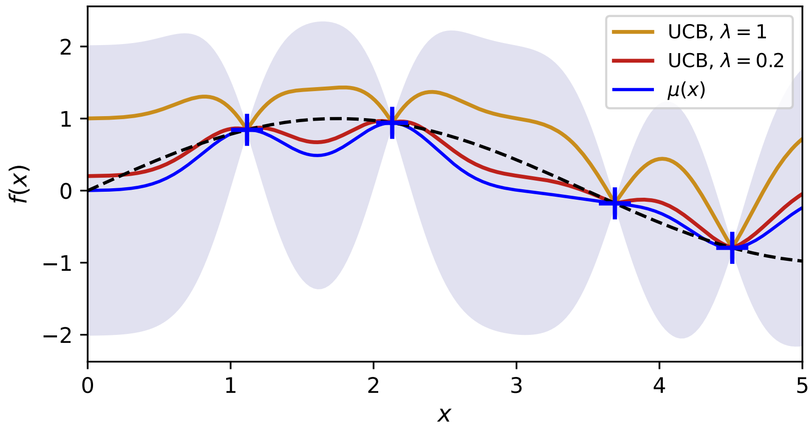


Figure 1.1 An UCB acquisition function with two λ settings demonstrated on a Bayesian model predicting mean $\mu(x)$, and variance (the light blue corresponds to two standard deviations from the mean), fitted to the black true function with four noisy observations.

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

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

where λ is an exploration parameter. The smaller the λ, the more the UCB exploits regions that are known to yield good solutions and vice versa. The UCB acquisition function is illustrated in Figure 1.1.

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

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

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 Φ 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 ξ, which forces PI to consider only points that are by at least ξ better than the incumbent:

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

Expected improvement

Expected improvement (EI) [10] 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 [10] 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 ϕ is the density of the normal distribution.

Even the expected improvement can be extended with parameter ξ 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 1.2.

Include other acquisitions if needed, like entropy search, or knowledge gradient

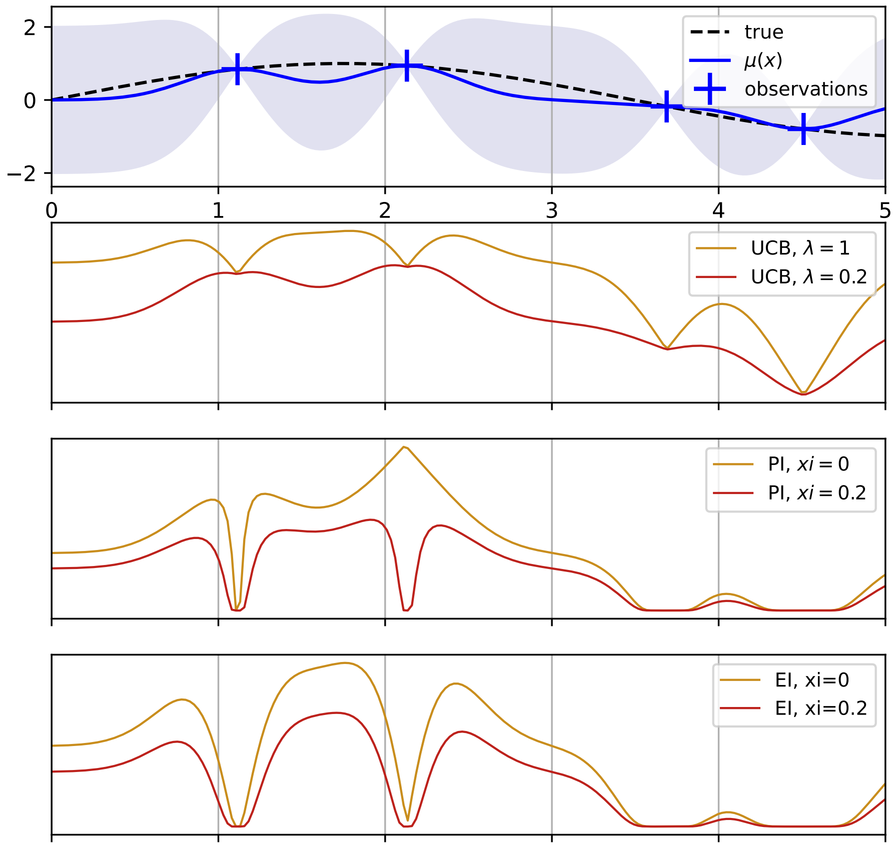


Figure 1.2 A comparison of different acquisition functions. The top image shows the real target function with a Bayesian model predicting mean and variance (the light blue area corresponds to two standard deviations). The other images show different acquisition functions from top to bottom: upper confidence 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 [11], and we use similar illustrations to present the topic. The definitions and the notation are taken from the textbook by Rasmussen [9]. 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

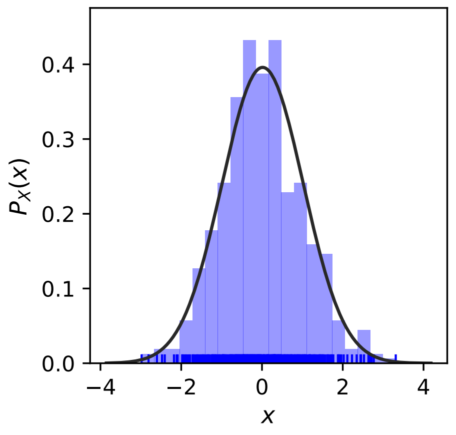
In order to describe the Gaussian process regression, we will start with the basics. A random variable X is Gaussian or normally distributed with mean μ and variance σ^2 if its probability density function is:

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

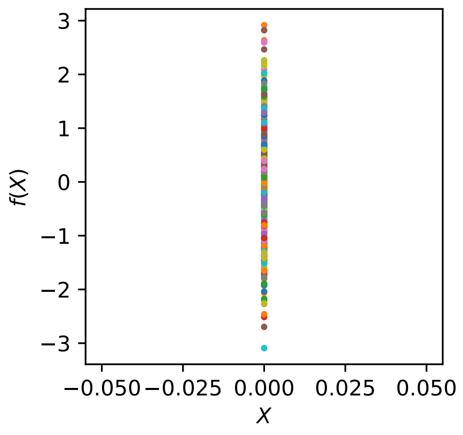
We denote that a random variable is normally distributed as $X \sim \mathcal{N}(\mu, \sigma^2)$. For illustration, we have sampled 500 points randomly from a univariate normal distribution into a vector $x_1 = [x_1^1, x_1^2, \dots, x_1^{500}]$. In Figure 1.3 we plot a histogram of the points and fit 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 1.4. 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

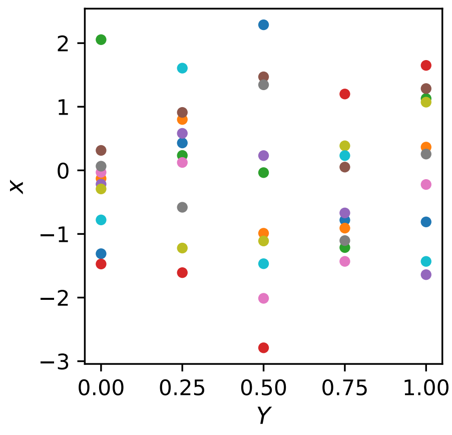


(a) Histogram of the samples with the fitted density as a black curve.

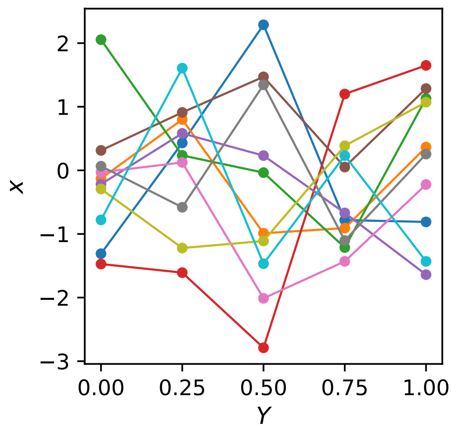


(b) Random samples plotted vertically along the Y-axis with a fixed X coordinate.

Figure 1.3 A visualization of sampling from a Gaussian distribution. A random variable $X \sim \mathcal{N}(0, 1)$ is sampled 500 times.



(a) Random samples



(b) Random samples connected by linear lines

Figure 1.4 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

$\boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}$ 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 $\Sigma_{ij} = \mathrm{cov}[X_i, X_j]$. The $\boldsymbol{\Sigma}$ 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

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

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 σ 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 1.5. Note that the samples illustrated in Figure 1.4 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.

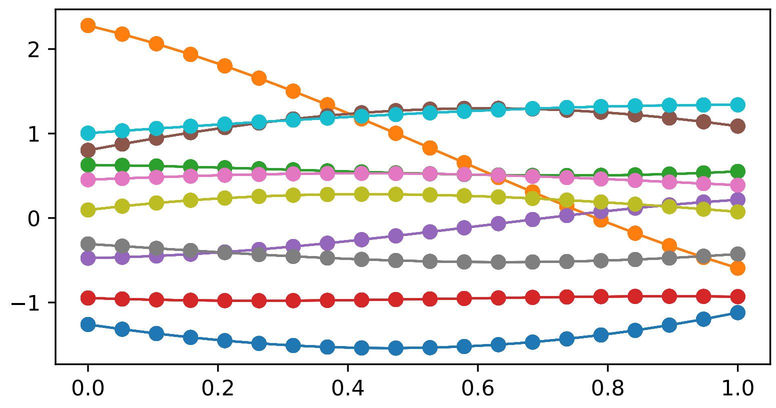


Figure 1.5 Samples from the 20-VN distribution with RBF covariance.

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 1.6, 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, $\boldsymbol{\mu} = [m(\mathbf{x_1}), \dots, m(\mathbf{x_n})]$ 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-

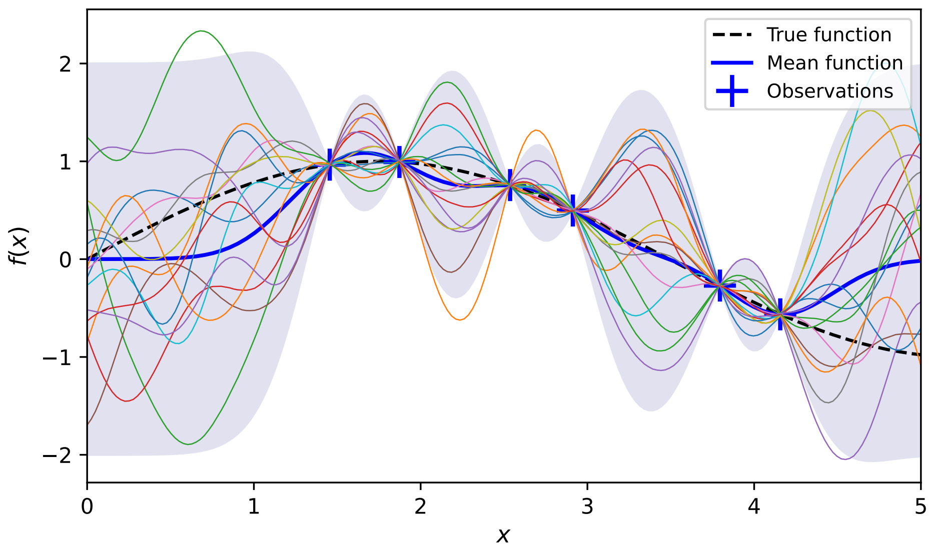


Figure 1.6 An example of Gaussian process regression. The black dashed line is the true target function. We drew six random samples and the function was evaluated at these points with noise (blue crosses). Then we sampled 15 functions from the posterior distribution given the test points. The blue line is the mean function and the light blue area is bounded at two standard deviations from the mean.

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

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

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 ϵ with variance σ_n^2. The noise is incorporated into the prior as $\operatorname{cov}(y) = \mathbf{K} + \sigma_n^2 \mathbf{I}$. 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

$$\bar{\mathbf{f}}_* \mid \mathbf{y}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}\big(\bar{\mathbf{f}}_*, \operatorname{cov}(\mathbf{f}_*)\big)\ ,$$

where

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

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 [7].

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. [12] 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 [12], we want to minimize the objective function $f(x)$ and the observations $\mathscr{D}$ are split into the better (lower) group $\mathscr{D}^{(l)}$ and the worse (greater) group $\mathscr{D}^{(g)}$ based on their objective value; for illustration see the top left part of Figure 1.7. More precisely, the top-quantile γ is computed in each iteration based on the number of observations $N = |\mathscr{D}|$ and y^γ is the top-γ-quantile objective value in the set of observations $\mathscr{D}$. Observations with $y \leq y^\gamma$ are assigned into the $\mathscr{D}^{(l)}$, and observations with $y > y^\gamma$ to the $\mathscr{D}^{(g)}$. The subsets are used to model $p(\mathbf{x} \mid y, \mathscr{D})$ with the assumption:

$$(\mathbf{x} \mid y, \mathscr{D}) := \begin{cases} p(\mathbf{x} \mid \mathscr{D}^{(l)}) & (y \leq y^\gamma) \\ p(\mathbf{x} \mid \mathscr{D}^{(g)}) & (y > y^\gamma) \end{cases}\,. \tag{1.1}$$

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 $y_1 \leq y_2 \leq \cdots \leq y_N$. Then the KDEs are estimated as

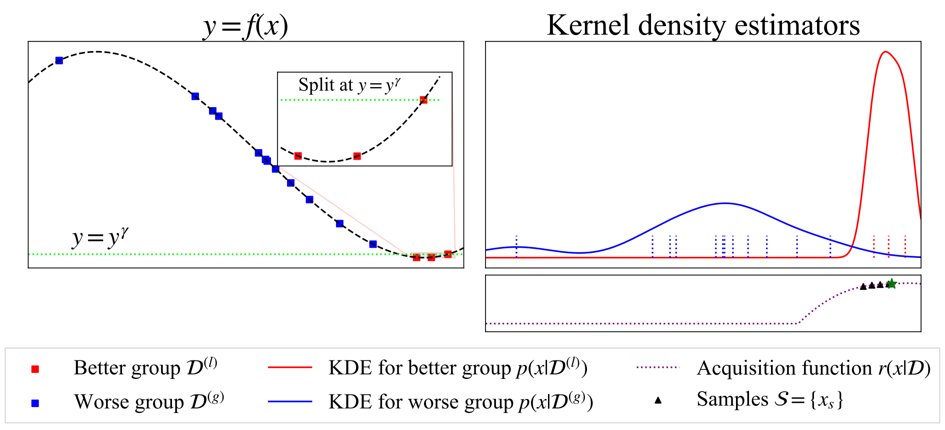


Figure 1.7 Example of the TPE algorithm. We are searching for the minimum of the target (black) function. The observations are split into $D^{(l)}$ and $D^{(g)}$ by the green line on the objective value of observations. For each group, a kernel density estimation is calculated. The acquisition function is derived from the densities and the algorithm chooses the next point to sample by finding the maximum (Source: Figure 1 from Watanabe [13]).

$$
\begin{aligned}
p(\mathbf{x} \mid \mathscr{D}^{(l)}) &= w_0^{(l)} p_0(\mathbf{x}) + \sum_{n=1}^{N^{(l)}} w_n k(\mathbf{x}, \mathbf{x}_n \mid b^{(l)}), \\
p(\mathbf{x} \mid \mathscr{D}^{(g)}) &= w_0^{(g)} p_0(\mathbf{x}) + \sum_{n=N^{(l)}+1}^{N} w_n k(\mathbf{x}, \mathbf{x}_n \mid b^{(g)}),
\end{aligned} \tag{1.2}
$$

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 1.7.

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. 1.1 as

$$
\mathbb{P}(y \leq y^{\gamma} \mid \mathbf{x}, \mathscr{D}) \overset{\text{rank}}{\simeq} r(\mathbf{x} \mid \mathscr{D}) := \frac{p(\mathbf{x} \mid \mathscr{D}^{(l)})}{p(\mathbf{x} \mid \mathscr{D}^{(g)})},
$$

where the $\overset{\text{rank}}{\simeq}$ symbol means the order isomorphic between the left-hand side and the right-hand side. See Figure 1.7 for illustration. The detailed pseudocode is provided in Algorithm 2.

Algorithm 2 Tree-structured Parzen estimator (TPE) (Source: Watanabe [13])

```
    $N_{\text{init}}$ (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 $\boldsymbol{x}_n$
 4:     $y_n := f(\boldsymbol{x}_n) + \epsilon_n$ ........ Evaluate the (expensive) objective function
 5:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_n, y_n)\}$
 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 $p(\boldsymbol{x} \mid \mathcal{D}^{(l)}), p(\boldsymbol{x} \mid \mathcal{D}^{(g)})$ based on Eq. (1.2) Use $\{w_n\}_{n=0}^{N+1}$ and $b^{(l)}, b^{(g)}$
12:     Sample $\mathcal{S} := \{\boldsymbol{x}_s\}_{s=1}^{N_s} \sim p(\boldsymbol{x} \mid \mathcal{D}^{(l)})$
13:     Pick $\boldsymbol{x}_{N+1} := \boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in \mathcal{S}} r(\boldsymbol{x} \mid \mathcal{D})$ The evaluations by the acquisition
    function
14:     $y_{N+1} := f(\boldsymbol{x}_{N+1}) + \epsilon_{N+1}$ ..... Evaluate the (expensive) objective function
15:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{\boldsymbol{x}_{N+1}, y_{N+1}\}$
```

Components and control parameters

In this section, we will look at how the different components and control parameters change the behavior of the algorithm. We start with the *splitting algorithm*, which is used to split the observations $\mathcal{D}$ into $\mathcal{D}^{(l)}$ and $\mathcal{D}^{(g)}$. Weighting Algorithm. Kernel Functions (numerical, categorical, univariate vs multivariate). Bandwidth. Non-informative prior. Will I have to go into such a details?

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. [14] 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 [15] 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, where Gaussian processes usually struggle (add citation). 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.

1.5 Multifidelity techniques

Even though Bayesian optimization techniques are very sample-efficient, our budget might not allow for enough full training runs to find a good hyperparameter configuration. One option for dealing with such strict budget constraints is the multifidelity approach. A new hyperparameter λ_{fid} is introduced that allows us to trade off runtime for the reliability of the information we gain from evaluation. Low fidelity values mean that the evaluations are cheap but unreliable. Fidelity close to the upper limit implies that the returned values are close to true objective values achievable with a full budget. What we are most interested in is finding λ_{fid} so that we use as few resources as possible and the rank correlation is mostly preserved between the models trained with low fidelity and the fully trained models. In other words, we want the low-fidelity models to reliably predict the ranking of the fully-trained models. In general, multifidelity HPO algorithms will start by evaluating cheap HPCs with low λ_{fid} values and then using the remaining budget to exploit the gathered data and train the most promising models with high λ_{fid}.

1.5.1 Early stopping

The simplest way to reduce the runtime of an algorithm is to stop the computation before it finishes. Swersky et al. [16] 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. [16] 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. [17], 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.

Successive halving

Hyperband

The Hyperband algorithm, developed by Li et al. [5], offers a completely different approach to dynamic resource allocation. It is a general approach with only a few assumptions. Authors originally developed it as an extension to speed up random search through early stopping.

The Hyperband internally uses the Successive Halving algorithm. Successive Halving works by uniformly allocating a budget to a set of hyperparameter configurations, then it throws out the worst half, and this process repeats until one configuration remains. The algorithm allocates exponentially more resources to more promising configurations, which usually speeds up the search considerably.

In theory, 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 the number of configurations n in the first round manually. Therefore, we have to choose between large n, which means training a lot of hyperparameter configurations with a smaller training time and small n, resulting in the exploration of fewer hyperparameter configurations that are trained for longer on average.

The optimal choice of n depends mainly on how hard is it to distinguish similarly performing hyperparameter configurations from each other. We know that the intermediate losses are noisy, so we have to account for the uncertainty, which the authors of the paper call the envelope, as well. Ideally, we would wait until the envelopes do not overlap. We can observe that more resources are needed if the envelopes are wider, 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. The more configurations we want to evaluate, the less budget is allocated to the best configurations. Therefore, there might not be enough time for them to converge, and we might select a worse hyperparameter configuration as a result.

The Hyperband addresses the problem of selecting the optimal value of n by considering several possible values of n for a fixed budget B. Specifically, a successive halving algorithm is repeatedly called with different values of n. A larger value of n corresponds to more aggressive early-stopping since the same amount of computational resources is distributed between more hyperparameter configurations. Furthermore, by resetting the search, Hyperband is hedging against bad instantiations of the randomly sampled configurations and their initialization.

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. [18] noticed its deficiency and proposed a new algorithm BOHB to fix this. They combine Hyperband with Bayesian optimization in a way that their weaknesses are compensated by the other algorithm. Bayesian optimization needs some initial trials to gather enough data for the internal models, so it performs like a random search for a while. This is where the strong low-budget performance of Hyperband is used. On the other hand, well-fitted Bayesian optimization models provide better suggestions later in the tuning process.

Another algorithm extending the Hyperband is the DEHB developed by Awad et al. [19], which uses an evolutionary optimization method instead of Bayesian optimization. More specifically, the DE stands for Differential Evolution. 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.

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. The DEHB is much more efficient on some benchmarks while performing similarly to the BOHB, the next-best HPO optimizer in the experiments. The DEHB has very strong performance early with a low budget, and the performance does not fall off even for large budgets, which it often does for the BOHB.

1.5.2 Subsampling

The second possibility is to start the exploration with a fraction of the training data. The intuition behind this approach is that even a small subset of the training data will contain most of the information about the structure of the dataset for the model to learn. That should be enough to approximate the performance on the full dataset while training faster.

Klein et al. [20] developed a Bayesian optimization algorithm FABOLAS. 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 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 the 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 [21]. They combine successive halving with progressively increasing 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 the successive halving, but the authors do not mention which surrogate model they 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. 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 will fewer resources used.

A method called DyHPO that is based on a novel Gaussian process kernel was developed by Wistuba et al. [22] They reviewed existing multifidielity approaches, including the Hyperband, BOHB, and DEHB. They have stated a conjecture that these gray-box methods suffer from a major issue — low-budget performances are not always a good indicator for the full budget performances. 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.

Another area where DyHPO should gain efficiency compared to approaches like Hyperband is that instead of pre-allocating the budget, DyHPO dynamically adapts the allocation of budgets after every HPO step. Therefore, DyHPO invests only a small budget on unpromising configurations. In the provided experiments, DyHPO showed statistically significant efficiency gains compared to other hyperparameter optimization methods such as DEHB, BOHB, and Hyperband.

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. [23] 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.

1.6 Benchmarks

Probably just include the comparisons in the text and delete this section. In this section, we review the literature comparing and benchmarking the approaches, which should further clarify which algorithm and when to use.

Li et al. [5] compare the Hyperband algorithm with three Bayesian optimization algorithms (SMAC, TPE, Spearmint) on CIFAR-10, rotated MNIST and SVHN datasets. They also include random search and 2x-random search as a baseline. The 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.

FABOLAS [20] - FABOLAS>MTBO>Hyperband.

Chapter 2

Methodology

2.1 Algorithms

2.2 Datasets and models

2.3 Evaluation Metric

2.4 Experimental Setup

Chapter 3

Results and discussion

3.1 Tabulated benchmarks

Should I include tabulated benchmarks? I think it is a good starting point for performance evaluation of HPO algorithms, most of the literature uses them as a standardized way of comparison. I would briefly describe the benchmark and include one or two plots where I could analyze how the algorithms behave, the results don't seem to have an obvious conclusion.

The text is written as notes for myself at this point.

Benchmark	Epochs	1x full eval (s)	Max t (s)	Full evals
lc-Fashion-MNIST	50	1200	7200	6
lc-airlines	50	1108	7200	6.5
lc-albert	50	934	7200	7.7
lc-covertype	50	650	7200	11
lc-christine	50	2376	7200	3
nas-cifar100	200	3649	21600	5.9
nas-cifar10	200	3649	18000	4.9
nas-ImageNet	200	10450	28800	2.7
fc-protein	100	254	3600	14.1
fc-naval	100	68	3600	53
fc-parkinsons	100	33.9	3600	109
fc-slice	100	354	3600	10

Table 3.1 Summary of the tabular benchmarks.

In this table, we compare some basic attributes of the benchmarks. How many epochs is allocated to fully train a model, how much time it takes to fully train a model, how much time the benchmark runs for and how many full evaluations is it possible to do in that time (with random search). For model-based methods, training of the HPO model takes from the total time

3.1.1 NAS-201

NAS-Bench-201 contains 15625 multi-fidelity configurations of computer vision architectures evaluated on 3 datasets. The search space is defined by cell-based structure – the cell operations are optimized and the cell is then used as a building block for the network. A cell is a DAG with N nodes. There are 7 nodes in each cell, including an input and an output node. 5 predefined operations (3x3 Conv, 1x1 Conv, AvgPool, MaxPool, skip connection). The total number of possible architectures in the search space is $5^6 = 15625$ because each edge can be one of 5 operations and there are 6 edges in the DAG.

Because we are only choosing the connections between the intermediate nodes out of the 5 possible values, The search space is a composition of categorical variables, which means that there is not much structure to exploit by Gaussian Processes. On the other hand, the early stopping could be decisive.

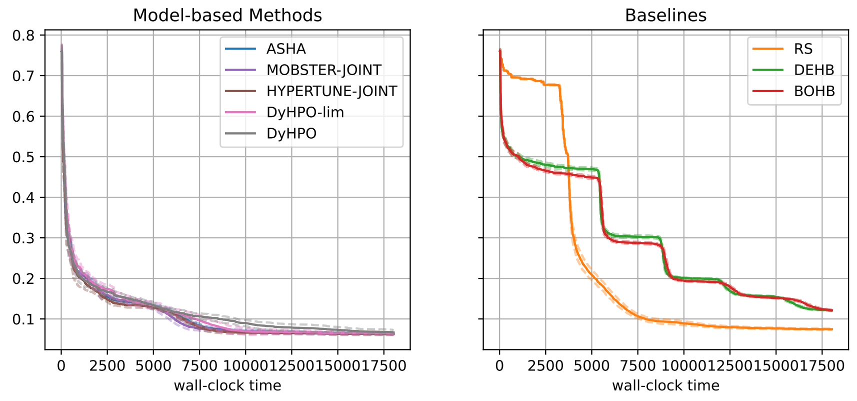


Figure 3.1 nas201-cifar10

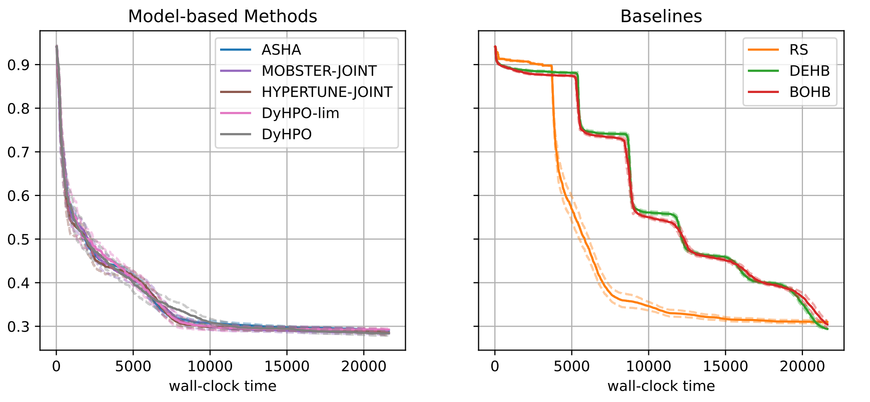


Figure 3.2 nas201-cifar100

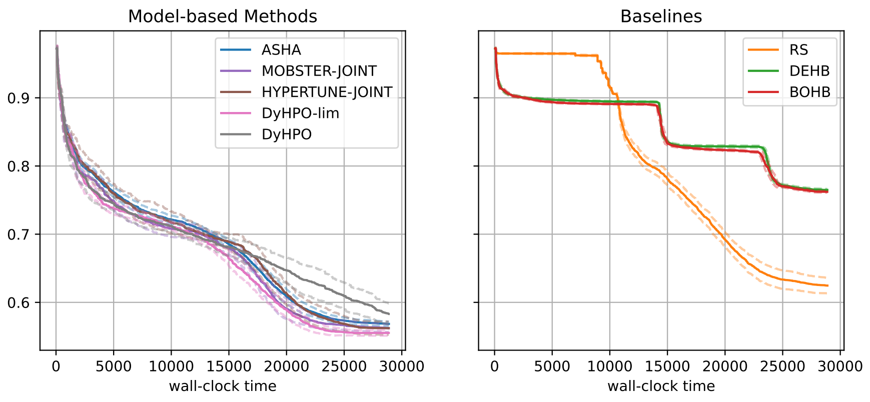


Figure 3.3 nas201-ImageNet16-120

The default settings of NAS-Bench are very constrained. For CIFAR100, the wall-time limit of 21600 seconds (5 hours) is enough for 4-6 full evaluations depending on the HPO algorithm. The Random search is initially flat, because the first combination is always sampled by the midpoint rule, and therefore is deterministic even though we have just categorical variables and midpoint rule doesn't make sense?. The cliffs for BOHB and DEHB mark an increase of the training budget. For the first 5000 seconds, they are allowed to train the network for one epoch only, then it increases to 3, 9, 81, and finally to 200, which is the maximum. Therefore, the comparison is not fair for this scheduling and they should be compared only at one chosen budget, for which the brackets parameter is well chosen.

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

3.1.2 LCBench

LCBench is a benchmark suite for studying the performance of Neural Architecture Search algorithms. The LC in LCBench stands for learning curve, LCBench tracks the performance of architectures throughout the search process. It contains 16 datasets from various domains. "lcbench": 2000 multi-fidelity Pytorch model configurations evaluated on many datasets.

LCBench provides training data for 2000 different configurations across different architectures and hyperparameters, each evaluated on 35 datasets over 50 epochs. Training, test, and validation loss are tracked, as well as accuracies. The following hyperparameters (4 float, 3 integer) were optimized:

All runs feature funnel-shaped MLP nets and use SGD with cosine

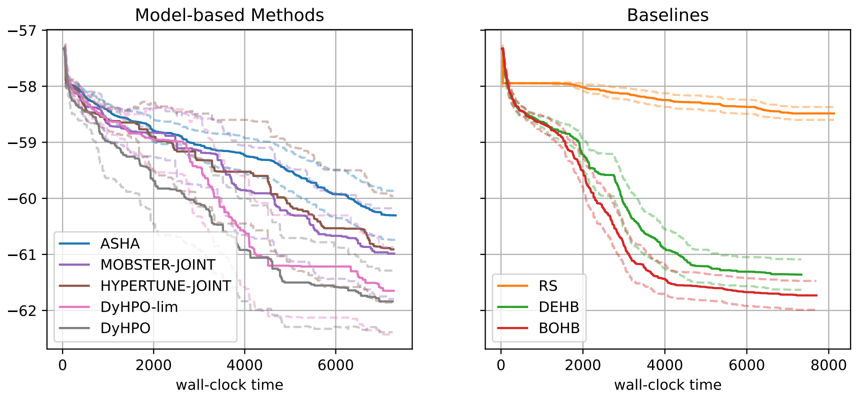


Figure 3.4 LCBench-airlines

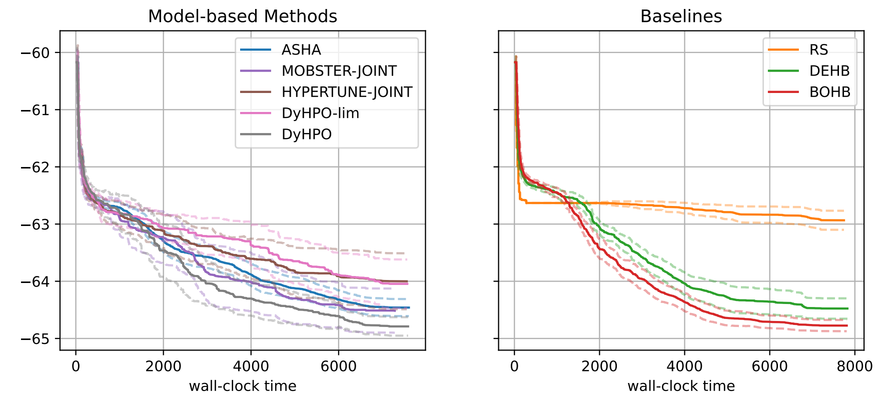


Figure 3.5 LCBench-albert

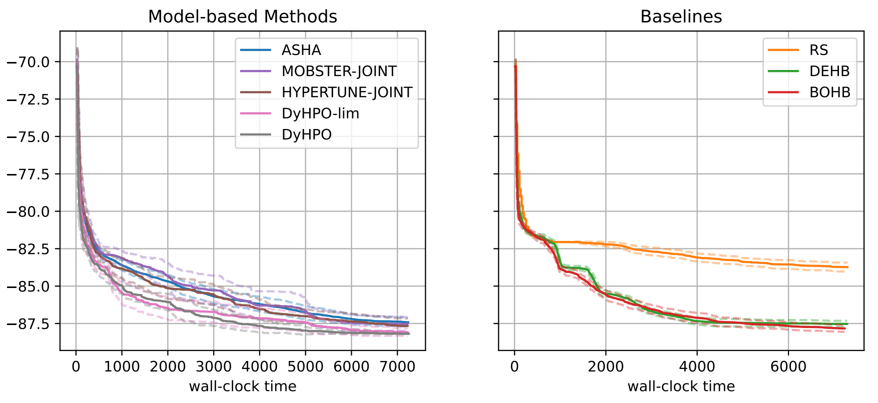


Figure 3.6 LCBench-Fashion-MNIST

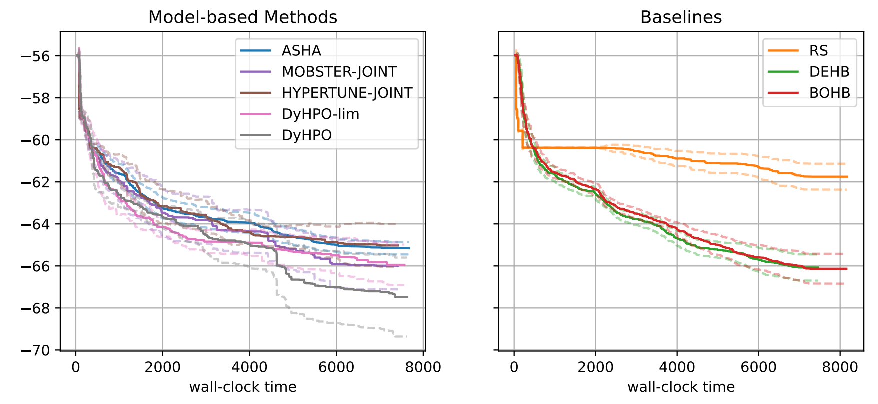


Figure 3.7 LCBench - covertype

Hyperparameter	Values
Learning rate	{0.0005, 0.001, 0.005, 0.01, 0.05, 0.1}
Batch size	{8, 16, 32, 64}
LR schedule	{cosine, fix }
Activation L1	{relu, tanh }
Activation L2	{relu, 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}

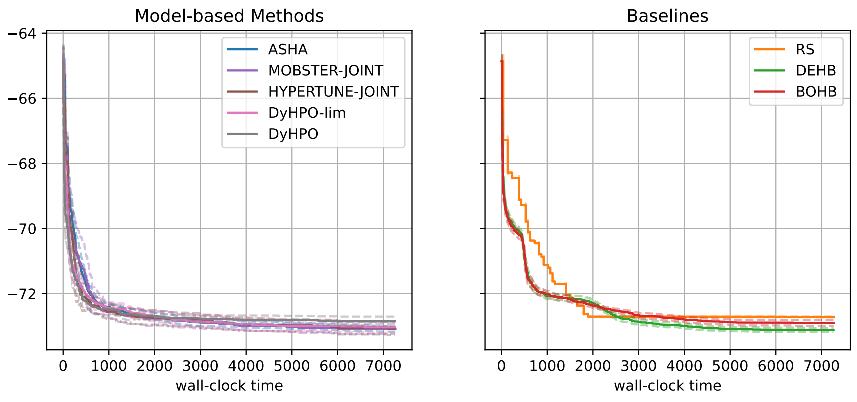


Figure 3.8 LCBench-christine

LCBench has the max_wallclock_time set to 7200 seconds, and max number of evaluations to 4000.

3.1.3 FCNet

The FCNet multi-fidelity benchmark contains 62208 configurations of MLP evaluated on 4 datasets (protein structure, slice localization, naval propulsion, parkinsons telemonitoring). The base architecture is two layer feed forward neural network followed by a linear output layer. The configuration space includes 4 architectural choices (number of units and activation functions for both layers), and 5 other hyperparameters (dropout rates per layer, batch size, initial learning rate, learning rate schedule). They discretized the search space and did an exhaustive evaluation of all reulting 62208 configurations. Each configuration was trained 4 times. Full learning curves are provided as well.

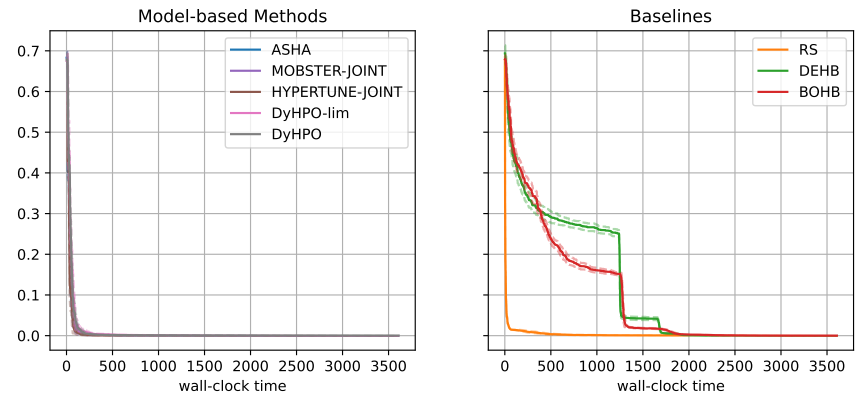


Figure 3.9 FCNet-naval

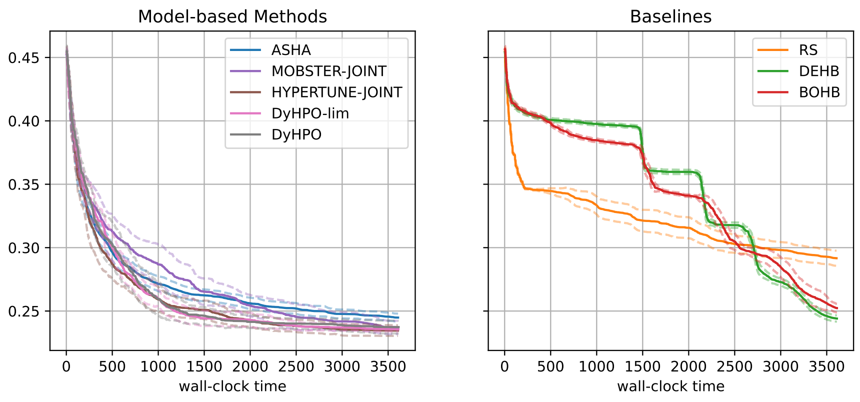


Figure 3.10 FCNet-protein

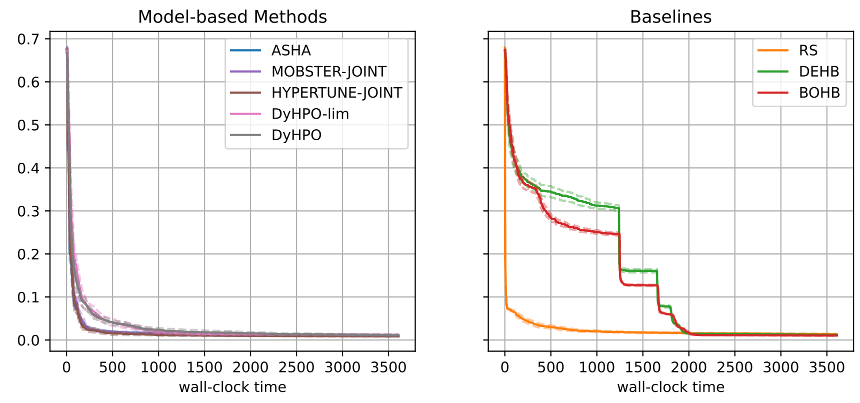


Figure 3.11 FCNet-parkinsons

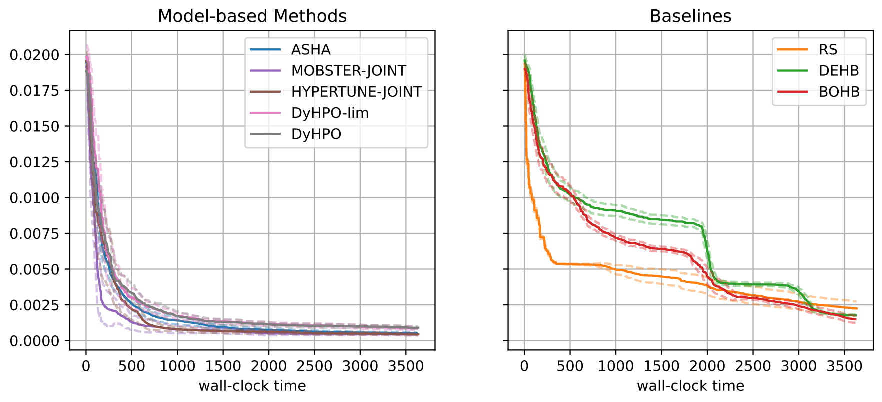


Figure 3.12 FCNet-slice

FCNet has default max_wallclock_time set to 3600. For FCNet-Naval, this translates to approximately 51 fully trained neural networks, since a full run of 100 epochs takes approximately 70 seconds. We can see that the midpoint rule of random search works well here. The brackets for DEHB and BOHB are again set incorrectly, and the algorithm spends too much time in low fidelity.

3.2 Real benchmarks

3.2.1 CIFAR10 - demo

Probably the most widely used image classification dataset is the CIFAR-10. We use it to evaluate the performance of Random Search, DEHB, Optuna and DyHPO hyperparameter optimization algorithms (not the final set of algorithms). The optimization was rerun 5 times for each algorithm with different seed but this seems to be too little, the means are not significantly different. The optimization problem consisted of six hyperparameters, three float and three integer. We used AdamW optimizer. Learning rate and the final value of cosine decay η_{min} were optimized. The neural network architecture consists of sequential convolutional layers with batch normalization. We optimize the number of convolutional layers and the number of filters. Then the fully connected layer follows and we optimize its size. The hyperparameters and their domains are summarized in the Table 3.2.

Hyperparameter	**Values**
Learning rate	$\{1e{-}4, 1e{-}1\}$
η_{min}	$\{1e{-}5, 0.99\}$
Dropout	$\{0.0, 1.0\}$
FC neurons	$\{8, 128\}$
Channels multiplier	$\{1, 8\}$
Conv layers	$\{1, 4\}$

Table 3.2 CIFAR-10 HPO optimization search space.

The network is trained for up to 70 epochs. The budget for the hyperparameter optimization is 17 full evaluations, which is equal to 1190 epochs. We compare the algorithms in classification accuracy on the validation set. We use the trial number on the x-axis, which neglects the cost of the hyperparameter optimization algorithm.

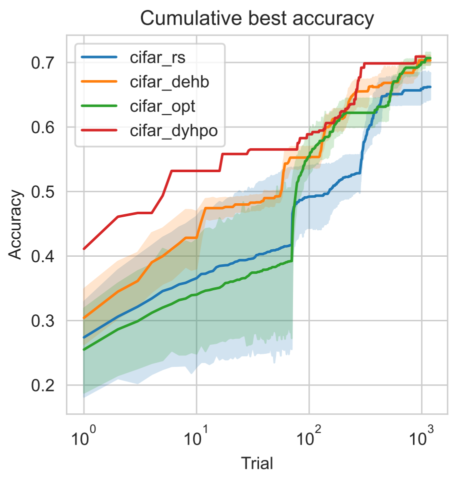


Figure 3.13 CIFAR-10 comparison of the best cumulative accuracy, 17 full evaluations (1190 epochs)

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.

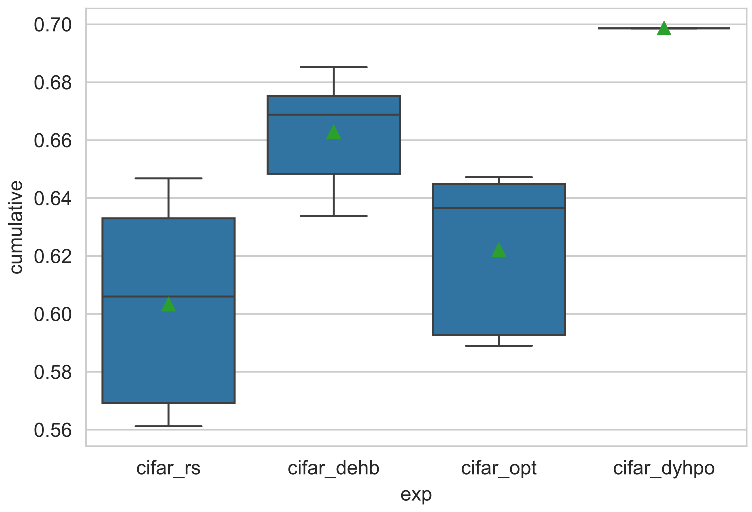


Figure 3.14 CIFAR-10 boxplot after 350 epochs.

3.2.2 SVHN

3.3 Discussion

Chapter 4

Related literature

Things that are not directly relevant, but related to HPO. Might not include this chapter in the final version. In this chapter, we mention other approaches in the literature that we did not use directly for solving the problem, but we think might be interesting for some readers.

4.1 Transfer learning

One advantage that an experienced practitioner will have over a classical HPO algorithm is that he will be good at generalizing and estimating good hyperparameter configurations across similar learning problems. The algorithm either depends on good bounds given by a user for efficient search, or it has to try a lot of configurations that do not perform well at all to find the bounds itself. The main idea of transfer learning is to use the experience from previous trials and similar problems in a new trial. This target function estimate should guide the search until the model is refined by new trials.

Collaborative hyperparameter tuning [24].

Efficient transfer learning method for automatic hyperparameter tuning [25].

Scalable hyperparameter transfer learning [26].

Pre-trained Gaussian processes for Bayesian optimization [27].

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] Varun Godbole et al. *Deep Learning Tuning Playbook*. Version 1.0. 2023. URL: `http://github.com/google-research/tuning_playbook`.

[4] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization.” In: *Journal of machine learning research* 13.2 (2012).

[5] 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.

[6] 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).

[7] Peter I Frazier. “A tutorial on Bayesian optimization”. In: *arXiv preprint arXiv:1807.02811* (2018).

[8] Jonas Mockus. “On Bayesian methods for seeking the extremum”. In: *Proceedings of the IFIP Technical Conference*. 1974, pp. 400–404.

[9] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006.

[10] 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.

[11] Jie Wang. “An intuitive tutorial to Gaussian processes regression”. In: *Computing in Science & Engineering* (2023).

[12] James Bergstra et al. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems* 24 (2011).

[13] Shuhei Watanabe. “Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance”. In: *arXiv preprint arXiv:2304.11127* (2023).

[14] 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).

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

[16] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. “Freeze-thaw Bayesian optimization”. In: *arXiv preprint arXiv:1406.3896* (2014).

[17] 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.

[18] 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.

[19] Noor Awad, Neeratyoy Mallik, and Frank Hutter. “Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization”. In: *arXiv preprint arXiv:2105.09821* (2021).

[20] Aaron Klein et al. “Fast bayesian optimization of machine learning hyperparameters on large datasets”. In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 528–536.

[21] 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.

[22] 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.

[23] 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.

[24] Rémi Bardenet et al. “Collaborative hyperparameter tuning”. In: *International conference on machine learning*. PMLR. 2013, pp. 199–207.

[25] Dani Yogatama and Gideon Mann. “Efficient transfer learning method for automatic hyperparameter tuning”. In: *Artificial intelligence and statistics*. PMLR. 2014, pp. 1077–1085.

[26] Valerio Perrone et al. “Scalable hyperparameter transfer learning”. In: *Advances in neural information processing systems* 31 (2018).

[27] Zi Wang et al. “Pre-trained Gaussian processes for Bayesian optimization”. In: *arXiv preprint arXiv:2109.08215* (2021).