

Hyperparameter Optimization in Machine Learning

Suggested Citation: Luca Franceschi, Michele Donini, Valerio Perrone, Aaron Klein, Cédric Archambeau, Matthias Seeger, Massimiliano Pontil and Paolo Frasconi (2024), “Hyperparameter Optimization in Machine Learning”, : Vol. XX, No. Preprint, pp 1–141. DOI: XXX.

Luca Franceschi

Michele Donini

Valerio Perrone

Aaron Klein

Cédric Archambeau

Matthias Seeger

Massimiliano Pontil

Paolo Frasconi

This article may be used only for the purpose of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit authors' approval.

Contents

1	Introduction	3
1.1	A Simple Illustrative Example	4
1.2	What Is Hyperparameter Optimization?	6
1.3	Why Is Hyperparameter Optimization Difficult?	6
1.4	Historical Remarks	7
1.5	Outline of the Monograph	9
1.6	Related Surveys	10
1.7	Notation	10
2	Hyperparameter Optimization	12
2.1	Learning Algorithms with Hyperparameters	12
2.2	The Hyperparameter Optimization Problem	15
2.3	Hyperparameter Optimization in the Literature	20
2.4	Practical Considerations for HPO Algorithms	21
2.5	Why not Tackling the Search Manually?	24
3	Elementary Algorithms	25
3.1	Grid Search	26
3.2	Random Search	28
3.3	Quasi-Random Search	30

4 Model-Based Methods	33
4.1 Model-Based HPO: A General Schema	34
4.2 Bayesian Optimization	36
4.3 Other Approaches to Model-Based HPO	48
4.4 Discussion	49
5 Multi-Fidelity Methods	53
5.1 Multi-Fidelity Optimization	53
5.2 Methods Based on Random Search	56
5.3 Model-Based Methods	61
6 Population-Based Methods	65
6.1 Evolutionary Algorithms	66
6.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)	67
6.3 Population-Based Training (PBT)	68
6.4 Particle Swarm Optimization	71
6.5 Discussion	72
7 Gradient-Based Optimization	73
7.1 Preliminaries	74
7.2 Hypergradients via Implicit Differentiation	75
7.3 Hypergradients via Iterative Differentiation	76
7.4 Fixed-Point Methods	80
7.5 Use Cases	81
7.6 Discussion	83
8 Further Topics	84
8.1 Optimizing for Multiple Objectives	84
8.2 Hyperparameter Schedules and Online HPO	90
8.3 Neural Architecture Search	92
8.4 Meta-Learning	96
8.5 Transfer of Hyperparameters Across Model Scale	99
9 Hyperparameter Optimization Systems	101
9.1 Open Source Libraries and Frameworks	101
9.2 Commercial Systems	102
9.3 Benchmarking	103

10 Research Directions	105
10.1 Response Functions for Unsupervised Learning	105
10.2 Learning to Optimize Hyperparameter	107
10.3 HPO and Foundation Models	108
Acknowledgements	110
References	111

Hyperparameter Optimization in Machine Learning

Luca Franceschi^{1,6}, Michele Donini², Valerio Perrone¹, Aaron Klein³, Cédric Archambeau², Matthias Seeger¹, Massimiliano Pontil⁴ and Paolo Frasconi⁵

¹*Amazon Web Services; work not related to position at Amazon*

²*Helsing*

³*ScaDS.AI, Leipzig University*

⁴*University College London & Istituto Italiano di Tecnologia*

⁵*Università di Firenze*

⁶*Work partially done while LF was at UCL and IIT*

ABSTRACT

Hyperparameters are configuration variables controlling the behavior of machine learning algorithms. They are ubiquitous in machine learning and artificial intelligence and the choice of their values determines the effectiveness of systems based on these technologies. Manual hyperparameter search is often time-consuming and becomes infeasible when the number of hyperparameters is large. Automating the search is an important step towards advancing, streamlining, and systematizing machine learning, freeing researchers and practitioners alike from the burden of finding a good set of hyperparameters by trial and error. In this survey, we present a unified treatment of hyperparameter optimization, providing the reader with examples, insights into the state-of-the-art, and numerous links to further reading. We cover the main families of techniques to automate hyperparameter search, often referred to as hyperparameter optimization or tuning, including

Luca Franceschi, Michele Donini, Valerio Perrone, Aaron Klein, Cédric Archambeau, Matthias Seeger, Massimiliano Pontil and Paolo Frasconi (2024), “Hyperparameter Optimization

in Machine Learning”, : Vol. XX, No. Preprint, pp 1–141. DOI: XXX.

random and quasi-random search, bandit-, model-, population-, and gradient-based approaches. We further discuss extensions, including online, constrained, and multi-objective formulations, touch upon connections with other fields, such as meta-learning and neural architecture search, and conclude with open questions and future research directions.

1

Introduction

In this chapter, we informally introduce the main subject of this monograph: algorithms for optimizing hyperparameters. Hyperparameters are configuration variables that are ubiquitous in machine learning methods and can strongly influence generalization and overall performance. The wide array of existing algorithms for optimizing hyperparameters is the product of several years of research. Their adoption is already widespread in academia and in the industry. Our goal is to provide a comprehensive introduction to the fundamentals of these methodologies, with the aim of encouraging their broader adoption – making them standard practice — and providing a solid foundation for further research in the field. Such efforts help strengthen and advance the current state-of-the-art, foster transparency and improve the reproducibility of scientific and engineering results. This is a timely necessity as the application of AI has rapidly expanded during the last decade and has solidified its position as a primary driver of innovation in many areas of science and industry.

The digitalization of human activities and interactions has led to the generation of data at an unprecedented scale. On the one hand, data can now be stored more affordably, thanks to advancements in microelectronics and the emergence of cloud computing. On the other hand, progress in hardware and low-power chip design have led to an exponential increase in comput-

ing capabilities for both cloud infrastructure, notably graphical processing units (GPUs), and on edge devices, like mobile phones. The convergence of these two key trends has been instrumental in the pervasive adoption of machine learning. Recent breakthroughs in algorithms and architectures for deep learning have further reinforced this foundation. Learning algorithms can now sift through vast amounts of data to discover and extract patterns that can then be used to guide decisions with little or even no human intervention. But this is largely due to a dramatic increase in model complexity, both in terms of size and structure. Modern architectures often comprise several intertwined modules, each introducing its own set of hyperparameters. Without systematic tools, configuring these hyperparameters jointly can be complex and overwhelming.

One notable and often cited illustration of the recent advances made in machine learning is AlphaGo (Silver *et al.*, 2017), a computer program developed at the London-based company *DeepMind*. AlphaGo won against one of the world champions of the game of Go, Lee Sedol, in 2016. This event generated considerable press coverage (and was even made into a movie) as it was believed at the time that no computer program could win a game of Go against a human before several decades to come. What is less known is that the success of AlphaGo critically depended on how a set of hyperparameters were adjusted automatically by another computer program (Chen *et al.*, 2018). This computer program relied on Bayesian optimization (see Section 4.2), a general black-box technique that in this case sequentially predicts and then evaluates the performance of learning algorithms, such as those behind AlphaGo, when their hyperparameters are set to specific values.

1.1 A Simple Illustrative Example

To illustrate the practical importance of carefully selecting the hyperparameters, we consider the problem of *sentiment analysis*. Yogatama *et al.* (2015) studied the effect of hyperparameters in this context. More specifically, the authors framed the problem as a standard binary classification problem as was commonly done in the literature, where the task of the classifier is to predict whether the text expresses a negative or a positive sentiment. They compared simple logistic regression, trained by stochastic gradient descent, to convolutional neural networks, which achieved state-of-the-art results at

Hyperparameter	Values	Method	Acc.
n_{min}	{1, 2, 3}	SVM-unigrams	88.62
n_{max}	{ $n_{min}, \dots, 3$ }	SVM-{1, 2}-grams	90.70
weighting scheme	{tf, tf-idf, binary}	SVM-{1, 2, 3}-grams	90.68
remove stop words?	{True, False}	NN-unigrams	88.94
regularization	{ ℓ_1, ℓ_2 }	NN-{1, 2}-grams	91.10
regularization strength	[$10^{-5}, 10^5$]	NN-{1, 2, 3}-grams	91.24
convergence tolerance	[$10^{-5}, 10^{-3}$]	LR (this work)	91.56
		Bag of words CNN	91.58
		Sequential CNN	92.22

(a) Hyperparameters.

(b) Accuracy.

Figure 1.1: Results on sentiment analysis reported in (Yogatama *et al.*, 2015). Hyperparameters and their domains (both continuous and discrete) are listed on the left. Here $[n_{min}, n_{max}]$ denotes the n -gram range. SVM: support vector machine; LR: logistic regression; NN: multi-layer perceptron; CNN: convolutional neural network.

the time of the publication. Figure 1.1a shows the hyperparameters that were searched over. These included the choice of text features used (e.g., removal of stop words or not), the type of regularization (e.g., L_1 or L_2) and optimization algorithm parameters (e.g., convergence tolerance). The experimental results they reported on the Amazon electronics dataset are reproduced in Figure 1.1b. Interestingly, they showed that an optimized logistic regression with a bag of words representation of the text performed similarly to a convolutional neural network and only slightly worse than a sequential convolutional neural network. Hence, by carefully optimizing the hyperparameters, the authors could achieve a result close to the state-of-the-art, yet using a much simpler model. In Section 2.1.1, we will come back to this example, explaining in detail the role of individual hyperparameters.

As this simple example reveals, claims about state-of-the-art performance improvements need to be interpreted carefully: empirical results can vary greatly depending on the chosen hyperparameters, with a possibly significant impact on the conclusions drawn. Unfortunately, it is not uncommon for published results to omit values for specific hyperparameters used in the experiments, or to omit details explaining how these values were chosen. In these cases, reproducibility may be hindered (Haibe-Kains *et al.*, 2020).

1.2 What Is Hyperparameter Optimization?

A typical supervised learning algorithm receives training data and outputs a predictor (e.g., a mapping from textual reviews to binary sentiments, as in our previous example). The quality of the predictor can then be measured on new data (the validation set) using an evaluation metric, such as the error rate. Since the predictor depends on the chosen hyperparameters, the validation error also depends on those hyperparameters. The mapping from hyperparameter values to the performance measured on the validation set is called the *response function*.

In a nutshell, hyperparameter optimization (HPO) consists in finding the hyperparameters that optimize the response function. What distinguishes the HPO problem from conventional optimization problems, such as those arising in convex or combinatorial optimization, is its *nested nature*: evaluating the response function at a specific point requires running the learning algorithm, which in turn often involves solving another optimization problem to fit a model to the available (training) data.

1.3 Why Is Hyperparameter Optimization Difficult?

First, the nested nature of hyperparameter optimization problems means that, in most cases, the response function is not available in closed form, but is defined only implicitly. The function may be stochastic (due to the inherent randomness of the training algorithm), have multiple local optima, exhibit wide flat regions, or be non-smooth, non-continuous, or non-differentiable. These characteristics rule out the application of standard gradient-based optimization algorithms. Moreover, evaluating the response function at a given hyperparameter assignment can be very expensive: for instance, training a modern deep architecture may require significant compute, ranging from several hours to several days on a GPU cluster. As a result, much of the field has focused on developing algorithms that make minimal assumptions, such as only requiring *observability* for instance. This perspective becomes clearer when we return to the example of AlphaGo: it does not matter whether the mathematical expression linking the hyperparameters of AlphaGo (i.e., how the algorithm is configured) to the probability of winning a game is explicitly known; what matters is that, for a given set of hyperparameters, we can

observe whether AlphaGo won or lost the game.

Second, the hyperparameter search space is often complex and heterogeneous. Some hyperparameters could be continuous (e.g., the learning rate), but others could be integer-valued (e.g., the number of layers in a deep neural network) or categorical to encode choices. Additional structure can also arise in the search space. Notably, *conditional* hyperparameters are variables that are only relevant depending on the value taken by others. For example, the number of units in the j -th layer of a neural network is only relevant if the network has at least j layers. If the domain to be searched is a small discrete set, an exhaustive search can be performed by evaluating all possible hyperparameter configurations within this set. Unfortunately, this is rarely the case. In most scenarios, the number of configurations can be infinite or even uncountable.

These challenges mean that, in practice, the focus of HPO is often to find “reasonably good” configurations as quickly as possible — which could still mean days of computation — rather than to find the global optimum of the response function. Hence, HPO consists in determining a small set of promising hyperparameters on which to evaluate the response function.

1.4 Historical Remarks

Although HPO has significantly developed in recent years, especially after the rise of deep learning, its foundations are much older. Here we briefly summarize some important landmarks. Model selection in statistics attracted significant attention in the 1970s and it is perhaps the earliest precursor to HPO in machine learning. Indeed, HPO itself is sometimes referred to as “model selection” — although the expression originally only denoted the simpler problem of finding the appropriate *dimension* of a model. Unlike modern approaches to HPO, which are based on a validation set, early model selection methods, such as Akaike’s information criterion (Akaike, 1974) and the Bayesian information criterion (Schwarz, 1978), were based on the idea of penalizing model complexity by adding a term to the likelihood function. In the context of small, analytically treatable model, penalties make sense as they avoid always selecting the largest (best fitting) model (Schwarz, 1978). A closely related idea, rooted in information theory, is the minimum description length principle (Rissanen, 1978), where model complexity is measured by the number of bits required to encode it.

Interestingly, some algorithmic ideas behind modern HPO methods are even older and can be traced back to well before the 1970s. The HPO problem in machine learning is related to optimal experimental design, an area that has a long tradition in statistics since the introduction of the *response surface methodology* (Box and Wilson, 1951); see (Myers *et al.*, 2016) for a recent account. One of the core ideas in that context is the approximation, by means of a surrogate model, of an otherwise unknown true response function that needs to be optimized. Model-based algorithms for HPO share the very same idea and the HPO objective function draws its name from these methodologies.

Other ideas behind modern HPO originate from a community at the intersection between AI and operations research, interested in a different class of problems: the configuration of heuristics and meta-heuristics in *problem solving*. The goal in these cases is not generalization as in machine learning, but finding solutions quickly, for example to certain instances of (weighted) Boolean satisfiability. The LEX system (Mitchell *et al.*, 1983) already embodied some ideas behind model-based approaches to HPO, which we cover in Chapter 4. LEX ran the to-be-configured algorithm to produce examples from which a *generalizer* is trained to produce better heuristics. Later, sequential model-based configuration (SMAC) was proposed to solve global optimization and propositional satisfiability problems (Hutter *et al.*, 2009b; Hutter *et al.*, 2011b). It was eventually applied to supervised learning (Snoek *et al.*, 2012; Thornton *et al.*, 2013). SMAC is related to Bayesian optimization (Močkus, 1975; Sacks *et al.*, 1989; Jones *et al.*, 1998a) as it is one of the techniques at the core of current model-based HPO approaches.

Yet another class of early approaches is based on the idea of racing, where candidate configurations are run in parallel and, as soon as sufficient statistical evidence is collected against a candidate, the run is stopped and the candidate replaced by a new one. This idea was initially formulated for model selection in machine learning (Maron and Moore, 1994), and then applied to configuration of meta-heuristics (Birattari *et al.*, 2002). Modern multi-fidelity strategies for HPO also discard unpromising candidates but in a more general setting (see Chapter 5).

Finally, population-based algorithms also originated in the area of problem solving with seminal works of Rechenberg (1965) and Fogel *et al.* (1965), boosted by the introduction of genetic inspired operators by Holland (1975). While their application to optimization in machine learning has a long tradition,

the use of population-based algorithms in HPO is more recent and preceded by the idea of gradient-based approaches (Bengio, 2000).

1.5 Outline of the Monograph

In this chapter, we introduced the HPO problem, motivating its importance and its place in machine learning. In the next chapter, we will formalize many concepts introduced here and further present several examples of hyperparameters and HPO problems. In Chapter 3, we will introduce elementary algorithms that search over a predefined or random grid of hyperparameters. While conceptually simple, random search strategies based on uniform sampling are attractive as they are embarrassingly parallel and exhibit an anytime behavior, meaning that stopping the search at any time yields a valid experiment. However, they require the evaluation of a relatively large number of hyperparameters to find a good solution. In Chapter 4, we will study model-based strategies, such as Bayesian optimization. These approaches learn a surrogate model of the response function based on the evaluations observed so far. Model-based approaches are sample-efficient, which means that they typically require evaluating a much smaller number of hyperparameters (compared to grid or random search) to find a satisfactory solution. However, they are inherently sequential, which can negatively impact the run time of the optimization algorithm. In Chapter 5, we will discuss how randomized approaches can be made resource-efficient by stopping unpromising evaluations early. We will also show that they can be married to model-based approaches in order to sample the hyperparameter search space in a non-uniform fashion. Chapter 6 covers population-based methods, stochastic algorithms for global optimization that draw inspirations from natural phenomena. When applied to HPO, some techniques in this family, such as population-based training, maintain a set of predictors whose hyperparameters are updated dynamically, unlike the aforementioned approaches where hyperparameters are fixed for the entire training duration. In Chapter 7, we will discuss how gradient-based approaches can be applied to the optimization of continuous hyperparameters through hypergradients, which is more time-efficient than search-based algorithms. Finally, we close the monograph by discussing advanced use cases. In particular, we study how to adapt HPO to constrained and multi-objective settings. Indeed, HPO algorithms provide a powerful set of tools to meet competing

requirements in applications (e.g., maximizing the predictive performance while mitigating the prediction bias) and satisfy practical constraints (e.g., maximal prediction latency). We also discuss connections with neural architecture search and meta-learning, and touch upon transfer learning of hyperparameters for foundation models. The two closing chapters are dedicated to an overview of HPO software frameworks (Chapter 9) and emerging topics and open questions in the field (Chapter 10).

1.6 Related Surveys

The topic of this monograph has been surveyed in some other papers, briefly by Hutter *et al.* (2015) and more extensively in the works of Bischi *et al.* (2023) and Yang and Shami (2020). Arlot and Celisse (2010) surveyed the model selection problem from a complementary statistical perspective. Specific aspects have been reviewed in other papers, in particular Bayesian optimization by Frazier *et al.* (2008), Shahriari *et al.* (2016), and Wang *et al.* (2023), neural architecture search by Ren *et al.* (2021) and Heuillet *et al.* (2024), evolutionary algorithms by Li *et al.* (2023), and meta-learning by Hospedales *et al.* (2020). HPO is a core topic in automated machine learning (AutoML). The subject has been covered extensively in the book by Hutter *et al.* (2019) and more recently in the survey by He *et al.* (2021).

This monograph aims to provide a broad methodological introduction to the problem settings and principal algorithmic approaches in HPO. It is intended to support both practitioners aiming to understand the principles behind the methods they use or consider adopting, and researchers entering the field or seeking a reference on specific algorithm classes or aspects of hyperparameter optimization.

1.7 Notation

We denote by \mathbb{R} and \mathbb{N} the set of real and natural numbers, respectively. For K a positive integer, we indicate by $[K] = \{1, \dots, K\}$ the subset of the natural numbers from 1 to K , included. We use calligraphic letters, like \mathcal{D} , to indicate sets, and script-style calligraphic, like \mathcal{D} to indicate collections. We reserve the Greek letter λ to denote hyperparameters, Λ to indicate hyperparameter spaces, and f to denote the response function. We use \mathbb{P} to indicate a probability

measure. Whenever confusion may arise, we further add a subscript like $\mathbb{P}_{x \sim v}$ to specify that the random variable x evaluated by \mathbb{P} follow the distribution v . We use \mathbb{E} with the same convention to indicate the expected value. We mostly reserve the letter p to indicate density or probability mass functions, and denote by $\mathcal{P}(\mathcal{X})$ the space of densities over the set \mathcal{X} . We denote by $\mathcal{N}(\mu, \Sigma)$ the multivariate Gaussian distribution with mean μ and covariance matrix Σ .

2

Hyperparameter Optimization

Here, we formalize hyperparameter optimization (HPO) along the concepts introduced in the previous chapter. We discuss the nested structure that characterizes an HPO problem and the archetypal settings encountered in supervised machine learning. To fix ideas, we revisit in greater detail the example of the introduction and showcase different application scenarios from the literature. We conclude the chapter with an overview of practical desiderata for HPO algorithms that will guide our discussions in the remainder of the monograph.

2.1 Learning Algorithms with Hyperparameters

Hyperparameters are variables that configure the behavior of a learning algorithm, shaping and modifying its inductive bias, i.e., the set of assumptions the algorithm implicitly relies on in order to generalize. We view a learning algorithm A as a higher-order, possibly stochastic, function that maps *data* and *hyperparameters* to a model h_λ , capable of accomplishing a given task:

$$A : \mathcal{D} \times \Lambda \rightarrow \mathcal{H}; \quad h_\lambda = A(\mathcal{D}, \lambda) \tag{2.1}$$

where \mathcal{D} denotes the collection of all finite datasets that the algorithm can process, Λ the *hyperparameter space* (the Cartesian product of the domains of individual hyperparameters), \mathcal{H} the *model space* that represents the class

of models that the algorithm may output, and $\lambda \in \Lambda$ a hyperparameter vector (or tuple). By writing $h_\lambda = A(\mathcal{D}, \lambda)$ in Eq. (2.1) we emphasize the dependence of the model returned by the algorithm on λ , but omit the dependence on the training data \mathcal{D} . This choice is consistent with the fact that one often considers the dataset “fixed and given” when setting up an HPO problem.

2.1.1 Logistic Regression for Sentiment Analysis: An HPO Perspective

As a concrete example, let us revisit in more detail the problem of sentiment analysis introduced in Chapter 1 and, in particular, the logistic regression model considered by Yogatama *et al.* (2015). Data in this case is a set of pairs $\{(x, y) \mid x \in \mathcal{X}, y \in \mathcal{Y}\}$ where \mathcal{X} is the space of documents and $\mathcal{Y} = \{0, 1\}$ is the space of labels, expressing a negative or a positive sentiment. The hyperparameter space of the learning algorithm is the Cartesian product $\Lambda = \Lambda^{(1)} \times \dots \times \Lambda^{(7)}$ of the seven domains reported in the column “Values” of Table 1.1a. The first four rows are related to a function $\Phi_\lambda : \mathcal{X} \rightarrow \mathbb{R}^{n_\lambda}$, the feature extractor, that maps documents to n_λ -dimensional vector representation. The last three rows are related to regularization and optimization. The model space is defined as follows:

$$\mathcal{H} = \left\{ h_\lambda : \mathcal{X} \rightarrow \mathcal{Y} \mid h_\lambda(x) = \mathbb{1}\{w^\top \Phi_\lambda(x) > 0\}, w \in \mathbb{R}^{n_\lambda}, \lambda \in \Lambda \right\}, \quad (2.2)$$

where $\mathbb{1}\{s\}$ is the indicator function: $\mathbb{1}\{s\} = 1$ if s is true and 0 otherwise. In other words, \mathcal{H} is the union of the spaces of all the n_λ -dimensional linear classifiers composed with feature extractor Φ_λ , as $\lambda^{(1)}, \lambda^{(2)}, \lambda^{(3)}$, and $\lambda^{(4)}$ vary in their respective domains $\Lambda^{(1)}, \Lambda^{(2)}, \Lambda^{(3)}$ and $\Lambda^{(4)}$.

In the context of a logistic regression model, learning consists in finding the parameter vector w that minimizes the negative log-likelihood of the data under this model. The first four hyperparameters determine in which space this minimization is carried out. For instance, the larger the range of the n -grams considered, the more input features and, consequently, parameters the model will have. This range is controlled by the hyperparameters $\lambda^{(1)}$ and $\lambda^{(2)}$. Unigrams alone ($\lambda^{(1)} = \lambda^{(2)} = 1$) are very likely not sufficient for sentiment analysis as the sentences $x_1 = \text{the restaurant is not busy but the food is tasty}$ and $x_2 = \text{the restaurant is busy but the food is not tasty}$ are indistinguishable since $\phi_\lambda(x_1) = \phi_\lambda(x_2)$. However,

using longer n -grams typically requires increasingly larger corpora to avoid overfitting, since the number of possible features (sequences of words of length n) grows exponentially with n .

The remaining hyperparameters, $\lambda^{(5)}$, $\lambda^{(6)}$, and $\lambda^{(7)}$, do not directly affect the model space, but rather determine the way in which the optimization is carried out. Specifically, the algorithm A outputs the model whose parameter vector w minimizes the training objective $\mathcal{J}_\lambda : \mathbb{R}^{n_\lambda} \times \mathcal{D} \rightarrow \mathbb{R}$ defined as follows:

$$\mathcal{J}_\lambda(w, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} -\log \mathbb{P}(y | \Phi_\lambda(x); w) + \lambda^{(6)} \Omega_{\lambda^{(5)}}(w), \quad (2.3)$$

where $\Omega_{\lambda^{(5)}}(w)$ is either the L^1 or the L^2 norm depending on the value of $\lambda^{(5)}$, and the probability that the model assigns a document to express a positive sentiment, encoded by $y = 1$, is given by

$$\mathbb{P}(1 | \Phi_\lambda(x); w) = \frac{1}{1 + e^{-w^\top \Phi_\lambda(x)}}.$$

Hyperparameter $\lambda^{(5)}$ exposes a binary choice between two different forms of regularization: the first favors sparse solutions, the second penalizes components of w with comparatively large values. Figure 2.1 offers a geometric interpretation of these regularization schemes. The hyperparameter $\lambda^{(6)}$ controls the strength of such regularization; if, for instance, $\lambda^{(5)} = L^1$, one can expect the number of 0's in the solution vector to be proportional to the value of $\lambda^{(6)}$. Finally, $\lambda^{(7)}$ controls the convergence tolerance of the numerical solver used to minimize the objective in Eq. (2.3).

Although logistic regression is a relatively simple algorithm, its practical application already exposes several hyperparameters. Other algorithms may require many more hyperparameters. This happens for example in deep learning, where feature extraction is internalized in the underlying models, and the optimization problems become more challenging.

The actual hyperparameter space depends on the application and how the problem is framed. For instance, the range of the regularization coefficient $\lambda^{(6)}$ in our example could in principle take any non-negative real value: it does not need to be limited to the interval $[10^{-5}, 10^5]$. Furthermore, one may want to use both types of regularization simultaneously in Eq. (2.3), instead of

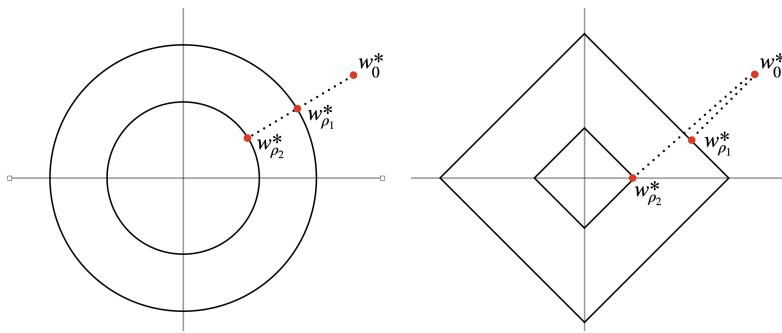


Figure 2.1: Geometrical representations of the L^2 (left) and L^1 (right) regularizers. w_0^* represents the minimizer of the training loss \mathcal{J}_λ when the regularization coefficient $\lambda^{(6)}$ is 0 (i.e. no regularization). $w_{\rho_1}^*$ and $w_{\rho_2}^*$ are, instead, the minimizers of the regularized losses for two different values of $\lambda^{(6)} = \rho_i$ for $i \in \{1, 2\}$, with $0 < \rho_1 < \rho_2$. These correspond to the projections of w_0^* onto L^2 and L^1 balls with varying radii that are inversely proportional to $\lambda^{(6)}$.

picking only one of the two, or one may consider the usage of also tetra-grams when extracting features. What to include in Λ is a design choice that can have important implications in practice. One of the reasons Yogatama *et al.* (2015) succeeded to achieve strong predictive performance with a simple linear classifier is that they included in Λ hyperparameters related to the feature extractor Φ_λ , whereas other authors opted for an *a priori fixed* map Φ instead of optimizing over it.

Hence, by exposing a (relatively) larger number of hyperparameters of the learning algorithm, such as parameters associated with the feature extractors or parameters associated with the optimizer or the regularizer, one can substantially expand the algorithm's model space \mathcal{H} . These additional degrees of freedom may in turn raise the odds of finding a better model for the task at hand, at the price of having to search over a possibly much larger hyperparameter space.

2.2 The Hyperparameter Optimization Problem

Hyperparameter configurations need to be evaluated according to a given performance metric, that we denote by \mathcal{M} . This metric takes as input a trained model and a *validation set*, and returns a scalar value, such as the validation

error¹. The objective of an HPO problem, the *response function*, is then constructed from the performance metric. Specifically, given a learning algorithm $A : \mathcal{D} \times \Lambda \rightarrow \mathcal{H}$, a metric $\mathcal{M} : \mathcal{H} \times \mathcal{D} \rightarrow \mathbb{R}$, a training set \mathcal{D} and a validation set \mathcal{V} , the hyperparameter optimization problem is defined as follows:

$$\min_{\lambda \in \Lambda} f(\lambda) = \mathcal{M}(h_\lambda, \mathcal{V}) \quad (2.4)$$

$$\text{such that } h_\lambda = A(\mathcal{D}, \lambda). \quad (2.5)$$

The *nested structure* of this problem involves minimizing a function whose dependence from its decision variables (the hyperparameters) is implicit through the execution of the learning algorithm. Thus, in principle, to evaluate f at a given point λ , we must run $A(\mathcal{D}, \lambda)$ until termination (e.g. convergence). Hence, evaluating the response function can be expensive, as many learning algorithms may take hours, or even days to terminate. We will discuss how we can relax this requirement using low-fidelity approximations in Chapter 5. The second major difficulty inherent to most HPO problems is the irregularity of the search space Λ . For instance, in our sentiment analysis example, the range of n -grams considered by the feature extractor Φ_λ is a discrete variable, ruling out the application of out-of-the-box continuous algorithms for optimizing f .

When A internally minimizes a training objective, such as the function \mathcal{J}_λ in the case of logistic regression, Problem (2.4)–(2.5) becomes an instance of *bilevel programming* (Colson *et al.*, 2007; Dempe, 2002; Dempe and Zemkoho, 2020) where the algorithm plays the role of the inner problem (Franceschi *et al.*, 2018). We will return on this view in Chapter 7 where we discuss gradient-based HPO techniques that explicitly leverage this structure. A key observation is that we cannot collapse the optimization problem into one level, where we jointly minimize both model parameters and hyperparameters. For instance, if we were to optimize the objective \mathcal{J}_λ from Eq. (2.3) jointly with respect to both w and λ , we would obtain as solutions the smallest values in Λ , namely $\lambda^{(6)} = 10^{-5}$ and $\lambda^{(7)} = 10^{-5}$, irrespective of the training set considered. This would defeat the purpose of these two hyperparameters which are limiting the model complexity.

¹In Section 8.1 we discuss the case where \mathcal{M} returns multiple scalars.

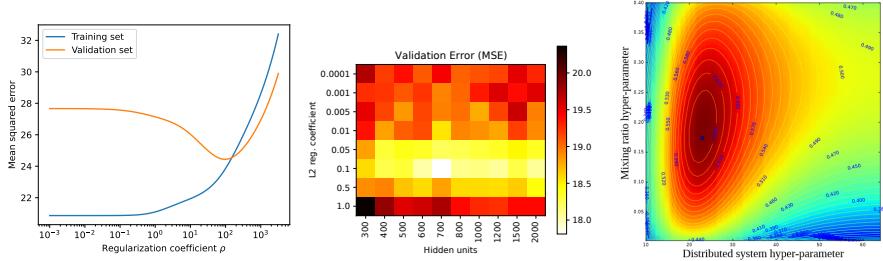


Figure 2.2: Three examples of response functions. From left to right the underlying algorithms are: ridge regression, two-layers neural network regression, Monte Carlo tree search (Chen *et al.*, 2017).

2.2.1 Estimating Generalization

Most of the material and techniques for HPO we covered in this monograph may be used regardless of the specific nature and meaning of \mathcal{M} and, hence, of f . However, since many hyperparameters control the inductive bias of their respective learning algorithms, the archetypal setting consists in a response function that estimates the generalization of the learning algorithm’s output models. Informally, a hypothesis generalizes if it “behaves sensibly” on data outside \mathcal{D} .

Formalizing this requirement satisfactorily may be a daunting task in certain scenarios (e.g., in generative modeling), but it is easy in the case of supervised learning, where we assume to have access to a loss function $\mathcal{L} : \mathcal{H} \times \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}$. We can define the generalization error of h_λ as the expected loss:

$$\mathcal{E}(h_\lambda) \doteq \mathbb{E}_{(x,y) \sim p} [\mathcal{L}(h_\lambda, (x, y))], \quad (2.6)$$

where p is the unknown distribution from which the training dataset \mathcal{D} was drawn independently. Widely used metrics such as classification error or mean-squared-error for regression can be interpreted as empirical estimates of Eq. (2.6):

$$\mathcal{M}(h_\lambda, \mathcal{V}) = \frac{1}{|\mathcal{V}|} \sum_{(x,y) \in \mathcal{V}} \mathcal{L}(h_\lambda, (x, y)), \quad (2.7)$$

where \mathcal{V} is a set of points drawn i.i.d. from p , dubbed *validation*, *held-out* or *development* set. For example, the sentiment analysis problem addressed by Yogatama *et al.* (2015) is an instance of supervised binary classification and

the task loss is the 0-1 error, namely $\mathcal{L}(h_\lambda, (x, y)) = \mathbb{1}\{h_\lambda(x) \neq y\}$. Note that in many cases, the internal structure of the learning algorithm, $A(\mathcal{D}, \lambda)$, also requires the minimization of a loss function \mathcal{J} with respect to the model's *parameters*, which may or may not coincide with \mathcal{L} . For example, in the case of classification, the 0-1 loss is intractable to minimize even for linear classifiers, and we usually resort to a convex surrogate such as the cross-entropy (Nguyen *et al.*, 2009).

Figure 2.2 shows some examples of other response functions. The leftmost plot concerns a simple ridge regression algorithm to regress median housing prices using the Boston dataset (Harrison Jr and Rubinfeld, 1978). The scalar hyperparameter controls the L^2 regularization and $\mathcal{L}(h_\lambda, (x, y)) = \|h_\lambda(x) - y\|^2$. The plot shows the mean loss computed on the training and validation data, respectively. The center plot represents the response function of a two-layer neural network on the same learning scenario. Here the two hyperparameters are an L^2 regularization coefficient (real-valued, vertical axis) and the number of units in the hidden layer (integer, horizontal axis). The rightmost plot, reproduced from the paper by Chen *et al.* (2018), represents instead the winning probability of a Go agent introduced in Chapter 1, and somewhat deviates from the standard setting we just discussed.

When estimating generalization of a model, it is *essential* to compute \mathcal{M} on a set of points that differs from \mathcal{D} . In fact, as learning algorithms adapt their hypotheses through \mathcal{D} , evaluating $\mathcal{M}(h_\lambda, \mathcal{D})$ would result in a distorted estimation of Eq. (2.6); e.g., underestimation when the model is under-regularized. This is illustrated in Figure 2.2, left. In order to obtain an unbiased estimate of the expected loss Eq. (2.6) one needs to draw points from the task distribution p *independently* both from each other and from h_λ . Otherwise, one carries over unwanted dependencies between training and validation.

In practice, one may split the entirety of the available data into two disjoint sets \mathcal{D} and \mathcal{V} and use the first as input to A and hold out the second for evaluating f .² More sophisticated approaches, such as *cross-validation*, involve splitting the data into K subsets and define the response function as a mean over K evaluation of \mathcal{M} , whereby at each round one subset is held out

²Note that a third *test set* is needed to estimate generalization capabilities of the hypothesis obtained after HPO!

for validation and the rest of the data is used for training (Stone, 1974; Kohavi, 1995). Cross-validation may reduce the variance of the estimate of Eq. (2.6) at the expense of increased computational cost.

2.2.2 Other Types of Response Functions

One may also be interested in optimizing hyperparameters with respect to performance measures not necessarily linked to generalization. For instance, \mathcal{M} could measure the inference latency or the memory requirements of h_λ when looking to deploy models for devices on the edge (Benmeziane *et al.*, 2021). Chen *et al.* (2018) take \mathcal{M} to be an estimate of the winning rate of the AlphaGo agent that plays with hyperparameters λ versus another AlphaGo agent that plays with a fixed baseline hyperparameter configuration λ_0 . Figure 2.2 (right) plots the response function (i.e. the winning rate) along two hyperparameter axes.

Furthermore, one may be interested in optimizing the hyperparameters of a learning algorithm simultaneously with respect to multiple measures (e.g., accuracy, inference latency and memory footprint) or in ensuring that h_λ satisfies some constraints regarding one or multiple measures. These scenarios give rise to *multi-objective* and *constrained* hyperparameter optimization, advanced topics which we will discuss in Chapter 8.

2.2.3 General Algorithm Configuration

Several algorithms, not only those employed in machine learning, require configuration constants that may affect both the quality of the solutions and the computational resources used to find solutions on specific instances (Schede *et al.*, 2022). While in HPO the emphasis is on the former aspect, in general algorithm configuration (AC) the focus is more typically on compute. A typical problem suitable for AC is propositional satisfiability (SAT), where proper choices of parameters can improve solving time significantly (Hutter *et al.*, 2009a; Hutter *et al.*, 2011b). Additional interesting examples can be found in the area of discrete optimization, where several parameters may be used to configure metaheuristics (Birattari, 2009). Entering into details would be out of scope for this survey. Still, we would like to point out a striking similarity between AC and HPO. In both cases, optimal (hyper) parameters depend on some unknown probability distribution, such as the data distribution for HPO,

or the distribution of future instances of a certain problem in AC. This requires resorting to data, such as a validation set or a set of problem instances, to formulate a proxy empirical optimization problem.

2.3 Hyperparameter Optimization in the Literature

Below we report a few notable milestones and examples of “success stories” of HPO in the literature.

- The paper by Bergstra and Bengio (2012), later extended by Bergstra *et al.* (2013), brought the attention of the community to random search as a sensible baseline compared to manual or grid search which, at the time of writing, were widespread approaches to HPO. We cover grid, random and more advanced quasi-random search techniques in Chapter 3.
- Snoek *et al.* (2012) used Gaussian-process based Bayesian optimization to optimize the hyperparameters of a 3-layer convolutional neural network (Krizhevsky *et al.*, 2012) on CIFAR-10. They reduced test error by 3% compared to the previous state-of-the-art that was achieved using manual search. These papers sparked interest in Bayesian optimization for HPO. Chapter 4 is devoted to model-based HPO and Bayesian optimization in the HPO context.
- Jamieson and Talwalkar (2016) and later Li *et al.* (2017) introduced bandit-based approaches to hyperparameter optimization. These methods draw inspiration from the multi-armed bandit literature, particularly the successive halving algorithm (Karnin *et al.*, 2013). They adaptively allocate resources to promising configurations while terminating unpromising ones early, emphasizing the concepts of budget and efficient compute allocation. These works have been particularly influential in the development of multi-fidelity optimization methods, which we discuss in Chapter 5.
- Zoph and Le (2017) parameterized the architecture of a convolutional and recurrent neural networks to discover new architectures. To sample new architectures they trained a recurrent neural network controller

based on reinforcement learning. This work marked the beginning of a sub-field of research dubbed neural architecture search (NAS), which we discuss in Section 8.3.

- As we mentioned in the introduction, Chen *et al.* (2018) used Gaussian-process-based Bayesian optimization to optimize the hyperparameter of the Monte-Carlo Tree-Search of Alpha GO. This improved the win-rate of Alpha GO from 50% to 66.5% in self-play games.

2.4 Practical Considerations for HPO Algorithms

The general formulation of Eq. (2.4) abstracts away specific aspects that are practically important and that collectively set aside HPO from a generic bilevel optimization problem. For this reason, the appraisal of an HPO algorithm should take into account several dimensions, as summarized in Table 2.1.

Efficiency and budget. The response function $f(\lambda)$ is typically stochastic, depending on random effects in the computation of $A(\mathcal{D}, \lambda)$. This may be due for example to weight initialization or to data ordering when performing stochastic gradient descent. It is also computationally demanding to evaluate. Moreover, HPO is typically only a step in a wider iterative process of model development and refinement. This means that in practice, the emphasis in HPO is to achieve a *sufficient decrease* of $f(\lambda)$ compared to a reference point (e.g., previously used default configuration) *within a given budget b* — expressed in time, number of function evaluations, monetary costs, etc. — rather than achieving an (approximate) optimum, which is the typical aim in other branches of optimization. Such a decrease needs to be traded off with compute and runtime costs. A good HPO algorithm should be *sample efficient*, such that it significantly decreases f with as few evaluations of f as possible. More importantly, it should be *wall-clock time efficient*, meaning that it finds good solutions as rapidly as possible. While number of evaluations and wall-clock time are related, they are not equivalent. For example, decision-making (i.e., choosing the next configuration to evaluate) counts towards wall-clock time and needs to be balanced against evaluation times of f for model-based HPO algorithms. When comparing different algorithms, time efficiency is captured by plotting the best value of f attained against wall-clock time and

Table 2.1: List of desirable properties of an HPO algorithm.

Characteristic	Brief Description
Sample efficiency	How many evaluations of f does the algorithm need to find satisfactory configurations?
Runtime efficiency	How long does the algorithm take to find satisfactory configurations?
Parallelizability:	How efficiently does the algorithm exploit parallel computational resources if:
Synchronous	<ul style="list-style-type: none"> • all parallel evaluations need to start at the same time?
Asynchronous	<ul style="list-style-type: none"> • evaluations can be started any time a resource becomes available?
Multi-Fidelity Support	Can the algorithm take advantage of cheaper approximate evaluations of f ?
Applicability	Does the algorithm have restriction on the type of search space Λ ?
Accessibility	Which type of access does the method need to the underlying learning algorithm and/or training data?
Ease of use	How difficult is to implement and tune the algorithm? What expertise is required to use it effectively?

computing the area under this curve. Some HPO methods tend to witness rapid initial descent of f , but may level off at a suboptimal value, while others spend more of their initial budget in order to ultimately achieve near-optimal performance.

Multi-fidelity. A powerful principle to increase time efficiency is to consider low-fidelity approximations of the response, which are cheaper to compute. For example, if we train a neural network for 64 epochs, we can access for free to b checkpoints, for all $b \leq 64$. Multi-fidelity HPO methods operate on such a sequence of fidelities $f_b(\lambda)$, where, in this case, $f(\lambda) = f_{64}(\lambda)$. By controlling the budget b for each trial evaluation λ (e.g., stopping training

after b epochs), they can often decrease $f(\lambda)$ with a much smaller number of full-fidelity evaluations. These methods will be discussed in detail in Chapter 5. One aspect that is worth noting when choosing an HPO algorithm is if it can leverage low-fidelity evaluation, and how the quality of such evaluation may impact the final result.

Synchronous and asynchronous parallelization. We can trade off time efficiency against computation cost by running several evaluations of f in parallel on (say) W workers. For a *synchronous* system, all W workers start new jobs at the same time. While this reduces wall-clock time, it often leads to idle time when workers need to wait for the slowest job to finish. In an *asynchronous* system, each worker can be assigned a new job independently of the status of the others. However, HPO decision-making for parallel execution systems comes with additional challenges. In the synchronous case, a batch of W new configurations has to be suggested in one go. Configurations in a batch should be “diverse”, in that they reveal non-redundant information about the optimum (Ginsbourger *et al.*, 2010; Desautels *et al.*, 2014; González *et al.*, 2016; Wilson *et al.*, 2018). While only a single new configuration needs to be proposed in the asynchronous case, up to $W - 1$ responses of running evaluations are not yet available (Snoek *et al.*, 2012). Not taking such pending evaluations into account runs the risk of increased redundancy in the proposed configurations. HPO can also be distributed across several computational nodes that may reside in networked computers. In this case, algorithms should also be parsimonious in terms of communication to reduce overhead as much as possible.

Applicability and accessibility. The applicability and accessibility of an HPO algorithm refer to the requirements it may have on the type of problem or the level of access needed to the underlying learning algorithm. Some algorithms may be restricted to certain types of search spaces, e.g., continuous vs. integer or categorical domains, or a mix of these. Furthermore, some algorithms can exploit conditionality in the search space, while others cannot. Some HPO methods like gradient-based techniques (Chapter 7) presuppose a white-box access to the underlying learning algorithm in terms of computational structure and/or data processing, while others assume the learning algorithm to be a black box. In between, there are a class of algorithms that

require a gray-box access, like the ability to start and stop the execution of the learning algorithm and be able to observe intermediate statistics. Although HPO has been traditionally described and tackled as a black-box optimization problem, more restrictive assumptions may be met in specific cases and HPO algorithm that exploit these assumptions typically show efficiency gains.

Ease of use. This aspect refers to the complexity involved in implementing, configuring, and effectively using an HPO algorithm. Some methods may be conceptually simple but require careful tuning of their hyperparameters or auxiliary components. Others may be more complex to implement from scratch or require deep expertise in the underlying techniques (e.g., Bayesian and multi-fidelity optimization) to use them effectively.

2.5 Why not Tackling the Search Manually?

In the next chapters we will introduce the main algorithms for HPO. One might question the need for sophisticated algorithms instead of just performing manual search. Indeed, searching for the best hyperparameters by hand can be alluring: getting started requires no new frameworks or assumptions, and one may be under the impression that the search could proceed organically, driven by hunches.

Yet, manual search can quickly become exhausting. Experienced practitioners may intuitively narrow the search, but novices face a dizzying array of choices. While flexible, tweaking hyperparameters without a systematic approach risks getting lost in an endless maze of configurations. Furthermore, with each trial, insights and analysis of results can guide exploration. While potentially beneficial to speed up the process, this risks overfitting by inadvertently exploiting patterns in the validation data. Even for experts, it may be difficult to make valid assumptions about the behavior of a learning algorithm, and it is not always obvious how to carry over expertise from one application domain to another (or from a class of models to another). Finally, manual search also suffers from poor reproducibility, as it is hard to track the steps a user followed to reach some final configuration.

3

Elementary Algorithms

In this chapter, we present simple non-adaptive, model-free and “memoryless” approaches to HPO, in that subsequent runs are not informed by any previous search history. These methods correspond to basic search schemes in global optimization that are conceptually simple, widely applicable, (mostly) easy to implement and straightforward to parallelize. Furthermore, these methods may work as initialization schemes for more advanced techniques like Bayesian optimization, as we shall see in later chapters.

We start with grid search, an approach that consists in dividing the search space into a grid of values and querying the response function on each of these values. Next, random search swaps the grid in favor of a fixed, simple distribution over the search space, drawing configurations from such distribution until a budget is exhausted. Finally, we move to quasi-random search strategies, approaches rooted in low-discrepancy sequence generation, in which the simple distributions of random search are replaced by more complex schemes that seek to explore the space more evenly. While grid search is a traditional and classic technique in HPO, consensus has grown in the field that random search should be considered as the baseline method for HPO (Bergstra and Bengio, 2012). Quasi-random search may offer benefits over random search, although to date we are not aware of any theoretical result supporting this.

3.1 Grid Search

Perhaps the first formal description of grid search (GS) dates back to the 1930s, appearing in a foundational work by Fisher (1935) on experimental design, under the name of *factorial design*. GS consists in evaluating $f(\lambda)$ over a pre-defined grid of hyperparameter configurations. For a generic hyperparameter $\lambda^{(i)}$ with domain $\Lambda^{(i)}$, we define the discretized domain $\bar{\Lambda}^{(i)}$ as a *finite* subset of $\Lambda^{(i)}$. The search grid is the Cartesian product

$$\bar{\Lambda} = \bar{\Lambda}^{(1)} \times \bar{\Lambda}^{(2)} \times \cdots \times \bar{\Lambda}^{(m)} \subset \Lambda.$$

The learning algorithm is then executed sequentially or in parallel on the resulting set of tuples and the returned solution is

$$\hat{\lambda} \in \arg \min_{\lambda \in \bar{\Lambda}} f(\lambda). \quad (3.1)$$

GS is a simple memoryless method that can be used to explore low-dimensional hyperparameter spaces. It requires very little implementation effort and parallelization is straightforward. Unfortunately, if d is the cardinality of the largest discretized domain, then $|\bar{\Lambda}| = O(d^m)$, making the method impractical for even moderate values of m . In addition, while Boolean and categorical hyperparameters naturally lend themselves to an evaluation on a predefined grid, unbounded integer and real-valued hyperparameters require manually setting bounds and discretization.

Extensive experimental practice has shown that some continuous hyperparameters, such as the learning rate, benefit from discretization on a logarithmic scale (Bengio, 2012). While requiring less expert knowledge than manual search, GS is very sensitive to the definition of the grid. To reduce the computational overhead, GS can be paired with manual search: the user alternates GS steps with adjustments and refinement of the grid values, e.g., using a finer grid “centered” around the best found configuration, or moving the search bounds of continuous hyperparameters, until a satisfactory result is found.

The top row of Figure 3.1 depicts equally distanced grid “samples” with 4, 9 and 16 points, on the unit square.

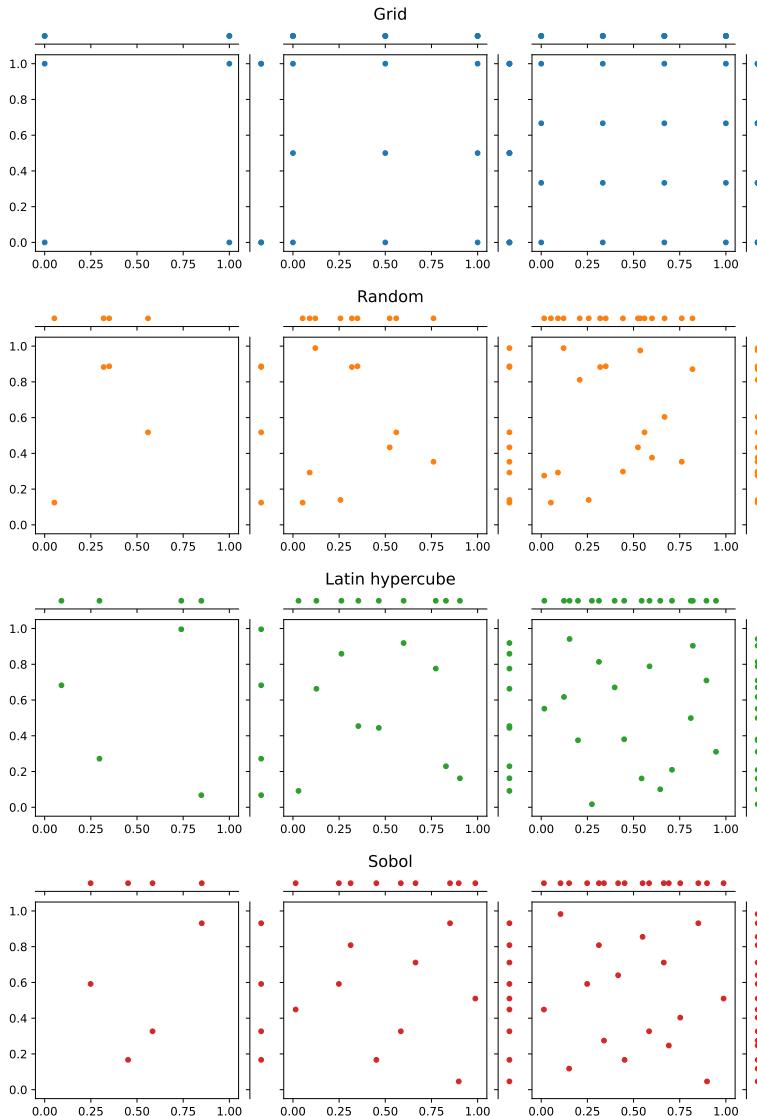


Figure 3.1: Non-adaptive sampling methods on the unit square for different sample sizes: 4 (left), 9 (center) and 16 (right) points. Each row represents a different schema. Grid and Latin hypercube sampling (LHS) require re-sampling every time we increase the sample size. Instead, for random and Sobol' we can keep on adding points, “reusing” previous samples. For each plot, we also show projections of the samples onto the two axes. We can see how grid samples cover much fewer points on the projections w.r.t. the other sampling methods. We can also observe the “unlucky run” phenomenon of (uniform) random sampling, where after nine draws the upper-right quadrant is still unvisited. LHS and Sobol' (with random shifts and scrambling) provide more even coverage of the space and its unidimensional projections.

3.2 Random Search

Random search (RS) was already considered in the 1950s as a method for global optimization (Brooks, 1958). It is perhaps the conceptually simplest procedure for HPO. RS consists in repeatedly drawing hyperparameter configurations from a predefined distribution ν over Λ , and it is readily parallelized. The solution after T trials is given by

$$\hat{\lambda}(T) = \arg \min_{\lambda \in \{\lambda_1, \dots, \lambda_T\}} f(\lambda), \quad (3.2)$$

where the trial points $\lambda_1, \dots, \lambda_T$ are drawn i.i.d. from $\nu \in \mathcal{P}(\Lambda)$. While in GS the total number of trial points is predetermined, one may keep on executing RS indefinitely. Termination conditions might involve approximate convergence checks, like lack of improvements after a certain amount of trials, or after a given budget is exhausted (see Section 2.4). For instance, one may run RS for 12 hours.

In principle, ν could be any distribution. However, when referring to random search for hyperparameter optimization, one typically implies the use of simple distributions (e.g., uniform or log-uniform distributions) that factorize independently along the configuration components, i.e., $\nu = \prod_{i=1}^m \nu^{(i)}$. For instance, if $\lambda^{(i)}$ is a categorical choice between N different neural architectures, then $\nu^{(i)}$ could be a categorical uniform distribution. Similarly, if $\lambda^{(i)}$ is the learning rate for stochastic gradient descent, then $\nu^{(i)}$ could be the log-uniform (or reciprocal) distribution on the interval $(10^{-5}, 10^{-1})$.

Using a distribution rather than a grid allows for a more natural treatment of a larger class of hyperparameters: RS does not require discretization of continuous spaces and also allows for distributions with non-compact supports.

The second row of Figure 3.1 shows uniform random samples with 4, 9 and 16 points on the unit square.

3.2.1 On the efficiency of random search

In an influential article, Bergstra and Bengio (2012) advocate for using RS as the “default” baseline method for HPO instead of manual or grid search. They argue that while RS retains the advantages of GS, such as reproducibility, conceptual simplicity, very low implementation overhead and ease of parallelization, it is also more efficient in higher dimensional spaces, especially

when the response function has a *low effective dimensionality*. The authors informally define this condition as when a subset of the hyperparameters accounts for most of the variation of the response function. For instance, for a bi-dimensional hyperparameter $\lambda = (\lambda^{(1)}, \lambda^{(2)})$ it could be that $f(\lambda) \approx g(\lambda^{(1)})$, for some $g : \Lambda^{(1)} \rightarrow \mathbb{R}$.

One way to reason about the success of RS is to assume that there is a sub-region $\Lambda^* \subset \Lambda$ that leads to satisfactory performance (i.e., where f is sufficiently small). Then, we can assess the probability that RS returns a solution in this sub-region (Brooks, 1958), assuming that hyperparameter assignments are drawn i.i.d. from ν :

$$\mathbb{P}[\hat{\lambda}(T) \in \Lambda^*] = 1 - (1 - \mathbb{P}[\lambda \in \Lambda^*])^T. \quad (3.3)$$

Eq. (3.3) says that the minimum number of trials needed to assure a satisfactory outcome with a given probability, say 95%, depends only on the probability of hitting Λ^* under ν . This can be independent of the search space dimension if the response function has a low effective dimensionality. This argument may explain the success of RS as a baseline in seemingly complex settings such as neural architecture search (Li and Talwalkar, 2020b), where repeated experimentation on a few well-known benchmark datasets has led to the development of well-tuned search spaces for neural architectures.

For a general search space, we should expect $\mathbb{P}[\lambda \in \Lambda^*]$ to decay exponentially in some $m' \leq m$. For concreteness, consider the case where $\Lambda = [0, 1]^m$ is the m -dimensional unit hypercube and $\Lambda^* = [0, 1/2]^{m'} \times [0, 1]^{m-m'}$ is the “good region”. The m' -dimensional $1/2$ -hypercube represents the important hyperparameters, while the $m - m'$ -dimensional unit hypercube represents the unimportant ones, as these coordinates do not matter in hitting Λ^* . If ν is the uniform distribution over Λ , then $\mathbb{P}[\lambda \in \Lambda^*] = 2^{-m'}$. Consequently, the number of trials needed to obtain a positive outcome with at least 95% probability is

$$T \geq \log(0.05) / \log(1 - 2^{-m'}).$$

For $m' = 3$ one already needs $T \geq 23$ draws to achieve a 95% probability of success. This illustrates that the number of trials needed grows exponentially in m' . At the same time, this also shows that RS is much more efficient compared to GS if $m' \ll m$.

3.2.2 Unlucky runs

One potential disadvantage of RS, due to its randomized nature, is that entire regions of Λ could be under-explored or completely neglected. As a consequence, a single execution of random search may not necessarily provide sufficient information to meaningfully modify v for another iteration of random search, should it be needed.

Let us illustrate this phenomenon. Continuing with our illustrative examples, say we divide the unit hypercube in 2^m 1/2 hypercubes I_j , for $j \in [2^m]$. We wish to have at least one configuration per hypercube (i.e., anywhere inside the 1/2 hypercube). While GS would need exactly 2^m points to achieve this, the probability under uniform sampling that RS hits all the hypercubes in 2^m draws is

$$\prod_{j=1}^{2^m-1} (1 - j/2^m).$$

For example, for $m = 3$ such probability is already $3/32$. This means that, in expectation, RS with $2^3 = 8$ draws covers all the hypercubes only once every ten runs.

3.3 Quasi-Random Search

One basic requirement for a stochastic generator of hyperparameter configuration is *consistency*: we would like that $\lim_{T \rightarrow \infty} \min_{k \in [T]} f(\lambda_k) = f(\lambda^*)$. RS with a uniform distribution is consistent, while GS is not. However, unlike GS, random search for any fixed budget may generate a set of configurations that misses large portions of the search space, as we just discussed. Intuitively, good sequences should cover evenly the search space. Also, their projections onto subspaces of Λ should be well spread, in order to speed up convergence when optimizing functions with low effective dimensionality — a property that GS misses. Quasi-random search methods seek to combine the advantages of grid and random search, with configuration-drawing strategies that combine randomness with deterministic steps.

Formalizations of these desiderata have been developed in the context of numerical integration and global optimization via the concepts of low dispersion and low discrepancy sequences (see e.g. Sobol', 1979; Niederreiter, 1988; Dick and Pillichshammer, 2010). Bousquet *et al.* (2017) study the

usage of such quasi-random sequences in HPO, finding that they consistently outperform random search (with a uniform distribution) on a series of deep learning tasks; see also (Godbole *et al.*, 2023, Chapter 7). The resulting methods are conceptually very similar to both GS and RS, being highly parallelizable, except that the set of hyperparameters $\lambda_1, \dots, \lambda_T$ are generated according to more complex procedures. We review two possible choices below, assuming that the search space is an m -dimensional unit hypercube for simplicity. Note that this is typically not at all a hard limitation: starting with samples on the unit interval, one can easily derive samples for a variety of distributions with appropriate reparameterizations (see, e.g. Devroye, 2006).

3.3.1 Latin hypercube sampling (LHS)

LHS (McKay *et al.*, 1979) is a stratified sampling scheme that directly combines grid and random search, in that it divides the search space into a grid, but samples configurations at random within regions of the grid.

LHS works as follows: choose a sequence length $n > 1$, which determines the number of hyperparameter configurations generated by a run of the sampling process. Divide each $[0, 1]$ interval of the unit hypercube into $n > 1$ parts, denoting by $I_s = [(s - 1)/n, s/n]$ the resulting intervals. This gives rise to n^m hypercubes of volume n^{-m} each. Then generate m random permutations of $[n]$, denoted by $\{\sigma_i\}_{i=1}^m$, $\sigma_i : [n] \rightarrow [n]$. For $k \in [n]$, sample hyperparameter configurations as follows:

$$\lambda_k \sim \mathcal{U}(H_k), \text{ where } H_k = I_{\sigma_1(k)} \times I_{\sigma_2(k)} \times \cdots \times I_{\sigma_m(k)}, \quad (3.4)$$

where $\mathcal{U}(H_k)$ is the uniform distribution on H_k . This generates n trial points; one may then generate further sequences of n configurations by repeating the process. Importantly, Latin hypercube sampling ensures that the projection on each axis of the n -long sequences places exactly one sample per interval.

The second last row of Figure 3.1 shows Latin hypercube samples on the unit square with 4, 9 and 16 points; namely, for $n \in \{2, 3, 4\}$.

3.3.2 Sobol' sequences

Sobol' sequences (Sobol', 1979) are a collection of separate one-dimensional sequences, one for each axis: when constructing an $m + m'$ dimensional

sequence one can reuse the m -dimensional one for the first coordinates. Furthermore, unlike LHS, Sobol' sequences are “extendible”, meaning that if $\{\lambda_1, \dots, \lambda_T\}$ is a Sobol' sequence, one can add new points $\lambda_{T+1}, \dots, \lambda_{T+T'}$ such that $\{\lambda_1, \dots, \lambda_{T+T'}\}$ remains a Sobol sequence. This is, in practice, a useful property shared with random search, as it allows one to avoid committing to a predefined budget T , offering flexibility to add more points as needed, without invalidating previous runs.

Let $j \in [m]$ be an axis, and let $\mathbb{F}_2 = \{0, 1\}$ with integer operations modulo 2. For each integer k , representing the iterate, let $k = \sum_{l \geq 0} a_l(k)2^l$ be the binary expansion of k , where $a_l(k) \in \mathbb{F}_2$ are binary coefficients (e.g., for $k = 3$, these would be $a_0(3) = 1$, $a_1(3) = 1$ and the rest 0's). Denote by $a(k)$ the resulting vector. Let C_j be a specific binary (infinite) “direction matrix” for axis j . Then the j -th coordinate of the k -th iterate of a Sobol' sequence can be written as

$$[\lambda_k]_j = \sum_l \frac{[C_j \cdot a(k - 1)]_l}{2^{l+1}}$$

where the matrix-vector multiplication $C^j \cdot a(k)$ is in \mathbb{F}_2 (i.e., the additions are modulo 2). The direction matrices C^j are constructed starting from primitive polynomials of \mathbb{F}_2 and direction numbers, which must be carefully chosen to ensure good coverage of the sequences. We refer the reader to Lemieux (2009, Chapter 5) for details.

Note that, given the directions matrices, Sobol' sequences are deterministic. It is common practice to obtain stochastic variants of such sequences. This is achieved by first adding random shifts, i.e. replacing λ_k with $\lambda_k + u$ (modulo the unit hypercube), where u is drawn uniformly from the interval $[0, 1]$, and by then randomly permuting the entries of the binary representation vector $a(k)$ (Owen, 1995). The bottom row of Figure 3.1 shows 4, 9, and 16 points randomized Sobol' sequences on the unit square.

3.3.3 Other schemes

Bousquet *et al.* (2017) experimented also with Halton and Hammersley sequences and found that the latter (with random permutations and shifts) performed well on the tested deep learning scenarios. Like Sobol' sequences, both Halton and Hammersley sequences can be extended. Lemieux (2009) presents other quasi-random low-discrepancy sequences. LHS, Sobol', Halton and Hammerley sampling are implemented in the `stats.qmc` module of SciPy.

4

Model-Based Methods

The techniques described in the previous chapter do not attempt to model the response function and are memoryless, i.e., the outcomes of previous evaluations do not influence the search. In contrast, model-based techniques use data gathered from past evaluations to build an explicit approximation of the objective function, known as a *surrogate model*, which is the central component of the family of HPO methods described in this chapter.

In general, the surrogate model takes as input a candidate configuration and, optionally, additional contextual information, such as the performance of partially trained models or the performance of the current best configuration (the incumbent). It then produces a prediction about how well the configuration is expected to perform. For example, the surrogate model might predict a distribution over performance scores. Details on the functional form of the surrogate depend on the specific instantiation of the model-based HPO method and will be discussed in the next sections. Regardless of the exact implementation, a key requirement of surrogate models is that they must be much cheaper to evaluate than the original objective function.

The landscape of model-based (MB) methods is strongly dominated by techniques falling under the umbrella of *Bayesian optimization* (BO), which we cover in Section 4.2. In BO, the surrogate model is a probabilistic mapping

that also captures the uncertainty of the estimation of the response function. Common implementations include Gaussian processes (Section 4.2.2) and tree Parzen estimators (Section 4.2.5). BO methods do not select the next configuration by directly minimizing the surrogate model. Rather, they use a second mapping, called *acquisition function*, which mediates the selection process. Typically, the acquisition function aims to balance the selection of the next configuration between two competing objectives:

- *Exploration*, which prioritizes evaluating the response function in regions where the surrogate model is highly uncertain, with the goal of gathering information to improve the surrogate model itself;
- *Exploitation*, which focuses on selecting configurations that the surrogate model predicts to perform well, aiming to improve the current best solution.

We examine various acquisition functions and their roles in Section 4.2.3, and present a practical example that contrasts BO to random search in Section 4.2.6. In Section 4.3, we briefly touch upon other approaches to model-based optimization that do not directly fall under the BO category. We conclude the chapter with an overall discussion of model-based methods in Section 4.4. In particular, we discuss the parallelizability of model-based methods — a significant practical bottleneck of MB-HPO methods — in Section 4.4.1.

4.1 Model-Based HPO: A General Schema

The vast majority of model-based methods are, in principle, sequential. At each iteration, the surrogate model is used to select the next hyperparameter configuration to evaluate, typically through the optimization of an acquisition function. Then, the response function is evaluated on the selected configuration by running the learning algorithm and assessing the resulting model performance. Finally, the surrogate model is updated with the newly collected data, or refitted on all data collected so far, and the process is repeated.

Algorithm 1 summarizes the typical steps of an MB-HPO method, where the selection of the next configuration is guided by the optimization of an acquisition function. The surrogate model is typically initialized by fitting it to several evaluations obtained during a warm-up stage, where configurations are

Algorithm 1 Model-based-HPO

Require: Response function f
Require: Warm-up dataset $C_0 = \{(\lambda_j, f(\lambda_j))\}_{j=1}^{n_0}$
Require: Class of surrogate functions $\hat{\mathcal{F}}$ and fitting routine
Require: Acquisition function α

```

1:  $\beta \leftarrow \min_j f(\lambda_j)$                                 ▷ Initialize incumbent performance
2: Fit  $\hat{f}_0$  to  $C_0$                                          ▷ Initialize the surrogate model
3: for  $k \leftarrow 1$  to  $T$  do
4:    $\lambda_k \leftarrow \arg \max_{\lambda'} \alpha(\lambda', \hat{f}_{k-1}, \beta)$       ▷ Select next hyperparameter
5:   Evaluate  $f(\lambda_k)$                                               ▷ Expensive step
6:   if  $f(\lambda_k) \leq \beta$  then
7:      $\beta \leftarrow f(\lambda_k)$                                          ▷ Update incumbent performance
8:   end if
9:    $C_k \leftarrow C_{k-1} \cup \{(\lambda_k, f(\lambda_k))\}$ 
10:  Fit  $\hat{f}_k$  to  $C_k$                                             ▷ Update the surrogate model
11: end for

```

generated using methods such as random search (see Section 3.2). We denote this initial configuration set as $C_0 = \{(\lambda_j, f(\lambda_j))\}$. New configurations are selected and evaluated, then added to the set of points $C_k = C_0 \cup \{(\lambda_i, f(\lambda_i))\}_{i=1}^k$, where $k \in [T]$ denotes the iteration index and $T \in \mathbb{N}^+$ is the total number of iterations. This set of points is then used to fit the k -th version of \hat{f} , which we denote by \hat{f}_k . We denote the acquisition function as:

$$\alpha : \Lambda \times \hat{\mathcal{F}} \times \mathcal{K} \rightarrow \mathbb{R}. \quad (4.1)$$

Here, $\hat{\mathcal{F}}$ denotes the class of surrogate functions and \mathcal{K} optional auxiliary information that may be needed by the acquisition function, including the validation performance of the incumbent — denoted by β in the algorithm, so that $\mathcal{K} = \mathbb{R}$.

One may interpret the interplay between fitting a surrogate model and choosing the next point to evaluate through an acquisition function as an iterative refinement of a sampling distribution $v_k \in \mathcal{P}(\Lambda)$ — implicitly defined through \hat{f}_k and α . In comparison, random and quasi-random search methods discussed in the previous chapter maintain a predetermined distribution $v \in \mathcal{P}(\Lambda)$ that does not change through iterations (cf. Section 3.2).

In the next section, we focus on model-based methods rooted in Bayesian optimization. We discuss specific implementations of surrogate models and their update strategies, as well as various acquisition functions.

4.2 Bayesian Optimization

BO has long been used as a method to globally optimize black-box functions, that is, functions for which we may only observe the value $f(\lambda)$ at any point $\lambda \in \Lambda$. It dates back at least to the 1970s with a series of works by J. Močkus (Močkus, 1975; Močkus *et al.*, 1978). Although traditionally suited for low-to medium-dimensional problems, BO has proven to be a powerful method for optimizing expensive, non-convex, and highly multi-modal functions. As a result, it has attracted considerable attention for its potential to tackle HPO problems, especially following the publication of three influential papers in the early 2010s (Bergstra *et al.*, 2011; Hutter *et al.*, 2011b; Snoek *et al.*, 2012). In this section, we give an overview of the general methodology in the HPO context. We refer the reader to the work by Jones *et al.* (1998a) for a more general perspective of the methodology and to the review by Shahriari *et al.* (2016) for further insights into practical aspects and applications to machine learning beyond HPO.

In Bayesian optimization, the surrogate model \hat{f} is a probabilistic model. Specifically, for any configuration $\lambda \in \Lambda$, the surrogate model $\hat{f}(\lambda)$ is a random variable whose distribution captures the uncertainty about the possible values of the true objective function $f(\lambda)$. This uncertainty has an *epistemic* nature, i.e., due to limited number of observations of configurations-responses pairs. The surrogate is updated using Bayesian inference, conditioning on the collected data C_k , to obtain a posterior distribution over functions \hat{f}_k . This posterior is then used to guide the selection of the next configuration via an acquisition function, as introduced in the previous section. The acquisition function encodes a strategy to balance exploration: sampling in regions of high uncertainty to improve the surrogate model, and exploitation: selecting configurations that are expected to yield good performance. In this context, the surrogate model's ability to represent epistemic uncertainty is crucial for enabling effective exploration.

Next, we discuss standard choices to design and fit surrogate models in BO, including Gaussian processes and tree Parzen estimators. We discuss

acquisition functions, focusing on the expected improvement (Močkus *et al.*, 1978). A classic instantiation of BO consists in modeling the surrogate with a Gaussian process and using expected improvement as the acquisition function. It is well-suited for the global optimization of smooth multivariate real functions, and it provides an accessible setting to implement BO for hyperparameter optimization.

4.2.1 Bayesian Linear Model

To introduce our first surrogate model, we start with a simple linear model h_w in a given feature space $\mathcal{V} \subseteq \mathbb{R}^d$, to which we add independent Gaussian noise with variance σ^2 :

$$h_w(\lambda) = \varphi(\lambda)^\top w + \varepsilon; \quad \varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2), \quad (4.2)$$

where $\varphi : \Lambda \rightarrow \mathcal{V}$ is a feature map and $w \in \mathbb{R}^d$ is a weight vector. Using linear maps directly in the input space would result in an overly simplistic model — we are trying to approximate a response function that can be highly non-convex and multimodal. However, mapping configurations into a suitable feature space allows us to capture much richer behaviors while retaining many of the benefits of the linear formulation (Murphy, 2012, Ch. 14). As evaluating f is expensive, we expect to work with only a few observations. The noise in Eq. (4.2) models potential errors (and inherent stochasticity) in the observations (we further discuss this aspect in Section 4.2.4).

Suppose now that we are at the k -th iteration of the algorithm, after having collected the observations $C_k = \{(\lambda_i, f(\lambda_i))\}_{i=1}^k$.¹ Let

$$Y_k = (f(\lambda_1), \dots, f(\lambda_k))^\top \in \mathbb{R}^k$$

be the vector of observed values, and denote by

$$X_k = (\varphi(\lambda_1), \dots, \varphi(\lambda_k)) \in \mathbb{R}^{d \times k}$$

the corresponding design matrix in the feature space (organized by columns). We are interested in obtaining a probability distribution over the output of f at a query point $\lambda_* \in \Lambda$, that is, in computing

$$y_* \sim \hat{f}_k(\lambda_*) = p(\cdot | x_*, X_k, Y_k),$$

¹In the following, we disregard the warm-up set, for simplicity.

where $x_* = \varphi(\lambda_*)$ denotes the feature representation of the query point λ_* . Here, the surrogate model $\hat{f}_k : \Lambda \rightarrow \mathcal{P}(\mathbb{R})$ produces a predictive distribution over real-valued outcomes, modeling the response function λ . The resulting distribution will be employed in Section 4.2.3, where the acquisition function uses the surrogate \hat{f}_k to guide the selection of the next configuration λ_{k+1} .

Modeling uncertainty away from observations. We could follow an empirical risk minimization approach, obtaining the weights $\bar{w}_k \in \mathbb{R}^d$ of the model in Eq. (4.2) by minimizing, for example, a (regularized) mean squared error over the dataset (X_k, Y_k) . We could then directly set $\hat{f}_k = h_{\bar{w}_k}$. Such an estimate would, however, leave us with a rather meager predictive distribution $p(\cdot | x_*, X_k, Y_k, \bar{w}_k)$, capable of capturing only our belief of the amount of noise present in the observations. As we discussed above, we would like the stochasticity of \hat{f} to (primarily) capture our uncertainty about the behavior of f in unexplored regions.

To better understand the difference between these two aspects, following an argument by Jones *et al.* (1998a), consider the case where f is a (non-linear) smooth deterministic function. If we were to follow an empirical risk minimization (ERM) approach, we could either omit the noise term ε in Eq. (4.2), or leave it for capturing modeling errors. However, the first choice would yield a single point estimate for any $\lambda \in \Lambda$, which is very unlikely to be accurate, especially for points far from those previously observed. Conversely, if we still wish to maintain a noise component, \hat{f}_k would trivially be inaccurate at λ_i for $i \in [k]$ (since f is deterministic). Thus, both ERM modeling options would leave us with a surrogate model that is *almost surely* inaccurate.

A way to escape this apparent stalemate is to replace the independent Gaussian noise of Eq. (4.2) with one that depends on the evaluation point, i.e. letting $\varepsilon : \Lambda \rightarrow \mathbb{R}$ be a function of λ . When fitting the model to the k observations, this choice allows having $\varepsilon_k(\lambda_i) = 0$ for $i \in [k]$ while still retaining $\varepsilon_k(\lambda) \neq 0$ for $\lambda \neq \lambda_i$. Furthermore, since we assumed f to be smooth, ε_k will also be smooth (as a difference of smooth functions). By continuity, we then have $\varepsilon_k(\lambda) \rightarrow \varepsilon_k(\lambda_i) = 0$ as $\lambda \rightarrow \lambda_i$. The Bayesian perspective offers a principled way to achieve this behavior, as we discuss next.

Deriving the posterior predictive distribution. We can reason about all linear models in the feature space that are consistent with the observations,

weighted by their posterior probability. In other words, we construct the surrogate model as a weighted average of all plausible h_w given the observed data. Formally, we marginalize over the weights w and set

$$\hat{f}_k(\lambda_*) = \int p(\cdot | x_*, X_k, Y_k, w) p(w | X_k, Y_k) dw \in \mathcal{P}(\mathbb{R}). \quad (4.3)$$

Given our modeling assumption of Eq. (4.2) and the hypothesis that the noise is independent, the first distribution simply reads as

$$p(\cdot | x_*, X_k, Y_k, w) = p(\cdot | x_*, w) = \mathcal{N}(h_w(\lambda_*), \sigma_\varepsilon^2) = \mathcal{N}(x_*^\top w, \sigma_\varepsilon^2). \quad (4.4)$$

By the Bayes' rule, $p(\cdot | X_k, Y_k)$ is instead given by

$$p(\cdot | X_k, Y_k) = \frac{p_y(\cdot | X_k, w)p(\cdot | X_k)}{p_y(\cdot | X_k)}. \quad (4.5)$$

Regarding the numerator of Eq. (4.5), we have that, for the same argument of Eq. (4.4),

$$p_y(\cdot | X_k, w) = \mathcal{N}(X_k^\top w, \sigma_\varepsilon^2 I).$$

This is the likelihood of the observations given the model. We must instead make a decision regarding the second term of the numerator, that is the distribution of the model parameters. We make the very natural assumption of independence from the design matrix and set

$$p(\cdot | X_k) = p = \mathcal{N}(0, \Sigma), \quad (4.6)$$

where Σ is a positive definite covariance matrix. This is a so-called *prior*. Equipped with Eq. (4.6), we can now also compute the denominator of Eq. (4.5) by marginalizing $p_y(\cdot | X_k)$ over w . We obtain

$$p_y(\cdot | X_k) = \int p_y(\cdot | X_k, w)p(w) dw = \mathcal{N}(0, X_k^\top \Sigma X_k + \sigma_\varepsilon^2 I). \quad (4.7)$$

We recognize, now, that the feature map only appears in a quadratic form where the (i, j) -th entry is given by $\varphi(\lambda_i)^\top \Sigma \varphi(\lambda_j)$: a dot product in the feature space. We then introduce the kernel corresponding to the feature map φ as follows:

$$\kappa_\Sigma(\lambda, \lambda') = \varphi(\lambda)^\top \Sigma \varphi(\lambda') = \langle \varphi(\lambda), \varphi(\lambda') \rangle_\Sigma \quad (4.8)$$

and rewrite Eq. (4.7) in terms of the Gram matrix $K = \{\kappa_\Sigma(\lambda_i, \lambda_j)\}_{i,j=1}^k \in \mathbb{R}^{k \times k}$ as

$$p_y(\cdot | X_k) = \mathcal{N}(0, K + \sigma_\varepsilon^2 I).$$

Finally, we have all the terms to compute Eq. (4.3); which is, again, Gaussian:

$$\hat{f}_k(\lambda_*) = p(\cdot \mid x_*, X_k, Y_k) = \mathcal{N}(k_*^\top(K + \sigma_\varepsilon^2 I)^{-1} Y_k, \kappa_\Sigma(\lambda_*, \lambda_*) - k_*^\top(K + \sigma_\varepsilon^2 I)^{-1} k_*) \quad (4.9)$$

where $k_* = \{\kappa_\Sigma(\lambda_*, \lambda_i)\}_{i=1}^k \in \mathbb{R}^k$ is the vector of kernel evaluations between the query and the observation points. Eq. (4.9), despite its apparent complexity, describes a rich *posterior* distribution over the values of f that is analytically computable and whose design depends essentially only on the definition of the kernel map in Eq. (4.8).

An example of kernel function is the squared exponential kernel:

$$\kappa_\Sigma(\lambda, \lambda') = \sigma_0 e^{-r_\Sigma^2(\lambda, \lambda')/2}; \quad r_\Sigma^2(\lambda, \lambda') = (\lambda - \lambda')^\top \Sigma^{-1} (\lambda - \lambda'),$$

where $\sigma_0 > 0$ is a scale parameter and Σ a diagonal matrix with positive length scales $\{\sigma_i\}_{i=1}^m$. Other examples include kernels from Matérn family, which contains mappings with various degrees of smoothness, defined as functions of r_Σ , introduced above (Williams and Rasmussen, 2006). Kernels intuitively encode the concept of distance or similarity between data points, and allow sidestepping the definition of explicit feature maps and covariance matrices. The series of “free parameters” that appears in Eq. (4.9) — that is, σ_ε , σ_0 and the diagonal entries of Σ when using the squared exponential or a Matérn kernel — are usually called *hyperparameters* in the Bayesian literature, and are not to be confused with λ . They do not need to be fixed constants but can instead be estimated from the observed data — see *automatic relevance determination* (MacKay, 1992; Williams and Rasmussen, 2006). Thus, the design choice left to the experimenter mainly boils down to the choice of the kernel function.

4.2.2 Gaussian Processes

Instead of working with explicit feature maps and weights, we can directly define a model through a kernel function, which leads to the framework of *Gaussian processes* (GPs) (Williams and Rasmussen, 2006; Murphy, 2012). In fact, the Bayesian linear model described above converges to a GP in the limit of infinitely many features, provided the prior over the weights is scaled appropriately (Neal, 1996).

A Gaussian process (GP) is a distribution over functions whose finite-dimensional marginals are multivariate Gaussian distributions. A GP is fully specified by a mean function $\mu : \Lambda \rightarrow \mathbb{R}$ and a covariance function $\kappa : \Lambda \times \Lambda \rightarrow \mathbb{R}$, also called the kernel. That is,

$$g \sim \mathcal{GP}(\mu, \kappa)$$

means g is a random function such that for any finite set of points $\lambda_1, \dots, \lambda_k \in \Lambda$, the random vector $(g(\lambda_1), \dots, g(\lambda_k))$ follows the multivariate Gaussian distribution:

$$(g(\lambda_1), \dots, g(\lambda_k)) \sim \mathcal{N}(M, \Upsilon)$$

where the mean vector is given by $M_i = \mu(\lambda_i)$ and the covariance matrix is given by $\Upsilon_{ij} = \kappa(\lambda_i, \lambda_j)$.

In Gaussian process-based Bayesian optimization, the surrogate model is a random function $\hat{f} \sim \mathcal{GP}(\mu, \kappa)$ taking values in the space of functions $\Lambda \rightarrow \mathbb{R}$. The update of the Gaussian process after observing C_k (the fitting step in Algorithm 1) corresponds to conditioning the prior on the dataset (X_k, Y_k) , and can be written as:

$$\hat{f} \sim \mathcal{GP}(\mu, \kappa | X_k, Y_k) = \mathcal{GP}(\mu_k, \kappa_k), \quad (4.10)$$

where typically μ is taken to be 0. While the posterior GP is technically an infinite-dimensional object, in practice we only require its marginal distributions at single query points λ_* (or small batches of points) — which are (multivariate) Gaussians — to compute acquisition functions and guide the search.

Our derivation of the Bayesian linear model in the previous section yields a zero-mean Gaussian process (since the prior over weights, Eq. (4.6), has mean zero) with covariance function

$$\kappa(\lambda, \lambda') = \kappa_\Sigma(\lambda, \lambda') + \sigma_e^2 \delta(\lambda, \lambda') = \varphi(\lambda)^\top \Sigma \varphi(\lambda') + \sigma_e^2 \delta(\lambda, \lambda'),$$

where $\delta(\lambda, \lambda')$ is the Dirac delta function, and the term $\sigma_e^2 \delta$ accounts for independent Gaussian observation noise. More specifically, the model derived in Eq. (4.9) corresponds to the posterior predictive distribution of the GP at a single query point λ_* , conditioned on the observed dataset (X_k, Y_k) . This marginal distribution over \mathbb{R} is a univariate Gaussian and provides the mean

$$\mu_k(\lambda_*) = k_*^\top (K + \sigma_e^2 I)^{-1} Y_k$$

and variance

$$\kappa_k(\lambda_*, \lambda_*) = \kappa_{\Sigma}(\lambda_*, \lambda_*) - k_*^\top (K + \sigma_e^2 I)^{-1} k_*$$

required for the evaluation of acquisition functions.

Limitations of Gaussian processes. Gaussian processes also come with a number of drawbacks that may limit their applicability to real-world scenarios. Most importantly, computing the posterior for k data points costs $\mathcal{O}(k^3)$, due to the inversion of Eq. (4.9). If we are able to sample f a few thousand times or more, GP-based BO starts to become intractable. It is also difficult to design kernel functions for high-dimensional Λ , or if the search space contains many integer, categorical, or conditional hyperparameters. In such situations, BO with other surrogate models can perform better, as shown in a comparative study by Eggensperger *et al.* (2013).

Snoek *et al.* (2012) tackle the first problem, proposing an extension of the expected improvement that accounts for pending executions. Snoek *et al.* (2015) tackles scalability issues by replacing the GP with a Bayesian neural network. Hutter *et al.* (2011b), Bergstra *et al.* (2011), Bergstra *et al.* (2013), Perrone *et al.* (2018), and Jenatton *et al.* (2017) focus on scalability and applicability to discrete and conditional search spaces by using random forests and tree Parzen estimators, which we will discuss in Section 4.2.5). More recent advances in the field (Springenberg *et al.*, 2016; Wu *et al.*, 2017; Klein *et al.*, 2017b) focus on replacing GPs with more practical and scalable approaches, exploiting additional information besides function evaluations (thus “dropping” the black-box assumption for f). The additional information could be, for example, the performance of A on a small subsets of D (Klein *et al.*, 2017b) or (possibly noisy) gradients (Wu *et al.*, 2017). Finally, Falkner *et al.* (2018) propose hybridizing BO with bandit-based methods (discussed in the next chapter) to improve performance in the first stage of HPO, when the surrogate model may still be unreliable.

4.2.3 Acquisition Functions

We now come to the second central component of a Bayesian optimization method: the acquisition function (AF). Suppose we have fitted our surrogate model \hat{f}_k to the current set of evaluations $C_k = \{(\lambda_j, f(\lambda_j))\}_{j=1}^k$. To decide

which point to evaluate next, BO optimizes an *acquisition function* $\alpha(\lambda, \hat{f}_k)$:

$$\lambda_{k+1} = \arg \max_{\lambda \in \Lambda} \alpha(\lambda, \hat{f}_k). \quad (4.11)$$

In this section, we primarily think of the surrogate model as a $\Lambda \rightarrow \mathbb{R}$ -valued random function modeled by a Gaussian process, as described above. However, most AFs discussed here can be applied to other models as well. As we will see, most common AFs depend on the posterior distribution of \hat{f}_k only through its marginal mean and variance

$$\mu_k(\lambda) = \mathbb{E}[\hat{f}_k(\lambda)], \quad \sigma_k(\lambda)^2 = \mathbb{V}[\hat{f}_k(\lambda)],$$

given by Eq. (4.9), where \mathbb{E} and \mathbb{V} denote expectation and variance, respectively.

A useful AF should guide the choice of the next candidate to gain as much new information about high-performing hyperparameters as possible, given the data C_k collected so far. In particular, it needs to balance the trade-off between exploration (i.e., looking away from points in C_k in order to increase coverage) and exploitation (i.e., refining the search by probing near current optima in C_k). We can gain intuition by considering two extremes. First, we could sample at our current best guess of where the minimum should be:

$$\alpha_{\text{exploit}}(\lambda | \hat{f}_k) = -\mu_k(\lambda).$$

Maximizing α_{exploit} constitutes an extreme form of exploitation, which typically gets stuck in suboptimal local optima. On the other extreme, we could sample where we know least about f :

$$\alpha_{\text{explore}}(\lambda, \hat{f}_k) = \sigma_k(\lambda).$$

Maximizing α_{explore} is entirely focused on exploration and may waste many evaluations in regions of poorly performing hyperparameters. However, a linear combination of the two,

$$\alpha_{\text{LCB}}(\lambda, \hat{f}_k) = -\mu_k(\lambda) + \eta \sigma_k(\lambda), \quad \eta > 0,$$

is a competitive and frequently used AF, called *lower-confidence bound* (LCB) (Srinivas *et al.*, 2010). The choice of η explicitly determines the trade-off between exploration and exploitation. Choosing η depends on the application and, unfortunately, there is no good default rule of thumb to set the value.

Expected improvement. A classic AF, introduced in the seminal work by Močkus *et al.* (1978) and frequently used in HPO libraries (Snoek *et al.*, 2012) and services (Perrone *et al.*, 2020), is the *expected improvement* (EI), defined as

$$\alpha_{\text{EI}}(\lambda, \hat{f}_k, \beta) = \mathbb{E} \left[\max \left\{ \beta - \hat{f}_k(\lambda), 0 \right\} \right].$$

Here, $\beta \in \mathbb{R}$ is a baseline, which is typically chosen as the minimum value of f observed so far: $\beta = \min_{k \in [K]} f(\lambda_k)$. In contrast to LCB, EI does not come with a free parameter to be set. EI makes an implicitly explore-exploit trade-off that has been reported to result in good performances in practice. A generalized version of EI replaces 0 by $\eta > 0$, which adjusts this trade-off with a cut-off hyperparameter. In addition, it can be shown to have a simple closed-form expression in terms of $\mu_k(\lambda)$ and $\sigma_k(\lambda)$:

$$\alpha_{\text{EI}}(\lambda, \hat{f}_k, \beta) = \sigma_k(\lambda) \nu(\lambda) \Phi(\nu(\lambda)) + \sigma_k(\lambda) \phi(\nu(\lambda)) \quad (4.12)$$

where

$$\nu(\lambda) = \frac{\beta - \mu_k(\lambda)}{\sigma_k(\lambda)}$$

and where Φ and ϕ denote the cumulative distribution function and the probability density function of the standard Gaussian distribution, respectively.

Eq. (4.11), where α is the expected improvement, and Eq. (4.10) specify a particular simple instantiation of a GP-based BO algorithm that may be used for HPO. As an illustrative example, two steps of the Bayesian optimization algorithm using EI as the acquisition function are shown in Figure 4.1.

Other acquisition functions. An earlier and simpler variant of EI is the probability of improvement (Kushner, 1964), given by

$$\alpha_{\text{PI}}(\lambda, \hat{f}_k, \beta) = \mathbb{P} \left[\hat{f}_k(\lambda) < \beta \right] = \Phi(\nu(\lambda)). \quad (4.13)$$

Another popular AF is known as *Thompson sampling* (TS) (Thompson, 1933; Hoffman *et al.*, 2014). The idea is to draw a sample path from our surrogate model and to search for its minimum point. Exact TS is intractable for a GP surrogate model, sample paths of which do not have a finite representation. In practice, we can draw a joint sample over a finite set of points or approximate the GP posterior by a Bayesian linear model (Hernández-Lobato *et al.*, 2014).

Other relevant families of AFs rely on information-theoretic criteria (Villemonteix *et al.*, 2009; Hennig and Schuler, 2012; Hernández-Lobato *et al.*,

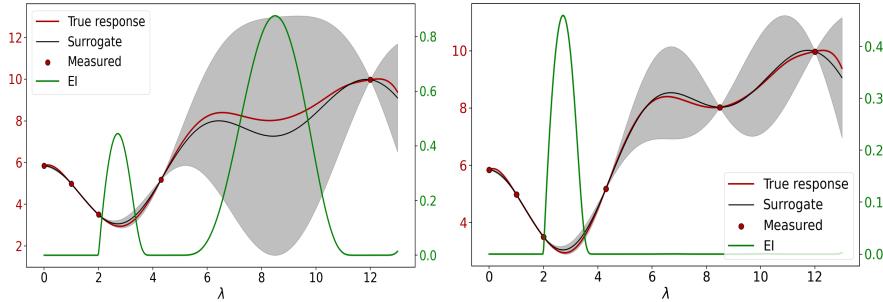


Figure 4.1: Left: Suppose we have measured the response function (plotted in red) on a small set of points. Fitting a Gaussian process on these points yields a mean function (plotted in black) with and a variance function (\pm one standard deviation is filled in gray). If λ falls between 7 and 11, the uncertainty is very high, which yields the right bump in the expected improvement (plotted in green), allowing us to *explore* a region where potentially a smallest response might be found. The left bump of the expected improvement falls in an *exploitation* region, which in this example looses against exploration. Hence, the next candidate is $\lambda = 8.5$. Right: once the response is measured at the new candidate, the Gaussian process is trained again. Now uncertainty shrinks significantly and the new expected improvement has just one bump around 2.7, which is already very close to the minimum of the true response. This example takes inspiration from Figure 11 in (Jones *et al.*, 1998b).

2014; Wang and Jegelka, 2017b). These typically are more complex to implement, come with higher computational cost, and require biased approximation, but have compelling theoretical foundations and often perform better than standard choices such as EI or LCB, achieving lower sample complexity. Given that the choice of the best AF can depend on the target function to optimize, one can also resort to a multi-armed bandit strategies to pick the most suitable AF from a portfolio of choices over the course of the optimization (Hoffman *et al.*, 2011; Shahriari *et al.*, 2016).

Optimizing acquisition functions in practice. The optimization of acquisition functions can itself pose a challenge. As a result, Eq. (4.11) is rarely solved exactly. Techniques for maximizing AFs include grid, random, and quasi-random search, discussed in the previous chapter. Because surrogate models are cheap to evaluate, it is typically feasible to sample the acquisition function at many candidate points. This allows even naive search strategies to perform reasonably well in practice. More sophisticated approaches rely on gradient-based optimization, which uses the derivatives of the acquisition

function to locate local maxima more efficiently. This is especially effective when the acquisition function is smooth and differentiable — such as in the case of GP-based posteriors where gradients can be computed analytically or via automatic differentiation. However, it is important to note that thorough optimization of the acquisition function is rarely essential. This is because the surrogate model is only an approximation of the true objective, and the acquisition function itself is a heuristic derived from it. Approximate maximization is typically sufficient to drive effective search progress for the overall HPO algorithm (Wilson *et al.*, 2018).

4.2.4 Aleatoric and Epistemic Uncertainty

The noise term ε introduced in Eq. (4.2) accounts for the inherent stochasticity of the learning algorithm—for example, due to random initialization when training a neural network. This kind of randomness gives rise to *aleatoric uncertainty*, which reflects intrinsic noise in the observations and may depend on the specific hyperparameter configuration.

However, as we saw in the previous sections, Bayesian optimization fundamentally relies on a surrogate model that not only predicts performance but also quantifies uncertainty. In particular, acquisition functions such as expected improvement and lower-confidence bound use the predictive variance of the surrogate model to guide exploration. This variance primarily captures *epistemic uncertainty* — uncertainty due to limited data. Crucially, this is the kind of uncertainty that can be reduced by collecting more observations, and that the acquisition function should leverage when choosing new points to evaluate.

Failing to distinguish between epistemic and aleatoric uncertainty can be problematic. If the surrogate model attributes too much of its predictive variance to aleatoric noise (which cannot be reduced), the acquisition function may over-prioritize regions where further evaluation is uninformative. Conversely, underestimating aleatoric noise can lead to overconfident predictions that neglect areas of genuine variability. To address this, heteroscedastic models—which explicitly account for input-dependent noise—can be employed. For example, Griffiths *et al.* (2021) train two distinct Gaussian processes aiming to predict the unknown response and the noise separately.

4.2.5 Tree Parzen Estimator

We now return to surrogate models and describe an implementation particularly well-suited for structured search spaces: the tree Parzen estimator (TPE). Unlike the Gaussian process framework, which models the conditional distribution of the response function given a configuration, TPE (Bergstra *et al.*, 2011; Bergstra *et al.*, 2013) adopts a different approach. It models two conditional densities over configurations, given a performance threshold: one for configurations with good performance, and the other for those with poor performance. Formally, in the TPE framework the surrogate model $\hat{f} : \Lambda \times \mathbb{R} \rightarrow \mathbb{R}^2$ is defined as

$$\hat{f}(\lambda, \beta) = (l_\beta(\lambda), g_\beta(\lambda)), \quad (4.14)$$

where l_β and g_β are conditional density estimates over Λ . The threshold β is typically set to the γ -quantile of the observed responses at iteration k , for some $\gamma \in (0, 1)$, e.g., $\gamma = 0.15$. This induces a $(\gamma, 1 - \gamma)$ partition of the data C_k , where the set of good-performing configurations is defined as

$$\{(\lambda, f(\lambda)) \in C_k \mid f(\lambda) < \beta\} \subset C_k,$$

based on the available observations and the corresponding partition induced by γ . At each iteration k , the densities l_β and g_β are fitted separately to the two subsets of C_k , so that

$$l_\beta(\lambda) \approx p(\lambda \mid f(\lambda) < \beta), \quad g_\beta(\lambda) \approx p(\lambda \mid f(\lambda) \geq \beta).$$

TPE uses as acquisition function the ratio of the two densities:

$$\alpha_{\text{TPE}}(\lambda, \hat{f}_k, \beta) = \frac{l_\beta(\lambda)}{g_\beta(\lambda)},$$

which assigns higher values to configurations that the surrogate model deems more likely to be good-performing than poor-performing. The ratio is monotonically related to the expected improvement acquisition function with threshold β .² In practice, to approximate the maximization of α , TPE samples a set of candidate configurations from l_β and selects the one that maximizes the ratio $l_\beta(\lambda)/g_\beta(\lambda)$. This generative process of sampling λ from l_β allows TPE to

²Note that unlike in GP-based BO, we cannot set β to the minimum observed value, as this would imply $\gamma = 0$ and yield an empty set for fitting l_β .

flexibly accommodate conditional or hierarchical search spaces by employing appropriate factorizations of l_β – for example, tree-structured models.

TPE is conceptually simple, naturally parallelizable, and handles complex, conditional search spaces well. Subsequent work has shown that the same acquisition objective can be approximated via probabilistic binary classification, which bypasses density estimation and leverages standard machine learning tools (Tiao *et al.*, 2021).

4.2.6 Bayesian Optimization Vs. Random Search: An Example

We illustrate the difference between model-based HPO and random search in a concrete example of tuning regularization parameters of a machine learning model to prevent overfitting. We tuned the L1 and L2 regularization terms, respectively alpha and lambda, of the XGBoost algorithm (Chen and Guestrin, 2016) on the UCI direct marketing binary classification dataset from Kaggle. The goal was to minimize the AUC. Figure 4.2 presents a comparison between random search and BO (model-based), which models the objective function through a GP and uses the expected improvement as the acquisition function.

Each experiment was conducted using 50 different random seeds, and the results were reported as the average and standard deviation. The left and middle plots reveal that BO consistently suggests better-performing hyperparameters than random search. The right plot further demonstrates that BO consistently outperforms random search across all evaluated hyperparameter configurations.

While BO is particularly effective in a sequential setting, random search remains a valuable approach, especially in distributed environments where many configurations can be evaluated in parallel, significantly reducing wall-clock time.

4.3 Other Approaches to Model-Based HPO

Not all model-based methods follow a Bayesian optimization framework. Ilievski *et al.* (2017) use a deterministic radial basis function model for \hat{f} which they interpolate on observations of the response function. Hazan *et al.* (2018) focus on Boolean hyperparameters and use a sparse Fourier basis model for \hat{f} , in conjunction with heuristics to estimate the most sensitive hyperparameter

entries. They then use a base global optimizer to perform HPO on a restriction of the original hyperparameter space. The works by Lorraine and Duvenaud (2018) and MacKay *et al.* (2019), related to gradient-based hyperparameter optimization (covered in Chapter 7), can also be interpreted as attempts to learn a surrogate of the entire learning algorithm (applied to a specific dataset), i.e., learning a mapping $\hat{f} : \Lambda \rightarrow \mathcal{H}$ that maps hyperparameters to trained models (cf. Eq. (2.1)).

Other approaches blend meta- and transfer-learning to model-based HPO by training a surrogate model on multiple datasets and tasks. In particular, (Feurer *et al.*, 2015) extracts meta-features from multiple datasets, including descriptive statistics (number of instance, classes, mean, variance and higher moments) and “landmarks” (i.e. the performances of a few quick-to-evaluate learning algorithms on the dataset). They then use this information to train a surrogate model that initializes a BO process, thereby leveraging prior knowledge to accelerate optimization. Fusi *et al.* (2018) follow instead a probabilistic matrix factorization approach akin to collaborative filtering in recommender systems, modeling the performance of configurations on datasets – therefore, users and items are “mapped to configurations *and* datasets”). The learned probabilistic representations are then used to guide the search of next configurations to try via the expected improvement acquisition function.

More sophisticated strategies to exploit parallel compute resources, either with random search or BO, are described in the next chapter.

4.4 Discussion

Model-based methods tend to be somewhat more complex than other HPO techniques, both conceptually and from an implementation standpoint. They require fitting a (probabilistic) surrogate model with relatively few function evaluations, and to update this fit online whenever new observations become available. Moreover, the optimal decision on where to evaluate f next needs to be approximated via an AF depending on the surrogate model, as we saw in Section 4.2.3. Thus, while there is a thriving ecosystem of free academic software, it is not rare that model-based HPO techniques are implemented and proposed as commercial products (Clark and Hayes, 2019; Golovin *et al.*, 2017a; Perrone *et al.*, 2020). The development of model-based techniques requires several design choices to be made, such as the concrete representation

of \hat{f} . Some configurations may be left to be set by the final user. While authors often provide “default values”, the presence of many such parameters may make some model-based techniques less accessible. See, e.g., the list of required inputs for the algorithms proposed by Hazan *et al.* (2018) or Falkner *et al.* (2018).

While the foremost representatives of these type of methods — tied to BO — are classically black-box procedures, model-based techniques may greatly benefit from direct access to the underlying learning algorithm. For instance, several works (Swersky *et al.*, 2014; Klein *et al.*, 2017b; Falkner *et al.*, 2018) use information gathered from partial execution of A to stop unpromising evaluations, which may be additionally resumed at a later stage (Swersky *et al.*, 2014) as we will discuss in more details in the next chapter. Before doing so, we discuss what is possibly the biggest practical limitation of conventional model-base methods: parallelizability.

4.4.1 Decision Making Under Parallelism

In its original formulation BO is sequential, which is unsuitable for many real-world scenarios where parallel computation resources are readily available in order to increase wall-clock time efficiency. A number of extensions of BO to parallel evaluation settings have been proposed. Although more time efficient, these techniques tend to be less sample efficient than sequential BO. The extensions we discuss aim thus to find a diverse set of configurations to preserve sample efficiency as much as possible.

For simplicity, assume a fixed set of W workers available for parallel evaluations. Recall from Section 2.4 that systems may either schedule evaluation jobs synchronously or asynchronously. In the synchronous case, we need to suggest batches of W configurations to be evaluated next. A simple way to do this would be to pick the W top-ranked configurations when optimizing our AF. However, this strategy risks suggesting very similar configurations, and their parallel evaluation may reveal little more information than any single one of them. A heuristic approach to suggest a more diverse batch is to run Thompson sampling W times with independently drawn sample paths (Kandasamy *et al.*, 2018a), but this can be quite expensive. Another idea is to select batch members greedily one at a time. For each configuration placed in the batch, the AF is modified by a local penalty function, ensuring that subsequently

selected points are at least some distance away (González *et al.*, 2016).

Independent of how configurations are suggested, synchronous job scheduling tends to be substantially less efficient than asynchronous. Namely, the time to process a batch is determined by the slowest one, while up to $W - 1$ workers are idle waiting. Such differences in training times are particularly pronounced when searching over parameters determining the model size. In an asynchronous system, whenever a worker becomes available, it can be assigned a new configuration. While BO needs to make single suggestions only (as opposed to batches), this has to be done in the presence of up to $W - 1$ pending evaluations, i.e. jobs which have been started for previously suggested configurations, but did not return a metric value yet. Ignoring pending evaluations runs the same risk of selecting highly redundant configurations in subsequent rounds. Fortunately, dealing with pending evaluations is quite related to ensuring diversity in batch selection, and local penalty methods apply just as well (González *et al.*, 2016; Alvi *et al.*, 2019). Simpler approaches start from the observation that in sequential BO, diversity is enforced by conditioning on observed points. The resulting very low posterior variance shrinks the AF to small values near such points. In the kriging believer heuristic (Ginsbourger *et al.*, 2010; Desautels *et al.*, 2014), we impute the response for each pending evaluation by its current posterior mean and condition on this pseudo-datapoint. A somewhat more principled approach is fantasizing (Snoek *et al.*, 2012; Wilson *et al.*, 2018), where pending target values are integrated out by Monte Carlo: they are sampled several times from the joint posterior over pending configurations, and the AF is averaged over these samples.

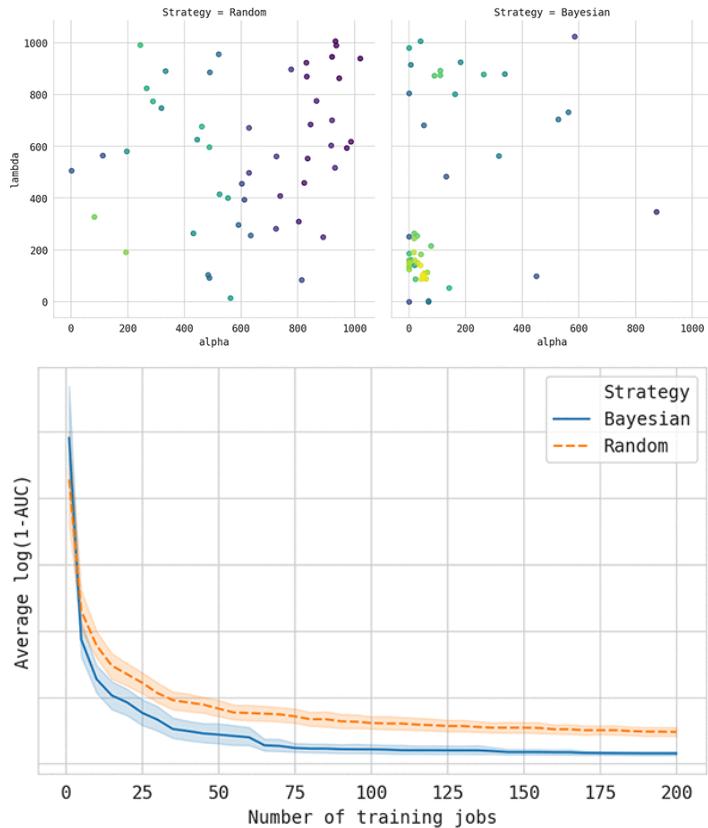


Figure 4.2: Top: Hyperparameters suggested by random search (left) and BO (right). The x -axis and y -axis are the hyperparameter values for α and λ and the color of the points represent the quality of the hyperparameters (yellow: better AUC; violet: worse AUC). Bottom: Best model score obtained so far (y -axis, where lower is better) as more hyperparameter evaluations are performed (x -axis).

5

Multi-Fidelity Methods

One of the main challenges of HPO is that training and validating a machine learning algorithm each time the objective function is evaluated can be expensive. This is particularly true for modern machine learning algorithms applied to large datasets, such as deep neural networks, which can often take hours or days to train. In this chapter, we consider how to obtain and exploit low-fidelity signals that serve as approximations of the objective function. These auxiliary metrics are faster to evaluate and often provide a strong enough signal to guide the optimization process.

We begin with an overview of different types of fidelities in Section 5.1. Section 5.2 explores methods for multi-fidelity optimization based on randomly sampling configurations. Finally, in Section 5.3, we examine multi-fidelity approaches that leverage probabilistic models to guide the search.

5.1 Multi-Fidelity Optimization

As noted in Section 2.5, an important aspect of manual search and hyperparameter tuning is a fast turnaround time. To achieve this, practitioners typically work with a subset of the training data during their initial explorations. They also do not wait for training to fully converge but instead check validation performance after a few iterations or epochs. While these early metrics are

not entirely reliable, they are useful for identifying promising regions of the search space. In this section, we discuss how these manual practices can be transformed into algorithmic tools that enable automated early stopping of underperforming runs.

Multi-fidelity HPO (Li *et al.*, 2017; Klein *et al.*, 2017b; Swersky *et al.*, 2014) extends the HPO formulation from Section 2.2 by taking the budget that we spend to train a hyperparameter configuration into explicit account. We assume access to inexpensive approximations of the true objective function $f(\lambda)$, denoted as $f_b(\lambda)$, where the fidelity level is determined by the budget $b \in [b_{min}, b_{max}]$.

If we set the budget to its maximum value, $b = b_{max}$, we recover the original objective function, $f_{b_{max}}(\lambda) = f(\lambda)$. From this perspective, standard HPO can be seen as a special case of multi-fidelity HPO. Additionally, we assume that the approximation of $f(\lambda)$ improves as b increases. However, the cost $c(\lambda, b)$ associated with evaluating $f_b(\lambda)$ —such as training time—also increases with b .

While we can freely choose b during the optimization process, our primary interest lies in the performance at the full budget b_{max} . More formally, our goal is (still) to find $\lambda^* \in \arg \min_{\lambda \in \Lambda} f_{b_{max}}(\lambda)$, which coincides with the formalization of Chapter 2. Next, we explore commonly used fidelity measures that can be leveraged to accelerate the optimization process.

5.1.1 Iterations

For iterative machine learning algorithms, such as learning in neural networks, we can interpret the number of iterations as the budget and use the performance at each iteration as an approximation of the final performance (Swersky *et al.*, 2014; Li *et al.*, 2017; Jamieson and Talwalkar, 2016). In the case of neural networks, we typically consider epochs as iterations rather than individual mini-batch updates to reduce the overhead of validation error computations.

Figure 5.1 (left) illustrates the validation loss after each epoch of training for different hyperparameter configurations of a multi-layer perceptron. Even after just a few epochs, we can visually distinguish between poorly performing and well-performing configurations. However, to reliably identify the best configuration, it is often necessary to evaluate the top-performing candidates for the full number of epochs.

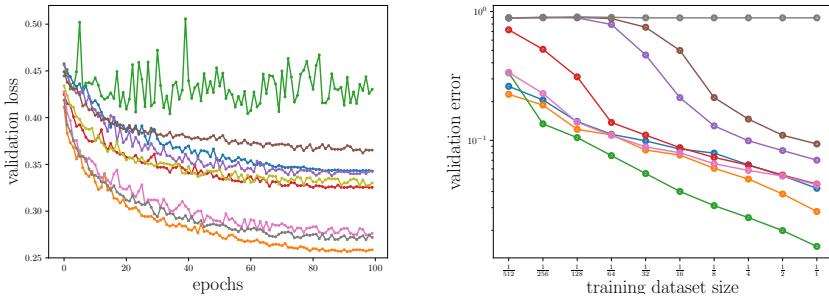


Figure 5.1: Left: Validation loss of different neural networks evaluated after each epoch of training. Right: Validation error of a support vector machine trained on different subsets of the training data. Each color represents a different hyperparameter configuration.

5.1.2 Training Dataset Subsets

The runtime of most machine learning algorithms scales linearly or even super-linearly with the number of training data points. Figure 5.1 (right) shows the validation error of a support vector machine trained on increasing subsets of the MNIST dataset, with each color representing a different hyperparameter configuration. As expected, the validation error decreases as the training dataset size increases. More interestingly, however, the ranking of hyperparameter configurations remains relatively consistent across dataset subsets (Bornshein *et al.*, 2020). In the context of deep neural networks, this observation is related to recent research on scaling laws (Kaplan *et al.*, 2020; Hoffmann *et al.*, 2022), which empirically demonstrate that the training loss of autoregressive transformer models (Vaswani *et al.*, 2017) often follows a power-law function with respect to the size of the training dataset.

In multi-fidelity optimization, we can exploit this phenomenon by interpreting the budget as the training dataset size (Klein *et al.*, 2017b; Swersky *et al.*, 2013). To evaluate $f_b(\lambda)$, we vary only the training dataset size while keeping the validation set fixed to avoid introducing additional noise. Most methods resample the training dataset uniformly at random for each evaluation of $f_b(\lambda)$, though more sophisticated strategies, such as importance sampling, can also be used to select dataset subsets (Ariaifar *et al.*, 2021). Importantly, global statistics such as the class distribution must be preserved to avoid biasing performance estimation.

5.1.3 Cross-validation

As discussed in Section 2.2.1, cross-validation (CV) is a widely used technique for estimating the generalization error of a machine learning method and reducing the risk of overfitting. Rather than relying on a single training/validation split, we divide the dataset into K subsets or folds, using each fold for validation in turn and averaging our target criterion across them. The main drawback of CV is its computational cost, which is K times higher than using a single fold. However, we can approximate it using a multi-fidelity approach by treating the number of folds, $b = 1, \dots, K$, as the budget.

Unlike the previous two fidelities, where performance typically improves with increasing budget, CV does not necessarily exhibit this behavior. Instead, the key assumption is that the variance of the generalization performance estimate decreases as the number of folds increases.

5.2 Methods Based on Random Search

In this section, we review algorithms that leverage multiple fidelities and propose new configurations through random sampling. Random search based multi-fidelity methods were first introduced in the multi-armed bandit (MAB) literature. We discuss how a simple MAB approach can dynamically halt the evaluation of poorly performing configurations. Additionally, we discuss how these methods can be adapted for efficient execution in large-scale distributed settings. The methods covered here assume that the evaluation costs, $c(\lambda, b)$, for any fixed λ are linear (or affine linear) in b .

5.2.1 Successive Halving

Successive halving (SH) (Karnin *et al.*, 2013; Jamieson and Talwalkar, 2016) is an MAB strategy that aims to identify the best hyperparameter configuration from a finite set of configurations. Given the maximum and minimum budgets, b_{max} and b_{min} , for evaluating a single hyperparameter configuration, we first define a set of rung levels:

$$\mathcal{R} = \{b_{min}\eta^k \mid k \in [T]\}.$$

We assume, for simplicity, that $b_{max}/b_{min} = \eta^T$, where $T \in \mathbb{N}$. Here, $\eta \geq 2$ defines the halving constant and determines how aggressively SH prunes

Algorithm 2 Successive-Halving-HPO

Require: Multi-fidelity response function f_b

Require: Initial budget b_{min} , halving constant $\eta \geq 2$, number of rungs T

Require: Set of n candidate configurations $C_0 = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$

```

1: for  $k \leftarrow 1$  to  $T$  do
2:    $r_k \leftarrow b_{min}\eta^k$                                  $\triangleright$  Define rung level
3:    $\mathcal{F} \leftarrow \{f_{r_k}(\lambda) \mid \lambda \in C_{k-1}\}$      $\triangleright$  Evaluate configurations on the rung
4:    $s \leftarrow \lfloor |C_{k-1}|/\eta \rfloor$                    $\triangleright$  Number of configurations to keep
5:    $C_k \leftarrow \arg \underset{\lambda \in C_{k-1}}{\text{top-}s} \mathcal{F}$      $\triangleright$  Keep  $s$  best performing configurations
6: end for

```

configurations.

Algorithm 2 shows pseudocode for SH. Given a set C_0 of n hyperparameter configurations, usually drawn uniformly at random, we evaluate them at the first rung level, i.e., each configuration is evaluated with a budget of b_{min} (Line 2). For convenience, the number of initial configurations is often set to $n = \eta^T$, but this can also be arbitrary. Afterwards, SH promotes $\frac{1}{\eta}$ of the configurations, sorted by their validation performance, to the next higher rung level $b_{min}\eta$ (Line 4). These steps are iterated until we reach the highest rung level, b_{max} . See Figure 5.2 for a visualization.

While conceptually simple, SH has demonstrated strong performance in the literature and has even outperformed more sophisticated Bayesian optimization techniques that operate only on the full budget, in terms of achieving the same performance in less wall-clock time (Jamieson and Talwalkar, 2016; Li *et al.*, 2017). However, its performance relies on a high correlation between the ranking of hyperparameter configurations across rung levels. In other words, the performance of a configuration at the first rung level should be representative of its performance at the highest rung level. Otherwise, configurations that perform best at b_{max} may be discarded too quickly.

5.2.2 Hyperband

The minimum budget b_{min} allocated to each configuration is an important meta-parameter of SH. If b_{min} is too small, we may filter out good configurations based on unreliable low fidelities, while if b_{min} is too large, our advantage over

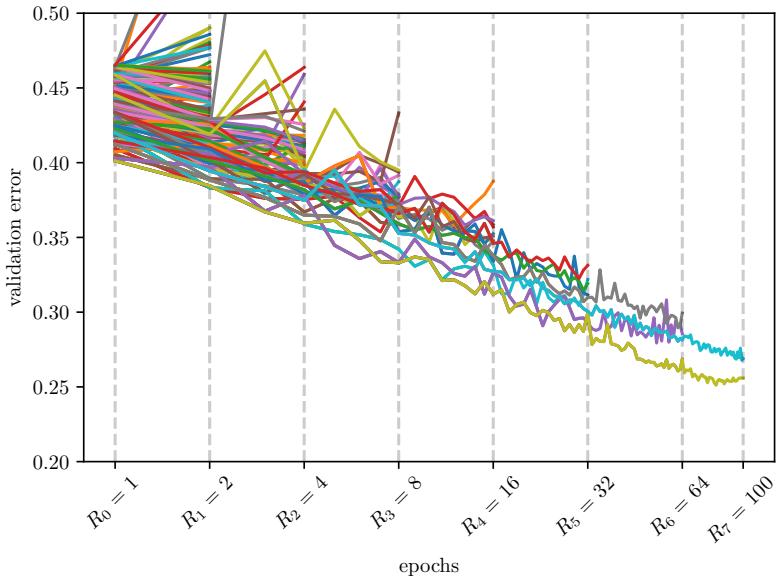


Figure 5.2: Successive halving splits the maximum budget b_{max} into a set of exponentially increasing rungs. Then, in each iteration, successive halving evaluates all configurations on the current rung and promotes the top $1/\eta$ configurations to the next rung. Here, half of the configurations are discarded at every rung, and their learning curve is interrupted. Only two configurations are evaluated for the full budget $b_{max} = 100$.

standard random search may diminish. Hyperband (Li *et al.*, 2017) addresses this issue by running different instances of SH in sequence. The l -th instance (or bracket) has rung levels

$$\mathcal{R}_l = \{b_{\min} \eta^k \mid k \in \{l, \dots, T\}\},$$

so $l = 0$ corresponds to SH with $T + 1$ rung levels, while $l = T$ corresponds to random search with a single level b_{\max} . Each bracket has its own number of initial configurations n_l , which is chosen such that the total budget spent in each bracket is roughly the same. The risk of choosing too small a value for b_{\min} is mitigated at a modest extra expense.

5.2.3 Synchronous Successive Halving

Recall the notation of synchronous versus asynchronous parallelization from Section 2.4, and assume that W workers are available. For synchronous scheduling, we can start $\min(W, M)$ jobs in parallel, where M is the number of free slots remaining in the currently processed rung. Here, $W - M$ workers remain idle, which is particularly wasteful for the higher rungs. For example, if $n = \eta^K$, the highest rung has only a single slot, so $W - 1$ workers will be idle for the maximum budget b_{\max} . The same issue arises for asynchronous scheduling. A remedy will be discussed in Section 5.2.4.

Hyperband inherits the idle worker issue noted above for synchronous scheduling. However, since brackets are independent of each other, we can work on several of them in parallel, which eliminates idle time for asynchronous scheduling.

5.2.4 Asynchronous Successive Halving (ASHA)

As discussed above, neither SH nor Hyperband can fully exploit parallel computation because they require synchronization among workers evaluating hyperparameter configurations for a specific run.

This also means that if a configuration is clearly superior to all others on a rung $r_i \in \mathcal{R}$, it may take a long time before it is promoted to the next rung level r_{i+1} . In fact, all configurations on the current rung r_i must be evaluated before any promotion decisions can be made.

For example, consider Figure 5.3. In a distributed setting, we refer to the evaluation of a hyperparameter configuration as a trial, and in this scenario,

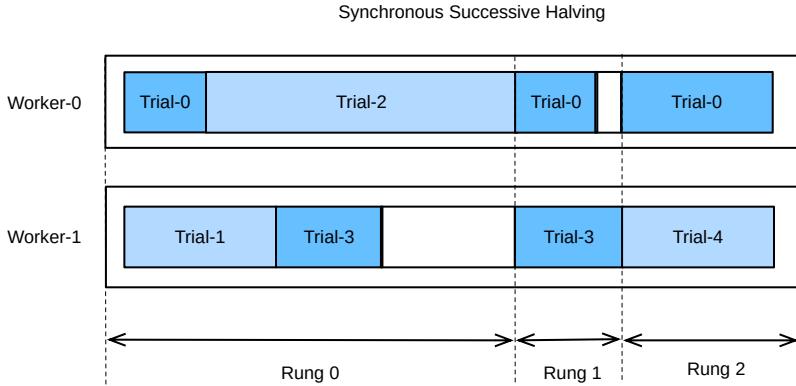


Figure 5.3: Scheduling behavior of Synchronous SH with two workers. A darker shade of blue indicates better-performing trials. Some trials may take significantly longer than others, resulting in idle workers until all trials in the current rung are completed.

we have two workers evaluating four different trials on Rung 0. If Trial-2 takes significantly longer than the others, worker 1 remains idle until all trials on Rung 0 are completed, delaying progress to Rung 1.

These shortcomings are addressed by Asynchronous Successive Halving (ASHA) (Li *et al.*, 2019). In this algorithm, rungs do not have a predefined size but grow dynamically over time. Additionally, promotion decisions are made more aggressively.

For example, assume that $\eta = 2$; a configuration is immediately promoted to the next rung if it ranks in the top 50% of the current rung. In other words, configurations are promoted as early as possible: once a rung has two entries, one of them advances to the next level. Figure 5.4 illustrates this process: as soon as Trial-1 is completed, evaluation of Trial-0 continues on Rung 1—assuming Trial-0 performs better than Trial-1 — rather than evaluating Trial-3 on Rung 0.

After evaluating Trial-3, no configurations can be promoted because, at this point, only two configurations have been evaluated on Rung 0, and one on Rung 1. As a result, the evaluation proceeds with a new trial at the lowest level.

In ASHA, all available computational resources are utilized immediately, ensuring that workers are never idle. Additionally, superior configurations are quickly promoted toward b_{max} without the need to wait for synchronization

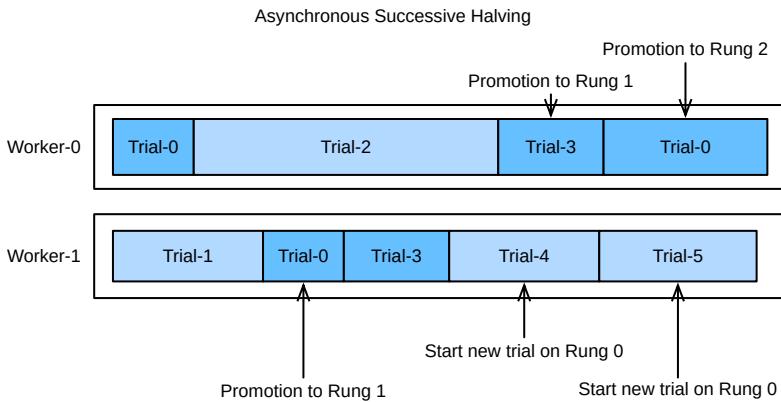


Figure 5.4: Scheduling behavior of ASHA. As in Figure 5.3, a darker shade of blue indicates better-performing trials. ASHA eliminates idle time by immediately promoting a trial to the next rung as soon as it meets the promotion criteria, rather than waiting for all trials in the current rung to be completed.

points. In practice, ASHA often significantly outperforms synchronous SH or Hyperband in terms of wall-clock time (Klein *et al.*, 2020a).

5.3 Model-Based Methods

So far, the multi-fidelity methods we have examined sample hyperparameter configurations purely at random. As we saw in Chapter 4, more sophisticated methods that use a probabilistic model to sample configurations typically require fewer function evaluations to approach the global optimum compared to random search-based approaches. In this section, we will discuss multi-fidelity strategies that leverage one or multiple probabilistic models to guide the search.

5.3.1 Fabolas

Fabolas (Klein *et al.*, 2017b) models the performance $f_b(\lambda)$ of a hyperparameter configuration λ across different subsets $b \in [b_{min}, b_{max}]$ of the training data.

Compared to classical Bayesian optimization (see Chapter 4), we cannot simply optimize standard acquisition functions as described in Section 4.2.3. For example, expected improvement without any further modifications tends

to evaluate mostly where $b = b_{max}$, since, as discussed above, here we expect the validation error to be lower than for smaller b . As a result, this approach would lead to over-exploitation, primarily evaluating configurations at the highest budget.

Instead, Fabolas is based on the entropy search method by Hennig and Schuler (2012), which models the probability

$$p_{\min}(\lambda^* | C) = \mathbb{P}\left[\lambda^* \in \arg \min_{\lambda \in \Lambda} f(\lambda)\right]$$

of λ being the global optimum, where C are the collected observations (see Section 4.2). The key idea of entropy search is to select, in each iteration, the configuration that provides the most information about p_{\min} , which we quantify by computing the KL divergence $KL[p_{\min}(\lambda | C) || \mathcal{U}(\lambda)]$ between p_{\min} and a uniform distribution $\mathcal{U}(\lambda)$, which serves as a prior assuming that each point $\lambda \in \Lambda$ is equally likely to be the optimum. In other words, entropy search selects, in each iteration, the candidate point λ that shifts p_{\min} from a uniform distribution to a highly peaked distribution around the global optimum.

For Fabolas, we model p_{\min} only at the highest budget $b = b_{max}$, since we aim to find $\lambda^* \in \arg \min_{\lambda \in \Lambda} f_{b_{max}}(\lambda)$. To balance information gain with the cost of evaluating λ , Fabolas maintains a second Gaussian process $p(c | \lambda, b)$ to model the wall-clock time $c(\lambda, b)$ required to evaluate $f_b(\lambda)$. In each iteration, Fabolas optimizes the following acquisition function:

$$\alpha(\lambda, b) = \frac{KL[p_{\min}(\lambda | C) || \mathcal{U}(\lambda)]}{\mu_c(\lambda, b)}$$

to select both the configuration λ and budget b that provide the most information about p_{\min} while remaining cost-effective to evaluate, where $\mu_c(\lambda, b) = \mathbb{E}_{p(c|\lambda,b)}[c(\lambda, b)]$ is the expected cost of evaluating λ using budget b .

5.3.2 BO-HB: Bayesian Optimization Hyperband

To combine the sample efficiency of Bayesian optimization with the strong anytime performance of Hyperband, Falkner *et al.* (2018) fit an independent TPE model (see Section 4.2.5) on each rung level $b_i \in \mathcal{R}$ to sample new configurations.

More specifically, once the proposed method BO-HB observes N_{\min} data points at the rung level b_i , it trains two kernel density estimators— $l(\lambda)$ and

$g(\lambda)$ —to model the probability of a configuration λ achieving a validation error lower or higher than a certain threshold, as defined in Eq. (4.14). However, the density estimators are only trained on all data points at the given rung level.

New configurations are then sampled according to the density ratio $l(\lambda)/g(\lambda)$ instead of being chosen uniformly at random. This process continues until N_{\min} observations have been collected at the highest rung level b_{\max} , after which new configurations are sampled exclusively from the model trained at this level.

While the original implementation of TPE by Bergstra *et al.* (2011) used a univariate kernel density estimator, Falkner *et al.* (2018) proposed using a multivariate kernel density estimator to better capture interactions between hyperparameters. To fit such a model, at least $N_{\min} = m + 1$ data points are required, where m is the dimensionality of the search space. Follow up work also considered more sophisticated modelling approaches, such as different GP variants (Klein *et al.*, 2020b; Wistuba *et al.*, 2022) or conformalized quantile regression (Salinas *et al.*, 2023).

Figure 5.5 presents a comparison by Falkner *et al.* (2018) of BO-HB with other approaches for optimizing the hyperparameters of a neural network. Hyperband initially outperforms other methods since it returns configurations much earlier than non-multi-fidelity methods such as random search or TPE. However, compared to model-based approaches like TPE, Hyperband eventually converges toward random search, as it samples configurations uniformly at random and, in later stages, requires evaluating them on larger budgets to improve performance. BO-HB, on the other hand, retains the early speed-up advantage of Hyperband while achieving a performance level similar to TPE in the later stages of the optimization process.

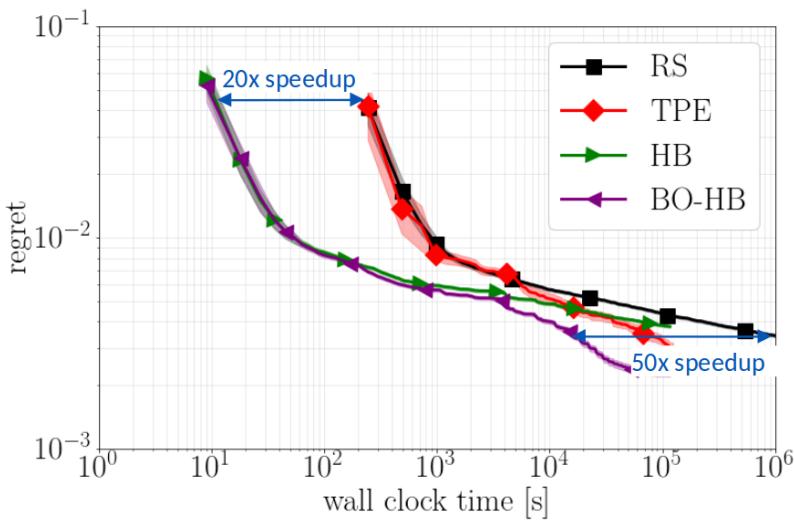


Figure 5.5: Comparison of random search (RS), TPE, Hyperband (HB) and BO-HB over time from Falkner *et al.* (2018). BO-HB keeps the strong anytime performance of Hyperband but because of the model samples better configurations over time.

6

Population-Based Methods

In this chapter, we discuss a class of randomized methods for global optimization that maintain a population of candidate solution, similar to multi-fidelity methods discussed in the previous chapter. Whereas successive halving and related techniques are rooted in the bandit problem and linked to hedging strategies, the methods we explore here draw inspiration from processes of natural evolution (Bäck *et al.*, 1997; Simon, 2013). Evolutionary algorithms date back to the beginning of computer science (Neumann, 1966) and have been used to tackle non-smooth and non-convex problems across many domains. Methods in this class typically rely on function evaluations only. Their application to hyperparameter optimization includes methods like CMA-ES (Hansen and Ostermeier, 1996), population-based training (PBT) (Jaderberg *et al.*, 2017) and particle swarm optimization (Lorenzo *et al.*, 2017), which we review below. Despite the lack of theoretical backing, population-based methods, especially evolutionary algorithms, have enjoyed continued attention in HPO, due to their adaptability to complex search spaces and their strong empirical performances (Loshchilov and Hutter, 2016).

6.1 Evolutionary Algorithms

Traditionally, evolutionary algorithms (EAs) distinguish between evolutionary strategies (ES) and genetic algorithms (GA). GA refers to a class of methods that deal with binary problems and discrete objective functions while ES is concerned with real-valued, continuous objectives.

The terminology of EA techniques is borrowed from biology: the set of points is called population and configurations are called individuals, chromosomes or genotypes. Iterations are called generations and the objective function is called fitness. Typically, at the end of each generation, the fitness of all individuals is observed, making EAs inherently parallel algorithms — traditional implementations usually requiring synchronization. After observing the fitness of the population, a series of transformations is applied to the individuals. Common types of transformations, which collectively implement explore/exploit strategies in this context, include the following one:

- *Selection*, global transformations that prime a new generation, typically by discarding individuals with low fitness and selecting a subset of fitter individuals to become “parents” for the next generation;
- *Recombination* or *crossover*, global transformations that consist in generating new individuals combining slices (e.g. subsets of hyperparameter coordinates) of two or more parents;
- *Mutation*, local modifications designed to escape local optima that introduce random changes in the individual, for instance adding independent Gaussian noise.

Other strategies like *elitism* (i.e., keep the best individuals unchanged) and *niching* (i.e., maintain a diverse population) are often used in conjunction to the explore/exploit strategies outlined above.

We outline this class of methods in Algorithm 3. We can think of individuals and generations as draws from N distributions $\{p_k^j\}$, where N is the population size. Here, the superscripts $j \in [N]$ index individuals and the subscripts $k \in [T]$ index generations, where T is the number of generations. These distributions implement the strategies described above, and evolve at each generation using global information from the current generation. For instance, elitism can be implemented by setting $p_k^j = \delta_{x_k^j}$ for the best performing

configurations, where δ is a Dirac delta distribution. In contrast to random or quasi-random search methods covered in Chapter 3, population-based methods keep in general N distinct distributions rather than a global one that are updated iteratively rather than fixed.

Algorithm 3 Population-Based-HPO

Require: Response function f

Require: Population size N and number of generations T

Require: Initial distributions $\{p_0^j\}_{j \in [N]}$

```

1: for  $k \leftarrow 1$  to  $T$  do
2:   for  $j \leftarrow 1$  to  $N$  do                                ▷ Typically executed in parallel
3:      $\lambda_k^j \sim p_{k-1}^j$                          ▷ Draw an individual
4:     Evaluate  $f(\lambda_k^j)$                       ▷ Evaluate its fitness (if needed)
5:   end for
6:   Update  $p_k^j$  from  $p_{k-1}^j$  using  $\{(\lambda_k^i, f(\lambda_k^i))\}_{i \in [N]}$     ▷ “Evolve”
7: end for

```

Various implementations of ES and GA have been proposed in HPO e.g. for tuning hyperparameters of support vector machines (Friedrichs and Igel, 2005), for neural architecture search (Real *et al.*, 2019; Miikkulainen *et al.*, 2024) (see Section 8.3 for a discussion on neural architecture search), parameters of games (Kunanusont *et al.*, 2017) and even entire machine learning algorithms (Real *et al.*, 2020). In the next sections, we review three particular implementations that have received notable attention.

6.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES (Hansen and Ostermeier, 1996; Hansen, 2016a; Loshchilov and Hutter, 2016) maintains a global multivariate Gaussian distribution from which it samples individuals for the new generations. That is, the generation at iteration $k + 1$ is drawn according to

$$\lambda_{k+1}^j \sim p_k^j = \mathcal{N}(\mu_k, \sigma_k^2 \Sigma_k) \quad \text{for } j \in [N]$$

where $N \in \mathbb{N}^+$ is the population size, μ_k and Σ_k are the distribution parameters and $\sigma_k > 0$ is an overall standard deviation, akin to a step size. Referring to Algorithm 3, this means in CMA-ES $p_k^j = \mathcal{N}(\mu_k, \sigma_k^2 \Sigma_k)$ for all $j \in [N]$.

The parameters of the distribution are evolved at each iteration following a selection and recombination principle. Suppose $\{\lambda_k^j\}_{j=1}^N$ are sorted in increasing objective value, i.e. $f(\lambda_k^1) \leq f(\lambda_k^2) \leq \dots \leq f(\lambda_k^N)$.¹ Then the mean is updated as a convex combination of the best $1 \leq M \leq N$ points:

$$\mu_{k+1} = \sum_{i=1}^M w_i \lambda_k^i,$$

where the weights w_i are positive and sum to 1. The threshold M and the recombination weights are parameter of the method. Hansen (2016a) suggests $M = N/2$ and $w_i \propto M - i + 1$ as reasonable defaults.

The covariance matrix is adapted similarly by reestimating it from the M best points of the previous generation. That is:

$$\Sigma_k = \sum_{i=1}^M w_i (\lambda_k^i - \mu_k)(\lambda_k^i - \mu_k)^\top \quad (6.1)$$

where the w_i are the recombination weights. In practice the update of Eq. (6.1) could be unreliable especially for small populations and Hansen *et al.* (2003) proposes to maintain a moving average for the covariance matrix. Current implementation of CMA-ES incorporate other strategies to improve robustness and efficiency of the algorithms like adaptation of the step-size σ_k and generalizations of Eq. (6.1) (see Hansen *et al.*, 2019).

6.3 Population-Based Training (PBT)

Jaderberg *et al.* (2017) propose a parallel method that maintains a population of model and hyperparameter pairs that are evolved as the models are trained. The procedure, devised in the context of learning deep neural networks in challenging scenarios like image generation or reinforcement learning, consists in interleaving neural network training (by gradient descent) with applications of evolutionary transformations of both the parameters of the models and the hyperparameter of the learning algorithm. PBT is an online hyperparameter optimization algorithm, as hyperparameters are updated alongside model parameters. In other words, in contrast to many techniques we discussed so

¹We recall we seek to minimize f , hence the first points are the best points.

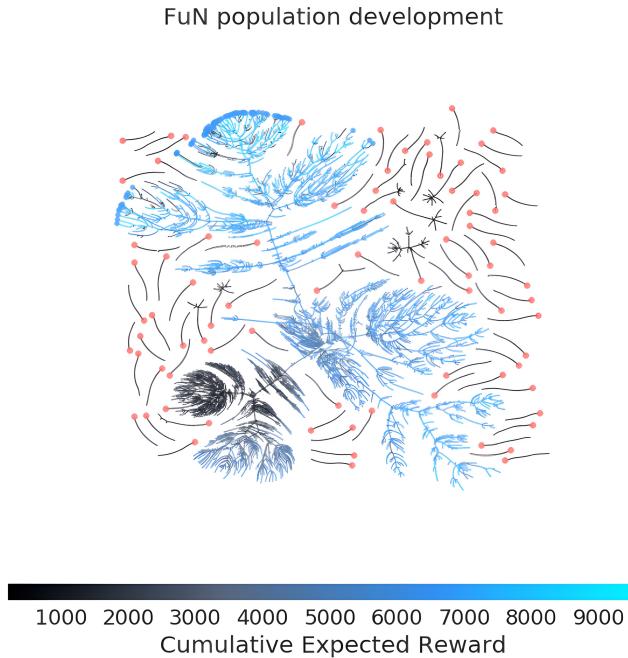


Figure 6.1: Evolution of models trained with PBT for Atari game-playing policies. Red dots represent initial points in the parameter and hyperparameter spaces. Paths represent progressions via parameter training. Branching points represent applications of the exploit and explore operations, namely, a copy of the parameters and hyperparameters of better performing individuals onto worse performing one and a subsequent perturbation of their hyperparameter configurations. The performance score is color-coded, with light blue color representing better performing models. Reproduced from (Jaderberg *et al.*, 2017).

far, PBT yields a hyperparameter sequence (or schedule) rather than a single hyperparameter configuration.²

Suppose one can “decompose” the learning algorithm of interest into a series of stages: $A = A_T \circ \dots \circ A_1$, where each map A_t represents a stage of the learning process that can be warm-started given the outcome of the previous stage. This is particularly straightforward for neural network training: given a space of weights \mathcal{W} , each $A_t : \mathcal{W} \times \Lambda \rightarrow \mathcal{W}$ could be, for instance, an epoch of gradient-based optimization of the network weights. Let $N \in \mathbb{N}$ be a population size and $\{(\theta_0^j, \lambda_0^j)\}_{j \in [N]}$ be a set of initial model weights

²We discuss in more generality hyperparameter schedules and online HPO in Section 8.2.

and hyperparameters, where $\theta_0^j \in \mathcal{W}$ and $\lambda_0^j \in \Lambda$. These pairs represent the “individuals” in PBT and they can be initialized drawing from suitable distributions.

Generations in PBT correspond to stages of the learning algorithm. At each generation $t \in [T]$ the weights are updated as follows:

$$\tilde{\theta}_t^j = A_t(\theta_{t-1}^j, \lambda_{t-1}^j) \quad \text{for } j \in [N].$$

Then, evolutionary transformations are applied to both parameters and hyperparameters before proceeding to the subsequent learning stage, on the basis of a fitness function $\mathcal{M} : \mathcal{H} \rightarrow \mathbb{R}$ (e.g. a validation error, cf. Section 2.2) evaluated on the current model weights.³ Suppose we sort individuals in ascending order so that $\mathcal{M}(\tilde{\theta}_t^1) \leq \mathcal{M}(\tilde{\theta}_t^2) \leq \dots \leq \mathcal{M}(\tilde{\theta}_t^N)$. Jaderberg *et al.* (2017) propose two types of operations:

- *Exploit*: a selection strategy that involves copying the parameters and hyperparameters of better performing individuals onto worst performing ones; for instance, one may set $(\theta_t^N, \lambda_t^N) = (\tilde{\theta}_t^1, \lambda_{t-1}^1)$. Jaderberg *et al.* (2017) suggest to replace the worst-performing 20% with individuals randomly drawn from the best-performing 20%;
- *Explore*: a mutation transformation that brings over the parameter form the last learning stage, but either resamples the hyperparameters from the starting distribution, or perturbs them, e.g. by setting $(\theta_t^j, \lambda_t^j) = (\tilde{\theta}_t^j, \lambda_{t-1}^j + \varepsilon)$ for a perturbation ε such that $\lambda_{t-1}^j + \varepsilon \in \Lambda$.

PBT further uses elitism, leaving the top performing individuals unchanged; that is, setting for some $M \in [N]$, $(\theta_t^i, \lambda_t^i) = (\tilde{\theta}_t^i, \lambda_{t-1}^i)$ for $i \in [M]$. Figure 6.1 present a visualization of a phylogenetic tree of game-playing policy functions for Atari games. The performance metric is the expected episodic reward, and the models deep neural networks trained using the Feudal Networks (FuN) framework (Vezhnevets *et al.*, 2017).

Tao *et al.* (2020) combine PBT with gradient-based optimization of the hyperparameters. The authors propose an “hyperparameter mutation” algorithm that learns a dynamic trade-off between local search via gradient-based

³Note that here we cannot explicitly refer to the response function f , as hyperparameters may change throughout iterations; see also the discussion on hyperparameter schedules in Section 8.2.

hyperparameter optimization and global exploration via PBT. We discussed gradient-based HPO methods in the next chapter.

6.4 Particle Swarm Optimization

Particle swarm optimization (PSO) methods draw inspiration from the social behavior of animals, like ant colonies or bird flocks (Kennedy and Eberhart, 1995), as well as from the evolution of particle trajectories in a physical system. The general idea is that each candidate solution evolves according to a simple equation that weights a momentum term, or velocity, and one or more attractor terms to steer the trajectory toward better performing points.

Assume for simplicity that $\Lambda = \mathbb{R}^m$, and let $\{(\lambda_1^j, v_1^j)\}_{j=1}^N$ be an initial population of hyperparameter and velocity pairs, both drawn randomly from suitable distributions (e.g., uniform distributions). Lorenzo *et al.* (2017) propose an application of PSO to hyperparameter optimization that uses the following equation:

$$\lambda_{k+1}^j = \lambda_k^j + v_k^j, \quad (6.2)$$

$$v_{k+1}^j = \alpha_1 v_k + \alpha_2 r_l \circ (\bar{\lambda}_k^j - \lambda_k) + \alpha_3 r_g \circ (\lambda^* - \lambda_k), \quad (6.3)$$

where $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{R}^+$ are three coefficients, $r_l, r_g \in \mathbb{R}^m$ are two randomly drawn vectors and \circ is the Hadamard product. The configuration $\bar{\lambda}_k^j$ and λ_k^* are, respectively, the best particle and population points found so far, that is,

$$\bar{\lambda}_k^j = \arg \min_{\lambda \in \bar{\Lambda}_k^j} f(\lambda) \quad \text{for } j \in [N],$$

where $\bar{\Lambda}_k^j = \{\lambda_t^j \in \Lambda \mid t \in [k-1]\}$, and

$$\lambda_k^* = \arg \min_{\lambda \in \Lambda_k^*} f(\lambda),$$

where $\Lambda_k^* = \{\lambda_t^j \in \Lambda \mid t \in [k-1], j \in [N]\}$. The second term in the velocity update of Eq. (6.3) can be interpreted as a local memory attractor that pushes the current configuration toward the individual best configuration found so far, while the third term is a global attractor that pushes toward the global best configuration found so far. The randomness introduced by r_l and r_g serves to encourage diversity throughout the search. Referring to Algorithm 3, each individual distribution is implicitly modeled by the Eqs. (6.2)-(6.3).

6.5 Discussion

Despite the lack of theoretical guarantees, population-based methods enjoy broad applicability and strong empirical performances. In addition, this class of methods is designed to readily scale to parallel and distributed compute systems. Although traditional EA and PSO implementation assume synchronicity, one may easily derive asynchronous versions of many algorithms. For instance in PSO, one may replace the global best configuration λ^* with the best known solution to each individual j . However, population-based algorithms require a considerable amount of customization due to the presence of several method parameters, task-specific heuristics and design choices. For instance, it might be challenging to define meaningful mutation or perturbation strategies, or implement effective niching. These considerations may undermine the applicability of these methods to novel learning algorithms and HPO problems, and may preclude accessibility to non-expert users. Besides, they typically require a considerable computational power, benefiting usually from large population sizes.

7

Gradient-Based Optimization

In some cases, the response function, $f(\lambda)$, is differentiable with respect to λ , enabling the application of optimization algorithms that require the gradients of their objective. Assuming $f(\lambda)$ to be differentiable excludes the direct optimization of discrete hyperparameters such as discrete choices, for example, which nonlinear activation function should be used in a layer of a neural network, or integer-valued hyperparameters, such as the dimension of a layer. There is, however, a distinct advantage of gradient-based methods for HPO, namely the possibility of tuning a (very) high-dimensional hyperparameter vector λ , which would be rather difficult to do with model-based algorithms. This opportunity allows us to completely rethink when a variable should be considered as a parameter or as a hyperparameter. Maclaurin *et al.* (2015) suggested considering the initial weights of a neural network as a large hyperparameter vector, an approach that in their own words effectively “blurs the distinction between learning and meta-learning.” Indeed, gradient-based methods reveal some deep connections between HPO and some forms of meta-learning, which we expand on in Section 8.4. For example, the idea of tuning initial weights from several learning tasks was successfully exploited by Finn *et al.* (2017) in their model-agnostic meta learner.

The crux of gradient-based HPO lies in computing approximate gradients

of the response function w.r.t. hyperparameters. In this chapter, we cover three main technical approaches: implicit differentiation (Section 7.2), iterative differentiation (Section 7.3) and fixed-point iterations (Section 7.4). Next, we explore some use cases in Section 7.5. We touch on extensions that allow for the gradient-based optimization of discrete hyperparameters in Section 7.5.1, and conclude with a discussion centered on computational considerations in Section 7.6.

7.1 Preliminaries

At a high level, gradient-based HPO methods start from an initial hyperparameter configuration $\lambda_0 \in \Lambda$ and iteratively update such configuration as follows:

$$\lambda_r = \text{Proj}_\Lambda \left[\lambda_{r-1} - \nu \hat{\nabla} f(\lambda_{r-1}) \right] \quad r = 1, 2, \dots \quad (7.1)$$

where $\hat{\nabla} f$ informally denotes some approximation of the gradient of the HPO objective f , $\nu > 0$ is a step-size and Proj_Λ is the projection onto Λ . Gradient-based optimization of the loss function inside the learning algorithm A is extremely common, especially in deep learning. Hence, to avoid confusion, we will call $\hat{\nabla} f$ the *hypergradient*.

Not every HPO problem is suitable for gradient-based optimization. Recalling the general formulation of the HPO problem in Section 2.2, one underlying assumption shared by all gradient-based HPO methods is that the performance metric \mathcal{M} be differentiable w.r.t. the model output of A . In this chapter, to simplify the notation we take the output of A to be a parameter vector $w \in \mathcal{W} \subseteq \mathbb{R}^d$ and let the performance metric take values in the weight space directly, that is $\mathcal{M} : \mathcal{W} \times \mathcal{D} \rightarrow \mathbb{R}$. Hence,

$$f(\lambda) = \mathcal{M}(w(\lambda), \mathcal{V}) = \mathcal{M}(A(\mathcal{D}, \lambda), \mathcal{V}). \quad (7.2)$$

In the following, it will be useful to explicitly write w as a function of λ , but we will drop \mathcal{D} and \mathcal{V} to simplify the notation.

In order to compute hypergradients, it is also necessary to specify at least some details of the learning algorithm $A(\mathcal{D}, \lambda)$ that appears in (2.4) and that we have mostly treated as an opaque box until now. Gradient-based HPO methods are distinguished mainly by the assumptions they make about A , which then translate into different computational schemes for $\hat{\nabla} f$. Next, we review the main approaches.

7.2 Hypergradients via Implicit Differentiation

Many learning algorithms may be conceptually expressed as solutions to optimization problems:

$$A(\mathcal{D}, \lambda) \in \arg \min_{u \in \mathcal{W}} \mathcal{J}(u, \lambda), \quad (7.3)$$

where $\mathcal{J} : \mathcal{W} \times \Lambda \rightarrow \mathbb{R}$ is an objective function (e.g., the training loss) that depends on the data, and $\mathcal{W} \subseteq \mathbb{R}^d$ is a parameter vector. We have seen a concrete example in Eq. (2.3). If \mathcal{J} is a smooth function w.r.t. u then, the first-order optimality condition implies that

$$\nabla_u \mathcal{J}(u, \lambda) = 0. \quad (7.4)$$

Notably, neural networks and many related deep learning techniques are formulated in these terms. If \mathcal{J} is twice differentiable w.r.t. u at a point $\bar{\lambda} \in \Lambda$ and $\partial_w^2 \mathcal{J}(u, \lambda) \in \mathbb{R}^{d \times d}$ is invertible, we can invoke the implicit function theorem (see, e.g., Krantz and Parks, 2012) to Eq. (7.4) which assures us that (locally) there exist a function $w : U \subset \Lambda \rightarrow \mathcal{W}$ such that $\nabla_u \mathcal{J}(u, \lambda)|_{u=w(\lambda)} = 0$ for all $\lambda \in U$, and $A(\mathcal{D}, \lambda) = w(\lambda)$. We can then locally express the HPO objective as the nested function $f(\lambda) = \mathcal{M}(w(\lambda), \mathcal{V})$. Now, if \mathcal{M} is differentiable, we have that

$$[\nabla f(\lambda)]^\top = \partial_\lambda f(\lambda) = \partial_w \mathcal{M}(w(\lambda), \mathcal{V}) \partial_\lambda w(\lambda) \quad \forall \lambda \in U, \quad (7.5)$$

where, again due to the implicit function theorem,

$$\partial_\lambda w(\lambda) = - \left[\partial_u^2 \mathcal{J}(u, \lambda)|_{u=w(\lambda)} \right]^{-1} \partial_{u,\lambda}^2 \mathcal{J}(u, \lambda)|_{u=w(\lambda)}. \quad (7.6)$$

Equations (7.5) and (7.6) give us the analytic expression of the exact gradient of the HPO objective $f(\lambda)$. Such expression is, however, only implicit. In fact, we almost never have an explicit formula for the map $w(\lambda)$.

Let us denote by w_T an approximate solution of the inner problem, e.g. found with an iterative optimization algorithm. Then, after substituting $w(\lambda)$ with w_T and plugging Eq. (7.6) into Eq. (7.5), one can set $q^\top = -\partial_u \mathcal{M} \left[\partial_u^2 \mathcal{J} \right]^{-1} \in \mathbb{R}^d$ and approximately solve the linear system

$$\partial_u^2 \mathcal{J}(u, \lambda)|_{u=w_T} q = -\nabla_u \mathcal{M}(u)|_{u=w_T}. \quad (7.7)$$

Call $q_{T,K}$ an approximate solution of Eq. (7.7). Then an approximate gradient of the outer objective is given by

$$\nabla_{T,K} f(\lambda) = \left[\partial_{u,\lambda}^2 \mathcal{J}(u, \lambda)|_{u=w_T} \right]^\top q_{T,K}. \quad (7.8)$$

The two possible sources of approximations stem from (i) the solution of the inner problem and (ii) of the linear system in Eq. (7.7). A particular implementation of this computation scheme depends on how Eq. (7.7) is solved. One classic procedure is the conjugate gradient (CG) method (Hestenes and Stiefel, 1952), which features a linear rate of convergence that depends on the conditioning number of the Hessian of the inner objective.

This computational approach has been studied by Larsen *et al.* (1996), Larsen *et al.* (1998), Bengio (2000), Chapelle *et al.* (2002), and Seeger (2007). More recently, Pedregosa (2016) proposed an approximation scheme whereby solutions to the inner problems and to the linear Eq. (7.7) become more precise as the number of HPO iterations r grows, proving convergence for algorithms which feature strongly convex (inner) objectives.

7.3 Hypergradients via Iterative Differentiation

Another approach to computing hypergradients involves explicitly accounting for an iterative training dynamics. Methods based on iterative differentiation further assume that solutions to the learning algorithm problem of Eq. (7.3) are sought (with approximation) by iterating an optimization dynamics for T steps:

$$s_t = \Phi_t(s_{t-1}, \lambda, \mathcal{D}) \quad t = 1, \dots, T. \quad (7.9)$$

We call $s_t = [w_t^\top, v_t^\top]^\top \in \mathbb{R}^d$ the “state” of the optimizer, which contains the current parameters, w_t , and a (possibly empty) set of auxiliary variables v_t such as velocities and moments for algorithms like Adam and RMSProp (Ruder, 2017a). This process yields a solution $w_T = w_T(\lambda)$. The transition maps $\Phi_t : \mathbb{R}^d \times \mathbb{R}^m \times \mathcal{D} \rightarrow \mathbb{R}^d$ are assumed to be differentiable and describe the actual operations associated with a specific optimizer. Typically, Φ_t depends explicitly on the current minibatch \mathcal{D}_t (e.g., for stochastic gradient descent) and it internally requires the computation of the inner objective gradients:

$$g_t \doteq \nabla_w \mathcal{J}(w, \lambda, \mathcal{D}_t) \Big|_{w=w_{t-1}}.$$

Note that Φ depends on λ both explicitly and implicitly via s_{t-1} . Also, λ might contain quantities related to the optimizer itself, such as the learning rate. As an example, the dynamics (7.9) for stochastic gradient descent with

momentum are

$$\begin{aligned} v_t &= \mu v_{t-1} + g_t \\ w_t &= w_{t-1} - \eta v_{t-1} \end{aligned} \quad (7.10)$$

where μ and η are the momentum term and the learning rate, and may be included in λ if desired.

In a nutshell, iterative differentiation schemes for gradient-based HPO rely on the application of automatic differentiation (AD) to the composite map $\mathcal{M} \circ \Phi_T \circ \dots \circ \Phi_1$. We briefly survey AD next, and refer the reader to the book by Griewank and Walther (2008) for an in-depth account.

7.3.1 Automatic Differentiation

Automatic (or algorithmic) differentiation (AD) (Griewank and Walther, 2008; Baydin *et al.*, 2017) is the standard tool for computing gradients in deep learning. Suppose we have a generic differentiable function $G : \mathbb{R}^N \mapsto \mathbb{R}^M$ with $(\eta_1, \dots, \eta_M) = G(\xi_1, \dots, \xi_M)$ and we are interested in computing the Jacobian matrix, with entries $\frac{\partial \eta_i}{\partial \xi_j}$. In AD, we begin by describing G as a computational graph (V, E) , which is a directed acyclic graph whose nodes V include inputs ξ_j , outputs η_i , and intermediate variables that are computed by applying elementary functions to inputs or other intermediate variables, that can be scalars, vectors or tensors. Elementary functions are typically atomic operations such as (tensor) sum, (Hadamard) product, and (elementwise) transcendental functions, whose partial derivatives are readily known. Edges E in the graph describe functional dependencies (i.e., they link intermediate variables to the elementary functions they are fed into). Forward propagation, following a topological sort of the graph, allows us to compute $G(\xi_1, \dots, \xi_M)$. Two AD strategies are available, forward and reverse mode, which are based on the definition of local derivatives and a message-passing procedure based on the standard chain rule of calculus.

In forward mode AD, the local derivative of v_k is defined as

$$\dot{v}_k \doteq \frac{\partial v_k}{\partial \xi_j}$$

and can be updated by the recursion

$$\dot{v}_k = \sum_{\ell \in \pi_k} \dot{v}_\ell \frac{\partial v_k}{\partial v_\ell} \quad (7.11)$$

initialized as $\xi_j = 1$, where π_k are v_k 's parents, and iterated following a topological sort of (V, E) . In the above equation, if v_k and v_ℓ are vector-valued, $\frac{\partial v_k}{\partial v_\ell}$ denotes a Jacobian matrix. The desired derivatives can be finally read as η_i . The same propagation allows us to compute the j -th column of the Jacobian and to obtain the whole Jacobian, where the propagation needs to be repeated for each input ξ_j . Assuming elementary functions can be computed in $O(1)$, the total computational cost is $O(N|E|)$. Forward mode is thus advantageous for functions with few inputs and many outputs. If on the other hand $N = O(|E|)$, as it happens for example for the objective function of neural networks, forward mode is as inefficient as computing derivatives numerically. It is therefore not surprising that forward mode is never used to train neural network models (although there is a remarkable exception when training recurrent networks on data streams (Williams and Zipser, 1989), a setting where reverse mode cannot be applied).

In reverse mode, the strategy used by the well known backpropagation algorithm (Rumelhart *et al.*, 1986), the local derivative of v_k , usually called *adjoint*, is defined as

$$\bar{v}_k \doteq \frac{\partial \eta_i}{\partial v_k}$$

and can be updated by the backward recursion

$$\bar{v}_k = \sum_{\ell \in \gamma_k} \bar{v}_\ell \frac{\partial v_\ell}{\partial v_k} \quad (7.12)$$

initialized as $\bar{\eta}_i = 1$, where γ_k are v_k 's children, and iterated following a reversed topological sort of (V, E) . The desired derivatives can be finally read as $\bar{\xi}_j$. The same propagation allows us to compute the i -th row of the Jacobian and to obtain the whole Jacobian, propagation needs to be repeated for each output η_i , for a total computational cost of $O(ME)$. Reverse mode is thus advantageous for functions with many inputs and few outputs. However, since the computation proceeds in a backward way, the memory used to store the values of intermediate nodes v_k cannot be freed until the end of the propagation, yielding a memory cost of $O(V)$.

7.3.2 Computing Hypergradients

We are now ready to apply the principles of AD to the differentiation of the validation error with respect to λ . A simplified view of the computational graph

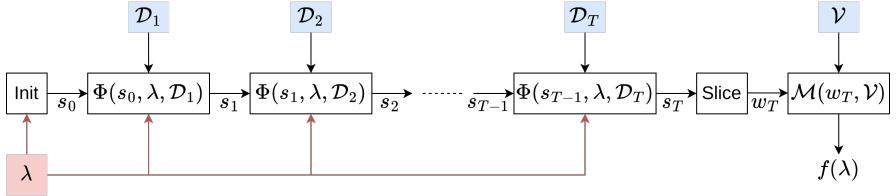


Figure 7.1: Computational graph of the response function for forward and reverse mode hypergradient calculation.

mapping λ into the response $f(\lambda)$ is shown in Figure 7.1, where minibatches \mathcal{D}_t and initial weights w_0 are supplied as given constants¹. Here the number of inputs M is the dimension of the hyperparameter vector and there is a scalar ($N = 1$) output. Forward and reverse calculations are presented in the work of Franceschi *et al.* (2017) and repeated below.

In the case of forward mode, the local variable is \dot{s}_t and is updated by the recursion

$$\dot{s}_t \doteq \frac{\partial s_t}{\partial \lambda} = \left. \frac{\partial \Phi(a, \lambda, \mathcal{D}_t)}{\partial a} \dot{s}_{t-1} \right|_{a=s_{t-1}} + \left. \frac{\partial \Phi(s_{t-1}, b, \mathcal{D}_t)}{\partial b} \right|_{b=\lambda} \quad (7.13)$$

where $\frac{\partial \Phi(a, \lambda, \mathcal{D}_t)}{\partial a}$ and $\frac{\partial \Phi(s_{t-1}, b, \mathcal{D}_t)}{\partial b}$ are $d \times d$ and $d \times m$ Jacobian matrices, respectively. The base step is $\dot{s}_0 = \frac{\partial s_0}{\partial \lambda}$ which is zero unless the hyperparameters directly affect the initial weights (e.g., their range, if randomly initialized, or even their values as in Maclaurin *et al.* (2015)). The desired gradient is finally obtained at the last node:

$$\nabla f(\lambda) = \left. \frac{\partial \mathcal{M}(u, \mathcal{V})}{\partial u} \right|_{u=w_T(\lambda)} \dot{s}_T$$

where w_T is the proper slice of s_T .

In the case of reverse mode, the adjoint \bar{s}_t is initialized as

$$\bar{s}_T = \left. \frac{\partial \mathcal{M}(u, \mathcal{V})}{\partial u} \right|_{u=w_T(\lambda)}$$

and updated as

$$\bar{s}_t \doteq \frac{\partial f}{\partial s_t} = \bar{s}_{t+1} \left. \frac{\partial \Phi(a, \lambda, \mathcal{D}_t)}{\partial a} \right|_{a=s_{t-1}}$$

¹Note that w_0 could be equally well included in λ but here we keep it distinct for clarity.

Since the variable λ is a parent of all s_t , Eq. (7.12) requires us to sum all the contributions:

$$\nabla f(\lambda) = \sum_{t=1}^T \bar{s}_t \left. \frac{\partial \Phi(s_{t-1}, b, \mathcal{D}_t)}{\partial b} \right|_{b=\lambda}. \quad (7.14)$$

It must be remarked the striking similarity between hypergradient calculations and gradient calculations in recurrent neural networks (RNN). Indeed, the computational graphs are identical, where s_t matches the “state” of an RNN, λ the weights, and Φ the state update function. Equations (7.13) and (7.14) closely resemble real-time recurrent learning (Williams and Zipser, 1989) and backpropagation through time (Werbos, 1990), respectively.

7.4 Fixed-Point Methods

Another possible approach that sits in between the iterative and the implicit differentiation schema is given by the so-called fixed-point method. The method is based on the observation that one can rewrite the condition of Eq. (7.4) as a fixed point equation of an appropriate contractive dynamics

$$w(\lambda) = \Phi(w(\lambda), \lambda), \quad (7.15)$$

which could be, for instance, gradient descent $\Phi(w, \lambda) = w - \eta \nabla \mathcal{J}$.² Then, the application of the implicit function theorem to Eq. (7.15) yields to

$$\frac{dw(\lambda)}{d\lambda} = \left(I - \frac{\partial \Phi(w(\lambda), \lambda)}{\partial w} \right)^{-1} \frac{\partial \Phi(w(\lambda), \lambda)}{\partial w \partial \lambda}$$

and, repeating the procedure described above, Eq. (7.7) becomes

$$(I - \partial_w \Phi(w_T, \lambda))^\top q = \nabla \mathcal{M}(w_T). \quad (7.16)$$

Starting from any point $q_{T,0} \in \mathbb{R}^d$, one can iterate

$$q_{T,k} = \partial_w \Phi(w_T, \lambda)^\top q_{T,k-1} + \nabla \mathcal{M}(w_T) \quad k \in [K] \quad (7.17)$$

to find an approximate fixed-point of Eq. (7.16) and then plug $q_{T,K}$ into Eq. (7.8) to obtain a hypergradient.

²As in the iterative case, the dynamics may be defined on an augmented state, for example when using gradient descent with momentum.

This computational scheme emerges in more general cases where Φ is not necessarily an optimization dynamics: Φ may represent a stable recurrent neural network (Miller and Hardt, 2019), a graph neural network (Scarselli *et al.*, 2009) or an equilibrium model (Bai *et al.*, 2019). In these contexts, the fixed-point algorithm is usually known as recurrent backpropagation (Almeida, 1987; Liao *et al.*, 2018). Warm-starting $q_{T,0}$ with the vector obtained at the previous optimization iteration (of λ) may speed up the convergence of Eq. (7.17).

7.5 Use Cases

Gradient-based approaches have mostly found application in academic contexts. Their flexibility, however, allows us to tune very high-dimensional hyperparameter vectors, as showcased in several papers.

One possibility is to attach a hyperparameter to every weight in a deep network. We have already mentioned the idea proposed by Maclaurin *et al.* (2015) regarding the initial values for the weights as hyperparameters. In the same paper, the authors also suggest a closely related opportunity where a regularization penalty is associated to every individual weights. The meta-learning technique presented by Franceschi *et al.* (2018) can be interpreted as treating all weights in the first layers of a deep network as hyperparameters, and optimize the loss on new tasks with respect to the weights in the classification head.

Another direction that has been studied is to attach hyperparameters to data. Maclaurin *et al.* (2015) demonstrated an extreme experiments where image pixels themselves are hyperparameters. In a more realistic scenario, hypergradients have been used to tune the hyperparameters of differentiable data augmentation procedures, yielding significant speedups without performance loss compared to other data augmentation searching methods (Lin *et al.*, 2019; Hataya *et al.*, 2020). A different opportunity is to assign a cost weight to each example in a supervised learning setting so that the overall loss for a training set of n examples is $\sum_{i=1}^n \lambda^{(i)} \mathcal{J}(h(x^{(i)}), y^{(i)})$. Here we have n hyperparameters $\lambda^{(i)}$. Franceschi *et al.* (2017) show how this idea can be exploited to filter out from the training set examples with noisy (or perhaps poisoned) labels. In this regard, it is interesting to note that (although not directly related to HPO) the problem of building training set attacks on linear models can be formulated in terms of bilevel optimization (Mei and Zhu, 2015) and has commonalities

with gradient-based HPO in its mathematical structure.

We finally note here that tuning the whole learning rate schedule has also been tackled with hypergradient approaches (Baydin *et al.*, 2018; Donini *et al.*, 2020; Micaelli and Storkey, 2021; Sharifnassab *et al.*, 2024). This specific problem is covered in more detail in Section 8.2.

7.5.1 Probabilistic Reparameterization of Discrete Hyperparameters

The presence of discrete hyperparameters in the response function — and therefore in the training objective or dynamics — prevents a straightforward application of gradient-based HPO techniques we just described. For some of such hyperparameters, however, it might make sense to reformulate them as random variable that follow some discrete distribution. For instance, consider a discrete binary choice $\lambda \in \{0, 1\}$ (e.g., indicating whether to use an L_1 or L_2 regularizer in Eq. (2.3)). Then, one may define a discrete binary random variable ι as follows:

$$\iota \sim \text{Ber}(\theta), \quad \theta \in [0, 1] \quad (7.18)$$

and reinterpret λ as a drawing of ι , where $\text{Ber}(\theta)$ is the Bernoulli distribution with success parameter θ . Similarly, one may reformulate hyperparameters encoding multiple choices via categorical distributions. Crucially, the distribution parameter θ is now continuous and one may consider optimizing the response function

$$g(\theta) = \mathbb{E}_{\iota \sim \text{Ber}(\theta)}[f(\iota)] = \theta f(1) + (1 - \theta)f(0) \quad (7.19)$$

instead of f , having that $\min_{\theta} g(\theta) = \min_{\lambda} f(\lambda)$. Although Eq. (7.19) might look like a circular argument, the value of such reformulation is unlocked when one computes (approximate) hypergradients via appropriate continuous relaxation or discrete gradient estimators (Niculae *et al.*, 2023). These techniques may be combined with the procedures described in this chapter, making it possible to avoid the exponential cost of computing f on all configurations. This is still an active area of research: successful examples include the optimization of graph structures in transductive learning (Franceschi *et al.*, 2019) and topological order of nodes in DAG learning (Zantedeschi *et al.*, 2023), deep learning architectures (Liu *et al.*, 2019) and dropout coefficients (Gal *et al.*, 2017).

7.6 Discussion

Implicit differentiation methods are “oblivious” to the way the approximate solution w_T has been found. This means that they have a computational advantage in memory over the reverse-mode differentiation as they do not require storing the entire dynamics. If implemented using algorithmic differentiation, they have a run-time advantage over forward-mode differentiation since they only deal with vector quantities (the $q_{T,K}$ ’s). However, for this very reason, they do not directly allow computing the gradient of hyperparameters related to the optimization of the inner problem, such as the initialization mapping or the learning rate. Furthermore, they are also quite sensitive to the proprieties of the inner problem (e.g., for the fixed-point method, the dynamics must be a contraction), and therefore are applicable to a narrower class of problems. A comparative theoretical study between the three main approaches covered in this chapter is presented in the work by Grazzi *et al.* (2020).

Computational disadvantages of plain iterative reverse mode are related to memory requirements. Indeed, the memory complexity is $O(TW)$, with W the number of weights, which can quickly become prohibitive when tuning the hyperparameters associated with modern networks. The problem has been addressed in different ways. Maclaurin *et al.* (2015) suggested to completely avoid the storage of s_t , recomputing it in the backward pass by reversing the learning dynamics of Eq. (7.9). Since the operation is numerically unstable, reconstructed initial weights may be very different from the original ones. As a remedy, they proposed an effective numerical strategy capable of overcoming the loss of information due to finite precision arithmetic. A general technique for reducing the memory requirements in reverse mode AD is *checkpointing* (Griewank and Walther, 2008, Ch. 12), where only a subset of the possible checkpoints is stored, namely w_{jC} for $j = 1, \dots, \lceil T/C \rceil$, being C a given window size. In so doing, one must redo the forward pass from jC to $(j+1)C - 1$ to recover the missing w_t , effectively doubling the computational effort. It is easy to see that the best window size is $O(\sqrt{T})$, which allows us to complete hypergradient calculations with a memory size of $O(W\sqrt{T})$. This technique has been considered by Shaban *et al.* (2019), together with a more radical truncation strategy similar to truncated backpropagation through time (Williams and Peng, 1990), which however introduces further bias in the computation of hypergradients.

8

Further Topics

In the last five chapters we covered the main classes of algorithms for hyperparameter optimization. Here, moving away from particular techniques, we discuss horizontally advanced topics in HPO such as multi-objective and constrained optimization (Section 8.1), online HPO and hyperparameter schedules (Section 8.2) and neural architecture search (Section 8.3). Next, we dedicate Section 8.4 to meta-learning, a closely related topic in the literature, discussing similarities and differences with the typical hyperparameter optimization setting. We conclude the chapter with a brief discussion on transfer of hyperparameters across model sizes, a recent topic that is relevant especially for the tuning of large foundational models (Section 8.5).

8.1 Optimizing for Multiple Objectives

In many real world settings there is not just one, but multiple, potentially conflicting, objectives of interest. For example, when working with hardware-constrained systems one desires predictive models that are accurate and have low resource consumption. Assume one needs to decide between two solutions λ_1, λ_2 . Each individual objective induces a distinct ordinal relationship between the candidates, but while λ_1 might dominate λ_2 in one objective, it might be worse in another. Because there is rarely a single best solution, the

multi-objective hyperparameter optimization (MO) problem is to identify the Pareto front — the set of all non-dominated solutions — to help users make an informed trade-off decision. More formally, consider a function $f : \Lambda \rightarrow \mathcal{Y}$ mapping points in domain Λ to a space \mathcal{Y} , which in the following we consider to be an n -dimensional objective space that we aim to minimize. For two points $\lambda_1, \lambda_2 \in \Lambda$, we say λ_1 weakly dominates λ_2 if $f(\lambda_1)_i \leq f(\lambda_2)_i$ for all objectives $i \in [n]$. We write $\lambda_1 \succeq \lambda_2$ to denote weak domination. Furthermore, λ_1 is said to dominate λ_2 if $\lambda_1 \succeq \lambda_2$ and $f(\lambda_1)_i < f(\lambda_2)_i$ for some objective i . We write $\lambda_1 > \lambda_2$ for domination. The Pareto front \mathcal{P}_f contains all nondominated points in Λ , formally:

$$\mathcal{P}_f = \{\lambda \in \Lambda \mid \nexists \lambda' \in \Lambda : \lambda' > \lambda\}.$$

Since the Pareto front is often infinite, multi-objective optimizers seek a finite approximation set $B \subset \mathcal{Y}$ of objective vectors near the Pareto front. The quality of B can be measured by the dominated hypervolume indicator \mathcal{H} (Zitzler and Thiele, 1998). Given a reference point r , $\mathcal{H}(B)$ equals the hypervolume dominated by B :

$$\mathcal{H}(B) = \text{Vol} \left(\{y \in \mathcal{Y} \subseteq \mathbb{R}^n \mid \exists y' \in B : y' \succeq y \wedge y \geq r\} \right).$$

Computing $\mathcal{H}(B)$ involves partitioning the space into hypercubes, an operation that scales exponentially with the number of objectives n . This exponential scaling poses a key bottleneck for hypervolume-based multi-objective optimizers.

Examples. A popular example of an MO scenario is the case of algorithmic fairness which aims to train accurate models subject to fairness criteria. Prior work in the field has proposed several fairness criteria, whose suitability depends on the application domain, and it has been shown that criteria, commonly referred to as acceptance-based and error-based, can conflict with each other (Friedler *et al.*, 2016; Verma and Rubin, 2018; Kleinberg, 2018). However, obtaining solutions that satisfy various definitions to some extent and expose empirical trade-offs between various definitions can still be important from a societal perspective. Another way to tackle MO scenarios is the case of gradient-free optimization with stochastic constraints. For example, when enhancing the accuracy of a machine learning model, considerations may include constraints on model memory or prediction latency. This occurs

in the practical application of deploying machine learning models on mobile devices, where memory cannot exceed a certain pre-specified threshold, or when models power devices with tight requirements on inference latency. The objective and constraint functions in such cases are real-valued and observed together. Alternatively, in tuning a deep neural network (DNN), one may aim to prevent training failures due to out-of-memory errors. Here, the constraint feedback is binary, indicating whether the constraint is violated, and the objective is not observed under constraint violation. It is common to treat constraints with equal importance to objectives, employing a surrogate model as discussed in Chapter 4. Unfeasible evaluations incur a cost, such as wasted resources or compute node failure, necessitating minimization of their occurrence. However, avoiding the unfeasible region should not compromise convergence to optimal hyperparameters. Often, optimal configurations lie at the boundary of the feasible region, presenting a challenge in balancing the costs of unfeasible evaluations and the need for exploration.

8.1.1 Multi-Objective Optimization

Initial efforts towards MO optimization for HPO (Jin, 2006) were inspired by evolutionary techniques (Yao, 1999; Nolfi and Parisi, 2002; Igel *et al.*, 2005; Friedrichs and Igel, 2005). To mitigate the large computational cost of evaluating a single hyperparameter configuration (e.g., training a DNN), sample-efficient Bayesian optimization (BO) techniques have been introduced (Hutter *et al.*, 2011b; Snoek *et al.*, 2012; Bergstra *et al.*, 2011).

Multi-objective Bayesian optimization (MBO) techniques can be categorized into three classes. First, *scalarization*-based MBO methods map the vector of all objectives to a scalar V (Knowles, 2006; Zhang *et al.*, 2009; Nakayama *et al.*, 2009; Paria *et al.*, 2019; Golovin *et al.*, 2020) and then use conventional single-objective BO techniques. The scalarization is defined by using a weight vector $w \in \mathbb{R}^n$. Some common choices for V are:

- Inner product between objective and weight vectors (also known as random weights): $V_{RW}(f(\lambda), w) = f(\lambda)^\top w$;
- The scalarization used in Knowles, 2006, called ParEGO:

$$V_{\text{ParEGO}}(f(\lambda), w) = \max_{j \in \{1, \dots, n\}} (w_j f(\lambda)_j) + \rho(f(\lambda)^\top w);$$

- The scalarization used in Golovin and Zhang, 2020:

$$V_{\text{Golovin}}(f(\lambda), w) = \min_{j \in \{1, \dots, n\}} (\max(0, f(\lambda)_j/w_j))^n.$$

Note that by optimizing different scalarizations, different solutions along the Pareto front can be obtained. The choice of V influences the subset of the Pareto front that is found, and the right selection might change depending on the task at hand.

A second class of MBO techniques builds on a performance measure of Pareto front approximations, namely the *dominated hypervolume* (Zitzler and Thiele, 1998). Both the expected hypervolume improvement (EHI) (Emmerich *et al.*, 2011) and probability hyper-improvement (PHI) (Keane, 2006) operate by extending their single-objective BO counterparts — expected improvement (Mockus *et al.*, 1978; Jones *et al.*, 1998b) and probability of improvement (Kushner, 1964), respectively. Other methods include step-wise uncertainty reduction (SUR) (Picheny, 2015), smsEGO (Ponweiser *et al.*, 2008; Wagner *et al.*, 2010) and expected maximin improvement (EMMI) (Svenson and Santner, 2010).

Finally, *information-theoretic* MBO approaches aim to select points that reduce uncertainty about the location of the Pareto front. Methods in this class tend to be more sample efficient and scale better with the number of objectives. PAL (Zuluaga *et al.*, 2013) iteratively reduces the size of a discrete uncertainty set. PESMO (Hernández-Lobato *et al.*, 2016) adapts the predictive entropy search (PES) (Henrández-Lobato *et al.*, 2014) criterion. Pareto-frontier entropy search (PFES) (Suzuki *et al.*, 2019) is suitable when dealing with decoupled objectives. MESMO (Belakaria *et al.*, 2019) builds on the max-value entropy-search criterion (Wang and Jegelka, 2017a) and enjoys an asymptotic regret bound. Building on MESMO, two recent works have proposed MF-OSEMO (Belakaria *et al.*, 2020b) and iMOCA (Belakaria *et al.*, 2020a), two multi-fidelity based information-theoretic MBO techniques which internally use multi-fidelity Gaussian processes.

8.1.2 Constrained Hyperparameter Optimization

When optimizing for multiple targets, some objectives may have predefined constraints on their values. For example, one might seek to optimize a given

objective function while enforcing a strict memory constraint. In this section, we focus on this scenario.

Formally, the goal in a constrained optimization scenario is to minimize the target response function $f(\lambda)$, subject to a constraint $c(\lambda) \leq \delta$. Here, we limit our attention to modeling the feasible region by a single function $c(x)$. As the latent constraints are pairwise conditionally independent, an extension to multiple constraints is straightforward, yet notationally cumbersome. Both $f(\lambda) : \Lambda \rightarrow \mathbb{R}$ and $c(x) : \Lambda \rightarrow \mathbb{R}$ are unknown and need to be queried sequentially. The constrained optimization problem is defined as follows:

$$\min_{\lambda \in \Lambda} f(\lambda), \quad \text{s.t. } c(\lambda) \leq \delta \quad (8.1)$$

where $\delta \in \mathbb{R}$ is a threshold on the constraint. For instance, in the case of BO, the latent functions $f(\lambda)$ and $c(\lambda)$ can be conditionally independent in our surrogate model, with different Gaussian process priors placed on them.

We consider two different setups, depending on what information is observed about the constraint. Many existing works (Gardner *et al.*, 2014; Gelbart *et al.*, 2014; Hernández-Lobato *et al.*, 2015) assume that *real-valued feedback* is obtained on $c(\lambda)$, just as for $f(\lambda)$. In this case, both latent functions can be represented as Gaussian processes with Gaussian noise. Unfortunately, this setup does not cover practically important use cases of constrained hyperparameter optimization. For example, if training a deep network fails with an out-of-memory error, we cannot observe the amount of memory requested just before the crash. Rather, we only know that an out-of-memory error occurred, which is a binary outcome. Covering such use cases requires handling *binary feedback* on $c(x)$, even though this is technically more difficult. We can assume an evaluation returns $z_f \sim \mathcal{N}(z_f | f(\lambda), \sigma_f^{-1})$ and $z_c \in \{-1, +1\}$, where $z_c = -1$ for a feasible, $z_c = +1$ for an unfeasible point. Here, z_f is the observed value of the objective function, while z_c is the observed value of the constraint. We never observe the latent constraint function $c(\lambda)$ directly. We assume $z_c \sim \sigma(z_c c(\lambda))$, where $\sigma(t) = \frac{1}{1+e^{-t}}$ is the logistic sigmoid, but other choices are possible. We can then rewrite the constrained optimization problem (8.1) as follows:

$$y_\star = \min_{\lambda \in \Lambda} \{f(\lambda) \mid \mathbb{P}[z_c = 1 | \lambda] = \sigma(c(\lambda)) \leq \sigma(\delta)\}.$$

This formulation is similar to the one proposed by Gelbart *et al.* (2014). The parameter $\sigma(\delta) \in (0, 1)$ controls the size of the (random) feasible region for

defining y_* . Finally, note that in the example of out of memory errors training failures, the criterion observation z_y is obtained only if $z_c = -1$: if a training run crashes, a validation error is not obtained for the queried configuration. Apart from an experiment by Gelbart *et al.* (2014), the case of constrained BO work covering the binary feedback case is covered by Perrone *et al.* (2019).

The most established technique to tackle constrained BO is constrained EI (cEI) (Gardner *et al.*, 2014; Gelbart *et al.*, 2014; Snoek *et al.*, 2015; Letham *et al.*, 2019). A separate regression model is used to learn the constraint function $c(\lambda)$, typically a Gaussian process, and EI is modified in two ways. First, the expected amount of improvement of an evaluation is computed only with respect to the current *feasible* minimum. Second, hyperparameters with a large probability of satisfying the constraint are encouraged by optimizing $cEI(\lambda) = \mathbb{P}[c(\lambda) \leq \delta] \cdot EI(\lambda)$, where $\mathbb{P}[c(\lambda) \leq \delta]$ is the posterior probability of λ being feasible under the constraint model, and $EI(\lambda)$ is the standard EI acquisition function, see Section 4.2.3.

Several issues with cEI have been noted by Hernández-Lobato *et al.* (2015). First, the current feasible minimum has to be known, which is problematic if all initial evaluations are unfeasible. A workaround is to use a different acquisition function, initially focused on finding a feasible point (Gelbart *et al.*, 2014). In addition, the probability of constraint violation is not explicitly accounted for in cEI. A confidence parameter was introduced by Gelbart *et al.* (2014), but it is only used to recommend the final hyperparameter configuration. Another approach was proposed by Hernández-Lobato *et al.* (2015), where PES is extended to the constrained case. Constrained PES (cPES) can outperform cEI and does not require the workarounds mentioned above. However, it is complex to implement, expensive to evaluate, and unsuitable for binary constraint feedback. Drawing from numerical optimization, different generalizations of EI to the constrained case were developed by Picheny *et al.* (2016) and by Ariaifar *et al.* (2019). The authors represent the constrained minimum by way of Lagrange multipliers, and the resulting query selection problem is solved as a sequence of unconstrained problems. An alternative option that does not require expensive computations is the extension of another acquisition function to the constrained case, namely Max-value Entropy Search (Perrone *et al.*, 2019). This acquisition function handles binary constraint feedback, does not require numerical quadrature (Picheny *et al.*, 2016), and does not come with a large set of extra hyperparameters (Ariaifar *et al.*, 2019).

8.2 Hyperparameter Schedules and Online HPO

The vast majority of learning algorithms do not produce a hypothesis in one shot, but rather refine it incrementally over the course of many iterations. In the previous chapters we already discussed algorithms designed around this observation, such as those leveraging multi-fidelity approaches (Chapter 5), and hypergradients (Section 7.3), as well as some evolutionary algorithms like population-based training (Section 6.3). When this is the case, it may be sensible to consider hyperparameter schedules rather than fixed values. One of the most prominent examples in the literature are the ubiquitous learning rate schedules when training neural nets (Bengio, 2012; Loshchilov and Hutter, 2017; Goyal *et al.*, 2017). In this section we overview such setup and briefly discuss how one may tackle the HPO problem as an instance of online optimization.

Mimicking the notation of Section 7.3, let us consider an algorithm $A : \Lambda \rightarrow \mathcal{H}$ as a composition of iterates $\Phi_t : \mathcal{H} \times \Lambda \rightarrow \mathcal{H}$.¹ Such transitions maps need not be necessarily differentiable, although several methods that exploit the iterative view rely on this assumption. This time, we let the hyperparameter configuration also vary as a function of the step. That is, given an initial model $h_0 \in \mathcal{H}$:

$$A(\lambda) = \Phi_T(\cdot, \lambda_T) \circ \Phi_{T-1}(\cdot, \lambda_{T-1}) \circ \cdots \circ \Phi_1(h_0, \lambda_1), \quad (8.2)$$

for some $T > 0$, where we view $\lambda \in \Lambda^T$ as a hyperparameter sequence. The iterations represented by the iterates Φ_t can correspond to different granularities or spans of operations within the training process, such as single stochastic gradient descent steps, entire epochs, or any other sequence of operations.

Hyperparameter iterates may be left “free to float”, in which case one can formulate a related HPO problem following the general setup of Section 2.2, but this time searching for configurations in Λ^T rather than in Λ . This is seldom done in practice due to the complications arising from an expanded search space, few exceptions being (Maclaurin *et al.*, 2015; Wu *et al.*, 2018). More commonly, the iterates are expressed as a function of the step t and a set of other fixed hyperparameters $\lambda' \in \Lambda'$. Learning rate schedules are a primary example. For instance, a triangular schedule (i.e. warmup followed by linear

¹We disregard the dependence of the maps on data, for notational simplicity.

decay) is given by:

$$\eta_t = \begin{cases} \eta_{\max}(t/T_{\max}) & \text{for } t \geq T_{\max} \\ \eta_{\max}(1 - (t - T_{\max})/(T - T_{\max})) & \text{for } T_{\max} < t \leq T \end{cases}$$

where $\lambda' = (\eta_{\max}, T_{\max})$. Once again, when defining a related HPO problem, one may fall back to the setting of Section 2.2, optimizing over Λ' rather than Λ . Along these lines, one may even let the hyperparameter iterates be a function of a state (e.g. the weights of the trainee model at step t), an approach that has been explored e.g. by Hansen (2016b) using reinforcement learning to fit a learning rate adjustment policy or Lorraine and Duvenaud (2018) using hyper-networks and gradient-based HPO.

Online HPO. One main advantage of explicitly taking an iterative view is that it enable recasting the HPO problem as an instance of online optimization (Shalev-Shwartz *et al.*, 2012), albeit a non-standard one. Online HPO techniques attempt to speed up the HPO procedure by finding a performing hyperparameter schedule in one go, during a single execution of the learning algorithm. A key difficulty is that one does not have access to the true objective \mathcal{M} . Indeed, being able to compute (an unbiased estimate of) \mathcal{M} requires terminating the learning algorithm, which would defeat the purpose of online HPO. In this sense, these techniques do not – or rather, cannot – strictly solve Problem (2.4)-(2.5), but rather rely on approximations, heuristics or surrogate objectives. This is akin to what we discussed in Chapter 5, however multi-fidelity techniques only use early observations to take decisions or fit models of the response surface, and do not change hyperparameters during execution. Examples of online HPO include (Orabona and Pál, 2016; Baydin *et al.*, 2018; Luketina *et al.*, 2016; Jaderberg *et al.*, 2017; Lorenzo *et al.*, 2017; MacKay *et al.*, 2019; Donini *et al.*, 2020; Vicol *et al.*, 2021; Micaelli and Storkey, 2021; Chandra *et al.*, 2022), with gradient-based techniques, and especially forward iterative differentiation, being proposed by various authors.

Another stream of recent research focuses on tuning hyperparameters of online learning algorithms, such as contextual bandit (Li *et al.*, 2010) methods like UCB (Auer *et al.*, 2002) or Thomson Sampling (Agrawal and Goyal, 2013). In these algorithms, hyperparameters affect the agent’s exploration-exploitation trade-off and must adapt in real time. Bouneffouf and Claeys

(2020) treat this as a nested bandit problem, tuning exploration parameters online via meta-bandits. Ding *et al.* (2022) generalize this idea with Syndicated Bandits, a multi-layer optimization scheme that tunes several hyperparameters simultaneously, with favorable regret guarantees. Kang *et al.* (2024) further extend this line by framing tuning as a continuum-armed bandit problem, enabling efficient search in continuous spaces. These methods depart from gradient-based HPO discussed above by tightly coupling hyperparameter adaptation with decision-time feedback, offering principled and scalable alternatives tailored to the online nature of bandit learning. Online HPO remains an area of active research.

8.3 Neural Architecture Search

Deep learning has revolutionized the traditional approach of hand-crafting features by enabling automatic feature learning during training. However, the design of neural network architectures still relies heavily on manual effort, involving substantial trial and error.

Following the problem definition described in Section 2.2, we can extend the idea of hyperparameter optimization (HPO) to automate the design of neural architectures — a process known as neural architecture search (NAS) (see also White *et al.* (2023) and Elsken *et al.* (2019) for an overview). NAS is closely related to HPO but focuses specifically on hyperparameters that define the architecture. At the same time, NAS introduces unique challenges not typically encountered in standard HPO tasks. Its search spaces tend to be higher-dimensional, entirely discrete, and often exhibit highly conditional structures (e.g., the number of units in a layer is only relevant if that layer exists, as determined by another hyperparameter). Furthermore, the high computational cost of training modern deep learning models makes NAS especially expensive and calls for even more efficient optimization techniques.

8.3.1 Search Space

A key question in NAS is how to define a search space to parameterize a neural network architecture. An expressive search space might lead to entirely new and innovative architecture but might be also prohibitively expensive to explore. On the other hand, a too restrictive space will lower the chance to improve

upon manually design architectures. We discuss two popular approaches to define NAS search spaces (White *et al.*, 2023): Macro Search Spaces and Cell Search Spaces.

Macro search spaces define the design of neural network architectures at a high level, focusing on configurable choices across the overall structure of the model rather than fine-grained details. Instead of searching over every possible low-level configuration, they assume a fixed architectural topology — such as a residual network or transformer — and explore how the major components within that structure can be customized.

These design decisions might include, for instance, whether to apply layer normalization before or after an operation, how many attention heads to use in each transformer layer, or what type of convolution is used in different stages of a CNN. The goal is to balance architectural flexibility with enough structure to keep the search space manageable. For example, MNASNet (Tan *et al.*, 2019) divides the architecture into blocks and tunes high-level choices like convolution type and kernel size per block.

Cell search spaces focus on smaller computational units — called cells — which are repeatedly stacked to form the full network. Unlike macro search spaces that configure existing architectures, cell-based approaches define a compact building block whose internal structure is optimized and then reused throughout the model allowing to find new topologies.

Typically, each cell is represented as a directed acyclic graph (DAG) where nodes correspond to intermediate feature representations and edges denote operations (e.g., convolutions, pooling). The search algorithm then determines the operations and how they are connected within the cell. This abstraction significantly reduces the size of the search space while retaining enough flexibility to discover high-performing architectures.

For instance, Zoph *et al.* (2018) introduced a cell-based NAS approach where each node in the DAG is assigned a specific operation, such as a 1x1 or 3x3 convolution. To manage complexity and enable spatial downsampling, they distinguished between two types of cells: normal cells, which preserve input dimensions, and reduction cells, which downsample feature maps using larger strides.

8.3.2 Neural Architecture Search as Hyperparameter Optimization

We can phrase NAS as a special type of hyperparameter optimization problem $\min_{\lambda \in \Lambda} f(\lambda)$ (see also Section 2.2), where the inputs λ define the architecture and $f(\lambda)$ represents the validation error after training the corresponding network.

While early work on automatically discovering neural network architectures dates back some time (for example (Kenneth *et al.*, 2009)), Bergstra *et al.* (2013) were the first to tackle this problem definition using TPE (see Section 4.2.5) to automatically design architectures for convolutional neural networks. Arguably, Zoph and Le (2017) coined the term neural architecture search. In their pioneering work, they proposed a controller based on a recurrent neural network to suggest new architectures for convolutional and recurrent neural networks. In each iteration, a set of candidate architectures is sampled and evaluated to collect observations, which are then used to update the controller using reinforcement learning. While they were able to discover architectures that outperformed the state-of-the-art at the time, their proposed method was prohibitively expensive, requiring up to 800 GPUs concurrently. Real *et al.* (2019) proposed an evolutionary approach that maintains a population of architectures (see Chapter 6). New candidates are sampled uniformly at random from the population and mutated along a single dimension. Afterward, the population is updated by removing the last entry.

Inspired by the success of Bayesian optimization in HPO, some NAS approaches proposed Bayesian optimization variants to tackle the often high-dimensional and structured search spaces of NAS. Kandasamy *et al.* (2018b) proposed a Gaussian process variant of Bayesian optimization that measures similarities across architectures using optimal transport approaches. Ru *et al.* (2021) argued for combining a Weisfeiler-Lehman graph kernel with a Gaussian process surrogate model instead, which allows for better interpretability of the final architectures. Similar to HPO, these Bayesian optimization approaches usually tend to be faster in terms of convergence speed than evolutionary algorithms such as regularized evolution (Ying *et al.*, 2019; Ru *et al.*, 2021).

While these approaches led to architectures outperforming state-of-the-art models at the time, the original methods required substantial computational resources since each network was trained from scratch. Nevertheless, these

black-box approaches can be quite effective for moderately sized neural networks and smaller search spaces. Furthermore, they can be easily extended to a multi-objective setting, for example, to simultaneously optimize energy consumption and validation performance. Building on recent work on HPO to accelerate the search process, in this setting we can also use more sophisticated approaches, such as multi-fidelity optimization (see Section 5) or transfer learning (Feurer *et al.*, 2015; Wistuba *et al.*, 2015; Salinas *et al.*, 2020). However, due to the ever-growing size of these networks, even these approaches quickly become infeasible.

8.3.3 Weight-Sharing Based Neural Architecture Search

A more radically different approach to accelerate NAS is the so-called weight-sharing NAS. The idea is to define a single super-network that contains all possible architectures in the finite search space. Each path through the super-network represents a single architecture. In the original work by Pham *et al.* (2018), this method accelerated the search by more than a factor of 1000.

In a similar spirit to gradient-based HPO (see Chapter 7), Liu *et al.* (2019) defined continuous auxiliary parameters for each architectural choice, which are learned by computing the gradient of the validation loss. This resembles the bi-level optimization problem described in Section 2.2; however, here the outer objective optimizes the auxiliary architectural parameters instead of other hyperparameters, and the inner objective optimizes the parameters of the neural network. To reduce computational overhead, Liu *et al.* (2019) proposed DARTS — a gradient-based NAS method — which alternates after each step between the inner and outer objectives. After training the super-network, the best architecture is selected based on the shared weights and then re-trained from scratch. However, several papers (Li and Talwalkar, 2020a; Yang *et al.*, 2020) reported that this formulation heavily relies on the search space and does not yield better results than just randomly sampling architectures.

Two-stage weight-sharing NAS (Cai *et al.*, 2020; Yu *et al.*, 2020) first trains the super-network by updating individual subnetworks in each iteration. After training the super-network, we compute the validation error $f(\lambda)$ by performing a single pass over the validation data. This substantially reduces the computational cost and allows us to apply hyperparameter optimization techniques, such as Bayesian optimization (see Chapter 4) or evolutionary

algorithms. For example, Cai *et al.* (2020) proposed the once-for-all model, which provides a single super-network that allows selecting the optimal architecture with weights for a given hardware, minimizing both latency on the target device and validation error.

Compared to black-box approaches, weight-sharing methods can be applied to higher-dimensional search spaces and neural networks with a larger parameter count. However, they require more substantial changes to the neural network implementation and therefore cannot be straightforwardly applied to existing codebases. Additionally, they are not easily applicable to more sophisticated cell search spaces defined using a context-free grammar (Ericsson *et al.*, 2024). While weight-sharing based methods are popular in the NAS literature, it is not straightforward to apply these techniques to HPO problems, because they only control the capacity of the network and not the training process itself.

8.4 Meta-Learning

We discuss here meta-learning since, from a certain point of view, it shares some interesting similarities to HPO. This section only focuses on this point of view, in the case of supervised learning. For a more general overview of meta-learning, see for example the survey by Hospedales *et al.* (2021) and references therein.

8.4.1 Definition

In meta-learning (also called learning-to-learn), we have a sequence of tasks $\mathcal{T}^1, \mathcal{T}^2, \dots, \mathcal{T}^N$, where each task $\mathcal{T}^j = (p^j, \mathcal{L}^j)$ consists of a distribution $p^j(x, y)$, $x \in \mathcal{X}$, $y \in \mathcal{Y}^j$ over data points and a loss function \mathcal{L}^j . Note that all tasks share the same input space, but their label spaces may differ. Tasks are sampled from a meta distribution \mathcal{P} over tasks. Unlike a traditional supervised algorithm (which, given a dataset, will produce a prediction function h), a meta-learning algorithm outputs a learning algorithm that is hoped to learn a good prediction function when presented with a new (unseen) task $\mathcal{T} = (p, \mathcal{L})$ sampled from the same meta distribution \mathcal{P} . The meta-training phase consists in computing the learning algorithm $A = ML(\mathcal{D}^1, \dots, \mathcal{D}^N)$ where $\mathcal{D}^j \sim p^j$ and ML is the meta-learning algorithm. The meta-test phase, in turn, consists

in computing the prediction function using the new task: $h = A(\mathcal{D})$, with $\mathcal{D} \sim p$. The concept is illustrated in Figure 8.1.

Meta-learning has its own specificity when compared to other settings dealing with several tasks. Compared to multitask learning (Caruana, 1994; Baxter, 2000; Ruder, 2017b), the set of tasks is not fixed in advance. Compared to transfer learning or domain adaptation (Pan and Yang, 2009; Torrey and Shavlik, 2010; Kouw and Loog, 2018), there are several “source” tasks. Typically, we are interested in the scenario where a very small dataset $\mathcal{D} \sim p$ will be available for the new task (few-shot learning) or even a single example per class (one-shot learning). The main insight is that when the tasks are related, as captured by the distribution \mathcal{P} , meta-learning reduces the sample complexity of learning new tasks drawn by the same distribution (Baxter, 2000; Maurer *et al.*, 2016). Note that while HPO typically involves the optimization of a “black-box” function, both the meta-training and the meta-test phases in meta-learning more usually involve the optimization of functions with a known structure, enabling the application of gradient-based techniques, as discussed below.

8.4.2 Meta-learning of Common Representations

One effective approach to meta-learning consists of using past tasks to build strong representations, which can be later used for few-shot learning on the new task. We briefly discuss this approach here, as it is the most closely related to the HPO problem (Franceschi *et al.*, 2018). As in (Baxter, 2000), let us introduce the family of hypothesis spaces $\mathbb{H}^j = \{\mathcal{H}^j\}$ where each $\mathcal{H}^j \in \mathbb{H}^j$ is a set of functions $h_{w,\lambda}^j : \mathcal{X} \mapsto \mathcal{Y}^j$ of the form $h_{w,\lambda}^j = \gamma_w^j \circ \Phi_\lambda$, i.e., obtained by composing a shared feature learning function $\Phi_\lambda : \mathcal{X} \mapsto \mathcal{F}$, parameterized by λ (also called meta-parameter), with a task-specific function $\gamma_w^j : \mathcal{F} \mapsto \mathcal{Y}_j$, parameterized by w . The set \mathcal{F} is a suitably chosen feature space, for example $\mathcal{F} = \mathbb{R}^d$. This type of function composition is common in several deep learning architectures where Φ is in charge of representation learning while γ could be a classification or regression head. The multitask network introduced in (Caruana, 1994) and the feature learning neural network described in (Baxter, 2000) are structured in that way, although all the weights λ and w are optimized jointly, unlike the problem formulated below. Note that we have deliberately used the notation λ for the parameters shared across the

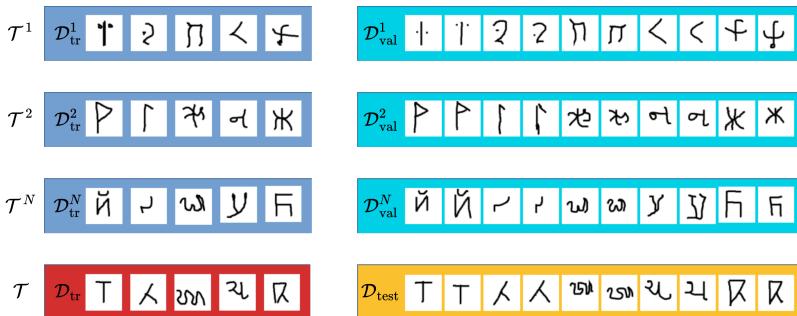


Figure 8.1: Tasks and datasets for meta-learning on Omniglot (Lake *et al.*, 2019). At the end of the meta-training phase, a small training set for the new task \mathcal{T} is provided and performance is evaluated on test data $\mathcal{D}_{\text{test}}$. A similar figure was shown in (Ravi and Larochelle, 2017) for tasks derived from Imagenet.

tasks, to highlight the fact that they play a role similar to hyperparameters. In order to construct a “response function,” for every source task \mathcal{T}^j , $j = 1, \dots, N$, we split the given dataset \mathcal{D}^j into a training set $\mathcal{D}_{\text{tr}}^j$ and a validation set $\mathcal{D}_{\text{val}}^j$ and define

$$f(\lambda) = \sum_{j=1}^N \mathcal{L}^j(w^j, \lambda, \mathcal{D}_{\text{val}}^j).$$

We are now ready to formulate the following bilevel program

$$\min_{\lambda} \quad f(\lambda) \quad (8.3)$$

$$\text{s.t.} \quad w^j \in \arg \min_u \mathcal{L}^j(w^j, \lambda, \mathcal{D}_{\text{tr}}^j) \quad (8.4)$$

where the lower level (or inner) objectives \mathcal{L}^j are empirical risk minimization on each training tasks, and the upper (or outer) objective is the validation error on the same tasks. If Φ and γ are realized by deep learning modules, the inner problem would be solved by some iterative gradient-based algorithm, implying that the solution would only be found approximately. Moreover, in general, the above problem is not guaranteed to have a solution. In (Franceschi *et al.*, 2018) it is shown that a solution is guaranteed to exist if γ is linear (i.e., the classification head consists of a single dense layer) and an algorithm based on hypergradients (using the method described in Section 7.3) is presented to solve the meta-learning problem. A closely related algorithm is proposed

in (Finn *et al.*, 2017), although it is not explicitly formulated as a bilevel program and the emphasis is on fine-tuning to the target task an architecture pretrained on the sequence of previous tasks. A solution based on implicit gradients (see Section 7.2) is offered in (Rajeswaran *et al.*, 2019).

Lastly, while we presented meta-learning as shared feature learning, we note that other tasks’ sharing knowledge mechanisms are possible. Notable examples are biased regularization (Denevi *et al.*, 2018; Denevi *et al.*, 2019; Balcan *et al.*, 2019), in which the meta-parameter λ describes a common mean among the tasks, hyper-network approaches, in which λ is associated to a function that outputs the parameters of a task-specific model (Zhao *et al.*, 2020; Zakerinia *et al.*, 2024), and conditional meta-learning (Denevi *et al.*, 2020; Denevi *et al.*, 2022; Rusu *et al.*, 2018; Wang *et al.*, 2020), in which λ is a conditioning function of side information specific to the task at hand.

8.5 Transfer of Hyperparameters Across Model Scale

A foundation model (Brown, 2020), such as a large language model, is a large-scale, pre-trained machine learning model, typically based on deep learning architectures like transformers (Vaswani *et al.*, 2017), that serves as a general-purpose backbone for various downstream tasks through fine-tuning or prompting. These models mark a cornerstone in artificial intelligence and have led to several breakthroughs in code generation, natural language processing, and common-sense reasoning. However, the expensive pre-training phase requires vast amounts of data and tremendous GPU resources.

While this also depends on a careful selection of hyperparameters, the hyperparameter optimization methods presented here are not directly applicable. Even efficient state-of-the-art multi-fidelity approaches (see Chapter 5) require at least a few training runs on the full budget, but for contemporary LLMs, this is infeasible.

Inspired by a theoretical perspective from tensor programs, a recent line of work (Yang *et al.*, 2022; Everett *et al.*, 2024) approaches this problem from a different angle. While hyperparameters of neural networks, such as the learning rate, change when we increase the width or depth of the network and need to be adjusted accordingly, the idea here is to change the parameterization of neural networks, such that the optimal hyperparameters stay the same when scaling from smaller models to larger models. This allows for finding the

optimal hyperparameters on a small scale and directly transferring them to the largest scale.

The original parameterization proposed by (Yang *et al.*, 2022), called μ -parameterization, initially provided only a theoretical foundation for scaling model width, however they also provided empirical results confirming its effectiveness for depth scaling. Recent work by (Everett *et al.*, 2024) has shown that even hyperparameters such as the learning rate can be transferred across different model widths using the standard parameterization of neural networks by adapting them on a per-layer basis.

While this line of research has shown promising results, it has so far been mostly theoretical and primarily focused on width scaling. Nevertheless, it has the potential to lead to more practical hyperparameter optimization methods that scale efficiently to large foundation models.

9

Hyperparameter Optimization Systems

In this chapter, we provide an overview of different open-source libraries and commercial systems that implement methods discussed in the previous chapters. HPO software landscape changes frequently as new problems, methods and application domains become relevant: the provided list here is by no means exhaustive and only represents a snapshot.

9.1 Open Source Libraries and Frameworks

The Python package Spearmint ([Snoek *et al.*, 2012](#)) is one of the first libraries that provides Bayesian optimization for hyperparameter optimization. It implements Bayesian optimization using Gaussian processes (see Section 4.2). At the same time, Bergstra *et al.* ([2011](#)) provided HyperOpt, offering Bayesian hyperparameter optimization by using the tree Parzen estimator (Section 4.2.5) instead of Gaussian processes (Section 4.2.2). Hutter *et al.* ([2011a](#)) introduced SMAC, which implemented Bayesian optimization strategies using Random Forests. Initially targeting at more generic algorithm configuration problems, such as configuring SAT solvers, it also offered functionality for HPO of machine learning methods. The current version ([Lindauer *et al.*, 2022](#)) is written in Python and also provides more advanced functionalities, such as multi-fidelity optimization (see Chapter 5).

Among the first Python libraries that implemented multi-fidelity HPO, such as Fabolas (Klein *et al.*, 2017b) or Hyperband (Li *et al.*, 2016), was the RoBO library by Klein *et al.* (2017a) and Dragonfly by Kandasamy *et al.* (2019).

While previous libraries, such as Spearmint, allowed to parallelize trials across processes on the same machine, more recent libraries focus more on a large-scale distributed setting to distribute the evaluation of trials across multiple instances or compute nodes. For example, RayTune (Liaw *et al.*, 2018), distributes the evaluation of hyperparameter configurations using the compute framework Ray. Another example is Orion, which supports both high-performance computing infrastructures and containerized environments. Salinas *et al.* (2022) recently presented Syne Tune, which supports different backends that allow to distribute the HPO process either locally or on the cloud. Furthermore, it also provides a backend to simulate the evaluation of expensive objective functions using surrogate models (see Section 9.3). This allows to simulate large-scale asynchronous HPO on a single compute core.

Recent PyTorch implementations of gradient-based hyperparameter optimization algorithms include the libraries HyperTorch (Grazzi *et al.*, 2020), higher (Grefenstette *et al.*, 2019) and Betty (Choe *et al.*, 2023). JaxOpt (Blondel *et al.*, 2022) is a Jax implementation that focuses on implicit differentiation.

Most HPO libraries or frameworks expect the objective function either as function handle, for example in Python, or as a single training script. The search space is usually defined outside the actual training script and hyperparameters are passed as input arguments. Optuna (Akiba *et al.*, 2019) follows a different approach by a so-called define-by-run API that allows users to dynamically define the search space instead of providing it in advance.

9.2 Commercial Systems

Since hyperparameter optimization plays such an integral role in the machine learning workflow, most machine learning platforms, for example *Weights and Biases*, provide some functionality to select hyperparameters. There are also several commercial systems specific for automated hyperparameter optimization. For example, Google Vizier (Golovin *et al.*, 2017b) is an internal service that contributed to several Google products and research efforts. Song *et al.* (2022) also introduces an open-source version of Google Vizier as a

stand-alone Python package. AWS SageMaker provides a fully automated service called Automated Model Tuner (Perrone *et al.*, 2021) for gradient-free optimization problems, which allows for large-scale HPO leveraging built-in SageMaker functionalities. Another example is Sigopt, which provides optimization-as-a-service for hyperparameter optimization utilizing Bayesian optimization techniques.

9.3 Benchmarking

A key challenge for an empirical driven research field such as HPO to make progress is the thorough evaluation of new approaches and comparison to baseline methods. However, due to the expensive nature of the underlying optimization problem, it is often prohibitively expensive to obtain statistically reliable results including multiple repetitions, which are necessary to characterize HPO algorithms that often come with intrinsic randomness.

To reduce the computational requirements and enable large-scale experimentation, recent work tries to emulate expensive HPO problems by using cheap-to-evaluate approximations of the underlying objective function. Eggensperger *et al.* (2015) introduced the idea of surrogate benchmarks that replace the original benchmark with the predictions of a machine learning model. This model is first trained in an offline step on data collected from the original benchmark. Unlike the probabilistic model used in Bayesian optimization, where data is generated iteratively, all data in this approach is generated simultaneously, and the model is trained only once. While using a machine learning model to approximate the objective function introduces potentially some bias, single function evaluations are now orders of magnitude cheaper, since they only require a single prediction.

Instead of using a machine learning model to approximate the true objective, Klein and Hutter (2019) first discretized the search space of a fully-connected neural network leading to a finite number of hyperparameter configurations and then performed an exhaustive search to collect the performance and runtime of each configuration. This allows to replace the actual training and validation of a hyperparameter configuration by simple table look-ups during the HPO process. To mimic the observation noise that comes with the HPO of neural networks, each HPO configuration was evaluated multiple times. During the HPO process, random entries for each hyperparameter configura-

tion are returned to simulate the observation noise. The idea of tabularized benchmarks has since been extended to neural architecture search (Ying *et al.*, 2019; Dong and Yang, 2020) (see Section 8.3) leading to several new benchmarks. While tabularized benchmarks return for each query truly observed values of the objective function instead of predictions of the model, Pfisterer *et al.* (2022) argued that the discretization of the search space eliminates the local structure of the objective function, leading actually to a higher bias than using surrogate models. The idea of surrogate benchmarks has been extended to define a generative model by Klein *et al.* (2019), which allows sampling an arbitrary set of new benchmarks.

To standardize the interface to different benchmarks, Eggensperger *et al.* (2013) introduced HPOLib, which consolidates different benchmarks from the literature. In follow-up work (Eggensperger *et al.*, 2021), HPOBench containerized these benchmarks to increase reproducibility of benchmarking results. HPO-B (Arango *et al.*, 2021) provides tablularized benchmarks based on OpenML datasets (Vanschoren *et al.*, 2014). In a similar vein, Zimmer *et al.* (2021) introduced LCBench, which provides full learning curves of neural network models trained on OpenML datasets. Salinas *et al.* (2022) also provided a unified interface to tabular and surrogate benchmarks in Syne-Tune, using a fast data format to accelerate the table look-ups. Another suite for surrogate-based benchmarking is YAHPO (Pfisterer *et al.*, 2022), consisting of different scenarios, including combined hyperparameter optimization and algorithm selection.

10

Research Directions

By now, hyperparameter optimization may be considered a mature field with well-established algorithmic solutions and with industry-level software frameworks. It is a crucial ingredient for automated machine learning, an active research area that has even started a conference in 2022. In this monograph, we have reviewed in depth the major classes of methods (random and quasi-random search, model-based, multi-fidelity, population-based, and gradient-based) and highlighted connections to closely related areas such neural architecture search and meta-learning.

Yet, in all classes of algorithms there remain open issues and interesting directions for improvement. In addition, the development of hybrid algorithms has so far been under-explored, but could potentially bring notable benefits in practical applications. We do not discuss these particular aspects here, but rather touch upon some general topics which we feel may become (or rather, are already becoming) broad and important research directions in HPO.

10.1 Response Functions for Unsupervised Learning

HPO requires a well-defined *response function* to be minimized (see Section 2.2), which in turn requires a well-defined performance metric. While suitable metrics are abundant for supervised learning, the same is not true for

unsupervised learning.

A notable example is *anomaly detection*, an area rich of methods, results, and applications (Pang *et al.*, 2021). The anomaly or novelty detection problem essentially consists of identifying the high probability region of the distribution behind the data, so that points outside this region can be detected as outliers. For this problem, Ma *et al.*, 2023 survey a number of internal (i.e., not using any ground truth labels) approaches for assessing outlier detection models and conclude that “none of the existing and adapted strategies would be practically useful.”

A second important example is generative models. In this case the goal is to construct a model of the data distribution such that sampling from that model produces fake data points that are indistinguishable from real data. Again, the literature is rich in terms of models, ranging from early generative adversarial networks (Goodfellow *et al.*, 2014) to normalizing flows (Rezende and Mohamed, 2015) and more recently probabilistic diffusion models (Ho *et al.*, 2020). In the particular case of images, popular metrics used to compare different generative models are the Inception Score (Salimans *et al.*, 2016) and the Fréchet inception distance (FID) (Heusel *et al.*, 2017). These metrics only capture one aspect of the generative process, namely fidelity. However, the problem of assessing the quality of a model is inherently more complicated: it is not sufficient to generate faithful points, we also need to cover all regions where the data distribution is high.

There have been interesting attempts to interpret this perspective in terms of precision and recall (Sajjadi *et al.*, 2018; Simon *et al.*, 2019), where the two supports of real and fake data are estimated and compared. Fake points off the support of real distribution are interpreted as false positives, and real points outside the support of the model’s distribution are interpreted as false negatives. The idea has sparked a number of practically computable metrics (Kynkänniemi *et al.*, 2019; Naeem *et al.*, 2020; Park and Kim, 2023) that however have not been shown to be effective to guide HPO. A unidimensional metric that aims to assess the similarity between the model and the real data distributions is the *overlapping area*. Computing the overlaps between these two distributions requires reliable density estimation, which is however difficult for high dimensional data. A simple trick to overcome the associated curse of dimensionality issue is to consider instead the distribution of intra-set distances (between pairs of real data points) and inter-set distances

(between pairs of fake points). This metric was introduced by Yang and Lerch (2020) for assessing the quality of generative models in symbolic music, but distances in that case are based on domain specific aspects of music such as rhythm or harmony, and cannot be easily extended to other types of data. A closely related metric based on maximum mean discrepancy (O’Bray *et al.*, 2021), popular in the context of graph generative models, has been shown to be complicated due to its strong sensitivity to hyperparameters in the definition of the metric itself, leading to a sort of hyper-hyperparameters.

A notable exception in the realm of unsupervised learning is the case of obtaining disentangled representations, which is the problem of identifying a small number of explanatory factors behind observed data (Bengio *et al.*, 2013; Locatello *et al.*, 2019), such as shape, color and position for image classification problems. For this particular case, evaluation metrics have been developed based on informativeness, separability and inference, and interpretability (Do and Tran, 2019). For this particular problem, a hyperparameter tuning algorithm was developed in (Duan *et al.*, 2019) using ranking scores. Still, the development of general purpose HPO algorithms for unsupervised learning remains an open problem.

10.2 Learning to Optimize Hyperparameter

In Section 8.4, we covered connections between HPO and meta-learning from a problem-setting point of view, highlighting the connections between meta-parameters of a meta-learning algorithm and hyperparameters of a standard algorithm. In addition, one may also think of meta-learning a hyperparameter optimization algorithm, in that the task distribution is made of HPO problems, each described by a distribution over datasets and response functions of one or several standard learning algorithms. Samples from such a task distribution may involve various datasets and learning algorithms and traces of evaluations of hyperparameter configurations and their response value (e.g., the validation error). The goal of the meta-learner in this context would be to learn a general strategy for efficiently searching the hyperparameter space based on data collected while optimizing hyperparameters across many different problems.

Early attempts use meta-data to warm start model-based methods, or learn loss curves; see (Hutter *et al.*, 2019, Chapter 2) for a survey. However, more recent works take this perspective one step further, motivated by successes of

transformer-based language modeling and learning to optimize, a sub-field of meta-learning (Chen *et al.*, 2022a). For example, Chen *et al.*, 2022b proposed using a large language model to directly generate promising hyperparameter configurations based on a natural language description of the dataset and learning algorithm. The language model is fine-tuned on a corpus of hyperparameter optimization traces, allowing it to capture complex relationships between hyperparameters and adapt its search strategy to the problem at hand.

The usage of rich meta-data, past experience, and the capacity to leverage similarities across standard learning algorithms may hold the promise of discovering HPO algorithm that are more efficient than classic approaches. However, significant challenges remain in developing truly general and robust meta-learning for HPO, including handling the diversity of problem types, scalability to high-dimensional spaces, and transferability across vastly different domains and algorithms.

10.3 HPO and Foundation Models

The emergence of foundation models (Bommasani *et al.*, 2021) as a new frontier for machine learning applications and research opens up a series of novel scenarios and challenges for hyperparameter optimization.

Zero-shot (Wei *et al.*, 2021) and in-context learning (Brown, 2020) have become common paradigms for utilizing large language models (LLM), both of which require choosing context (or prompt) templates and in-context examples. LLMs' performances at downstream tasks may be sensitive to such choices. These choices are often carried out manually by inspecting few rounds of interactions (prompt engineering). However, both problems can be formulated as hyperparameter optimization tasks, whereby the hyperparameters may be discrete choices over in-context examples or prompt templates, or even entire sequences over tokens. The resulting search space may be large and invariably discrete, and the response functions often involve several potentially expensive and potentially noisy calls to the LLM, which render the resulting HPO problems particularly challenging. Initial attempts include AutoPrompt (Shin *et al.*, 2020), a gradient-based approach (see Chapter 7) that optimizes for discrete “trigger” tokens; Self-Adaptive In-Context Learning (Wu *et al.*, 2022), which proposes a selection-then-rank framework for selecting instance-dependent examples; TRIPLE (Shi *et al.*, 2024) which tackles template optimization

under a best arm identification angle (algorithms proposed include successive halving; see Section 5.2.1); HbBoPs (Schneider *et al.*, 2024), which leverages a structure-aware Gaussian Process surrogate combined with Hyperband to efficiently select prompts in a black-box LLM setting; Black-box Discrete Prompt Learning (Diao *et al.*, 2022) which uses a policy gradient estimator to optimize the prompt templates, among others. Further research is needed to study more efficient and scalable methods for optimizing prompts and in-context examples for large language models. Advances in reinforcement learning and gradient-based approaches with discrete gradient estimators (see Section 7.5.1) may prove useful in these directions.

In addition, the emergence of agents and agent networks where various LLMs interact and act by accessing to external tools (Schick *et al.*, 2023; Yang *et al.*, 2024) provides another compelling application ground for HPO, in particular considering that many applications leverage closed-source LLMs queried through APIs. These agent-based systems introduce new hyperparameters to optimize, such as the selection and configuration of external tools available to the agents, including documentation, output formatting and error reporting, as well as the communication protocols and the interactions graph between agents — see, e.g., GPTSwarm (Zhuge *et al.*, 2024). Optimizing these hyperparameters presents unique challenges due to the complex interactions between agents, tools, and the underlying LLMs. Traditional HPO methods may need to be adapted or extended to handle the dynamic and potentially non-stationary nature of these systems. Furthermore, the use of closed-source LLMs via APIs introduces additional constraints, such as rate limits and usage costs, which must be considered in the optimization process.

Acknowledgements

The work of PF and MP was partially supported by the European Union NextGenerationEU and the Italian National Recovery and Resilience Plan through the Ministry of University and Research (MUR), under Projects CAI4DSA (CUP B13C23005640006) and PE0000013 (CUP J53C22003010006), respectively.

References

- Agrawal, S. and N. Goyal. (2013). “Thompson sampling for contextual bandits with linear payoffs.” In: *Proceedings of the 30th International Conference on Machine Learning*. PMLR. 127–135.
- Akaike, H. (1974). “A New Look at the Statistical Model Identification.” *IEEE Transactions on Automatic Control*. 19(6): 716–723. doi: [10.1109/TAC.1974.1100705](https://doi.org/10.1109/TAC.1974.1100705).
- Akiba, T., S. Sano, T. Yanase, T. Ohta, and M. Koyama. (2019). “Optuna: A Next-generation Hyperparameter Optimization Framework.” In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
- Almeida, L. B. (1987). “A learning rule for asynchronous perceptrons with feedback in a combinatorial environment.” In: *Proceedings, 1st First International Conference on Neural Networks*. Vol. 2. 609–618.
- Alvi, A., B. Ru, J. Calliess, J. Roberts, and M. Osborne. (2019). “Asynchronous Batch Bayesian Optimisation with Improved Local Penalisation.” In: *International Conference on Machine Learning 36*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. 253–262.
- Arango, S. P., H. Jomaa, M. Wistuba, and J. Grabocka. (2021). “HPO-B: A Large-Scale Reproducible Benchmark for Black-Box HPO based on OpenML.” In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

- Ariafar, S., Z. Mariet, D. Brooks, J. Dy, and J. Snoek. (2021). “Faster and More Reliable Tuning of Neural Networks: Bayesian Optimization with Importance Sampling.” In: *International Conference on Artificial Intelligence and Statistics (AISTATS’21)*.
- Ariafar, S., J. Coll-Font, D. Brooks, and J. Dy. (2019). “ADMMBO: Bayesian Optimization with Unknown Constraints using ADMM.” *Journal of Machine Learning Research*. 20(123): 1–26.
- Arlot, S. and A. Celisse. (2010). “A Survey of Cross-Validation Procedures for Model Selection.” *Statistics Surveys*. 4(none): 40–79. ISSN: 1935-7516. doi: [10.1214/09-SS054](https://doi.org/10.1214/09-SS054). URL: <https://projecteuclid.org/journals/statistics-surveys/volume-4/issue-none/A-survey-of-cross-validation-procedures-for-model-selection/10.1214/09-SS054.full>.
- Auer, P., N. Cesa-Bianchi, and P. Fischer. (2002). “Finite-time analysis of the multiarmed bandit problem.” *Machine learning*. 47: 235–256.
- Bäck, T., D. B. Fogel, and Z. Michalewicz. (1997). “Handbook of evolutionary computation.” *Release*. 97(1): B1.
- Bai, S., J. Z. Kolter, and V. Koltun. (2019). “Deep equilibrium models.” In: *Advances in Neural Information Processing Systems*. 688–699.
- Balcan, M.-F., M. Khodak, and A. Talwalkar. (2019). “Provable Guarantees for Gradient-Based Meta-Learning.” In: *International Conference on Machine Learning*. 424–433.
- Baxter, J. (2000). “A model of inductive bias learning.” *Journal of Artificial Intelligence Research*. 12: 149–198.
- Baydin, A. G., R. Cornish, D. Martinez-Rubio, M. Schmidt, and F. Wood. (2018). “Online Learning Rate Adaptation with Hypergradient Descent.” In: *6th International Conference on Learning Representations*. URL: <https://openreview.net/forum?id=BkrsAzWAb>.
- Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. (2017). “Automatic differentiation in machine learning: a survey.” *Journal of Machine Learning Research*. 18: 1–43. url: <http://jmlr.org/papers/v18/papers/v18/17-468.html>.
- Belakaria, S., A. Deshwal, and J. R. Doppa. (2019). “Max-value Entropy Search for Multi-Objective Bayesian Optimization.” In: *Advances in Neural Information Processing Systems*. 7823–7833.

- Belakaria, S., A. Deshwal, and J. R. Doppa. (2020a). “Information-Theoretic Multi-Objective Bayesian Optimization with Continuous Approximations.” *arXiv preprint arXiv:2009.05700*.
- Belakaria, S., A. Deshwal, and J. R. Doppa. (2020b). “Multi-Fidelity Multi-Objective Bayesian Optimization: An Output Space Entropy Search Approach.” In: *AAAI*. 10035–10043.
- Bengio, Y. (2000). “Gradient-based optimization of hyperparameters.” *Neural computation*. 12(8): 1889–1900.
- Bengio, Y. (2012). “Practical recommendations for gradient-based training of deep architectures.” In: *Neural networks: Tricks of the trade*. Springer. 437–478.
- Bengio, Y., A. Courville, and P. Vincent. (2013). “Representation Learning: A Review and New Perspectives.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 35(8): 1798–1828. URL: <https://ieeexplore.ieee.org/document/6472238>.
- Benmeziane, H., K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang. (2021). “A Comprehensive Survey on Hardware-Aware Neural Architecture Search.” *arXiv preprint arXiv:2101.09336*.
- Bergstra, J. and Y. Bengio. (2012). “Random search for hyper-parameter optimization.” *Journal of Machine Learning Research*. 13(Feb): 281–305. URL: <http://www.jmlr.org/papers/v13/bergstra12a.html>.
- Bergstra, J., D. Yamins, and D. D. Cox. (2013). “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures.” *International Conference on Machine Learning*. 28: 115–123.
- Bergstra, J. S., R. Bardenet, Y. Bengio, and B. Kégl. (2011). “Algorithms for hyper-parameter optimization.” In: *Advances in Neural Information Processing Systems*. 2546–2554.
- Birattari, M. (2009). *Tuning Metaheuristics - A Machine Learning Perspective. Studies in Computational Intelligence*. Springer-Verlag.
- Birattari, M., T. Stützle, L. Paquete, and K. Varenntrapp. (2002). “A Racing Algorithm for Configuring Metaheuristics.” In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation. GECCO'02*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 11–18. ISBN: 978-1-55860-878-8.

- Bischl, B., M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer. (2023). “Hyperparameter Optimization: Foundations, Algorithms, Best Practices, and Open Challenges.” *WIREs Data Mining and Knowledge Discovery*. 13(2): e1484. ISSN: 1942-4795. doi: [10.1002/widm.1484](https://doi.org/10.1002/widm.1484). URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1484>.
- Blondel, M., Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. (2022). “Efficient and modular implicit differentiation.” *Advances in neural information processing systems*. 35: 5230–5242.
- Bommasani, R., D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. (2021). “On the opportunities and risks of foundation models.” *arXiv preprint arXiv:2108.07258*.
- Bornschein, J., F. Visin, and S. Osindero. (2020). “Small Data, Big Decisions: Model Selection in the Small-Data Regime.” In: *In Proceedings of the 37th International Conference on Machine Learning (ICML’20)*.
- Bouneffouf, D. and E. Claeys. (2020). “Hyper-parameter tuning for the contextual bandit.” *arXiv preprint arXiv:2005.02209*.
- Bousquet, O., S. Gelly, K. Kurach, O. Teytaud, and D. Vincent. (2017). “Critical hyper-parameters: No random, no cry.” *arXiv preprint arXiv:1706.03200*.
- Box, G. E. P. and K. B. Wilson. (1951). “On the Experimental Attainment of Optimum Conditions.” *Journal of the Royal Statistical Society: Series B (Methodological)*. 13(1): 1–38. ISSN: 2517-6161. doi: [10.1111/j.2517-6161.1951.tb00067.x](https://doi.org/10.1111/j.2517-6161.1951.tb00067.x). URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1951.tb00067.x> (accessed on 12/20/2021).
- Brooks, S. H. (1958). “A discussion of random methods for seeking maxima.” *Operations research*. 6(2): 244–251.
- Brown, T. B. (2020). “Language models are few-shot learners.” *arXiv preprint arXiv:2005.14165*.
- Cai, H., C. Gan, T. Wang, Z. Zhang, and S. Han. (2020). “Once-for-All: Train One Network and Specialize it for Efficient Deployment.” In: *International Conference on Learning Representations (ICLR’20)*.

- Caruana, R. (1994). “Learning Many Related Tasks at the Same Time with Backpropagation.” In: *Advances in Neural Information Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994]*. Ed. by G. Tesauro, D. S. Touretzky, and T. K. Leen. MIT Press. 657–664. URL: <http://papers.nips.cc/paper/959-learning-many-related-tasks-at-the-same-time-with-backpropagation>.
- Chandra, K., A. Xie, J. Ragan-Kelley, and E. Meijer. (2022). “Gradient descent: The ultimate optimizer.” *Advances in Neural Information Processing Systems*. 35: 8214–8225.
- Chapelle, O., V. Vapnik, O. Bousquet, and S. Mukherjee. (2002). “Choosing multiple parameters for support vector machines.” *Machine learning*. 46(1-3): 131–159.
- Chen, T., X. Chen, W. Chen, H. Heaton, J. Liu, Z. Wang, and W. Yin. (2022a). “Learning to optimize: A primer and a benchmark.” *Journal of Machine Learning Research*. 23(189): 1–59.
- Chen, T. and C. Guestrin. (2016). “Xgboost: A scalable tree boosting system.” In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 785–794.
- Chen, Y., M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. Freitas. (2017). “Learning to Learn without Gradient Descent by Gradient Descent.” In: *International Conference on Machine Learning*. 748–756. URL: <http://proceedings.mlr.press/v70/chen17e.html> (accessed on 08/29/2017).
- Chen, Y., A. Huang, Z. Wang, I. Antonoglou, J. Schrittwieser, D. Silver, and N. de Freitas. (2018). “Bayesian Optimization in AlphaGo.” *CoRR*. abs/1812.06855. arXiv: 1812.06855. URL: <http://arxiv.org/abs/1812.06855>.
- Chen, Y., X. Song, C. Lee, Z. Wang, R. Zhang, D. Dohan, K. Kawakami, G. Kochanski, A. Doucet, M. Ranzato, et al. (2022b). “Towards learning universal hyperparameter optimizers with transformers.” *Advances in Neural Information Processing Systems*. 35: 32053–32068.
- Choe, S. K., W. Neiswanger, P. Xie, and E. Xing. (2023). “Betty: An Automatic Differentiation Library for Multilevel Optimization.” In: *The Eleventh International Conference on Learning Representations*. URL: https://openreview.net/forum?id=LV_MeMS38Q9.
- Clark, S. and P. Hayes. (2019). “SigOpt Web page.” <https://sigopt.com>. URL: <https://sigopt.com>.

- Colson, B., P. Marcotte, and G. Savard. (2007). “An overview of bilevel optimization.” *Annals of operations research*. 153(1): 235–256.
- Dempe, S. (2002). *Foundations of bilevel programming*. Springer Science & Business Media.
- Dempe, S. and A. Zemkoho. (2020). “Bilevel optimization.” In: *Springer optimization and its applications*. Vol. 161. Springer.
- Denevi, G., C. Ciliberto, R. Grazzi, and M. Pontil. (2019). “Learning-to-learn stochastic gradient descent with biased regularization.” In: *Proc. 36th International Conference on Machine Learning*. Vol. 97. PMLR. 1566–1575.
- Denevi, G., C. Ciliberto, D. Stamos, and M. Pontil. (2018). “Learning to learn around a common mean.” In: *Advances in Neural Information Processing Systems*. 10169–10179.
- Denevi, G., M. Pontil, and C. Ciliberto. (2020). “The advantage of conditional meta-learning for biased regularization and fine tuning.” *Advances in Neural Information Processing Systems*. 33: 964–974.
- Denevi, G., M. Pontil, and C. Ciliberto. (2022). “Conditional meta-learning of linear representations.” *Advances in Neural Information Processing Systems*. 35: 253–266.
- Desautels, T., A. Krause, and J. Burdick. (2014). “Parallelizing Exploration-Exploitation Tradeoffs in Gaussian Process Bandit Optimization.” *Journal of Machine Learning Research*. 15(1): 3873–3923.
- Devroye, L. (2006). “Nonuniform random variate generation.” *Handbooks in operations research and management science*. 13: 83–121.
- Diao, S., Z. Huang, R. Xu, X. Li, Y. Lin, X. Zhou, and T. Zhang. (2022). “Black-box prompt learning for pre-trained language models.” *arXiv preprint arXiv:2201.08531*.
- Dick, J. and F. Pillichshammer. (2010). *Digital nets and sequences: discrepancy theory and quasi-Monte Carlo integration*. Cambridge University Press.
- Ding, Q., Y. Kang, Y.-W. Liu, T. C. M. Lee, C.-J. Hsieh, and J. Sharpnack. (2022). “Syndicated bandits: A framework for auto tuning hyperparameters in contextual bandit algorithms.” *Advances in Neural Information Processing Systems*. 35: 1170–1181.

- Do, K. and T. Tran. (2019). “Theory and Evaluation Metrics for Learning Disentangled Representations.” In: *International Conference on Learning Representations*. URL: <https://openreview.net/forum?id=HJgK0h4Ywr>.
- Dong, X. and Y. Yang. (2020). “NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search.” In: *International Conference on Learning Representations (ICLR’20)*.
- Donini, M., L. Franceschi, O. Majumder, M. Pontil, and P. Frasconi. (2020). “MARTHE: Scheduling the Learning Rate Via Online Hypergradients.” In: *IJCAI*.
- Duan, S., L. Matthey, A. Saraiva, N. Watters, C. Burgess, A. Lerchner, and I. Higgins. (2019). “Unsupervised Model Selection for Variational Disentangled Representation Learning.” In: *International Conference on Learning Representations*. URL: <https://openreview.net/forum?id=SyxL2TNtvr>.
- Eggensperger, K., F. Hutter, H. Hoos, and K. Leyton-Brown. (2015). “Efficient Benchmarking of Hyperparameter Optimizers via Surrogates.” In: *Proceedings of the 29th National Conference on Artificial Intelligence (AAAI’15)*.
- Eggensperger, K., P. Müller, N. Mallik, M. Feurer, R. Sass, A. Klein, N. Awad, M. Lindauer, and F. Hutter. (2021). “HPOBench: A Collection of Reproducible Multi-Fidelity Benchmark Problems for HPO.” In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
- Eggensperger, K., M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. (2013). “Towards an empirical foundation for assessing bayesian optimization of hyperparameters.” In: *NIPS workshop on Bayesian Optimization in Theory and Practice*. 1–5.
- Elsken, T., J. H. Metzen, and F. Hutter. (2019). “Neural Architecture Search: A Survey.” *Journal of Machine Learning Research*. 20(55): 1–21.
- Emmerich, M. T., A. H. Deutz, and J. W. Klinkenberg. (2011). “Hypervolume-based expected improvement: Monotonicity properties and exact computation.” In: *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE. 2147–2154.
- Ericsson, L., M. Espinosa, C. Yang, A. Antoniou, A. J. Storkey, S. B. Cohen, S. McDonagh, and E. J. Crowley. (2024). “einspace: Searching for Neural Architectures from Fundamental Operations.”

- Everett, K. E., L. Xiao, M. Wortsman, A. A. Alemi, R. Novak, P. J. Liu, I. Gur, J. Sohl-Dickstein, L. P. Kaelbling, J. Lee, and J. Pennington. (2024). “Scaling Exponents Across Parameterizations and Optimizers.” In: *Proceedings of the 41st International Conference on Machine Learning (ICML’24)*.
- Falkner, S., A. Klein, and F. Hutter. (2018). “BOHB: Robust and efficient hyperparameter optimization at scale.” *International Conference on Machine Learning*.
- Feurer, M., J. T. Springenberg, and F. Hutter. (2015). “Initializing bayesian hyperparameter optimization via meta-learning.” In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*.
- Finn, C., P. Abbeel, and S. Levine. (2017). “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks.” In: *Proceedings of the 34th International Conference on Machine Learning*. 1126–1135.
- Fisher, R. A. (1935). *The Design of Experiments*. London: Oliver and Boyd.
- Fogel, L., A. Owens, and M. Walsh. (1965). “Intelligent Decision-Making through a Simulation of Evolution.” *Simulation*. 5(4): 267–279. doi: [10.1177/003754976500500413](https://doi.org/10.1177/003754976500500413).
- Franceschi, L., M. Donini, P. Frasconi, and M. Pontil. (2017). “Forward and reverse gradient-based hyperparameter optimization.” In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 1165–1173.
- Franceschi, L., P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. (2018). “Bilevel Programming for Hyperparameter Optimization and Meta-Learning.” In: *International Conference on Machine Learning*. 1563–1572.
- Franceschi, L., M. Niepert, M. Pontil, and X. He. (2019). “Learning Discrete Structures for Graph Neural Networks.” In: *International Conference on Machine Learning*. 1972–1982.
- Frazier, P., W. Powell, and S. Dayanik. (2008). “A Knowledge-Gradient Policy for Sequential Information Collection.” *SIAM Journal on Control and Optimization*. 47(5): 2410–2439.
- Friedler, S. A., C. Scheidegger, and S. Venkatasubramanian. (2016). “On the (im)possibility of fairness.” *arXiv preprint arXiv:1609.07236*.
- Friedrichs, F. and C. Igel. (2005). “Evolutionary tuning of multiple SVM parameters.” *Neurocomputing*. 64: 107–117.

- Fusi, N., R. Sheth, and M. Elibol. (2018). “Probabilistic matrix factorization for automated machine learning.” In: *Advances in Neural Information Processing Systems*. 3348–3357.
- Gal, Y., J. Hron, and A. Kendall. (2017). “Concrete dropout.” *Advances in neural information processing systems*. 30.
- Gardner, J., M. Kusner, Z. Xu, K. Weinberger, and J. Cunningham. (2014). “Bayesian optimization with inequality constraints.” In: *Proceedings of the International Conference on Machine Learning (ICML)*. 937–945.
- Gelbart, M. A., J. Snoek, and R. P. Adams. (2014). “Bayesian Optimization with Unknown Constraints.” In: *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI)*. 250–259.
- Ginsbourger, D., R. Le Riche, and L. Carraro. (2010). “Kriging is Well-Suited to Parallelize Optimization.” In: *Computational Intelligence in Expensive Optimization Problems*. Springer. 131–162.
- Godbole, V., G. E. Dahl, J. Gilmer, C. J. Shallue, and Z. Nado. (2023). “Deep Learning Tuning Playbook.” URL: http://github.com/google-research/tuning_playbook.
- Golovin, D. *et al.* (2020). “Random Hypervolume Scalarizations for Provable Multi-Objective Black Box Optimization.” *arXiv preprint arXiv:2006.04655*.
- Golovin, D., B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. (2017a). “Google Vizier: A service for black-box optimization.” In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.
- Golovin, D., B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. (2017b). “Google Vizier: A Service for Black-Box Optimization.” In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 1487–1495. doi: [10.1145/3097983.3098043](https://doi.org/10.1145/3097983.3098043). URL: <https://doi.org/10.1145/3097983.3098043>.
- Golovin, D. and Q. Zhang. (2020). “Random hypervolume scalarizations for provable multi-objective black box optimization.” *arXiv preprint arXiv:2006.04655*.
- González, J., Z. Dai, P. Hennig, and N. Lawrence. (2016). “Batch Bayesian Optimization via Local Penalization.” In: *Workshop on Artificial Intelligence and Statistics 19*. Ed. by A. Gretton and C. Robert. 648–657.

- Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. (2014). “Generative Adversarial Networks.” *CoRR*. abs/1406.2661. arXiv: 1406.2661. URL: <http://arxiv.org/abs/1406.2661>.
- Goyal, P., P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. (2017). “Accurate, large minibatch sgd: Training imagenet in 1 hour.” *arXiv preprint arXiv:1706.02677*.
- Grazzi, R., L. Franceschi, M. Pontil, and S. Salzo. (2020). “On the iteration complexity of hypergradient computation.” In: *International Conference on Machine Learning*. PMLR. 3748–3758.
- Grefenstette, E., B. Amos, D. Yarats, P. M. Htut, A. Molchanov, F. Meier, D. Kiela, K. Cho, and S. Chintala. (2019). “Generalized Inner Loop Meta-Learning.” *arXiv preprint arXiv:1910.01727*.
- Griewank, A. and A. Walther. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Second Edition. SIAM.
- Griffiths, R.-R., A. A. Aldrick, M. Garcia-Ortegon, V. Lalchand, and A. A. Lee. (2021). “Achieving Robustness to Aleatoric Uncertainty with Heteroscedastic Bayesian Optimisation.” *Machine Learning: Science and Technology*. 3(1): 015004. ISSN: 2632-2153. doi: [10.1088/2632-2153/ac298c](https://doi.org/10.1088/2632-2153/ac298c). URL: <https://dx.doi.org/10.1088/2632-2153/ac298c>.
- Haibe-Kains, B., G. A. Adam, A. Hosny, F. Khodakarami, L. Waldron, B. Wang, C. McIntosh, A. Goldenberg, A. Kundaje, and et al. (2020). “Transparency and reproducibility in artificial intelligence.” *Nature*. 586(7829): E14–E16. ISSN: 1476-4687. doi: [10.1038/s41586-020-2766-y](https://doi.org/10.1038/s41586-020-2766-y). URL: <http://dx.doi.org/10.1038/s41586-020-2766-y>.
- Hansen, N. (2016a). “The CMA evolution strategy: A tutorial.” *arXiv preprint arXiv:1604.00772*.
- Hansen, N., Y. Akimoto, and P. Baudis. (2019). “CMA-ES/pycma on Github.” Zenodo, DOI:10.5281/zenodo.2559634. doi: [10.5281/zenodo.2559634](https://doi.org/10.5281/zenodo.2559634). URL: <https://doi.org/10.5281/zenodo.2559634>.
- Hansen, N., S. D. Müller, and P. Koumoutsakos. (2003). “Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES).” *Evolutionary computation*. 11(1): 1–18.

- Hansen, N. and A. Ostermeier. (1996). “Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation.” In: *Proceedings of IEEE international conference on evolutionary computation*. IEEE. 312–317.
- Hansen, S. (2016b). “Using deep Q-learning to control optimization hyperparameters.” *arXiv preprint arXiv:1602.04062*.
- Harrison Jr, D. and D. L. Rubinfeld. (1978). “Hedonic housing prices and the demand for clean air.” *Journal of environmental economics and management*. 5(1): 81–102.
- Hataya, R., J. Zdenek, K. Yoshizoe, and H. Nakayama. (2020). “Faster AutoAugment: Learning Augmentation Strategies Using Backpropagation.” In: *Computer Vision – ECCV 2020*. Ed. by A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm. Cham: Springer International Publishing. 1–16. ISBN: 978-3-030-58595-2. doi: [10.1007/978-3-030-58595-2_1](https://doi.org/10.1007/978-3-030-58595-2_1).
- Hazan, E., A. Klivans, and Y. Yuan. (2018). “Hyperparameter optimization: A spectral approach.” *International Conference on Learning Representations*.
- He, X., K. Zhao, and X. Chu. (2021). “AutoML: A survey of the state-of-the-art.” *Knowl. Based Syst.* 212: 106622. doi: [10.1016/j.knosys.2020.106622](https://doi.org/10.1016/j.knosys.2020.106622). URL: <https://doi.org/10.1016/j.knosys.2020.106622>.
- Hennig, P. and C. J. Schuler. (2012). “Entropy search for information-efficient global optimization.” *Journal of Machine Learning Research*. 98888(1): 1809–1837.
- Henrández-Lobato, J. M., M. W. Hoffman, and Z. Ghahramani. (2014). “Predictive Entropy Search for Efficient Global Optimization of Black-Box Functions.” In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1. NIPS’14*. Montreal, Canada: MIT Press. 918–926.
- Hernández-Lobato, D., J. Hernandez-Lobato, A. Shah, and R. Adams. (2016). “Predictive entropy search for multi-objective Bayesian Optimization.” In: *International Conference on Machine Learning*. 1492–1501.
- Hernández-Lobato, J. M., M. A. Gelbart, M. W. Hoffman, R. P. Adams, and Z. Ghahramani. (2015). “Predictive Entropy Search for Bayesian Optimization with Unknown Constraints.” In: *Proceedings of the International Conference on Machine Learning (ICML)*. 1699–1707.

- Hernández-Lobato, J. M., M. W. Hoffman, and Z. Ghahramani. (2014). “Predictive Entropy Search for Efficient Global Optimization of Black-box Functions.” *Tech. rep.* preprint arXiv:1406.2541.
- Hestenes, M. R. and E. Stiefel. (1952). “Methods of conjugate gradients for solving.” *Journal of research of the National Bureau of Standards*. 49(6): 409.
- Heuillet, A., A. Nasser, H. Arioui, and H. Tabia. (2024). “Efficient Automation of Neural Network Design: A Survey on Differentiable Neural Architecture Search.” *ACM Comput. Surv.* 56(11): 270:1–270:36. ISSN: 0360-0300. doi: [10.1145/3665138](https://doi.org/10.1145/3665138). URL: <https://doi.org/10.1145/3665138>.
- Heusel, M., H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. (2017). “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.” In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett. 6626–6637. URL: <https://proceedings.neurips.cc/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html>.
- Ho, J., A. Jain, and P. Abbeel. (2020). “Denoising Diffusion Probabilistic Models.” In: *Advances in Neural Information Processing Systems 33*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin.
- Hoffman, M., B. Shahriari, and N. de Freitas. (2014). “On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning.” In: *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*. 365–374.
- Hoffman, M. D., E. Brochu, and N. de Freitas. (2011). “Portfolio Allocation for Bayesian Optimization.” *UAI*.
- Hoffmann, J., S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre. (2022). “Training Compute-Optimal Large Language Models.” *arXiv:2203.15556 [cs.CL]*.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The MIT Press.
- Hospedales, T., A. Antoniou, P. Micaelli, and A. Storkey. (2020). “Meta-learning in neural networks: A survey.” *arXiv preprint arXiv:2004.05439*.

- Hospedales, T. M., A. Antoniou, P. Micaelli, and A. J. Storkey. (2021). “Meta-Learning in Neural Networks: A Survey.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*: 1–1. ISSN: 1939-3539. doi: [10.1109/TPAMI.2021.3079209](https://doi.org/10.1109/TPAMI.2021.3079209).
- Hutter, F., H. H. Hoos, K. Leyton-Brown, and T. Stuetzle. (2009a). “ParamILS: An Automatic Algorithm Configuration Framework.” *Journal of Artificial Intelligence Research*. 36(Oct.): 267–306. ISSN: 1076-9757. doi: [10.1613/jair.2861](https://doi.org/10.1613/jair.2861). URL: <https://jair.org/index.php/jair/article/view/10628>.
- Hutter, F., H. H. Hoos, and K. Leyton-Brown. (2011a). “Sequential model-based optimization for general algorithm configuration.” In: *International conference on learning and intelligent optimization*. Springer. 507–523.
- Hutter, F., H. H. Hoos, and K. Leyton-Brown. (2011b). “Sequential model-based optimization for general algorithm configuration.” In: *Int. Conf. on Learning and Intelligent Optimization*. Springer. 507–523.
- Hutter, F., H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. (2009b). “An Experimental Investigation of Model-based Parameter Optimisation: SPO and Beyond.” In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. GECCO '09*. New York, NY, USA: ACM. 271–278. ISBN: 978-1-60558-325-9. doi: [10.1145/1569901.1569940](https://doi.acm.org/10.1145/1569901.1569940). URL: <http://doi.acm.org/10.1145/1569901.1569940> (accessed on 05/09/2018).
- Hutter, F., L. Kotthoff, and J. Vanschoren. (2019). *Automated machine learning: methods, systems, challenges*. Springer Nature.
- Hutter, F., J. Lücke, and L. Schmidt-Thieme. (2015). “Beyond Manual Tuning of Hyperparameters.” en. *KI - Künstliche Intelligenz*. 29(4): 329–337. ISSN: 0933-1875, 1610-1987. doi: [10.1007/s13218-015-0381-0](https://doi.org/10.1007/s13218-015-0381-0). URL: <http://link.springer.com/10.1007/s13218-015-0381-0>.
- Igel, C., S. Wiegand, and F. Friedrichs. (2005). “Evolutionary optimization of neural systems: The use of strategy adaptation.” In: *Trends and Applications in Constructive Approximation*. Springer. 103–123.
- Ilievski, I., T. Akhtar, J. Feng, and C. A. Shoemaker. (2017). “Efficient hyperparameter optimization for deep learning algorithms using deterministic RBF surrogates.” In: *Thirty-First AAAI Conference on Artificial Intelligence*.
- Jaderberg, M., V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, et al. (2017). “Population based training of neural networks.” *arXiv preprint arXiv:1711.09846*.

- Jamieson, K. and A. Talwalkar. (2016). “Non-stochastic best arm identification and hyperparameter optimization.” In: *Artificial Intelligence and Statistics*. 240–248.
- Jenatton, R., C. Archambeau, J. González, and M. Seeger. (2017). “Bayesian optimization with tree-structured dependencies.” In: *International Conference on Machine Learning*. PMLR. 1655–1664.
- Jin, Y. (2006). *Multi-objective Machine Learning*. Vol. 16. Springer Science & Business Media.
- Jones, D. R., M. Schonlau, and W. J. Welch. (1998a). “Efficient global optimization of expensive black-box functions.” *Journal of Global optimization*. 13(4): 455–492.
- Jones, D. R., M. Schonlau, and W. J. Welch. (1998b). “Efficient global optimization of expensive black-box functions.” *Journal of Global optimization*. 13(4): 455–492.
- Kandasamy, K., A. Krishnamurthy, J. Schneider, and B. Poczos. (2018a). “Parallelised Bayesian Optimisation via Thompson Sampling.” In: *21st International Conference on Artificial Intelligence and Statistics*. 133–142.
- Kandasamy, K., W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing. (2018b). “Neural Architecture Search with Bayesian Optimisation and Optimal Transport.” In: *Proceedings of the 31th International Conference on Advances in Neural Information Processing Systems (NeurIPS’18)*.
- Kandasamy, K., K. R. Vysyaraju, W. Neiswanger, B. Paria, C. R. Collins, J. Schneider, B. Poczos, and E. P. Xing. (2019). “Tuning Hyperparameters without Grad Students: Scalable and Robust Bayesian Optimisation with Dragonfly.” In: *Journal of Machine Learning Research 2020, Special Issue on Bayesian Optimization*.
- Kang, Y., C.-J. Hsieh, and T. Lee. (2024). “Online continuous hyperparameter optimization for generalized linear contextual bandits.” *TMLR*.
- Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. (2020). “Scaling Laws for Neural Language Models.” *arXiv:2001.08361 [cs.LG]*.
- Karnin, Z., T. Koren, and O. Somekh. (2013). “Almost optimal exploration in multi-armed bandits.” In: *International Conference on Machine Learning 30*. Ed. by A. Globerson and F. Sha. Omni Press.
- Keane, A. J. (2006). “Statistical improvement criteria for use in multiobjective design optimization.” *AIAA journal*. 44(4): 879–891.

- Kennedy, J. and R. Eberhart. (1995). “Particle swarm optimization.” In: *Proceedings of ICNN’95-international conference on neural networks*. Vol. 4. ieee. 1942–1948.
- Kenneth, S., D. D’Ambrosio, and J. Gauci. (2009). “A hypercube-based encoding for evolving large-scale neural networks.” *Artif. Life.* 15(2): 185–212. issn: 1064-5462. doi: [10.1162/artl.2009.15.2.15202](https://doi.org/10.1162/artl.2009.15.2.15202).
- Klein, A., Z. Dai, F. Hutter, N. Lawrence, and J. Gonzalez. (2019). “Meta-Surrogate Benchmarking for Hyperparameter Optimization.” In: *Proceedings of the 32th International Conference on Advances in Neural Information Processing Systems (NeurIPS’19)*.
- Klein, A., S. Falkner, N. Mansur, and F. Hutter. (2017a). “RoBO: A Flexible and Robust Bayesian Optimization Framework in Python.” In: *NeurIPS Workshop on Bayesian Optimization (BayesOpt’17)*.
- Klein, A. and F. Hutter. (2019). “Tabular Benchmarks for Joint Architecture and Hyperparameter Optimization.” *arXiv:1905.04970 [cs.LG]*.
- Klein, A., L. Tiao, T. Lienart, C. Archambeau, and M. Seeger. (2020a). “Model-based Asynchronous Hyperparameter and Neural Architecture Search.” *Tech. rep.* No. 2003.10865 [cs.LG]. ArXiv.
- Klein, A., L. C. Tiao, T. Lienart, C. Archambeau, and M. Seeger. (2020b). “Model-based Asynchronous Hyperparameter Optimization.” *arXiv:2003.10865 [cs.LG]*.
- Klein, A., S. Falkner, S. Bartels, P. Hennig, and F. Hutter. (2017b). “Fast bayesian optimization of machine learning hyperparameters on large datasets.” In: *Artificial Intelligence and Statistics*. 528–536.
- Kleinberg, J. (2018). “Inherent trade-offs in algorithmic fairness.” *SIGMETRICS*.
- Knowles, J. (2006). “ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems.” *IEEE Transactions on Evolutionary Computation*. 10(1): 50–66.
- Kohavi, R. (1995). “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.” In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. 1137–1145. url: <http://ijcai.org/Proceedings/95-2/Papers/016.pdf>.
- Kouw, W. M. and M. Loog. (2018). “An introduction to domain adaptation and transfer learning.” *arXiv preprint arXiv:1812.11806*.

- Krantz, S. G. and H. R. Parks. (2012). *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media.
- Krizhevsky, A., I. Sutskever, and G. Hinton. (2012). “ImageNet Classification with Deep Convolutional Neural Networks.” In: *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS’12)*.
- Kunanusont, K., R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas. (2017). “The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement.” In: *2017 IEEE Congress on Evolutionary Computation (CEC)*.
- Kushner, H. J. (1964). “A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise.” *Journal of Basic Engineering*. 86(1): 97–106. ISSN: 0021-9223. doi: [10.1115/1.3653121](https://doi.org/10.1115/1.3653121). eprint: https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/86/1/97/5763745/97_1.pdf. URL: <https://doi.org/10.1115/1.3653121>.
- Kynkänniemi, T., T. Karras, S. Laine, J. Lehtinen, and T. Aila. (2019). “Improved Precision and Recall Metric for Assessing Generative Models.” In: *Advances in Neural Information Processing Systems*. Vol. 32. URL: <https://proceedings.neurips.cc/paper/2019/hash/0234c510bc6d908b28c70ff313743079-Abstract.html>.
- Lake, B. M., R. Salakhutdinov, and J. B. Tenenbaum. (2019). “The Omnistlot Challenge: A 3-Year Progress Report.” *Current Opinion in Behavioral Sciences*. Artificial Intelligence 29(Oct.): 97–104. ISSN: 2352-1546. doi: [10.1016/j.cobeha.2019.04.007](https://doi.org/10.1016/j.cobeha.2019.04.007). URL: <https://www.sciencedirect.com/science/article/pii/S2352154619300051>.
- Larsen, J., L. K. Hansen, C. Svarer, and M. Ohlsson. (1996). “Design and regularization of neural networks: the optimal use of a validation set.” In: *Neural Networks for Signal Processing [1996] VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*. IEEE. 62–71.
- Larsen, J., C. Svarer, L. N. Andersen, and L. K. Hansen. (1998). “Adaptive Regularization in Neural Network Modeling.” In: *Neural Networks: Tricks of the Trade*. Springer. 113–132.
- Lemieux, C. (2009). *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer series in statistics. Dordrecht: Springer. doi: [10.1007/978-0-387-78165-5](https://doi.org/10.1007/978-0-387-78165-5). URL: <https://cds.cern.ch/record/1338331>.

- Letham, B., B. Karrer, G. Ottino, and E. Bakshy. (2019). “Constrained Bayesian Optimization with Noisy Experiments.” *Bayesian Analysis*. 14(2): 495–519.
- Li, L., K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. (2019). “Massively Parallel Hyperparameter Tuning.” *Tech. rep.* No. 1810.05934v4 [cs.LG]. ArXiv.
- Li, L. and A. Talwalkar. (2020a). “Random Search and Reproducibility for Neural Architecture Search.” In: *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*.
- Li, L. and A. Talwalkar. (2020b). “Random search and reproducibility for neural architecture search.” In: *Uncertainty in Artificial Intelligence*. 367–377.
- Li, L., W. Chu, J. Langford, and R. E. Schapire. (2010). “A contextual-bandit approach to personalized news article recommendation.” In: *Proceedings of the 19th international conference on World wide web*. 661–670.
- Li, L., K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. (2016). “Hyperband: A novel bandit-based approach to hyperparameter optimization.” *arXiv preprint arXiv:1603.06560*.
- Li, L., K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. (2017). “Hyperband: A novel bandit-based approach to hyperparameter optimization.” *The Journal of Machine Learning Research*. 18(1): 6765–6816.
- Li, N., L. Ma, G. Yu, B. Xue, M. Zhang, and Y. Jin. (2023). “Survey on Evolutionary Deep Learning: Principles, Algorithms, Applications, and Open Issues.” *ACM Comput. Surv.* 56(2): 41:1–41:34. ISSN: 0360-0300. doi: [10.1145/3603704](https://doi.org/10.1145/3603704). url: <https://doi.org/10.1145/3603704>.
- Liao, R., Y. Xiong, E. Fetaya, L. Zhang, K. Yoon, X. Pitkow, R. Urtasun, and R. Zemel. (2018). “Reviving and Improving Recurrent Back-Propagation.” In: *International Conference on Machine Learning*. 3088–3097.
- Liaw, R., E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. (2018). “Tune: A Research Platform for Distributed Model Selection and Training.” *arXiv preprint arXiv:1807.05118*.

- Lin, C., M. Guo, C. Li, X. Yuan, W. Wu, J. Yan, D. Lin, and W. Ouyang. (2019). “Online Hyper-Parameter Learning for Auto-Augmentation Strategy.” In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 6579–6588. URL: https://openaccess.thecvf.com/content_ICCV_2019/html/Lin_Online_Hyper-Parameter_Learning_for_Auto-Augmentation_Strategy_ICCV_2019_paper.html.
- Lindauer, M., K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. (2022). “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization.” *Journal of Machine Learning Research*. 23(54): 1–9. URL: <http://jmlr.org/papers/v23/21-0888.html>.
- Liu, H., K. Simonyan, and Y. Yang. (2019). “DARTS: Differentiable Architecture Search.” *International Conference on Learning Representations*.
- Locatello, F., S. Bauer, M. Lucic, G. Raetsch, S. Gelly, B. Schölkopf, and O. Bachem. (2019). “Challenging Common Assumptions in the Unsupervised Learning of Disentangled Representations.” In: *Proceedings of the 36th International Conference on Machine Learning*. PMLR. 4114–4124. URL: <https://proceedings.mlr.press/v97/locatello19a.html>.
- Lorenzo, P. R., J. Nalepa, M. Kawulok, L. S. Ramos, and J. R. Pastor. (2017). “Particle swarm optimization for hyper-parameter selection in deep neural networks.” In: *Proceedings of the genetic and evolutionary computation conference*. 481–488.
- Lorraine, J. and D. Duvenaud. (2018). “Stochastic hyperparameter optimization through hypernetworks.” *arXiv preprint arXiv:1802.09419*.
- Loshchilov, I. and F. Hutter. (2016). “CMA-ES for hyperparameter optimization of deep neural networks.” *arXiv preprint arXiv:1604.07269*.
- Loshchilov, I. and F. Hutter. (2017). “Sgdr: Stochastic gradient descent with warm restarts.” *International Conference on Learning Representations*.
- Luketina, J., M. Berglund, K. Greff, and T. Raiko. (2016). “Scalable gradient-based tuning of continuous regularization hyperparameters.” In: *International Conference on Machine Learning*. 2952–2960.
- Ma, M. Q., Y. Zhao, X. Zhang, and L. Akoglu. (2023). “The Need for Unsupervised Outlier Model Selection: A Review and Evaluation of Internal Evaluation Strategies.” *ACM SIGKDD Explorations Newsletter*. 25(1): 19–35. ISSN: 1931-0145, 1931-0153. doi: [10.1145/3606274.3606277](https://doi.org/10.1145/3606274.3606277). URL: <https://dl.acm.org/doi/10.1145/3606274.3606277>.

- MacKay, D. J. (1992). “Bayesian interpolation.” *Neural computation*. 4(3): 415–447.
- MacKay, M., P. Vicol, J. Lorraine, D. Duvenaud, and R. Grosse. (2019). “Self-Tuning Networks: Bilevel Optimization of Hyperparameters using Structured Best-Response Functions.” *International Conference on Learning Representations*.
- Maclaurin, D., D. Duvenaud, and R. Adams. (2015). “Gradient-based hyperparameter optimization through reversible learning.” In: *International Conference on Machine Learning*. 2113–2122.
- Maron, O. and A. Moore. (1994). “Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation.” In: *Advances in Neural Information Processing Systems*. Vol. 6. Morgan-Kaufmann. URL: <https://proceedings.neurips.cc/paper/1993/hash/02a32ad2669e6fe298e607fe7cc0e1a0-Abstract.html>.
- Maurer, A., M. Pontil, and B. Romera-Paredes. (2016). “The benefit of multitask representation learning.” *Journal of Machine Learning Research*. 17(81): 1–32.
- McKay, M. D., R. J. Beckman, and W. J. Conover. (1979). “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code.” *Technometrics*. 42(1): 55–61.
- Mei, S. and X. Zhu. (2015). “Using Machine Teaching to Identify Optimal Training-Set Attacks on Machine Learners.” In: *AAAI*. 2871–2877.
- Micaelli, P. and A. J. Storkey. (2021). “Gradient-based hyperparameter optimization over long horizons.” *Advances in Neural Information Processing Systems*. 34: 10798–10809.
- Miikkulainen, R., J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, *et al.* (2024). “Evolving deep neural networks.” In: *Artificial intelligence in the age of neural networks and brain computing*. Elsevier. 269–287.
- Miller, J. and M. Hardt. (2019). “Stable recurrent models.” *International Conference on Learning Representations*.

- Mitchell, T. M., P. E. Utgoff, and R. Banerji. (1983). “Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics.” In: *Machine Learning: An Artificial Intelligence Approach*. Ed. by R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Symbolic Computation*. Berlin, Heidelberg: Springer. 163–190. ISBN: 978-3-662-12405-5. doi: [10.1007/978-3-662-12405-5_6](https://doi.org/10.1007/978-3-662-12405-5_6). URL: https://doi.org/10.1007/978-3-662-12405-5_6.
- Mockus, J., V. Tesis, and A. Zilinskas. (1978). “The application of Bayesian methods for seeking the extremum.” *Towards global optimization*. 2(117-129): 2.
- Močkus, J. (1975). “On Bayesian Methods for Seeking the Extremum.” In: *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*. Ed. by G. I. Marchuk. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 400–404. ISBN: 978-3-540-37497-8. doi: [10.1007/3-540-07165-2_55](https://doi.org/10.1007/3-540-07165-2_55).
- Močkus, J., V. Tesis, and A. Zilinskas. (1978). “The application of Bayesian methods for seeking the extremum.” *Towards global optimization*. 2(117-129): 2.
- Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.
- Myers, R., D. C. Montgomery, and C. Anderson Cook. (2016). *Response surface methodology: process and product optimization using design experiments*. Fourth Edition. John Wiley & Sons.
- Naeem, M. F., S. J. Oh, Y. Uh, Y. Choi, and J. Yoo. (2020). “Reliable Fidelity and Diversity Metrics for Generative Models.” In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR. 7176–7185. URL: <https://proceedings.mlr.press/v119/naeem20a.html>.
- Nakayama, H., Y. Yun, and M. Yoon. (2009). *Sequential approximate multiobjective optimization using computational intelligence*. Springer Science & Business Media.
- Neal, R. M. (1996). *Bayesian Learning for Neural Networks. Lecture Notes in Statistics*. No. 118. Springer.
- Neumann, J. v. (1966). “Theory of self-reproducing automata.” Edited by Arthur W. Burks.

- Nguyen, X., M. J. Wainwright, and M. I. Jordan. (2009). “On Surrogate Loss Functions and F-Divergences.” *The Annals of Statistics*. 37(2): 876–904. ISSN: 0090-5364, 2168-8966. doi: [10.1214/08-AOS595](https://doi.org/10.1214/08-AOS595). URL: <https://projecteuclid.org/journals/annals-of-statistics/volume-37/issue-2/On-surrogate-loss-functions-and-f-divergences/10.1214/08-AOS595.full>.
- Niculae, V., C. F. Corro, N. Nangia, T. Mihaylova, and A. F. Martins. (2023). “Discrete latent structure in neural networks.” *arXiv preprint arXiv:2301.07473*.
- Niederreiter, H. (1988). “Low-discrepancy and low-dispersion sequences.” *Journal of number theory*. 30(1): 51–70.
- Nolfi, S. and D. Parisi. (2002). “Evolution of Artificial Neural Networks.” In: *In Handbook of brain theory and neural networks*. MIT Press. 418–421.
- O’Bray, L., M. Horn, B. Rieck, and K. Borgwardt. (2021). “Evaluation Metrics for Graph Generative Models: Problems, Pitfalls, and Practical Solutions.” In: *International Conference on Learning Representations*. URL: <https://openreview.net/forum?id=tBtoZYKd9n>.
- Orabona, F. and D. Pál. (2016). “Coin betting and parameter-free online learning.” In: *Advances in Neural Information Processing Systems*. 577–585.
- Owen, A. B. (1995). “Randomly permuted (t, m, s)-nets and (t, s)-sequences.” In: *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing: Proceedings of a conference at the University of Nevada, Las Vegas, Nevada, USA, June 23–25, 1994*. Springer. 299–317.
- Pan, S. J. and Q. Yang. (2009). “A survey on transfer learning.” *IEEE Transactions on knowledge and data engineering*. 22(10): 1345–1359.
- Pang, G., C. Shen, L. Cao, and A. V. D. Hengel. (2021). “Deep Learning for Anomaly Detection: A Review.” *ACM Computing Surveys*. 54(2): 38:1–38:38. ISSN: 0360-0300. doi: [10.1145/3439950](https://doi.org/10.1145/3439950). URL: <https://doi.org/10.1145/3439950>.
- Paria, B., K. Kandasamy, and B. Póczos. (2019). “A Flexible Framework for Multi-Objective Bayesian Optimization using Random Scalarizations.” In: *UAI*.

- Park, D. and S. Kim. (2023). “Probabilistic Precision and Recall Towards Reliable Evaluation of Generative Models.” In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 20099–20109. URL: https://openaccess.thecvf.com/content/ICCV2023/html/Park_Probabilistic_Precision_and_Recall_Towards_Reliable_Evaluation_of_Generative_Models_ICCV_2023_paper.html.
- Pedregosa, F. (2016). “Hyperparameter optimization with approximate gradient.” In: *International Conference on Machine Learning*. 737–746.
- Perrone, V., H. Shen, A. Zolic, I. Shcherbatyi, A. Ahmed, T. Bansal, M. Donini, F. Winkelmoelen, J. Jenatton R. Faddoul, B. Pogorzelksa, M. Miladinovic, K. Kenthapadi, M. Seeger, and C. Archambeau. (2021). “Amazon Sage-Maker Automatic Model Tuning: Scalable Gradient-Free Optimization.” In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.
- Perrone, V., R. Jenatton, M. W. Seeger, and C. Archambeau. (2018). “Scalable hyperparameter transfer learning.” In: *Advances in Neural Information Processing Systems*. 6845–6855.
- Perrone, V., I. Shcherbatyi, R. Jenatton, C. Archambeau, and M. Seeger. (2019). “Constrained Bayesian Optimization with Max-Value Entropy Search.” eprint: [NeurIPSWorkshoponMeta-Learning](#).
- Perrone, V., H. Shen, A. Zolic, I. Shcherbatyi, A. Ahmed, T. Bansal, M. Donini, F. Winkelmoelen, R. Jenatton, J. B. Faddoul, B. Pogorzelksa, M. Miladinovic, K. Kenthapadi, M. Seeger, and C. Archambeau. (2020). “Amazon SageMaker Automatic Model Tuning: Scalable Black-box Optimization.” *arXiv:2012.08489*.
- Pfisterer, F., L. Schneider, J. Moosbauer, M. Binder, and B. Bischl. (2022). “YAHPO Gym - An Efficient Multi-Objective Multi-Fidelity Benchmark for Hyperparameter Optimization. In *International Conference on Automated Machine Learning (Main Track)*.
- Pham, H., M. Guan, B. Zoph, Q. Le, and J. Dean. (2018). “Efficient Neural Architecture Search via Parameters Sharing.” In: *Proceedings of the 35th International Conference on Machine Learning (ICML’18)*.
- Picheny, V., R. B. Gramacy, S. Wild, and S. Le Digabel. (2016). “Bayesian optimization under mixed constraints with a slack-variable augmented Lagrangian.” *Advances in Neural Information Processing Systems* 29.

- Picheny, V. (2015). “Multiobjective optimization using Gaussian process emulators via stepwise uncertainty reduction.” *Statistics and Computing*. 25(6): 1265–1280.
- Ponweiser, W., T. Wagner, D. Biermann, and M. Vincze. (2008). “Multiobjective optimization on a limited budget of evaluations using model-assisted S -metric selection.” In: *International Conference on Parallel Problem Solving from Nature*. Springer. 784–794.
- Rajeswaran, A., C. Finn, S. M. Kakade, and S. Levine. (2019). “Meta-learning with implicit gradients.” In: *Advances in Neural Information Processing Systems*. 113–124.
- Ravi, S. and H. Larochelle. (2017). “Optimization as a model for few-shot learning.” *International Conference on Learning Representations*. (Accessed on 08/29/2017).
- Real, E., A. Aggarwal, Y. Huang, and Q. V. Le. (2019). “Regularized evolution for image classifier architecture search.” In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 4780–4789.
- Real, E., C. Liang, D. R. So, and Q. V. Le. (2020). “Automl-zero: Evolving machine learning algorithms from scratch.” *arXiv preprint arXiv:2003.03384*.
- Rechenberg, I. (1965). “Cybernetic Solution Path of an Experimental Problem.” *Royal Aircraft Establishment Library Translation*. (1122).
- Ren, P., Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang. (2021). “A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions.” *ACM Comput. Surv.* 54(4): 76:1–76:34. ISSN: 0360-0300. doi: [10.1145/3447582](https://doi.org/10.1145/3447582). URL: <https://doi.org/10.1145/3447582>.
- Rezende, D. and S. Mohamed. (2015). “Variational Inference with Normalizing Flows.” In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. *Proceedings of Machine Learning Research*. Lille, France: PMLR. 1530–1538. URL: <https://proceedings.mlr.press/v37/rezende15.html>.
- Rissanen, J. (1978). “Modeling by shortest data description.” *Automatica*. 14(5): 465–471.
- Ru, B., X. Wan, X. Dong, and M. Osborne. (2021). “Interpretable Neural Architecture Search via Bayesian Optimisation with Weisfeiler-Lehman Kernels.” In: *International Conference on Learning Representations (ICLR'21)*.

- Ruder, S. (2017a). “An Overview of Gradient Descent Optimization Algorithms.” *arXiv:1609.04747 [cs]*. June. eprint: [1609.04747](https://arxiv.org/abs/1609.04747). URL: <http://arxiv.org/abs/1609.04747>.
- Ruder, S. (2017b). “An Overview of Multi-Task Learning in Deep Neural Networks.” doi: [10.48550/arXiv.1706.05098](https://doi.org/10.48550/arXiv.1706.05098). eprint: [1706.05098](https://arxiv.org/abs/1706.05098).
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams. (1986). “Learning representations by back-propagating errors.” *nature*. 323(6088): 533–536.
- Rusu, A. A., D. Rao, J. Sygnowski, O. Vinyals, R. Pascanu, S. Osindero, and R. Hadsell. (2018). “Meta-learning with latent embedding optimization.” *arXiv preprint arXiv:1807.05960*.
- Sacks, J., W. J. Welch, T. J. Mitchell, and H. P. Wynn. (1989). “Design and Analysis of Computer Experiments.” *Statistical Science*. 4(4): 409–423. ISSN: 0883-4237, 2168-8745. doi: [10.1214/ss/1177012413](https://doi.org/10.1214/ss/1177012413). URL: <https://projecteuclid.org/journals/statistical-science/volume-4/issue-4/Design-and-Analysis-of-Computer-Experiments/10.1214/ss/1177012413.full>.
- Sajjadi, M. S. M., O. Bachem, M. Lucic, O. Bousquet, and S. Gelly. (2018). “Assessing Generative Models via Precision and Recall.” In: *Advances in Neural Information Processing Systems*. Vol. 31. URL: <https://proceedings.neurips.cc/paper/2018/hash/f7696a9b362ac5a51c3dc8f098b73923-Abstract.html>.
- Salimans, T., I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen. (2016). “Improved Techniques for Training GANs.” In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf.
- Salinas, D., J. Golebiowsk, A. Klein, M. Seeger, and C. Archambeau. (2023). “Optimizing Hyperparameters with Conformal Quantile Regression.” In: *Proceedings of the 40th International Conference on Machine Learning (ICML’23)*.
- Salinas, D., M. Seeger, A. Klein, V. Perrone, M. Wistuba, and C. Archambeau. (2022). “Syne Tune: A Library for Large Scale Hyperparameter Tuning and Reproducible Research.” In: *First Conference on Automated Machine Learning (Main Track)*.

- Salinas, D., H. Shen, and V. Perrone. (2020). “A quantile-based approach for hyperparameter transfer learning.” In: *Proceedings of the 37th International Conference on Machine Learning (ICML’20)*.
- Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. (2009). “The graph neural network model.” *IEEE Transactions on Neural Networks*. 20(1): 61–80.
- Schede, E., J. Brandt, A. Tornede, M. Wever, V. Bengs, E. Hüllermeier, and K. Tierney. (2022). “A Survey of Methods for Automated Algorithm Configuration.” *Journal of Artificial Intelligence Research*. 75(Oct.): 425–487. issn: 1076-9757. doi: [10.1613/jair.1.13676](https://doi.org/10.1613/jair.1.13676). url: <https://jair.org/index.php/jair/article/view/13676>.
- Schick, T., J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. (2023). “Toolformer: Language models can teach themselves to use tools.” *Advances in Neural Information Processing Systems*. 37.
- Schneider, L., M. Wistuba, A. Klein, J. Golebiowski, G. Zappella, and F. A. Merra. (2024). “Hyperband-based Bayesian Optimization for Black-box Prompt Selection.” *arXiv preprint arXiv:2412.07820*.
- Schwarz, G. (1978). “Estimating the dimension of a model.” *The annals of statistics*. 6(2): 461–464.
- Seeger, M. (2007). “Cross-validation optimization for large scale hierarchical classification kernel methods.” In: *Advances in neural information processing systems*. 1233–1240.
- Shaban, A., C.-A. Cheng, N. Hatch, and B. Boots. (2019). “Truncated Back-propagation for Bilevel Optimization.” In: *The 22nd International Conference on Artificial Intelligence and Statistics*. 1723–1732.
- Shahriari, B., K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. (2016). “Taking the Human Out of the Loop: A Review of Bayesian Optimization.” *Proceedings of the IEEE*. 104(1): 148–175. issn: 1558-2256. doi: [10.1109/JPROC.2015.2494218](https://doi.org/10.1109/JPROC.2015.2494218).
- Shalev-Shwartz, S. et al. (2012). “Online learning and online convex optimization.” *Foundations and Trends® in Machine Learning*. 4(2): 107–194.

- Sharifnassab, A., S. Salehkaleybar, and R. Sutton. (2024). “MetaOptimize: A Framework for Optimizing Step Sizes and Other Meta-Parameters.” doi: [10.48550/arXiv.2402.02342](https://doi.org/10.48550/arXiv.2402.02342). eprint: [2402.02342](https://arxiv.org/abs/2402.02342). URL: <http://arxiv.org/abs/2402.02342>.
- Shi, C., K. Yang, Z. Chen, J. Li, J. Yang, and C. Shen. (2024). “Efficient prompt optimization through the lens of best arm identification.” *arXiv preprint arXiv:2402.09723*.
- Shin, T., Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh. (2020). “Autoprompt: Eliciting knowledge from language models with automatically generated prompts.” *arXiv preprint arXiv:2010.15980*.
- Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, and et al. (2017). “Mastering the game of Go without human knowledge.” *Nature*. 550(7676): 354–359. ISSN: 1476-4687. doi: [10.1038/nature24270](https://doi.org/10.1038/nature24270). URL: <http://dx.doi.org/10.1038/nature24270>.
- Simon, D. (2013). *Evolutionary optimization algorithms*. John Wiley & Sons.
- Simon, L., R. Webster, and J. Rabin. (2019). “Revisiting Precision Recall Definition for Generative Modeling.” In: *Proceedings of the 36th International Conference on Machine Learning*. PMLR. 5799–5808. URL: <https://proceedings.mlr.press/v97/simon19a.html>.
- Snoek, J., H. Larochelle, and R. P. Adams. (2012). “Practical bayesian optimization of machine learning algorithms.” In: *Advances in neural information processing systems*. 2951–2959.
- Snoek, J., O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, M. Prabhat, and R. P. Adams. (2015). “Scalable Bayesian Optimization Using Deep Neural Networks.” In: *International Conference on Machine Learning*. 2171–2180.
- Sobol’, I. (1979). “On the systematic search in a hypercube.” *SIAM Journal on Numerical Analysis*. 16(5): 790–793.
- Song, X., S. Perel, C. Lee, G. Kochanski, and D. Golovin. (2022). “Open Source Vizier: Distributed Infrastructure and API for Reliable and Flexible Black-box Optimization.” In: *Automated Machine Learning Conference, Systems Track (AutoML-Conf Systems)*.
- Springenberg, J. T., A. Klein, S. Falkner, and F. Hutter. (2016). “Bayesian optimization with robust Bayesian neural networks.” In: *Advances in neural information processing systems*. 4134–4142.

- Srinivas, N., A. Krause, S. Kakade, and M. Seeger. (2010). “Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design.” *ICML*: 1015–1022.
- Stone, M. (1974). “Cross-validatory choice and assessment of statistical predictions.” *Journal of the Royal Statistical Society: Series B (Methodological)*. 36(2): 111–133.
- Suzuki, S., S. Takeno, T. Tamura, K. Shitara, and M. Karasuyama. (2019). “Multi-objective Bayesian Optimization using Pareto-frontier Entropy.” *arXiv preprint arXiv:1906.00127*.
- Svenson, J. D. and T. J. Santner. (2010). “Multiobjective optimization of expensive black-box functions via expected maximin improvement.” *The Ohio State University, Columbus, Ohio*. 32.
- Swersky, K., J. Snoek, and R. P. Adams. (2013). “Multi-task Bayesian optimization.” In: *Advances in Neural Information Processing Systems 26*. 2004–2012.
- Swersky, K., J. Snoek, and R. P. Adams. (2014). “Freeze-thaw Bayesian optimization.” *arXiv preprint arXiv:1406.3896*.
- Tan, M., B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. (2019). “MnasNet: Platform-Aware Neural Architecture Search for Mobile.” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR’19)*.
- Tao, Z., Y. Li, B. Ding, C. Zhang, J. Zhou, and Y. Fu. (2020). “Learning to mutate with hypergradient guided population.” *Advances in Neural Information Processing Systems*. 33: 17641–17651.
- Thompson, W. R. (1933). “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples.” *Biometrika*: 285–294.
- Thornton, C., F. Hutter, H. H. Hoos, and K. Leyton-Brown. (2013). “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms.” In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 847–855.
- Tiao, L. C., A. Klein, M. W. Seeger, E. V. Bonilla, C. Archambeau, and F. Ramos. (2021). “BORE: Bayesian Optimization by Density-Ratio Estimation.” In: *Proceedings of the 38th International Conference on Machine Learning*. 10289–10300.

- Torrey, L. and J. Shavlik. (2010). “Transfer learning.” In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global. 242–264.
- Vanschoren, J., J. van Rijn, B. Bischl, and L. Torgo. (2014). “OpenML: Networked Science in Machine Learning.” *SIGKDD Explorations*.
- Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. (2017). “Attention is All you Need.” In: *Advances in Neural Information Processing Systems*. 6000–6010.
- Verma, S. and J. Rubin. (2018). “Fairness definitions explained.” *FairWare*.
- Vezhnevets, A. S., S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. (2017). “Feudal networks for hierarchical reinforcement learning.” In: *International conference on machine learning*. PMLR. 3540–3549.
- Vicol, P., L. Metz, and J. Sohl-Dickstein. (2021). “Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies.” In: *International Conference on Machine Learning*. PMLR. 10553–10563.
- Villemonteix, J., E. Vazquez, and E. Walter. (2009). “An informational approach to the global optimization of expensive-to-evaluate functions.” *Journal of Global Optimization*. 44(4): 509–534.
- Wagner, T., M. Emmerich, A. Deutz, and W. Ponweiser. (2010). “On expected-improvement criteria for model-based multi-objective optimization.” In: *International Conference on Parallel Problem Solving from Nature*. Springer. 718–727.
- Wang, R., Y. Demiris, and C. Ciliberto. (2020). “A structured prediction approach for conditional meta-learning.” *Advances in Neural Information Processing Systems*.
- Wang, X., Y. Jin, S. Schmitt, and M. Olhofer. (2023). “Recent Advances in Bayesian Optimization.” *ACM Comput. Surv.* 55(13s): 287:1–287:36. issn: 0360-0300. doi: [10.1145/3582078](https://doi.org/10.1145/3582078). url: <https://doi.org/10.1145/3582078>.
- Wang, Z. and S. Jegelka. (2017a). “Max-Value Entropy Search for Efficient Bayesian Optimization.” In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML’17*. Sydney, NSW, Australia: JMLR.org. 3627–3635.
- Wang, Z. and S. Jegelka. (2017b). “Max-value entropy search for efficient Bayesian optimization.” *International Conference on Machine Learning*.

- Wei, J., M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. (2021). “Finetuned language models are zero-shot learners.” *arXiv preprint arXiv:2109.01652*.
- Werbos, P. J. (1990). “Backpropagation Through Time: What it Does and How To Do It.” *Proceedings of the IEEE*. 78(10): 1550–1560.
- White, C., M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter. (2023). “Neural Architecture Search: Insights from 1000 Papers.” *arXiv:2301.08727 [cs.LG]*.
- Williams, C. K. and C. E. Rasmussen. (2006). *Gaussian processes for machine learning*. Vol. 2. MIT press Cambridge, MA.
- Williams, R. J. and J. Peng. (1990). “An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories.” *Neural Comput.* 2(4): 490–501. doi: [10.1162/neco.1990.2.4.490](https://doi.org/10.1162/neco.1990.2.4.490). URL: <https://doi.org/10.1162/neco.1990.2.4.490>.
- Williams, R. J. and D. Zipser. (1989). “A learning algorithm for continually running fully recurrent neural networks.” *Neural computation*. 1(2): 270–280.
- Wilson, J., F. Hutter, and M. Deisenroth. (2018). “Maximizing acquisition functions for Bayesian optimization.” In: *NeurIPS*. 9906–9917.
- Wistuba, M., A. Kadra, and J. Grabocka. (2022). “Supervising the Multi-Fidelity Race of Hyperparameter Configurations.” In: *Proceedings of the 36th International Conference on Advances in Neural Information Processing Systems (NeurIPS’22)*.
- Wistuba, M., N. Schilling, and L. Schmidt-Thieme. (2015). “Learning hyperparameter optimization initializations.” In: *IEEE International Conference on Data Science and Advanced Analytics (DSAA)*.
- Wu, J., M. Poloczek, A. G. Wilson, and P. Frazier. (2017). “Bayesian optimization with gradients.” In: *Advances in Neural Information Processing Systems*. 5267–5278.
- Wu, L., F. Tian, Y. Xia, Y. Fan, T. Qin, L. Jian-Huang, and T.-Y. Liu. (2018). “Learning to teach with dynamic loss functions.” In: *Advances in Neural Information Processing Systems*. 6466–6477.
- Wu, Z., Y. Wang, J. Ye, and L. Kong. (2022). “Self-adaptive in-context learning: An information compression perspective for in-context example selection and ordering.” *arXiv preprint arXiv:2212.10375*.

- Yang, A., P. M. Esperança, and F. M. Carlucci. (2020). “NAS valuation is frustratingly hard.” In: *International Conference on Learning Representations (ICLR’20)*.
- Yang, L.-C. and A. Lerch. (2020). “On the Evaluation of Generative Models in Music.” *Neural Computing and Applications*. 32(9): 4773–4784. ISSN: 0941-0643, 1433-3058. doi: [10.1007/s00521-018-3849-7](https://doi.org/10.1007/s00521-018-3849-7). URL: <http://link.springer.com/10.1007/s00521-018-3849-7>.
- Yang, G., E. J. Hu, I. Babuschkin, S. Sidor, X. Liu, D. Farhi, N. Ryder, J. Pachocki, W. Chen, and J. Gao. (2022). “Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer.” *arXiv:2203.03466 [cs.LG]*.
- Yang, J., C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. (2024). “Swe-agent: Agent-computer interfaces enable automated software engineering.” *arXiv preprint arXiv:2405.15793*.
- Yang, L. and A. Shami. (2020). “On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice.” *Neurocomputing*. 415(Nov.): 295–316. ISSN: 09252312. doi: [10.1016/j.neucom.2020.07.061](https://doi.org/10.1016/j.neucom.2020.07.061). URL: <https://linkinghub.elsevier.com/retrieve/pii/S0925231220311693>.
- Yao, X. (1999). “Evolving artificial neural networks.” *Proceedings of the IEEE*. 87(9): 1423–1447.
- Ying, C., A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. (2019). “NAS-Bench-101: Towards Reproducible Neural Architecture Search.” In: *Proceedings of the 36th International Conference on Machine Learning (ICML’19)*.
- Yogatama, D., L. Kong, and N. A. Smith. (2015). “Bayesian Optimization of Text Representations.” In: *In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics. 2100–2105.
- Yu, J., P. Jin, H. Liu, G. Bender, P. J. Kindermans, M. Tan, T. Huang, X. Song, R. Pang, and Q. Le. (2020). “BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models.” In: *The European Conference on Computer Vision (ECCV’20)*.
- Zakerinia, H., A. Behjati, and C. H. Lampert. (2024). “More Flexible PAC-Bayesian Meta-Learning by Learning Learning Algorithms.” *arXiv preprint arXiv:2402.04054*.

- Zantedeschi, V., L. Franceschi, J. Kaddour, M. J. Kusner, and V. Niculae. (2023). “DAG Learning on the Permutahedron.” *arXiv preprint arXiv:2301.11898*.
- Zhang, Q., W. Liu, E. Tsang, and B. Virginas. (2009). “Expensive multiobjective optimization by MOEA/D with Gaussian process model.” *IEEE Transactions on Evolutionary Computation*. 14(3): 456–474.
- Zhao, D., S. Kobayashi, J. Sacramento, and J. von Oswald. (2020). “Meta-learning via hypernetworks.” In: *4th Workshop on Meta-Learning at NeurIPS 2020 (MetaLearn 2020)*. NeurIPS.
- Zhuge, M., W. Wang, L. Kirsch, F. Faccio, D. Khizbulin, and J. Schmidhuber. (2024). “Language agents as optimizable graphs.” *arXiv preprint arXiv:2402.16823*.
- Zimmer, L., M. Lindauer, and F. Hutter. (2021). “Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- Zitzler, E. and L. Thiele. (1998). “Multiobjective optimization using evolutionary algorithms—a comparative case study.” In: *International conference on parallel problem solving from nature*. Springer. 292–301.
- Zoph, B. and Q. V. Le. (2017). “Neural architecture search with reinforcement learning.” *International Conference on Learning Representations*.
- Zoph, B., V. Vasudevan, J. Shlens, and Q. V. Le. (2018). “Learning Transferable Architectures for Scalable Image Recognition.” In: *IEEE Conference on Computer Vision and Pattern Recognition*. 8697–8710. doi: [10.1109/CVPR.2018.00907](https://doi.org/10.1109/CVPR.2018.00907). URL: http://openaccess.thecvf.com/content_5C_cvpr%5C_2018/html/Zoph%5C_Learning%5C_Transferable%5C_Architectures%5C_CVPR%5C_2018%5C_paper.html.
- Zuluaga, M., G. Sergent, A. Krause, and M. Püschel. (2013). “Active learning for multi-objective optimization.” In: *International Conference on Machine Learning*. 462–470.