

## GOAT method: Green Orthogonal Array Tuning method

Authors	Ranković, Nevena; Rankovic, Dragica
Published in	Alexandria Engineering Journal
DOI	<a href="https://doi.org/10.1016/j.aej.2025.10.044">10.1016/j.aej.2025.10.044</a>
Publication Date	2025-12
Document Version	publishersversion
Link	<a href="https://research.tilburguniversity.edu/en/publications/61cc83c0-ddf3-4a37-ba2d-83892f42a4d7">https://research.tilburguniversity.edu/en/publications/61cc83c0-ddf3-4a37-ba2d-83892f42a4d7</a>
Citation	Ranković, N & Rankovic, D 2025, 'GOAT method : Green Orthogonal Array Tuning method', Alexandria Engineering Journal, vol. 133, pp. 13-41. <a href="https://doi.org/10.1016/j.aej.2025.10.044">https://doi.org/10.1016/j.aej.2025.10.044</a>
Download Date	2026-05-17 12:09:35
Rights	<p>General rights</p> <p>Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.</p> <ul style="list-style-type: none"> <li>- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.</li> <li>- You may not further distribute the material or use it for any profit-making activity or commercial gain</li> <li>- You may freely distribute the URL identifying the publication in the public portal"</li> </ul> <p>Take down policy</p> <p>If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.</p>



## Original Article

## GOAT method: Green Orthogonal Array Tuning method

Nevena Ranković<sup>a</sup>, Dragica Ranković<sup>b</sup><sup>a</sup> Department of Intelligent Systems, Tilburg School of Humanities and Digital Sciences, Tilburg University, Warandelaan 2, Tilburg, 5037 AB, North-Brabant, The Netherlands<sup>b</sup> Department of informatics, School of Computing, Union University, Belgrade, Knez Mihajlova 6, Belgrade, 11 000, Serbia

## ARTICLE INFO

## Keywords:

Hyper-parameter optimization  
 Robust design of experiment  
 Orthogonal array tuning method  
 Eco-efficiency

## ABSTRACT

This paper is a natural extension of our previous work and introduces an eco-efficient and integrated approach to hyper-parameter optimization (HPO) using Taguchi's orthogonal array tuning method (OATM), which forms the basis for our GOAT (Green Orthogonal Array Tuning) method across leading models in Machine Learning (ML), Deep Learning (DL), and Graph Neural Networks (GNNs): XGBoost, LightGBM, CatBoost, LSTM, GRU, GGNN, and GGSNN. Taguchi's method requires fewer than 10 experiments and just 11 s of running time for all models, demonstrating its efficiency. GGSNN emerges as the best-performing model overall. A comprehensive case study on software estimation, using 46 publicly available datasets, highlights the method's ability to reduce time and energy consumption while improving accuracy, promoting sustainable practices and high-impact real-world applications.

## 1. Introduction

The essential process in machine learning involves addressing optimization challenges while considering green issues, such as computational complexity and time constraints. When constructing, for example, an ML model, parameters (often called weighting coefficients) are first assigned and then refined through an optimization process to minimize the objective function or achieve optimal accuracy. Similarly, HPO methods strive to enhance the model's structure by finding the best hyper-parameter (HP) settings [1]. This approach is crucial not only for traditional machine learning (ML) but also for deep learning (DL) and graph neural networks (GNNs). However, these processes often demand substantial computational power, which results in higher energy usage and a greater environmental footprint [2]. The complexity and time required for optimization in ML, DL, and GNNs add another layer of challenge, necessitating efficient algorithms and methodologies to minimize both environmental and computational costs. In this section, we explore the fundamental concepts of mathematical optimization and hyper-parameter optimization in the context of these leading-edge models, highlighting the importance of sustainable practices and efficient computational strategies.

## 1.1. Mathematical optimization problem definition

Mathematical optimization plays an essential role in selecting the best solution from a range of specified candidates to maximize or

minimize an objective function, which is crucial in fields such as machine learning, deep learning, and graph neural networks. Optimization problems are typically divided into two main types: constrained and unconstrained [3]. In the context of unconstrained optimization, the variable  $z$  is free to assume any value within the real number set, represented as  $\min_{z \in \mathbb{R}} g(z)$ , where  $g(z)$  refers to the objective function to be optimized. In contrast, constrained optimization requires that  $z$  meet certain conditions, written as  $\min_z g(z)$ , subject to  $c_k(z) \leq 0$ , for  $k = 1, \dots, n$ , and  $d_l(z) = 0$ , for  $l = 1, \dots, q$ , with  $z \in Z$ . Here,  $c_k(z)$  denotes inequality constraints, and  $d_l(z)$  represents equality constraints. The feasible region  $S$  is defined as  $S = \{z \in Z \mid c_k(z) \leq 0, d_l(z) = 0\}$ . The goal of optimization is to identify values of  $z$  that minimize or maximize  $g(z)$  while satisfying these constraints. This process is pivotal for improving performance in ML, and is also critical in DL and GNNs, where both model parameters and network structures are optimized to achieve better outcomes. Advanced methods may involve regularization techniques or the use of Lagrange multipliers to solve such constrained problems effectively.

Achieving a global optimum can be challenging, and often only local optima are found. For a feasible region  $S$ , a global minimum is a point  $z^* \in S$  that satisfies:

$$g(z^*) \leq g(z) \quad \text{for all } z \in S, \quad (1)$$

\* Corresponding author.

E-mail address: [n.rankovic@uvt.nl](mailto:n.rankovic@uvt.nl) (N. Ranković).<https://doi.org/10.1016/j.aej.2025.10.044>

Received 13 April 2025; Received in revised form 28 August 2025; Accepted 30 October 2025

Available online 12 November 2025

1110-0168/© 2025 The Authors. Published by Elsevier B.V. on behalf of Faculty of Engineering, Alexandria University. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

whereas a local minimum is a point  $z^* \in S$  within a neighborhood  $M$ , such that:

$$g(z^*) \leq g(z) \quad \text{for all } z \in M \cap S. \quad (2)$$

For convex functions, any local minimum is also a global minimum, defined by the property:

$$g(tz_1 + (1-t)z_2) \leq tg(z_1) + (1-t)g(z_2) \quad (3)$$

An optimization problem is convex if  $g(z)$  is convex and  $S$  is a convex set, formulated as:

$$\min_z g(z) \quad \text{subject to } z \in S. \quad (4)$$

Non-convex functions, which are commonly encountered in ML and HPO tasks, contain multiple local optima. This makes it difficult to identify the global optimum. Classical optimization techniques include gradient descent, which updates the parameters by moving in the direction opposite to the gradient  $\nabla f(x)$ , but this method can only ensure finding the global optimum when the function  $f(x)$  is convex. The update rule is given by  $x_{k+1} = x_k - \eta \nabla f(x_k)$ . On the other hand, Newton’s method offers quicker convergence by employing the inverse of the *Hessian matrix*, though it comes with increased computational complexity. Its update rule is expressed as  $x_{k+1} = x_k - H^{-1} \nabla f(x_k)$  [4].

The conjugate gradient method improves convergence by searching along conjugate directions, which are determined from previously gathered data points. While it achieves faster convergence compared to standard gradient descent, the calculations involved are more intricate. Additionally, heuristic approaches, which rely on empirical rules, can often approximate global optima in relatively few iterations. However, they do not provide guarantees of identifying the true global solution [5].

In DL, optimization is formulated as  $\min_{\theta} L(\theta; x, y)$ , where  $\theta$  are model parameters and  $L$  is the loss function. In GNNs, optimization involves both weights and network structures:  $\min_{\theta, A} L(\theta, A; G)$ , where  $A$  is the adjacency matrix of graph  $G$ .

Thus, optimization in ML, DL, and GNNs involves complex and high-dimensional landscapes, requiring sophisticated methods to achieve effective and sufficient solutions. These techniques are fundamental for building robust and accurate models across various applications.

### 1.2. Hyper-parameter optimization problem definition

Efficiently navigating the hyper-parameter space using optimization methods is crucial for determining the optimal hyper-parameter values during the development of ML models [6]. Hyper-parameters are highly adjustable, and their effect on generalization performance frequently hinges on intricate and nuanced configurations. Hyper-parameter optimization algorithms aims to automatically find a reliable hyper-parameter configuration  $\lambda \in \Delta_e$  for a given ML algorithm  $I_\lambda$ . The search space  $\Delta_e \subset \Delta$  encompasses each pertinent hyper-parameter for optimization and their corresponding ranges, as represented by:

$$\Delta_e = \Delta_{e1} \times \Delta_{e2} \times \dots \times \Delta_{ei} \quad (5)$$

where  $\Delta_{ei}$  represents a restricted subset of the domain of the  $i$ -th hyper-parameter  $\Delta_i$ , which can be continuous, discrete, or categorical. This hybrid search space can also involve dependent hyper-parameters, resulting in a hierarchical configuration. A hyper-parameter  $\lambda_i$  is considered conditional on  $\lambda_j$  when  $\lambda_i$  is only active if  $\lambda_j$  belongs to a specific subset of  $\Delta_j$ , and otherwise,  $\lambda_i$  is inactive and does not influence the model learner [7]. An example can be found in the hyper-parameters of a kernelized model such as the Support Vector Machine (SVM), where both the choice of kernel and its corresponding hyper-parameters require adjustment. These conditional relationships often form hierarchical structures within the search space, typically represented by directed acyclic graphs.

### 1.3. Objective function and evaluation

The general HPO problem is formulated as:

$$\lambda^* = \arg \min_{\lambda \in \Delta} c(\lambda) = \arg \min_{\lambda \in \Delta} GE(I, J, \rho, \lambda) \quad (6)$$

where  $\lambda^*$  represents the theoretical optimal configuration, and  $c(\lambda)$  is the estimated generalization error with fixed  $I, J$ , and  $\rho$ . The goal is to estimate and minimize the generalization error  $GE(I, J, \rho, \lambda)$  of a learner  $I_\lambda$  for a particular hyper-parameter setting  $\lambda$ , using a re-sampling split  $J = ((J_{\text{train},1}, J_{\text{test},1}), \dots, (J_{\text{train},B}, J_{\text{test},B}))$ . The function  $c(\lambda)$  is considered an opaque system because it lacks an explicit formula or an analytic gradient. Furthermore, evaluating  $c(\lambda)$  can be time-consuming, leading to a computationally expensive optimization challenge, as noted by [8].

### 1.4. Optimization problem formulation

For less complex cases, every hyper-parameter can assume unrestricted real values, meaning the feasible set  $X$  of hyper-parameters can be represented as an  $n$ -dimensional real-valued vector space. However, in most practical scenarios, ML model hyper-parameters come from various domains and often have constraints, leading to more complex constrained optimization problems. For instance, the number of features selected in a decision tree must fall between 0 and the total available features, while the number of clusters in  $k$ -means is restricted by the number of data points. Additionally, categorical hyper-parameters are confined to a fixed set of options, such as choosing activation functions or optimizers in neural networks. Consequently, the feasible domain  $X$  often has a complex structure, adding further challenges to the optimization process [9].

In general, the objective of an HPO problem is to find:

$$x^* = \arg \min_{x \in X} f(x) \quad (7)$$

where  $f(x)$  is the objective function to be minimized, such as the error rate or the root mean squared error (RMSE);  $x^*$  is the hyper-parameter configuration that produces the optimal value of  $f(x)$ ; and a hyper-parameter  $x$  can take any value within the search space  $X$ . The objective function  $f(x)$  could be formulated as:

$$f(x) = \frac{1}{n} \sum_{i=1}^n L(y_i, \hat{y}_i(x)) \quad (8)$$

where  $L(y_i, \hat{y}_i(x))$  is the loss function comparing the true labels  $y_i$  and the predicted labels  $\hat{y}_i(x)$ , and  $n$  is the number of data points.

#### 1.4.1. Steps in the HPO process

The key stages in the HPO process are as follows:

---

#### Algorithm 1 Hyper-Parameter Optimization (HPO) Process

---

**Input:** Dataset  $D$ , ML Algorithm  $I$ , Search Space  $\Delta_e$ , Performance Metric  $\rho$ ;

**Output:** Best-performing HP configuration  $\lambda^*$ ;

**Step 1: Define Objective and Metrics:**

**Step 2: Select and Classify HPs:**

Identify and classify HPs (continuous, discrete, categorical, conditional);

Determine optimization technique;

**Step 3: Train Baseline Model:**

Train model with default HPs and evaluate performance;

**Step 4: Optimize HPs:**

Define initial search space  $\Delta_e$  and sampling strategy;

Evaluate configurations, refine search space based on performance;

**Step 5: Finalize Best Configuration:**

Continue optimization until convergence;

Return best configuration  $\lambda^*$ ;

---

### 1.5. Challenges with traditional optimization techniques

Most commonly used, often called traditional optimization techniques are not well aligned for HPO due to several factors:

**1. Non-convexity and Non-differentiability:** The objective function in ML models is typically non-convex and non-differentiable, making conventional optimization methods, which are tailored for convex or differentiable problems, inadequate. These methods often return local optima rather than global ones. Additionally, the lack of smoothness in the objective function can cause traditional derivative-free optimization techniques to struggle with poor performance. In these cases, the optimization problem is challenging due to the inability to rely on gradients or convexity properties to ensure convergence to a global solution. For example:

$$\nabla f(x) = \left( \frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_n} \right) \quad (9)$$

**2. Mixed Variable Types:** ML model hyper-parameters can include continuous, discrete, categorical, and conditional variables. Traditional numerical optimization techniques, which are designed to handle only continuous or numerical variables, are often inadequate for HPO problems due to the presence of these diverse variable types. The optimization problem might involve:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{subject to:} \quad & g_i(x) \leq 0, \quad i = 1, \dots, m, \\ & h_j(x) = 0, \quad j = 1, \dots, p, \\ & x \in X. \end{aligned} \quad (10)$$

**3. Computational Cost:** Training an ML model on large-scale datasets is computationally expensive. HPO techniques often times choose to use data sampling to approximate objective function values. Effective HPO optimization techniques should accommodate these approximate values, but many *black-box* optimization (BBO) approaches demand exact function evaluations, making them less compatible with HPO problems under time and resource constraints.

**4. HP Tuner:** We propose defining the hyper-parameter tuner  $\tau : D, I, \Delta_e, \rho \rightarrow \hat{\lambda}$ , which estimates the true optimal configuration  $\lambda^*$  as  $\hat{\lambda}$  based on a dataset  $D$ , an inducer  $I$ , a search space  $\Delta_e$  for optimization, and a target metric  $\rho$ . The resampling splits  $J$  can either be provided to  $\tau$  or handled internally, allowing for adaptive splitting or multi-fidelity optimization [10].

$$\tau : D, I, \Delta_e, \rho \rightarrow \hat{\lambda} \quad (11)$$

Therefore, appropriate optimization algorithms should be applied to HPO problems to identify the optimal HP configurations for ML models. These algorithms must effectively handle the unique challenges of HPO, ensuring efficient and accurate model tuning. Additionally, it is crucial to consider green issues such as computational complexity and energy consumption. Given the high resource demands of DL and GNNs, optimization algorithms must strive to minimize computational costs and environmental impact. This requires innovative, eco-efficient optimization strategies that balance performance with sustainability. Efficient HPO not only enhances model accuracy but also contributes to reducing the carbon footprint of extensive computational tasks, addressing both environmental and time constraints inherent in advanced ML, DL, and GNN applications. The graphical representation of the General HPO process with inner risk minimization can be seen in Fig. 1.

Finally, it is worth noting that while Taguchi methods and orthogonal arrays are not new and have been extensively applied in fields such as manufacturing, chemistry, and physics, their application in the context of modern machine learning HPO, particularly within software engineering workflows, remains limited. The novelty of this work lies not in the underlying statistical technique, but in the way we have engineered it into a scalable, eco-efficient framework, namely GOAT,

that emphasizes traceable execution, platform-agnostic integration, and compatibility with DevOps practices.

The rest of the paper is structured as follows: Section 2 reviews leading-edge research on HPO algorithms across domains such as ML, DL, and Graph Neural Networks, highlighting the research gap addressed by this work. Section 3 explores established HPO algorithms and techniques, including AutoML and Taguchi's orthogonal array tuning method, comparing their time complexities. Section 4 presents an overview of the selected models and the GOAT method, showcasing the best-performing ones and identifying key HPs. Section 5 discusses numerical simulations. Finally, Section 6 concludes with remarks on the research's limitations and potential future ideas for development.

## 2. Related work

In this section, we will review leading-edge research related to the hyper-parameter optimization problems across various domains. We begin by exploring its applications in machine learning, deep learning, and graph neural networks, followed by an examination of eco-efficiency across these areas. Finally, we identify key research gaps, some of which our research aims to address, contributing to the advancement of knowledge in this field.

### 2.1. Hyper-parameter optimization in machine learning

When selecting optimization methods for machine learning models, the choice largely depends on the types of HPs involved. Bayesian Optimization HyperBand is ideal when random subsets of data represent the entire dataset well, as it efficiently optimizes various hyper-parameters. Bayesian Optimization models work best with smaller hyper-parameter spaces, while Particle Swarm Optimization is suited for larger spaces. The authors found that the Bayesian Optimization - Tree-structured Parzen Estimator achieved an MSE of 59.40 in 0.33 s for SVM, while Bayesian Optimization HyperBand achieved 97.44% accuracy in 3.84 s for KNN [11]. The research [12] systematically reviews multi-objective HPO algorithms published between 2014 and 2020, classifying them into metaheuristic-based, metamodel-based, and hybrid methods. The study highlights the relevance of multi-objective HPO in optimizing trade-offs between performance measures and suggests future research directions, particularly in hybrid algorithms and low fidelity methods [13]. This survey, along with the other reviews [14], also underscores the potential benefits of ensemble methods and advanced single-objective techniques in multi-objective contexts, while noting gaps in applying early stopping criteria and low fidelity approaches to multi-objective HPO [15]. Moreover, in the paper by [16], the authors introduce Sherpa, a HPO library primarily designed for machine learning models. While it is broadly applicable across various machine learning tasks, Sherpa is also effective for deep learning, where computationally intensive, iterative evaluations are common. The library provides researchers with interchangeable optimization algorithms and a real-time interactive dashboard, allowing for efficient HP exploration and tuning on both single machines and distributed clusters, making it a versatile tool for developing robust machine learning models.

### 2.2. Hyper-parameter optimization in deep learning

The authors [17] have proposed DEEP-BO (Diversified, Early-termination-Enabled, and Parallel Bayesian Optimization), a robust algorithm for HPO in deep neural networks, built on four key strategies: variety, early stopping, parallel execution, and transformation of the cost function. DEEP-BO consistently outperformed existing methods across six benchmarks, demonstrating top or near-top performance, especially in parallel processing scenarios. This study highlights the need for robust Bayesian Optimization enhancements to address the challenges of chosen deep neural network tuning. Building on the previous paper, deep learning excels in fields like computer vision and

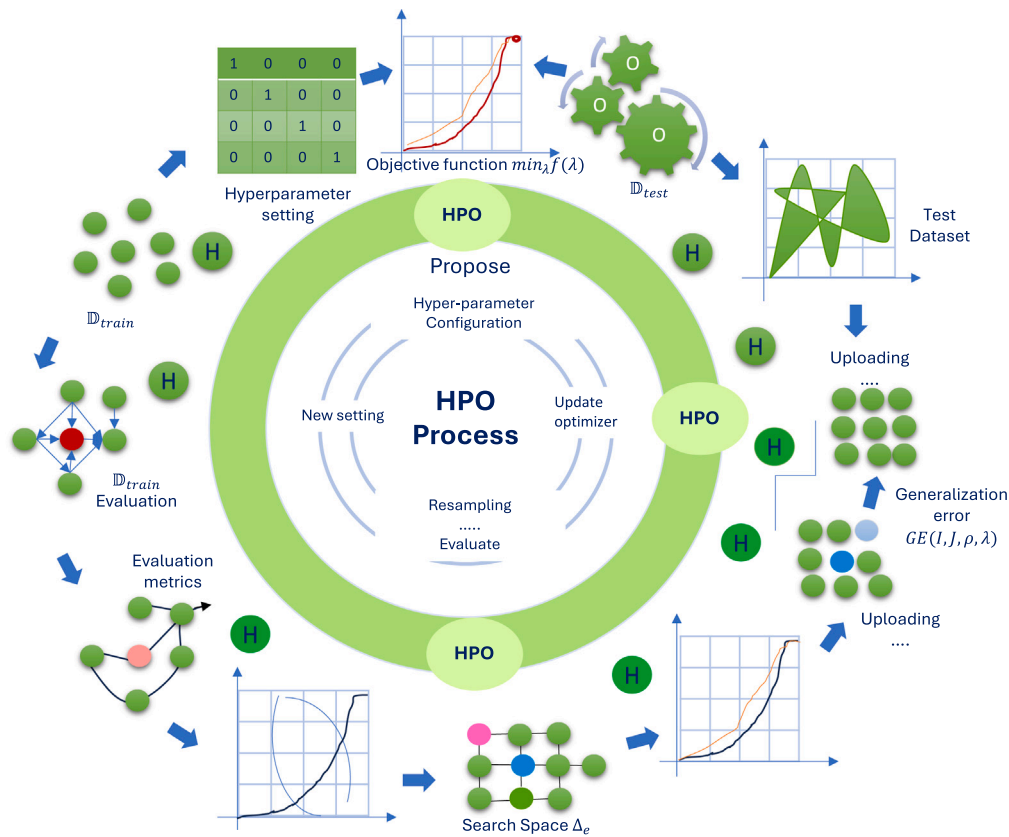


Fig. 1. General HPO process.

natural language processing but requires careful HP tuning due to the complexity of neural network architectures. The authors [18] applied Bayesian HPO to the CIFAR-10 dataset, demonstrating that it effectively enhances model performance by finding optimized values for HPs. The results showed a 6.2% reduction in validation error when using a graphical processing unit compared to a CPU, underscoring the importance of efficient HPO in improving deep learning models. The current paper by [19] compares the Weighted Random Search (WRS) method with several state-of-the-art algorithms specifically for Convolutional Neural Networks (CNNs). The comparison focuses on classification accuracy achieved within the same number of HP combinations. The experiments demonstrate that the WRS algorithm consistently outperforms the other methods, highlighting its effectiveness in optimizing CNN HPs and further reinforcing the importance of robust optimization techniques in deep learning. The study by [20] examines the use of a genetic algorithm to optimize HPs, demonstrating improvements over grid and random search methods. By incorporating verification time into the fitness evaluation and developing an auxiliary function, the study effectively reduces the dimensionality of the optimization problem. Using the MNIST and motor fault diagnosis datasets, the genetic algorithm achieved desirable HPs within a short time frame, highlighting its efficiency and potential for reducing optimization time in deep learning development. The paper [21] focuses on comparing different algorithms – Grid Search, Bayesian Optimization, and Genetic Algorithm – for finding the best hyperparameters in neural networks. The study highlights that hyperparameters, unlike internal model parameters, cannot be estimated directly from data and require tuning to achieve optimal accuracy. Using the Santander Customer Transaction Prediction dataset, the evaluation considered various metrics like accuracy, MSE, and AUC. The results showed that the Genetic Algorithm outperformed the others, particularly in identifying optimal hidden layer configurations and optimizers, though all algorithms showed comparable performance in some aspects.

### 2.3. Hyper-parameter optimization in Graph Neural Networks

In the research by [22] introduced HyperGene, an innovative framework utilizing Graph Neural Networks (GNNs) for hypergraph representation learning. This framework achieved notable improvements, such as a 5.69% increase in hyperedge classification accuracy and a 42.80% reduction in pre-training time, demonstrating its effectiveness in both transductive and inductive settings. In another study, the authors proposed a tree-structured mutation strategy within a genetic algorithm to enhance HPO for GNNs. This approach addresses the challenge of balancing exploration and exploitation during optimization, particularly when computational resources are limited, making the HP tuning process more efficient and effective [23]. The research by [24] introduced KGTuner, a two-stage algorithm that efficiently explores HP configurations on small subgraphs and transfers the best-performing configurations to the full graph for fine-tuning. This method resulted in a 9.1% average relative improvement in tuning HPs for knowledge graph embeddings, showcasing its utility in large-scale graph learning tasks. A novel transfer learning paradigm, Graph Pre-Training and Prompt Tuning (GPPT), for GNNs is introduced by [25]. GPPT achieved a 4.29% improvement in few-shot graph analysis and accelerated model convergence by up to 4.32 times. This method addresses the gap between pretext and downstream tasks, enabling more efficient adaptation of pre-trained models to new tasks. Finally, a research by [26] focused on improving HPO in scenarios where hyper-gradients are unavailable. The authors proposed a method using cubic regularization to avoid local optima and accelerate convergence. This approach demonstrated its effectiveness on both simulated and real-world data, proving to be a valuable technique for challenging HPO problems.

### 2.4. Eco-efficiency across domains

High computational demands and significant energy consumption pose substantial challenges in deploying artificial intelligence (AI)

tools, particularly on resource-constrained devices such as edge platforms and IoT nodes. Traditional deep learning models and hyperparameter optimization (HPO) techniques often exacerbate these issues, rendering them impractical for eco-friendly applications. Addressing these concerns, the FSpINN framework introduces memory-efficient and energy-efficient Spiking Neural Networks (SNNs) that maintain high accuracy during training and inference. By reducing computational requirements, employing fixed-point quantization, and integrating memory and energy considerations into the optimization process, FSpINN achieves 7.5× memory savings and enhances energy efficiency by 3.5× during training and 1.8× during inference, without compromising accuracy [27]. Similarly, the Greedy Hyperparameter Optimization (GHO) algorithm has been developed to facilitate on-the-fly learning in deep neural networks (DNNs). Designed for faster computation on edge devices, GHO optimizes each hyper-parameter individually, leading to a locally optimal solution that aspires to global optimality. Empirical studies demonstrate that GHO is over 3× more energy-efficient and 2× faster than existing state-of-the-art HPO methods across various DNNs and datasets [28].

### 2.5. Research gaps

Our research aims to bridge existing knowledge gaps by developing eco-efficient optimization strategies for HPO in ML, DL, and for deeper and more complex representations such as GNNs. Recognizing the high computational demands of these models, we focus on reducing both computational complexity and energy consumption. We will explore the Taguchi orthogonal array tuning method, which can significantly reduce the number of experiments required across state-of-the-art models like XGBoost, LightGBM, CatBoost, LSTM, GRU, GGNN, and GGSNN. Through a comprehensive case study on software estimation using 46 publicly available datasets, we will demonstrate how this method minimizes experiments and running time in ML, DL, and GNNs, particularly within the contexts of COCOMO, COSMIC, and UCP models. This approach not only enhances model accuracy but also contributes to reducing the carbon footprint associated with extensive computational tasks.

## 3. Well-established HPO algorithms

In this section, we will provide a detailed explanation of the HPO process employed to explore and optimize the performance of various machine learning, deep learning, and graph neural network models. The methodology is structured into several key subsections, each addressing different aspects of the HPO process. We begin with the sub-Section 3.1, which provides an overview of the datasets that will be utilized across three approaches: machine learning, deep learning, and graph neural networks within the software project estimation field. Following this, sub-Section 3.2 introduces model-free algorithms, which serve as foundational techniques for initial model tuning. Moving forward, sub-Section 3.3 explores gradient-based optimization techniques, which are essential for refining model performance by adjusting hyperparameters (HPs) in a systematic manner. We will also cover Bayesian optimization in sub-Section 3.4, multi-fidelity optimization algorithms in sub-Section 3.5, and metaheuristic algorithms in sub-Section 3.6, each of which offers unique advantages in different scenarios. Additionally, we will discuss advanced topics such as bilevel inference perspective in sub-Section 3.7, nested resampling, meta-overfitting, and threshold tuning in sub-Section 3.8, which are crucial for fine-tuning models in complex environments.

The methodology further encompasses the use of AutoML tools, discussed in sub-Section 3.9, which automate the process of model selection and hyper-parameter tuning, significantly reducing the manual effort required. A particular focus is placed on Taguchi's Orthogonal Array Tuning Method (OATM), which is explored in sub-Section 3.10. This method promises to reduce the number of required experiments

by focusing on critical factors, thereby optimizing model performance efficiently. Finally, we conclude the methodological overview with an examination of existing HPO implementations, which have been instrumental in advancing the field and serve as benchmarks in sub-Section 3.11.

### 3.1. Data overview and pre-processing

In the experimental phase, we used 46 distinct datasets were employed from three widely-used methodologies for first effort and later cost estimation in software project development, all of which are publicly accessible via the PROMISE repository (<http://promise.site.uottawa.ca/SERepository/>). The first method included 24 datasets from the COCOMO approach, covering a total of 366 projects. The second method utilized 19 datasets from the FPA approach, with a combined total of 1808 projects. Lastly, the UCP method featured 81 projects. For both the COCOMO and FPA approaches, projects were divided into four categories based on size: small, medium, large, and very large. In contrast, the UCP approach classified projects into three size categories, focusing particularly on very large-scale projects. These projects are highly diverse in terms of technology, programming languages, and other factors, which is reflected in the notably high standard deviation. For instance, in the COCOMO datasets, the smallest project value is 8.4, while the largest is 8211. Furthermore, the datasets exhibit imbalance due to the unequal distribution of samples and values across clusters and datasets within the different methods. An Exploratory Data Analysis (EDA) on the selected datasets supporting these observations for the three approaches is summarized in Table 1. Fig. 2 provides visualizations highlighting the class imbalance, as well as discussions regarding the distributions of Actual Effort, Functional Size, and Real Effort across all three approaches. To address the class imbalance, sampling techniques were applied to the training set using the RandomOverSampler and RandomUnderSampler from the imbalanced-learn library [29]. The optimal sampling technique was identified by comparing the average accuracy across different methods for both baseline and untuned models. This selected method was subsequently applied to create the combined training and validation datasets for the final models.

Initially, to handle missing data, a standard approach is to impute the missing values using the mean of the corresponding attribute. Considering the diverse nature of the datasets, which vary in terms of project size, programming languages, and technologies, it is crucial to implement min-max normalization for scaling [30,31] within a defined range of [0, 1]. This normalization ensures that the scaled dataset, derived from the original, effectively aligns the diverse parameter units saving the statistical characteristics of the original data. After feature selection, outlier removal, and the application of both min-max normalization and data augmentation techniques, the resulting datasets no longer contained missing values. For every input variable, the minimum, maximum, and mean values, along with the standard deviation, were calculated for the Actual Effort, Functional Size, and Real Effort metrics for each approach, as presented in Table 1.

### 3.2. Model-free algorithms

**Grad student descent (GSD)** or “Trial and Error”, is a fundamental method for hyper-parameter tuning [32]. This approach involves entirely manual tuning and is extensively utilized by students and researchers [33,34]. The process is simple: after building an ML model, students experiment with various hyper-parameter values based on their prior experience, educated guesses, or by examining previously evaluated results. This iterative approach continues until the student either runs out of time (typically due to an approaching deadline) or obtains satisfactory results. Consequently, this method requires significant prior knowledge and expertise to identify the optimal hyper-parameters within a constrained time period [11].

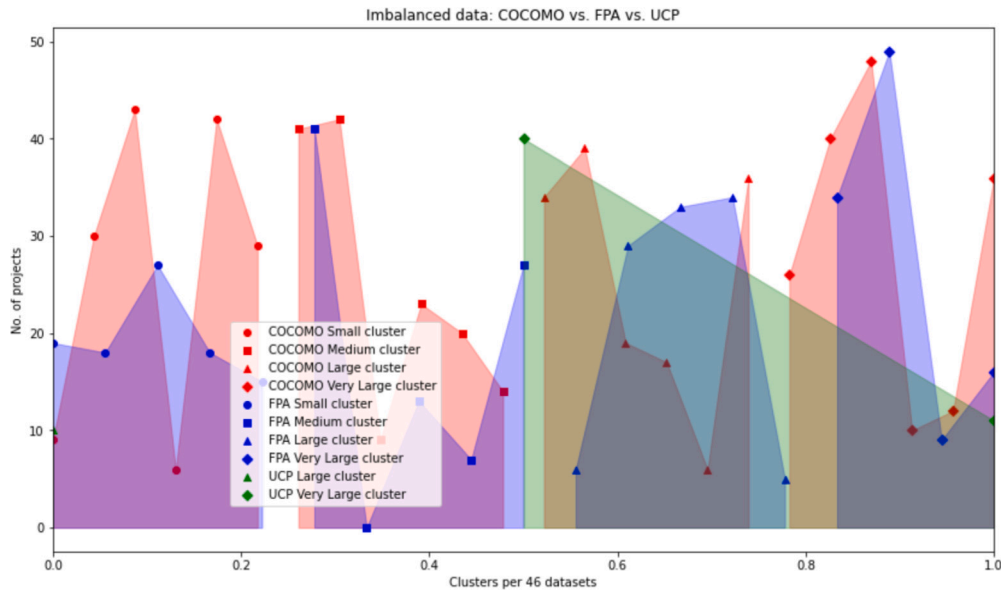


Fig. 2. Imbalanced data across all three software estimation approaches.

Table 1

Basic statistics about the datasets used in all three approaches: COCOMO, FPA, and UCP.

COCOMO - Actual Effort target						
Dataset	Clusters	No. of projects	Min	Max	Mean	Std. dev.
COCOMO2000	Small cluster	107	8.4	99	58.29	19.14
	Medium cluster	155	103	497.7	252.93	111.16
	Large cluster	51	515.2	987	724.57	134.11
	Very large cluster	53	1057.7	8211	2207.99	1386.63
Total (24 datasets)		366	8.4	8211.0	810.95	412.76
FPA - Funtional size target						
Dataset	Clusters	No. of projects	Min	Max	Mean	Std. dev.
COSMIC FPA	Small cluster	505	1	99	43.08	13.59
	Medium cluster	902	100	499	258.08	58.89
	Large cluster	225	500	561	712.41	149.00
	Very large cluster	176	1003	2045	2130.65	1123.06
Total (19 datasets)		1808	1.0	2045.0	656.15	282.85
UCP - Real Effort target						
Dataset	Clusters	No. of projects	Min	Max	Mean	Std. dev.
UCP	Large cluster	10	2692.06	3246.56	2988.39	233.23
	Very large cluster	71	5775	7970	6806.62	329.54
Total (3 datasets)		81	2692.06	7970.0	5533.87	297.43

Mathematical formulation for GSD procedure can be summarized as follows:

**Step 1:** Initialization, defining the set of HPs  $\mathcal{H}$ .

**Step 2:** Iteration, for each  $h_i \in \mathcal{H}$ , evaluate performance  $f(h_i)$  where  $f(h_i)$  represents a specific hyper-parameter configuration.

**Step 3:** Selection, choose the HP  $h^*$  that yields the best performance:

$$h^* = \arg \max_{h_i \in \mathcal{H}} f(h_i) \quad (12)$$

Manual tuning becomes impractical for various problems due to factors such as the large number of hyper-parameters, complex models, time-consuming model evaluations, and non-linear interactions between hyper-parameters [35]. These challenges have led to increased research into methods for automating hyper-parameter optimization [36]. Automated techniques, such as Grid Search and Random Search, have been developed to tackle these issues by systematically exploring the hyper-parameter space, thereby reducing the need for manual effort.

**Grid search (GS)** is a widely used method for exploring hyper-parameter configuration space. It is often considered an exhaustive or

brute-force search method, as it evaluates all possible hyper-parameter combinations within a predefined grid of configurations [37]. GS operates by evaluating the Cartesian product of a user-specified finite set of values. However, GS cannot independently exploit well-performing regions further. To identify global optima, the following procedure must be manually performed:

**Step 1:** Begin with a broad search space and a larger step size.

**Step 2:** Refine the search space and reduce the step size by analyzing the top-performing HP configurations.

**Step 3:** Iterate this refinement process several times until an optimal solution is achieved.

Grid search is easy to implement and parallelize, but its main drawback is inefficiency in high-dimensional HP spaces. The number of evaluations increases exponentially with the number of hyper-parameters, a phenomenon known as the curse of dimensionality [21,38]. For GS, if there are  $k$  parameters, each with  $n$  distinct values, the computational complexity increases exponentially at a rate of  $O(n^k)$ .

Let  $\mathcal{H}$  be the HP configuration space, and  $|\mathcal{H}|$  be the cardinality of this space:

$$|\mathcal{H}| = \prod_{i=1}^k n_i \quad (13)$$

where  $n_i$  is the number of distinct values for the  $i$ th hyper-parameter. The number of evaluations required can be expressed as:

$$E = \prod_{i=1}^k n_i \quad (14)$$

If the time complexity for evaluating a single configuration is  $T$ , the total complexity for grid search becomes:

$$T_{\text{total}} = T \cdot \prod_{i=1}^k n_i \quad (15)$$

Therefore, GS is effective only when the hyper-parameter configuration space is relatively small.

To overcome certain limitations of Grid Search (GS), **Random Search (RS)** was introduced. RS operates similarly to GS but instead of exhaustively testing all possible values, it randomly selects a predefined number of samples within the upper and lower bounds of the search space [39]. These selected samples are evaluated until the given budget is exhausted. The theoretical premise of RS is that, with a sufficiently large configuration space, global optima or close approximations can be found. RS is capable of exploring a larger search space compared to GS within a limited budget [40].

A key advantage of RS is its ability to be easily parallelized and efficiently allocate resources, as each evaluation is independent. In contrast to GS, RS samples a fixed number of parameter combinations from a specified distribution, improving system efficiency by minimizing time spent on underperforming regions [41]. Since the number of evaluations in RS is predetermined as  $n$  before the optimization begins, the computational complexity is  $O(n)$ .

Mathematically, let  $\mathcal{H}$  represent the HP configuration space, and  $\mathcal{H}_{\text{RS}}$  be the set of randomly sampled configurations:

$$\mathcal{H}_{\text{RS}} = \{h_1, h_2, \dots, h_n\} \subset \mathcal{H} \quad (16)$$

The probability  $P$  of sampling a particular configuration  $h$  in the space  $\mathcal{H}$  is:

$$P(h) = \frac{1}{|\mathcal{H}|} \quad (17)$$

For each sample  $h_i$ , the evaluation time complexity  $T$  remains constant, making the total time complexity for RS:

$$T_{\text{total}} = T \cdot n \quad (18)$$

Given enough budget, RS can approximate the global optimum  $h^*$ :

$$h^* \approx \arg \max_{h \in \mathcal{H}_{\text{RS}}} \text{Performance}(h) \quad (19)$$

Although RS is more efficient than GS for large search spaces, it still performs unnecessary function evaluations since it does not exploit previously well-performing regions.

In conclusion, a significant drawback of both RS and GS is that each evaluation during their iterative processes is independent of previous evaluations. As a result, time may be wasted by repeatedly exploring poorly performing regions [42]. This issue can be alleviated by employing other optimization techniques, such as Bayesian optimization, which leverages past evaluation results to guide future evaluations [43]. Bayesian optimization builds a probabilistic model to estimate the performance of configurations and selects the next configuration to evaluate based on this model:

$$h_{\text{next}} = \arg \max_{h \in \mathcal{H}} \text{AcquisitionFunction}(h|\text{Model}) \quad (20)$$

Thus, Bayesian optimization reduces the number of evaluations needed to find the optimum by focusing on promising regions of the search space. This optimization we will explain in the detail in sub-Section 3.4.

### 3.3. Gradient-based optimization

Gradient descent is a classic optimization method that computes the gradient of variables to determine the best direction for moving towards the optimum [44]. Starting from a randomly chosen data point, the algorithm proceeds in the direction opposite to the steepest gradient, guiding it to the next point. This process continues until convergence, at which point a local optimum is found. In the case of convex functions, this local optimum is also the global optimum. The time complexity of gradient-based algorithms for optimizing  $k$  hyper-parameters (HPs) is  $\mathcal{O}(n \cdot k)$ . For certain machine learning algorithms, it is possible to compute the gradient for specific HPs, enabling the use of gradient descent to optimize them [45]. Although gradient-based methods offer faster convergence towards local optima compared to the previously discussed techniques, they have several limitations. First, these methods are limited to continuous hyper-parameters, as categorical ones lack gradient directions. Second, they are most effective for convex functions, as they may only locate a local optimum in the case of non-convex functions. Consequently, gradient-based techniques are confined to situations where the hyper-parameter gradient can be calculated, such as optimizing the learning rate in neural networks (NN).

The gradient descent update rule for a hyper-parameter  $\theta$  is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta_t) \quad (21)$$

where  $\eta$  is the learning rate, and  $\nabla_{\theta} J(\theta_t)$  is the gradient of the objective function  $J$  with respect to  $\theta$ .

For convex functions, the objective function  $J(\theta)$  satisfies:

$$J(\theta_1) - J(\theta_2) \geq \nabla J(\theta_2)^T (\theta_1 - \theta_2) \quad (22)$$

For non-convex functions, reaching the global optimum is not guaranteed, as the gradient descent may converge to a local optimum. The convergence criterion for gradient descent can be expressed as:

$$\|\nabla_{\theta} J(\theta_t)\| < \epsilon \quad (23)$$

where  $\epsilon$  is a small positive value indicating that the gradient is close to zero.

Overall, gradient-based algorithms are useful for optimizing continuous hyper-parameters in convex functions but are limited by their inability to handle categorical hyper-parameters and their potential to converge to local rather than global optima in non-convex functions. Thus, while gradient descent is effective in some scenarios, it is not guaranteed to find global optima for all machine learning algorithms.

### 3.4. Bayesian optimization

Bayesian optimization (BO) is well-known iterative algorithm widely used for hyper-parameter optimization (HPO) problems. Comparing to the grid search (GS) and random search (RS), BO selects future evaluation points based on previously obtained results. BO uses two key components to determine the next hyper-parameter configuration: a surrogate model and an acquisition function [46]. The surrogate model fits all the currently observed data points to approximate the objective function. After constructing the predictive distribution using the surrogate model, the acquisition function evaluates different points by balancing exploration and exploitation. Exploration involves sampling in unexplored areas, while exploitation focuses on sampling in promising regions where the global optimum is likely to be, based on the posterior distribution [47]. BO effectively balances these processes to identify the optimal regions and avoid missing better configurations in unexplored areas.

Mathematical formulation for BO procedure can be summarized as follows:

**Step 1:** Develop a probabilistic model that approximates the objective function.

**Step 2:** Use this model to determine the best HP settings.

**Step 3:** Test these HP settings on the real objective function and assess their performance.

**Step 4:** Refine the probabilistic model using the latest results.

**Step 5:** Continue iterating through steps 2–4 until the predefined number of iterations is completed.

BO becomes more prominent than GS and RS by updating the surrogate model after each evaluation. Running the surrogate model is often way less expensive than evaluating the entire objective function, allowing BO to detect optimal hyper-parameter combinations more efficiently [43]. However, since BO relies on previously tested values, it is inherently a sequential method and is challenging to parallelize. Nevertheless, in most cases it can identify near-optimal hyper-parameter combinations within a few iterations. Common surrogate models used in BO include Gaussian processes (GP), random forests (RF), and tree-structured Parzen estimators (TPE). Consequently, there are three main types of BO algorithms based on their surrogate models: BO-GP, BO-RF, and BO-TPE. An alternative name for BO-RF is Sequential Model-based Algorithm Configuration (SMAC).

**Gaussian Process (GP)** is a common surrogate model used in Bayesian Optimization (BO) for approximating the objective function. Assuming that the function  $h$ , with a mean  $\nu$  and covariance  $\tau^2$ , is a realization of a Gaussian Process, the predictions follow a normal distribution:

$$p(z|w, C) = \mathcal{N}(z|\mu', \sigma'^2) \quad (24)$$

In this context,  $C$  denotes the configuration space of hyper-parameters, and  $z = h(w)$  represents the evaluation result for each hyper-parameter value  $w$ . Once a set of predictions is made, the next points to evaluate are selected from the confidence intervals generated by the BO-GP model. Each newly evaluated data point is incorporated into the sample set, and the BO-GP model is updated with this new information. This process is repeated until a stopping criterion is reached. The time complexity for applying a BO-GP model to a dataset of size  $n$  is  $\mathcal{O}(n^3)$ , and the space complexity is  $\mathcal{O}(n^2)$ . A major drawback of BO-GP is its cubic time complexity with respect to the number of data points, limiting its potential for parallelization. Additionally, it is mainly used for optimizing continuous variables [48].

**Sequential Model-based Algorithm Configuration (SMAC)** is another popular surrogate model used in Bayesian Optimization, leveraging an ensemble of regression trees. Assuming a Gaussian model  $\mathcal{N}(z|\mu', \sigma'^2)$ , where  $\mu'$  and  $\sigma'^2$  represent the mean and variance of the regression function  $q(w)$ :

$$\mu' = \frac{1}{|A|} \sum_{j=1}^A q_j(w) \quad (25)$$

where  $A$  is the set of regression trees, and  $q_j(w)$  is the prediction of the  $j$ th tree. The SMAC process involves constructing a Random Forest (RF) with  $A$  regression trees, each created by sampling  $n$  instances with replacement from the training data. For each tree, a split node is chosen from  $d$  hyper-parameters. To control computational costs, predefined values are used for the minimum number of instances required for further splits and the number of trees to be grown. The RF model estimates the mean and variance for each new configuration. The time complexity for fitting the model and predicting variances using SMAC is  $\mathcal{O}(\log n)$ , which is significantly lower than that of BO-GP [49].

**The Tree-structured Parzen Estimator (TPE)** is another widely-used surrogate model in Bayesian Optimization. Unlike BO-GP, which defines a predictive distribution, BO-TPE constructs two density functions,  $l(w)$  and  $h(w)$ , as generative models for the domain variables. The observed results are divided into two groups: good and poor outcomes, based on a predefined percentile  $\alpha$ . These two groups are then modeled using simple Parzen windows:

$$p(w|z, C) = \begin{cases} l(w) & \text{if } z < \alpha, \\ h(w) & \text{otherwise.} \end{cases} \quad (26)$$

During the acquisition phase, the expected improvement is calculated as the ratio of these two density functions, which guides the selection of new configurations for evaluation. The Parzen estimators are structured as a tree, preserving the conditional dependencies, allowing TPE to effectively manage conditional hyper-parameters. The time complexity of BO-TPE is  $\mathcal{O}(n \log n)$ , which is lower than that of BO-GP [50].

BO methods are particularly effective for many hyper-parameter optimization (HPO) problems, even when the objective function  $h$  is stochastic, non-convex, or discontinuous. However, a notable limitation of BO models is their tendency to converge to local rather than global optima if they do not strike the right balance between exploration and exploitation. Additionally, BO models are difficult to parallelize due to the interdependence of their intermediate results.

### 3.5. Multi-fidelity optimization algorithms

A significant challenge in HPO is the lengthy execution time, which escalates as the hyper-parameter search space expands and dataset size increases. This process can take hours, days, or even longer [51]. To address the constraints of time and computational resources, multi-fidelity optimization techniques are often used. These approaches shorten the execution time by working with subsets of the dataset or feature space [52]. Multi-fidelity optimization combines both low-fidelity and high-fidelity evaluations, making them suitable for practical applications [53]. Low-fidelity evaluations involve assessing a smaller subset at a lower cost, albeit with reduced generalization performance. In contrast, high-fidelity evaluations examine a larger subset, yielding better generalization performance but at a greater computational expense. Multi-fidelity optimization algorithms discard underperforming configurations after each round of hyper-parameter evaluation on generated subsets, ensuring that only the top-performing configurations are tested on the full training dataset. The expected execution time  $T_{\text{total}}$  can be expressed as:

$$T_{\text{total}} = \sum_{i=1}^n T_i \quad (27)$$

where  $T_i$  is the execution time for each hyper-parameter configuration  $i$  and  $n$  is the number of configurations.

Multi-fidelity optimization algorithms, particularly bandit-based algorithms, have demonstrated success in addressing deep learning optimization problems [54]. Two prevalent bandit-based techniques are successive [55] and Hyperband [56]. The cost function  $C$  in multi-fidelity optimization can be represented as:

$$C = \sum_{i=1}^k c_i \cdot f_i \quad (28)$$

where  $c_i$  is the cost of the  $i$ th fidelity evaluation and  $f_i$  is the frequency of evaluation.

These techniques efficiently manage the trade-off between cost and performance by progressively narrowing down the hyper-parameter configurations, focusing computational resources on the most promising candidates. Theoretically, exhaustive methods can identify the optimal HP combination by evaluating all possible combinations. However, practical applications must consider various constraints, including limited time and resources, which are collectively referred to as budgets (B). To address the inefficiencies of Grid Search (GS) and Random Search (RS), successive halving algorithms were proposed [56].

The core process of applying successive halving algorithms for HPO is as follows: Initially, consider  $n$  different sets of hyper-parameter combinations, where each set is evaluated with an equally allocated budget ( $b_1 = B/n$ ). After each round of evaluations, the least successful half of the hyper-parameter configurations are eliminated. The remaining higher-performing configurations advance to the next round, where they are given a budget twice the size of the previous one ( $b_{i+1} = 2 \cdot b_i$ ).

This cycle is repeated until the best hyper-parameter configuration is identified.

**Successive halving** is more efficient than Random Search (RS) but is affected by the balance between the number of hyper-parameter configurations and the budget allocated to each configuration [57]. Thus, the primary concern in successive halving is how to distribute the budget effectively, deciding whether to test fewer configurations with a larger budget or more configurations with a smaller budget.

The budget allocation for each iteration can be expressed as:

$$b_i = \frac{B}{2^i} \quad (29)$$

where  $B$  is the total budget, and  $i$  is the iteration number.

Additionally, the performance evaluation function for a configuration  $x$  can be defined as:

$$r(x) = \frac{1}{|B|} \sum_{r \in B} (r(x) - \hat{r})^2 \quad (30)$$

where  $B$  is a set of regression trees in the forest,  $r(x)$  is the performance of  $x$ , and  $\hat{r}$  is the mean performance.

In cases where higher budgets are allocated, the performance function  $g(x)$  can be adjusted as:

$$g(x) = \begin{cases} f(x) & \text{if } y > y', \\ g(x) & \text{otherwise.} \end{cases} \quad (31)$$

This adjustment ensures that higher performance configurations are given priority in the evaluation process, optimizing the allocation of resources.

Hyperband [56] overcomes the limitations of successive halving algorithms by dynamically determining an appropriate number of configurations. It manages the trade-off between the number of hyper-parameter configurations  $n$  and their allocated budgets by partitioning the total budget  $B$  into smaller portions and assigning these to each configuration ( $b = B/n$ ). Successive halving is then utilized as a subroutine on each set of random configurations to identify poorly-performing hyper-parameter configurations, thereby improving efficiency. The main steps of the Hyperband algorithm are as follows:

#### Algorithm 2 Hyperband

**Input:**  $b_{\max}$ ,  $b_{\min}$

1: Calculate  $s_{\max} = \log\left(\frac{b_{\max}}{b_{\min}}\right)$

2: **For**  $s = s_{\max}$  **down to** 0 **do**

3: Determine the budget  $n$  for the current iteration.

4: Sample  $n$  configurations.

5: Perform successive halving on the sampled configurations.

6: **End For**

7: **Return** the best configuration found.

First, the budget constraints  $b_{\min}$  and  $b_{\max}$  are defined based on factors such as the total number of data points, the minimum number of instances required to train a valid model, and the available resources. The number of configurations  $n$  and the budget allocated to each configuration are determined using  $b_{\min}$  and  $b_{\max}$ . Configurations are then sampled according to  $n$  and  $b$ , and passed to the successive halving procedure. Successive halving eliminates poorly-performing configurations and allows better-performing ones to advance to the next iteration. This cycle continues until the optimal hyper-parameter configuration is found. By employing successive halving, Hyperband achieves a computational complexity of  $\mathcal{O}(n \log n)$  [56].

Bayesian Optimization HyperBand (BOHB) [58] is a more advanced HPO technique that combines Bayesian Optimization with Hyperband, utilizing the strengths of both approaches while addressing their limitations. While the original Hyperband algorithm uses random search to explore the hyper-parameter space, which can be inefficient, BOHB replaces random search with Bayesian Optimization (BO), leading to higher performance and faster execution by efficiently utilizing parallel resources for hyper-parameter optimization.

In BOHB, the Tree-structured Parzen Estimator (TPE) serves as the surrogate model for Bayesian Optimization, using multidimensional kernel density estimators. The computational complexity of BOHB remains  $\mathcal{O}(n \log n)$  [58].

#### Algorithm 3 BOHB Algorithm

1. **Initialization:** Define the total budget  $B$  and split it into multiple configurations.

2. **Sampling:** Use TPE to sample a set of hyper-parameter configurations.

3. **Evaluation:** Allocate a small budget to each configuration and evaluate their performance.

4. **Successive Halving:** Iteratively increase the budget for well-performing configurations and eliminate poorly-performing ones.

5. **Update Model:** Update the TPE model with the new performance data and repeat the sampling and evaluation process.

The performance function for each configuration  $z$  can be represented as:

$$J(z) = \frac{1}{|A|} \sum_{s \in A} (q(z) - \bar{v})^2 \quad (32)$$

where  $J(z)$  is the loss function,  $A$  is the set of regression trees in the ensemble,  $q(z)$  represents the performance of  $z$ , and  $\bar{v}$  is the mean performance.

The budget allocation per iteration can be calculated as:

$$b_i = \frac{B}{2^i} \quad (33)$$

where  $b_i$  is the budget for the  $i$ -th iteration, and  $B$  is the total budget.

BOHB has demonstrated superior performance over many other optimization techniques when tuning Support Vector Machine (SVM) and Deep Learning (DL) models [58]. However, a BOHB constraint is that evaluations on subsets with small budgets must be representative of evaluations on the entire training set; otherwise, BOHB may exhibit slower convergence compared to standard BO models.

### 3.6. Metaheuristic algorithms

Metaheuristic algorithms [59] comprise a broad class of techniques, often inspired by natural and biological phenomena, and are frequently utilized to address complex optimization problems. Unlike traditional optimization approaches, metaheuristics are particularly effective for solving non-convex, non-continuous, and non-smooth optimization challenges [60]. A prominent subset of metaheuristics is population-based optimization algorithms (POAs), which include methods such as genetic algorithms (GAs), evolutionary algorithms, evolutionary strategies, and particle swarm optimization (PSO). POAs function by generating and iteratively refining a population of candidate solutions. Each candidate is evaluated over successive generations until the global optimum is located [61]. The main differences between various POAs lie in the mechanisms they use for generating and selecting populations [62].

POAs are inherently parallelizable, as a population of  $N$  individuals can be evaluated simultaneously across up to  $N$  threads or machines [63]. Among POAs, genetic algorithms and particle swarm optimization are particularly popular for tackling HPO problems.

For example, the performance of the  $i$ th individual can be evaluated using a fitness function  $f$ :

$$f_i = f(x_i) \quad (34)$$

where  $x_i$  represents the  $i$ th individual in the population.

The overall process can be encapsulated by:

$$x_{\text{new}} = \text{Crossover}(\text{Mutation}(x_{\text{selected}})) \quad (35)$$

This iterative approach continues until the algorithm converges to the global optimum or a predefined termination criterion is met.

**Algorithm 4** Generic POA Steps

1. **Initialization:** Create an initial set of potential solutions.
2. **Evaluation:** Measure the fitness of each solution within the population.
3. **Selection:** Select individuals based on their fitness levels to form a pool for recombination.
4. **Crossover and Mutation:** Perform crossover and mutation to produce the next generation of solutions.
5. **Iteration:** Repeat the evaluation and selection process for a number of generations until convergence is achieved.

Genetic Algorithm (GA) [11] is a popular metaheuristic algorithm inspired by evolutionary theory. It operates on the principle that individuals with better survival capabilities and adaptability are more likely to survive and pass on their traits to future generations. Over successive generations, better individuals are more likely to thrive, while less fit individuals gradually disappear, eventually leading to the identification of the global optimum [64]. In applying Genetic Algorithms (GA) to Hyper-Parameter Optimization (HPO) problems, each chromosome represents a specific hyper-parameter configuration, where its decimal value corresponds to the actual input value of the hyper-parameter. Chromosomes are made up of genes, represented as binary digits. Crossover and mutation operations are applied to these genes to explore the search space. The population encompasses all potential values within the initialized chromosome ranges, and a fitness function is used to assess the performance of these configurations [64].

**Algorithm 5** Genetic Algorithm Process

1. **Initialization:** Randomly generate an initial population of chromosomes.
2. **Evaluation:** Assess the performance of each individual using a fitness function that represents the objective function of a machine learning model.
3. **Selection, Crossover, Mutation:** Select well-performing chromosomes, perform crossover to combine traits, and apply mutations to introduce variations.
4. **Iteration:** Repeat the evaluation and genetic operations until the stopping condition is met.
5. **Output:** Return the optimal HP configuration.

The fitness function  $f$  for an individual  $x$  can be expressed as:

$$f(x) = \text{PerformanceMetric}(x) \quad (36)$$

where  $\text{PerformanceMetric}(x)$  measures how well the hyper-parameter configuration  $x$  performs.

Crossover and mutation operations can be represented as:

$$x_{\text{new}} = \text{Crossover}(\text{Mutation}(x_{\text{selected}})) \quad (37)$$

The GA's computational complexity is  $\mathcal{O}(n^2)$  [65], making it sometimes inefficient due to its low convergence speed.

Despite its simplicity and ease of implementation, GA introduces additional hyper-parameters such as population size, crossover rate, and mutation rate. Additionally, GA is inherently sequential, which can hinder parallelization efforts and affect its efficiency.

Particle Swarm Optimization (PSO) [66] is an evolutionary algorithm widely used for optimization problems. Inspired by the social behavior of biological populations, PSO enables a group of particles, or swarm, to explore the search space in a semi-random manner [36]. The algorithm identifies optimal solutions through cooperation and information distributing among individual particles in the swarm.

In PSO, a swarm  $S$  consists of  $n$  particles. Each particle represents a potential solution and has a position  $x_i$  and a velocity  $v_i$ . The particles update their positions and velocities based on their own experience and the experience of their neighbors, following these rules:

**Algorithm 6** Particle Swarm Optimization (PSO) Process

1. **Initialization:** Randomly initialize the positions  $x_i$  and velocities  $v_i$  of all particles.
2. **Evaluation:** Calculate the fitness of each particle based on a predefined fitness function  $f(x_i)$ .
3. **Update Velocities and Positions:** For each particle, update its velocity and position using the formulas:

$$\begin{aligned} v_i(t+1) &= w \cdot v_i(t) + c_1 \cdot r_1 \cdot (p_i - x_i(t)) \\ &\quad + c_2 \cdot r_2 \cdot (g - x_i(t)) \\ x_i(t+1) &= x_i(t) + v_i(t+1) \end{aligned} \quad (38)$$

where  $w$  is the inertia weight,  $c_1$  and  $c_2$  are cognitive and social coefficients,  $r_1$  and  $r_2$  are random numbers between 0 and 1,  $p_i$  is the personal best position, and  $g$  is the global best position.

4. **Iteration:** Repeat the evaluation and update steps until convergence or termination criteria are met.

Compared to Genetic Algorithms (GA), PSO is easier to implement as it does not involve crossover and mutation operations. In GA, all chromosomes share information, causing the entire population to move uniformly towards the optimal region. In PSO, only the best particle's information is shared, resulting in a more directed and efficient search process [67]. The computational complexity of PSO is  $\mathcal{O}(n \log n)$  [68], and it typically converges faster than GA. Additionally, the independent operation of particles allows for easy parallelization, enhancing model efficiency [36]. However, PSO requires proper population initialization to avoid local optima, especially for discrete hyper-parameters [69]. This initialization can be challenging and may necessitate prior experience or the use of population initialization techniques like opposition-based optimization [70] and space transformation search methods [71]. Incorporating these techniques can increase execution time and resource requirements.

## 3.7. Bilevel inference perspective

Hyper-Parameter Optimization (HPO) aims to find an approximately optimal hyper-parameter configuration (HPC)  $\hat{\lambda} = \tau(D, I, \Delta, \rho)$  by optimizing it with respect to the resampled performance  $c(\lambda) = GE(I, J, \rho, \lambda)$ . This process is essentially risk minimization with respect to hyper-parameters, where the goal is to find optimal parameters  $\hat{\lambda}$  that minimize the risk of the predictor  $\hat{f}$ . The objective is to ensure the predictor performs well on validation data, evaluated via:

$$\hat{\lambda}, \hat{\theta} = \arg \min_{(\lambda, \theta)} \sum_{i=1}^m \rho(y_i, F_{\lambda, \theta}) \quad (39)$$

Here  $\hat{f}_{\lambda, \theta} = I(D_{\text{train}}, \hat{\lambda})$  and  $F_{\lambda, \theta}$  is the prediction matrix of  $\hat{f}$  on validation data, for a pointwise loss function. This formulation applies to a single holdout split ( $J_{\text{train}}, J_{\text{test}}$ ) to illustrate the connection between risk minimization and HPO. This concept can be generalized for arbitrary resampling with multiple folds, but the complexity arises from the need to fit one or multiple models  $f$  during its computation, highlighting its black-box nature. Conceptually, this can be seen as a bilevel inference mechanism, where the parameters  $\hat{\theta}$  of  $f$  are estimated in the first level for a given HPC, and the hyper-parameters  $\hat{\lambda}$  are inferred in the second level.

Bilevel optimization offers two key advantages over directly minimizing the joint risk of both parameters and hyper-parameters [72]. First, learners are typically designed so that optimizing the first-level parameters becomes more computationally efficient when hyper-parameters are fixed, often making the first-level problem convex, while the combined problem is not. Second, since bilevel optimization focuses on minimizing the generalization error at the higher level, it usually results in a model that is less susceptible to overfitting. In the second level of the bilevel approach, the risk function to be minimized

is the generalization error (GE) instead of the empirical risk  $R_{\text{emp}}(\theta)$ . In this context, the process of hyper-parameter optimization (HPO) can be naturally interpreted as second-level inference [72].

We can define a learner with integrated tuning as a mapping  $T : D \rightarrow H, D \rightarrow I(D)$ , which maps a dataset  $D$  to a model  $\hat{f}_{\hat{\lambda}}$ , where the hyper-parameter configuration (HPC) is set to  $\hat{\lambda}$ , optimized by the tuner  $\tau$  on  $D$ , and trained on the entire dataset. This procedure involves a two-step process where tuning precedes the final model fitting. If such a learner is evaluated using cross-validation, we arrive at the concept of nested cross-validation (CV).

Resampling methods tackle this by repeatedly partitioning the data into training and test sets, applying an estimator to each split, and then aggregating the performance metrics. A resampling strategy can be represented by a vector of splits  $J = \{(J_{\text{train},1}, J_{\text{test},1}), \dots, (J_{\text{train},B}, J_{\text{test},B})\}$ , where  $J_{\text{train},i}$  and  $J_{\text{test},i}$  are index vectors, and  $B$  denotes the number of splits. The estimator is given by:

$$GE_{J_{\text{train},J_{\text{test}}}}(I, \lambda, n_{\text{train}}, \rho) = \rho(y_{J_{\text{test}}}, F_{J_{\text{test}}}(I(D_{\text{train}}, \lambda))) \quad (40)$$

Aggregating these, we get:

$$\begin{aligned} GE(I, J, \rho, \lambda) &= \arg \left( GE_{J_{\text{train},1}J_{\text{test},1}}(I, \lambda, |J_{\text{train},1}|, \rho), \dots, \right. \\ &\quad \left. GE_{J_{\text{train},B}J_{\text{test},B}}(I, \lambda, |J_{\text{train},B}|, \rho) \right) \\ &= \arg \left( \rho(y_{J_{\text{test},1}}, F_{J_{\text{test},1}}(I(D_{\text{train},B}, \lambda))), \dots, \right. \\ &\quad \left. \rho(y_{J_{\text{test},B}}, F_{J_{\text{test},B}}(I(D_{\text{train},B}, \lambda))) \right) \end{aligned} \quad (41)$$

The aggregator ( $\arg$ ) is often the value of mean. For this to be a valid estimator, the training set sizes should be approximately equal, i.e.,  $n_{\text{train}} \approx n_{\text{train},1} \approx \dots \approx n_{\text{train},B}$ .

### 3.8. Nested resampling, meta-overfitting and threshold tuning

The performance evaluation of any learning algorithm should always be performed through resampling on independent test sets to ensure an unbiased estimation of its generalization error [73]. This step is crucial because assessing the model on the same dataset used for training can lead to an overly optimistic bias. To combat this, resampling is applied to reduce the generalization error, and an additional outer resampling loop is integrated around the inner HPO-resampling procedure, referred to as nested resampling. In this nested framework, an outer training set is selected for model building, while an outer test set is held out. Each candidate hyper-parameter configuration (HPC)  $\zeta$  is evaluated using inner resampling on the outer training set. The best-performing HPC  $\zeta$  is then used to fit the final model on the outer training set, which is subsequently tested on the outer test set. This process repeats across all outer loops, and at the end, the results from the outer test sets are combined.

For example, consider a binary classification task where the model  $\zeta$  ignores the data and predicts class labels randomly but in a balanced manner, resulting in a true misclassification error of 0.5. Tuning such a learner through random search is ineffective, as  $\zeta$  has no impact, but more tuning iterations increase the chance of randomly finding a model that correctly labels some data by chance. This optimistic bias worsens with more tuning iterations, smaller datasets, or higher variability in the generalization error estimator. To prevent this bias, nested resampling is employed, as described in [74]. In nested resampling, the dataset is split into outer training and test sets. Each candidate hyper-parameter  $\zeta$  is evaluated via inner resampling on the outer training set, and the best-performing configuration is selected to train the final model, which is evaluated on the outer test set. The process is repeated over all outer loops, and the test performances are aggregated.

Various resampling strategies can be applied to both inner and outer loops of the nested resampling process, with nested cross-validation (CV) and nested holdout being the most popular [75]. Nested holdout is commonly referred to as the train-validation-test method. The objective

of nested CV is to generate a performance distribution across the outer test sets, where the selected hyper-parameters during CV are only temporary, intended for estimating generalization error [76]. If final model parameters are required for further investigation, the tuning should be rerun on the entire dataset, which implies a final round of tuning for second-level inference. While nested resampling provides an unbiased assessment of the HPO process, its primary purpose is performance estimation and does not directly enhance the model itself. Bias in performance estimation is typically not problematic for optimization, as long as the ranking of the evaluated HPCs remains consistent. However, repeated evaluations can introduce randomness or overfitting to the inner resampling folds, potentially leading to the selection of suboptimal HPCs—a phenomenon known as over-tuning, meta-overfitting, or over-searching. This issue is akin to multiple hypothesis testing, though unlike regularization in empirical risk minimization, few approaches currently exist to mitigate overfitting in HPO.

Most classifiers produce probabilities or real-valued decision scores rather than direct class labels, yet many metrics require predicted class labels [77]. These scores are converted to labels using a threshold  $t$ , where a score  $f(x)$  is classified as 1 if  $f(x) \geq t$  and 0 otherwise. For binary classification, the default thresholds are generally  $t = 0.5$  for probabilities and  $t = 0$  for decision scores. However, optimal thresholds may vary based on the metric and classifier. Threshold tuning is the process of optimizing  $t$  to enhance model performance. Although the threshold is technically a hyper-parameter, it can be optimized separately after the model has been built.

Once models are fitted and predictions obtained for all test sets through resampling, the joint test set scores can be compared against the test set label to find the optimal threshold  $\hat{t}$ :

$$\hat{t} = \arg \min_t (\hat{y}, F \geq t) \quad (42)$$

where the Iverson bracket is evaluated component-wise, with  $t \in [0, 1]$  for probabilistic classifiers and  $t \in \mathbb{R}$  for score-based classifiers. Since  $t$  is a scalar,  $\hat{t}$  can be found easily via a line search. This two-step approach ensures that each HPC is paired with its optimal threshold, and the performance  $c(\lambda)$  is defined as the optimal value for  $\lambda$  in combination with  $\hat{t}$ .

For multi-class classification, class probabilities or scores  $\hat{\pi}_k(x)$  are divided by threshold weights  $w_k$ , and the class with the maximum  $\frac{\hat{\pi}_k(x)}{w_k}$  is chosen as the predicted label. The weights  $w_k$  are optimized similarly to  $t$  in binary classification. In practice, threshold tuning is often a post-processing step in the machine learning pipeline.

### 3.9. AutoML

To achieve more flexible and efficient pipelining, especially within the context of AutoML, we can represent the pipeline as a directed acyclic graph (DAG) [78]. This approach is particularly valuable for HPO for both Machine learning and Deep learning and as well as for more complex interrelationships that graph neural networks have the ability to create. In this DAG-based pipeline, the initial node ingests the raw data, and the final node produces the predictions. Each node in the graph corresponds to a specific task, such as pre-processing, learning, post-processing, or a control operation that manages the flow of data to subsequent nodes. This structure enables more flexible and adaptable workflows, customized for specific datasets and tasks [79].

Considering examples from Figs. 3, 4, and 5, which illustrate pipelines for machine learning with CatBoost, LightGBM, and XGBoost; deep learning with LSTM and GRU; and graph neural networks with GGNN and GGSNN, we provide a detailed explanation of the hyper-parameter optimization process across these models. Violin plots in Fig. 3 each represent a key hyper-parameter: for XGBoost, it is `nrounds`; for LightGBM, it is `eta`; and for CatBoost, it is `tree_depth`. The violin plots in Fig. 4 show the key parameter for LSTM and GRU, such as `number_of_units`. Finally, the violin plots in Fig. 5 show

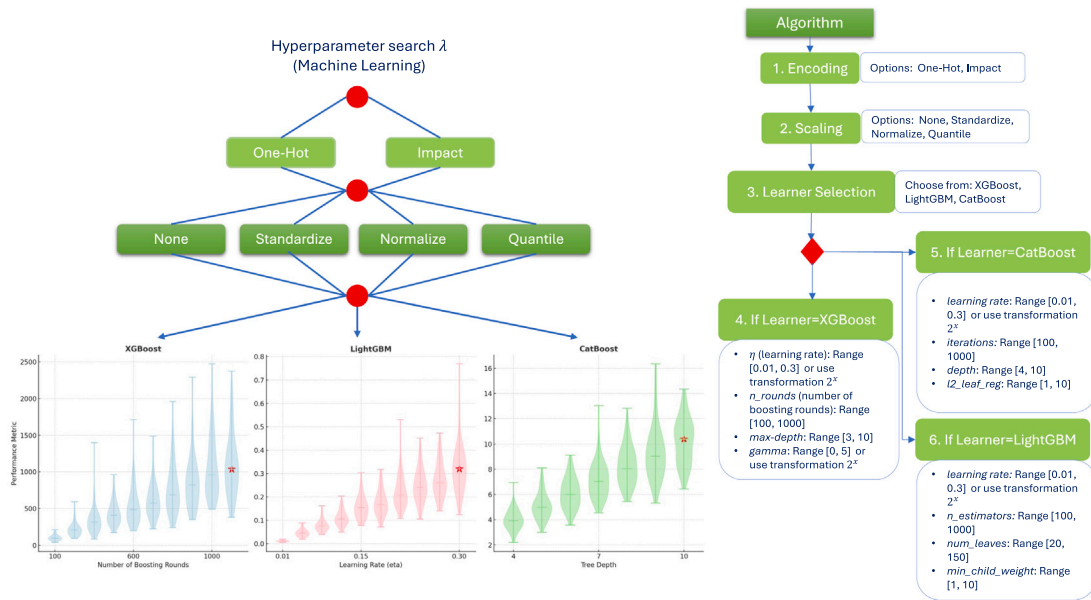


Fig. 3. Hyperparameter search space ( $\lambda$ ) and corresponding pseudo-code - Machine Learning.

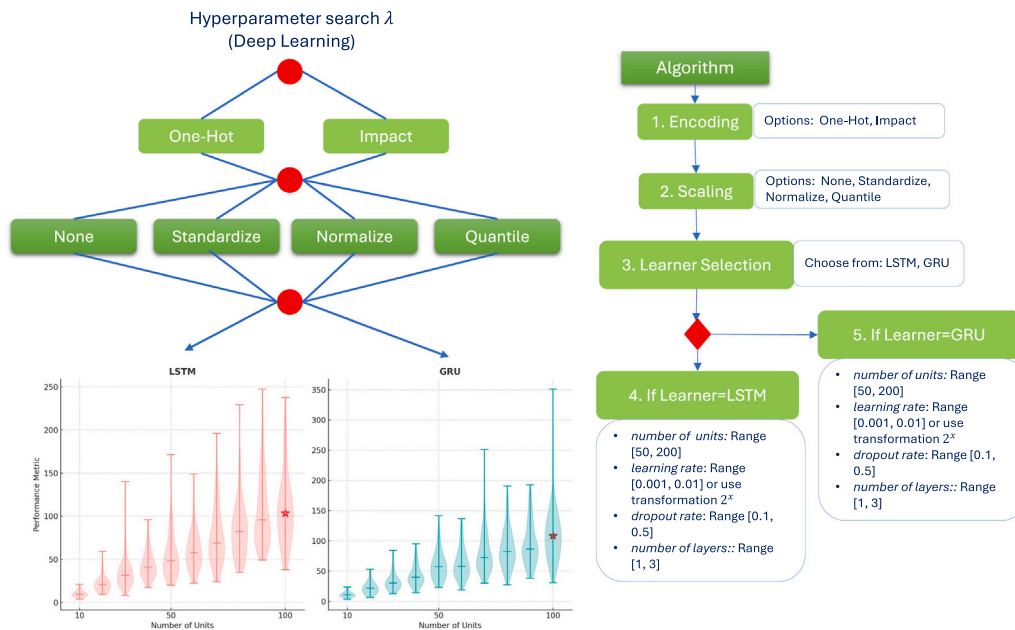


Fig. 4. Hyperparameter search space ( $\lambda$ ) and corresponding pseudo-code - Deep Learning.

the key parameter for GGNN and GGSNN, such as `number_of_hidden_units`. These figures depict multiple mutually exclusive pre-processing steps and various algorithmic choices within each pipeline. The branching operators in the pipeline are configured through categorical parameters, which later define the data flow and create multiple “modeling paths” within the graph. This flexibility results in a complex hyper-parameter space, where different nodes and their corresponding HPs become active depending on the branching configurations, forming a hierarchical search space. Choices between these steps and algorithms are represented by branching operators, configured through categorical parameters that determine the flow of data, resulting in multiple “modeling paths” within the graph. The hyper-parameter space created by such a flexible pipeline is inherently complex. Depending on the settings of branching hyper-parameters, different nodes and their corresponding hyper-parameters become active, creating a hierarchical search space. This is particularly relevant for ML, DL, and GNNs, where

the selection of pre-processing steps, model architectures, training procedures, and postprocessing steps can vary widely.

For example, in DL, the pipeline might include choices between different neural network architectures (e.g., CNNs, RNNs, transformers), different optimization algorithms, and varying regularization techniques. In GNNs, the pipeline might include choices between graph convolutional networks (GCNs), graph attention networks (GATs), and other GNN variants, as well as decisions on how to handle graph preprocessing and feature extraction.

In the context of AutoML, combining a DAG-based pipeline with an efficient tuner is crucial. The tuner can explore the hierarchical search space to identify the best hyper-parameters for each component of the pipeline. By doing so, it can configure the pipeline in a data-dependent manner, ensuring optimal performance across a wide range of datasets and tasks. The key principle of AutoML is to automate the

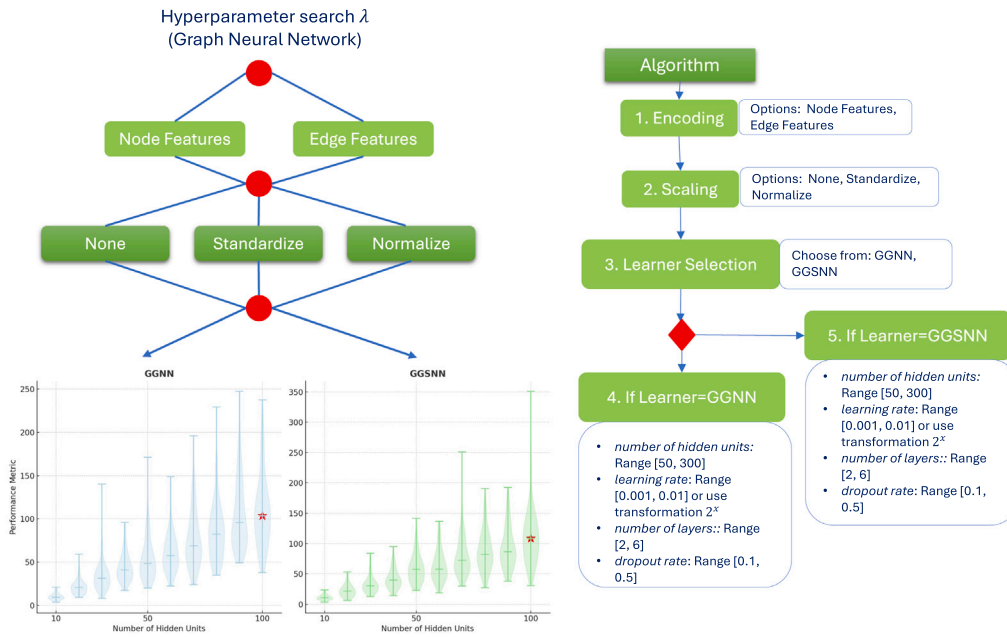


Fig. 5. Hyperparameter search space ( $\lambda$ ) and corresponding pseudo-code - Graph Neural Networks.

process of pipeline configuration, including the selection of preprocessing steps, model architectures, and hyper-parameters. This automation leverages advanced search algorithms and HPO techniques to navigate the complex hyper-parameter space effectively [80].

By integrating a flexible pipeline with a robust AutoML framework, we can achieve state-of-the-art performance in ML, DL, and GNN applications, as demonstrated by works such as [81,82], and [83].

### 3.10. Taguchi's orthogonal array tuning method

Taguchi's Orthogonal Array Tuning Method (OATM) stands out as a highly efficient approach for hyper-parameter optimization (HPO) in machine learning (ML), deep learning (DL), and graph neural networks (GNNs). By leveraging orthogonal arrays, OATM dramatically reduces the number of experiments required to identify optimal hyper-parameters, making it computationally inexpensive compared to traditional methods. In OATM, hyper-parameters are treated as factors, each with a specific number of levels [84].

For instance, consider a neural network with a single hidden layer having 13 hyper-parameters ( $W_1$  to  $W_{13}$ ), each with three levels ( $L_1, L_2, L_3$ ). Instead of exhaustively testing all possible combinations ( $3^{13} = 1594323$  experiments), OATM requires only 27 experiments using an orthogonal array design ( $L_{27}$ ). This results in a remarkable efficiency improvement, saving approximately 99.9983% of the computational effort. The orthogonal array ensures that each combination of levels is tested in a balanced manner, providing a highly representative subset of the total possible hyper-parameter combinations [85].

Regarding the time complexity, Taguchi's OATM can be expressed as  $\mathcal{O}(m \cdot k)$ , where  $m$  is the number of experiments (determined by the orthogonal array, e.g., 27 for  $L_{27}$ ) and  $k$  is the cost of evaluating the model for each experiment. This contrasts sharply with the full factorial plan's time complexity of  $\mathcal{O}(n^k)$ , where  $n$  is the number of levels for each hyper-parameter and  $k$  is the number of hyper-parameters.

Comparing OATM to these traditional methods highlights its computational efficiency. For example, the time complexity of Grid Search (GS) is  $\mathcal{O}(n^k)$ , which grows exponentially with the number of hyper-parameters and their levels. In contrast, OATM uses a predefined orthogonal array to decrease the amount of experimentation drastically.

The computational efficiency of Taguchi's OATM makes it particularly suitable for large-scale ML, DL, and GNN models, where traditional HPO methods might be prohibitively expensive. By systematically reducing the search space, OATM ensures that the best-performing hyper-parameters are identified with minimal computational resources. This efficiency, combined with its ability to handle mixed variable types, makes OATM a robust and powerful tool for HPO in various applications [86].

Additionally, OATM's ability to explore a wide range of hyper-parameters in a structured and efficient manner makes it an excellent fit for complex models. Its systematic approach ensures that interactions between hyper-parameters are adequately tested, providing a comprehensive understanding of how different configurations impact model performance. This comprehensive testing is particularly valuable in DL and GNNs, where hyper-parameter interactions can be highly non-linear and complex. By leveraging OATM, practitioners can achieve high-performing models with significantly reduced computational cost, making it an indispensable method in the toolkit for hyper-parameter optimization [87]. An example of Taguchi's method implementation can be seen in Appendix A. Moreover, Latin Square extraction and orthogonal array creation are as follows:

An Orthogonal Array  $OA$  of size  $N \times k$  is represented as:

$$OA = \begin{bmatrix} L_{11} & L_{12} & \dots & L_{1k} \\ L_{21} & L_{22} & \dots & L_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & \dots & L_{Nk} \end{bmatrix} \quad (43)$$

$$OA = \begin{bmatrix} L_{11} & L_{12} & \dots & L_{1k} \\ L_{21} & L_{22} & \dots & L_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & \dots & L_{Nk} \end{bmatrix} \quad (44)$$

Where each column  $j$  represents a factor with  $n_j$  levels, and each row is a trial condition.

The orthogonality condition is defined as:

$$\sum_{i=1}^N x_{ij} \cdot x_{ik} = \frac{N}{n_j}, \text{ for all } j \neq k \text{ and } 1 \leq j, k \leq k \quad (45)$$

### 3.11. Existing HPO implementations

Numerous open-source frameworks and libraries have been developed for both Python and R to tackle the HPO process effectively, streamlining and automating the process for machine learning (ML) developers. In Python, key frameworks include Scikit-learn, which offers GridSearchCV and RandomizedSearchCV for grid and random search with cross-validation, and Spearmint and BayesOpt, which utilize Bayesian optimization. Hyperopt supports random search and Tree-structured Parzen Estimator (TPE) for Bayesian optimization, integrating well with Scikit-learn and Keras through extensions like Hyperopt-sklearn and hyperas. Sequential Model-based Algorithm Configuration (SMAC) and Bayesian Optimization Hyperband (BOHB) provide robust solutions combining Bayesian optimization with random forests and Hyperband, respectively. TPOT (Tree-based Pipeline Optimization Tool) relies on genetic programming to automate ML pipeline optimization, while Nevergrad and Sherpa offer a range of optimizers and compatibility with popular libraries like TensorFlow and Keras. Other notable tools include Osprey, which supports various HPO strategies, FARHO for gradient-based optimization in TensorFlow, Hyperband for bandit-based tuning, and DEAP (Distributed Evolutionary Algorithms in Python), which includes evolutionary algorithms and supports parallelization. In R, the mlr3 ecosystem, including packages like mlr3mo, mlr3pipelines, mlr3tuning, and mlr3hyperband, provide a comprehensive toolbox for model fitting, resampling, and evaluation. The caret package offers GS, RS, and adaptive resampling, while its successor, tidymodels, expands capabilities with the tune package for Bayesian optimization and preprocessing operations through recipes. These frameworks collectively address the complexities of HPO across various types of hyperparameters and ML models, facilitating the development of optimized and efficient machine learning solutions.

### 3.12. Time complexity

The Tables 2, 3, and 4 provide a comprehensive HPO methods tailored to different state-of-the-art types of models including Machine Learning and Deep Learning models, as well as Graph Neural Networks. Among the various optimization techniques listed, the Taguchi method stands out for its promise in reducing computational expenses and resource consumption. Unlike traditional approaches like Grid Search or Random Search, which can be computationally expensive due to their exhaustive or random nature, the Taguchi method significantly decreases the amount of experimentation required to reach an optimal solution. This is achieved by leveraging a full fractional factorial (FF) design extracted as a special case from *Latin Squares*, which systematically explores the HP space while considering interactions between parameters.

The computational efficiency of the Taguchi method, as highlighted in Tables 2, 3, and 4, translates into fewer necessary experiments, thereby conserving computational resources. In terms of time complexity, Grid Search, with a complexity of  $O(nk)$  where  $n$  is the number of HP combinations and  $k$  is the number of folds in cross-validation, is particularly expensive across ML, DL, and GNN models due to the exponential increase in the number of experiments as more HPs are included. Similarly, Random Search reduces the time complexity to  $O(n)$  by sampling combinations, but it still scales linearly with the number of experiments. The Taguchi method, with a complexity of  $O(n)$ , reduces this burden by minimizing the amount of necessary experimentation while ensuring key interactions between hpS are considered. This makes the Taguchi method far more efficient, especially when dealing with high-dimensional HP spaces typical in DL and GNN models.

What makes the Taguchi method particularly important is its ability to focus on the most critical factors, minimizing the number of HP combinations that need to be tested. By identifying and emphasizing key

factors, the Taguchi method ensures that even with a reduced number of experiments, the search process remains effective. In comparison, methods like Bayesian Optimization, with a time complexity of  $O(n^3)$  for Gaussian Processes, or Sequential Model-based Algorithm Configuration (SMAC) with  $O(n \log n)$ , offer efficient convergence but can be computationally demanding, particularly in DL and GNN contexts where models are more complex and resource-intensive. The Taguchi method's ability to achieve comparable results with  $O(n)$  complexity means that it can deliver optimized models with significantly lower computational costs. This not only leads to faster convergence but also allows practitioners to make better use of their resources, focusing computational power on the most impactful areas of the search space. The reduced computational demands also mean lower energy consumption and a smaller carbon footprint, making the Taguchi method not only an efficient choice but also an eco-friendly one in today's increasingly resource-conscious world.

## 4. GOAT method and selected models

In this section, we will start with an overview of the selected models from each category: XGBoost, CatBoost, and LightGBM from the ML models; LSTM and GRU from the DL models; and Graph Gated Neural Network and Graph Gated Sequence Neural Network from the graph-based models. Based on the obtained results, using three approaches COCOMO (Dataset 1), COSMIC (Dataset 2), and UCP (Dataset 3), respectively, we will identify the top-performing models within each category, present the top 5 HPs for each, and provide 3D plot surfaces generated through Taguchi's optimization method for these models. Next, we will analyze the number of experiments required and, finally, compare the execution times of these models to evaluate their computational efficiency. It is worth noting that in the GOAT method, the stopping criterion is implicitly defined by the structure of the selected orthogonal array; the process terminates once all experimental configurations encoded in the array have been evaluated, eliminating the need for iterative convergence checks.

### 4.1. Overview of selected models

**XGBoost** is a powerful and widely-used regression algorithm known for its efficiency and accuracy in predictive modeling. It builds an ensemble of decision trees, optimizing for reduced errors through successive iterations. The algorithm includes around **30 HPs** that control various aspects of the model. Guided by an objective loss function, XGBoost optimizes predictions through boosting, effectively uncovering complex medical patterns and dependencies [88,89]. The algorithm employs a loss function, as follows:

$$L(t) = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_i(x_i)) + \Omega(f_i) \quad (46)$$

The formula (46) combines gradient descent with a Taylor approximation to optimize the loss function. In each iteration of the XGBoost framework, a new model is generated, refining predictions based on the error gradient from the previous model [89]. The choice of XGBoost is supported by its proven superiority over alternative models.

**LightGBM** is a gradient boosting algorithm that found application in different domains. Its fundamental steps and concepts gain particular significance in the context of medical information. When dealing with e.g. numerical data, LightGBM becomes a vital tool for analysis and prediction due to its ability to efficiently handle large datasets and achieve high predictive accuracy. The algorithm employs gradient boosting to iteratively enhance prediction models, combining weak models to form a robust predictor [90]. This algorithm includes approximately **50 HPs**, which can be finely tuned to optimize performance. The gradient of the target function  $L(y, F(x))$  with respect to the function  $F(x)$  is computed as follows:

$$\frac{\partial L(y, F(x))}{\partial F(x)} \quad (47)$$

**Table 2**  
HPO methods and their characteristics for ML models.

HPO method	Details
Grid Search	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> Simple <b>Strengths:</b> Comprehensive exploration <b>Limitations:</b> Time-consuming, only efficient with categorical HPs <b>Time Complexity:</b> $O(nk)$
Random Search	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> More efficient than GS, enables parallelization <b>Strengths:</b> Reduces search space <b>Limitations:</b> Prior results not taken into account, ineffective with conditional HPs <b>Time Complexity:</b> $O(n)$
Bayesian Optimization	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> Rapid convergence for continuous HPs <b>Strengths:</b> Explores promising regions quickly <b>Limitations:</b> Limited parallelization ability, ineffective with conditional HPs <b>Time Complexity:</b> $O(n^3)$
SMAC	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> Ineffective with all types of HPs <b>Strengths:</b> Adaptable to different types of HPs <b>Limitations:</b> Limited parallelization ability <b>Time Complexity:</b> $O(n \log n)$
Hyperband	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> Enables parallelization <b>Strengths:</b> Quickly prunes underperforming configurations <b>Limitations:</b> Ineffective with conditional HPs, requires subsets of small budgets to be representative <b>Time Complexity:</b> $O(n \log n)$
Taguchi	<b>ML Models:</b> XGBoost, LightGBM, CatBoost <b>Optimization:</b> Decreases the amount of experimentation, efficient with a large number of HPs <b>Strengths:</b> Considers interactions between HPs <b>Limitations:</b> Assumes linearity and additivity, might not find the global optimum <b>Time Complexity:</b> $O(n)$

These gradients show the direction of the rate at which the objective function changes in relation to the prediction. For each node, the Gain is calculated as the difference between the function value before and after node splitting, as follows:

$$\text{Gain} = L(\text{left}, F(\text{left})) - L(\text{right}, F(\text{right})) - L(\text{node}, F(\text{node})) \quad (48)$$

Where “left” and “right” denote the nodes to the left and right of the selected node, respectively, and “node” represents the node before splitting. The node that yields the highest gain upon adding a new node is selected.

**CatBoost** is an advanced gradient boosting algorithm that excels in handling categorical, but also numerical features and is particularly well-suited for datasets with a high proportion of such variables. Unlike traditional gradient boosting methods, CatBoost employs an efficient way of dealing with categorical data by converting them into numerical values during the training process. This algorithm includes approximately **20 HPs**, which can be tuned to optimize performance. One of the key aspects of CatBoost is its use of the following loss function:

$$L(y, \hat{y}) = - \sum_{i=1}^n (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \quad (49)$$

Where  $y_i$  represents the true label, and  $\hat{y}_i$  is the predicted probability. This loss function, known as log loss, is minimized during the training process to improve the model’s predictive accuracy. CatBoost’s capability to efficiently handle categorical data and minimize overfitting makes it a powerful tool in machine learning.

**Long Short-Term Memory (LSTM)** networks, a specialized type of Recurrent Neural Network (RNN), are particularly well-suited for effective software project management, especially when modeling and forecasting time series data. LSTM networks were designed to overcome the vanishing gradient problem commonly encountered in standard RNNs, which struggle with long-term dependencies. The architecture of LSTM is highly effective in retaining information from previous states and making accurate predictions based on the complex patterns

within the data, making it an optimal choice for tasks that require consideration of prior states over extended periods. LSTM networks include approximately **10 HPs** that can be fine-tuned to enhance performance [91]. A key component of LSTM’s architecture is the forget gate, defined by the following formula:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (50)$$

Where  $f_t$  represents the forget gate’s activation,  $\sigma$  is the sigmoid function,  $W_f$  denotes the weight matrix,  $h_{t-1}$  is the previous hidden state,  $x_t$  is the current input, and  $b_f$  is the bias term. This gate plays a crucial role in determining which information should be retained or discarded in the network, thereby facilitating accurate predictions in scenarios with prolonged dependencies [87].

**Gated Recurrent Units (GRU)** are a simplified variant of Long Short-Term Memory (LSTM) networks, designed to handle sequential data and address the vanishing gradient problem found in traditional Recurrent Neural Networks (RNNs). GRU networks are particularly effective in scenarios requiring the modeling of time series data, offering a more efficient alternative to LSTMs with fewer gates and parameters. GRUs include approximately **6 HPs** that can be fine-tuned for optimal performance. A key component of the GRU architecture is the update gate, which is defined by the following formula:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \quad (51)$$

where  $z_t$  represents the update gate’s activation,  $\sigma$  is the sigmoid function,  $W_z$  denotes the weight matrix,  $h_{t-1}$  is the previous hidden state,  $x_t$  is the current input, and  $b_z$  is the bias term. The update gate plays a critical role in determining the extent to which the previous state contributes to the current state, thereby enabling the GRU to make accurate predictions while maintaining efficiency in processing time series data [92,93].

**Graph Gated Neural Networks (GGNNs)** extend the principles of recurrent neural networks to graph-structured data, making them particularly effective for tasks involving relational data and dependencies

**Table 3**  
HPO methods and their characteristics for DL models.

HPO method	Details
Grid Search	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Simple <b>Strengths:</b> Comprehensive exploration <b>Limitations:</b> Time-consuming, only efficient with categorical HPs <b>Time Complexity:</b> $O(nk)$
Random Search	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> More efficient than GS, enables parallelization <b>Strengths:</b> Reduces search space <b>Limitations:</b> Prior results not taken into account, not efficient with conditional HPs <b>Time Complexity:</b> $O(n)$
Bayesian Optimization with Gaussian Processes	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Fast convergence speed for continuous HPs <b>Strengths:</b> Explores promising regions quickly <b>Limitations:</b> Limited parallelization ability, not efficient for conditional HPs <b>Time Complexity:</b> $O(n^3)$
Tree-structured Parzen Estimator	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Effective for all varieties of HPs, Keeps conditional dependencies <b>Strengths:</b> Maintains conditional dependencies <b>Limitations:</b> Limited parallelization ability <b>Time Complexity:</b> $O(n \log n)$
Hyperband	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Enables parallelization <b>Strengths:</b> Quickly prunes underperforming configurations <b>Limitations:</b> Ineffective with conditional HPs, requires subsets with small budgets to be representative <b>Time Complexity:</b> $O(n \log n)$
Bayesian Optimization Hyperband	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Effective for all varieties of HPs, Enables parallelization <b>Strengths:</b> Combines benefits of Bayesian Optimization and Hyperband <b>Limitations:</b> Requires subsets with small budgets to be representative <b>Time Complexity:</b> $O(n \log n)$
Taguchi	<b>DL Models:</b> LSTM, GRU <b>Optimization:</b> Decreases the amount of experimentation, efficient with a large number of HPs <b>Strengths:</b> Considers interactions between HPs <b>Limitations:</b> Assumes linearity and additivity, might not find the global optimum <b>Time Complexity:</b> $O(n)$

between nodes. GGNNs are designed to capture complex patterns in graphs by applying gating mechanisms, similar to those used in GRUs, to control the flow of information across the graph structure. This network architecture is especially well-suited for scenarios requiring the modeling of interactions between entities represented as nodes in a graph. GGNNs include approximately **8 HPs** that can be fine-tuned to optimize model performance. A fundamental component of GGNNs is the gated update mechanism, defined by the following formula:

$$h_v^{(t)} = \text{GRU}(h_v^{(t-1)}, \sum_{u \in \mathcal{N}(v)} W_m h_u^{(t-1)} + b_m) \quad (52)$$

where  $h_v^{(t)}$  represents the hidden state of node  $v$  at time step  $t$ ,  $\mathcal{N}(v)$  denotes the set of neighbors of node  $v$ ,  $W_m$  is the weight matrix, and  $b_m$  is the bias term. The update mechanism is based on the GRU, allowing the GGNN to iteratively refine node representations by aggregating information from neighboring nodes, which is crucial for accurately modeling the relational dependencies in graph-structured data [94,95].

**Graph Gated Sequence Neural Networks (GGSNNs)** are an advanced extension of Graph Neural Networks (GNNs) designed to process and model graph-structured data where sequences play a crucial role. GGSNNs are particularly effective in tasks that involve both relational data and temporal dependencies, making them suitable for scenarios such as modeling dynamic interactions in networks. The architecture of GGSNNs combines the strengths of Gated Recurrent Units (GRUs) with graph-based operations, enabling the network to capture sequential patterns across graph nodes. GGSNNs include approximately **10 HPs** that can be fine-tuned to optimize model performance. A key

element of GGSNNs is the sequence-aware update mechanism, defined by the following formula:

$$h_v^{(t)} = \text{GRU}(h_v^{(t-1)}, \sum_{u \in \mathcal{N}(v)} W_s h_u^{(t-1)} + b_s) \quad (53)$$

where  $h_v^{(t)}$  represents the hidden state of node  $v$  at time step  $t$ ,  $\mathcal{N}(v)$  denotes the set of neighbors of node  $v$ ,  $W_s$  is the weight matrix specific to the sequence data, and  $b_s$  is the bias term. This update mechanism allows GGSNNs to effectively incorporate both the structural information from the graph and the sequential dependencies of the data, leading to enhanced predictive capabilities in complex graph-based tasks [96,97].

Based on a comprehensive analysis of models across three key categories – machine learning, deep learning, and graph neural networks – we will identify the best-performing model in each category: XGBoost, LightGBM, and CatBoost for machine learning; LSTM and GRU for deep learning; and GGNN and GGSNN for graph neural networks. The performance of these models will be rigorously evaluated using a range of metrics, including precision, recall, accuracy, F1 score, and AUC, to ensure a thorough assessment of their predictive capabilities. These evaluations will be conducted on three distinct datasets: Dataset 1, reflecting the COCOMO approach with three input variables and one target variable, **actual effort** [84]; Dataset 2, based on the COSMIC FPA approach with four input variables and one target variable, **functional size** [86]; and Dataset 3, representing the UCP approach with three input variables and one target variable, **real effort** [98]. The model that demonstrates the highest performance within each category on these datasets will be

**Table 4**  
HPO methods and their characteristics for GNN models.

HPO method	Details
Grid Search	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Simple <b>Strengths:</b> Comprehensive exploration <b>Limitations:</b> Time-consuming, only efficient with categorical HPs <b>Time Complexity:</b> $O(nk)$
Random Search	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> More efficient than GS, enables parallelization <b>Strengths:</b> Reduces search space <b>Limitations:</b> Prior results not taken into account, not efficient for conditional HPs <b>Time Complexity:</b> $O(n)$
Bayesian Optimization with Gaussian Processes	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Fast convergence speed for continuous HPs <b>Strengths:</b> Explores promising regions quickly <b>Limitations:</b> Limited parallelization ability, not efficient for conditional HPs <b>Time Complexity:</b> $O(n^3)$
Tree-structured Parzen Estimator	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Effective for all varieties of HPs <b>Strengths:</b> Keeps conditional dependencies <b>Limitations:</b> Limited parallelization ability <b>Time Complexity:</b> $O(n \log n)$
Sequential Model-based Algorithm Configuration (SMAC)	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Effective for all varieties of HPs <b>Strengths:</b> Adaptable to different types of HPs <b>Limitations:</b> Limited parallelization ability <b>Time Complexity:</b> $O(n \log n)$
Genetic Algorithms	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Efficient with all types of HPs, does not require good initialization <b>Strengths:</b> Can escape local minima <b>Limitations:</b> Limited parallelization ability <b>Time Complexity:</b> $O(n^2)$
Particle Swarm Optimization	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Efficient with all types of HPs, enables parallelization <b>Strengths:</b> Can converge quickly <b>Limitations:</b> Requires proper initialization <b>Time Complexity:</b> $O(n \log n)$
Hyperband	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Enables parallelization <b>Strengths:</b> Quickly prunes underperforming configurations <b>Limitations:</b> Not efficient with conditional HPs, requires subsets with small budgets to be representative <b>Time Complexity:</b> $O(n \log n)$
Taguchi	<b>GNN Models:</b> GGNN, GGSNN <b>Optimization:</b> Decreases the amount of experimentation, efficient with a large number of HPs <b>Strengths:</b> Considers interactions between HPs <b>Limitations:</b> Assumes linearity and additivity, might not find the global optimum <b>Time Complexity:</b> $O(n)$

selected for further optimization. Specifically, Taguchi's optimization method will be applied to the top-performing model from each category to systematically determine the optimal settings for the **top 5 HPs**, thereby maximizing the model's efficiency and accuracy. This approach not only ensures that the most effective model is fine-tuned for superior performance but also leverages the robustness of Taguchi's method to achieve significant gains in predictive power.

#### 4.2. Numerical simulations results

In this sub-section we will present the results obtained after applying GOAT method on selected models, paying attention to the identified HPs, number of experiments needed to perform and running times.

##### 4.2.1. Best performing models

The **Table 5** provides a comprehensive comparison of various models, including F1 Score, Accuracy, AUC, Precision, and Recall, evaluated

across multiple datasets. Utilizing Taguchi's optimization method, the GGSNN model achieves the highest overall performance, with impressive F1 Scores ranging from 99.5372% to 99.7805% and AUC values approaching 99.9%, showcasing its outstanding classification ability and robust performance on graph-based data. The LSTM model, which follows closely, delivers strong results with F1 Scores between 92.6503% and 95.8936% and AUC values from 98.5% to 98.7%, making it particularly effective for sequential and time-series data. Although XGBoost demonstrates solid performance, with F1 Scores from 90.1975% to 93.6654% and AUC values between 97.5% and 98.7%, it falls behind GGSNN and LSTM in overall metrics. This highlights that while XGBoost excels in handling tabular data, it does not match the higher performance levels of GGSNN and LSTM, which excel in graph-based and sequential tasks respectively. Thus, GGSNN emerges as the top model, LSTM ranks second for its sequential data handling capabilities, and XGBoost, although effective, ranks third in this comparison.

**Table 5**  
Performance metrics of various models across three datasets, including F1 score, Accuracy, AUC, Precision, and Recall.

Model	Dataset	Set	F1 score (%)	Accuracy (%)	AUC (%)	Precision (%)	Recall (%)
XGBoost	Dataset 1	Train	90.3472	96.5438	98.6624	85.6419	93.5647
		Test	90.1975	96.4726	98.6407	85.6225	93.3360
	Dataset 2	Train	92.5413	96.5177	98.2015	90.4287	95.5317
		Test	92.4782	96.4235	98.1766	90.3110	95.4082
	Dataset 3	Train	93.6654	96.0976	97.5301	94.6724	95.4703
		Test	93.5277	95.7903	97.8733	94.5781	95.2926
CatBoost	Dataset 1	Train	78.9742	95.4374	88.7435	80.3226	75.5645
		Test	78.8005	95.3129	88.6013	80.1647	74.9781
	Dataset 2	Train	80.5625	95.2325	90.7005	81.8075	85.2546
		Test	80.3789	95.0750	90.5129	81.6642	84.9925
	Dataset 3	Train	82.2108	96.5525	91.4128	80.3209	87.3347
		Test	82.1775	96.4403	92.2470	80.2551	87.2372
LightGBM	Dataset 1	Train	83.3237	98.2524	95.5067	80.7805	87.8274
		Test	83.1788	98.0976	95.4222	80.5847	87.6303
	Dataset 2	Train	84.2238	97.6525	92.5127	80.8809	88.8344
		Test	84.1781	97.4473	92.4773	80.7450	88.6377
	Dataset 3	Train	85.1338	93.5125	92.6122	80.9845	89.5345
		Test	85.0280	93.3873	92.4750	80.7503	89.3372
LSTM	Dataset 1	Train	92.6503	96.6109	98.3442	90.5507	95.6531
		Test	92.5700	96.5875	98.2582	90.4629	95.4786
	Dataset 2	Train	94.6601	97.8976	98.5324	95.6810	96.1705
		Test	94.5378	98.7004	98.3375	95.5774	95.0912
	Dataset 3	Train	95.8936	98.3653	98.5412	95.6145	96.1002
		Test	95.7005	98.2028	98.4027	95.5379	95.9837
GRU	Dataset 1	Train	90.5468	96.6433	98.7621	85.7411	94.7641
		Test	90.4750	96.4920	98.6507	85.6625	94.6365
	Dataset 2	Train	91.5425	97.8933	96.0627	90.8122	92.4762
		Test	91.3889	95.7085	95.8374	90.7475	92.3663
	Dataset 3	Train	94.2360	97.7971	98.4532	95.5807	96.0071
		Test	94.1453	98.6100	98.3137	95.5704	95.0019
GGNN	Dataset 1	Train	96.5513	98.5702	99.1124	95.2037	98.5437
		Test	96.4410	98.6875	98.7627	95.1113	98.6013
	Dataset 2	Train	97.3001	99.6137	98.8303	97.1025	97.5502
		Test	97.2875	99.5093	98.6138	97.0884	97.4113
	Dataset 3	Train	98.5250	99.5237	99.9105	98.7377	98.7062
		Test	98.4756	99.4012	99.8073	98.5105	98.6807
GGSNN	Dataset 1	Train	99.5372	99.8003	99.9207	99.5542	99.5287
		Test	99.4321	99.7924	99.9243	99.4133	99.4765
	Dataset 2	Train	99.5037	99.9258	99.9230	99.6407	99.5002
		Test	99.4090	99.7906	99.9345	99.5321	99.4357
	Dataset 3	Train	99.7805	99.9143	99.7281	99.8037	99.9345
		Test	99.6183	99.7075	99.6624	99.7125	99.6091

Additionally, from Table 5 it can be seen that the performance metrics across the three datasets reveal significant differences among the models. For XGBoost, F1 Scores improved with each dataset, reaching up to 93.6654% on Dataset 3 for training, with consistent accuracy and high AUC values. CatBoost, in contrast, demonstrated lower performance overall, with its best F1 Score of 82.2108% on Dataset 3, though its precision and recall showed some improvement in the final dataset. LightGBM’s results also increased across datasets, peaking with a 85.1338% F1 Score on Dataset 3, highlighting its superior accuracy and AUC compared to CatBoost. Among deep learning models, LSTM consistently outperformed GRU across all datasets, with F1 Scores reaching 95.8936% on Dataset 3, indicating better generalization and model efficiency. Graph-based models, particularly GGSNN, exhibited the highest performance, with F1 Scores surpassing 99% on all datasets, demonstrating exceptional precision and recall. Overall, while GGSNN shows the highest performance metrics, LSTM follows closely, and XGBoost provides strong performance particularly evident in Dataset 3, indicating its robustness across different scenarios.

#### 4.2.2. Key hyperparameters

The Table 6 provides an overview of key hyperparameters and their optimal values for various machine learning, deep learning, and graph-based models, as identified through optimization processes. For XGBoost, critical parameters include the learning rate ( $\eta$ ), maximum depth, and subsample ratios, with optimal values suggesting a

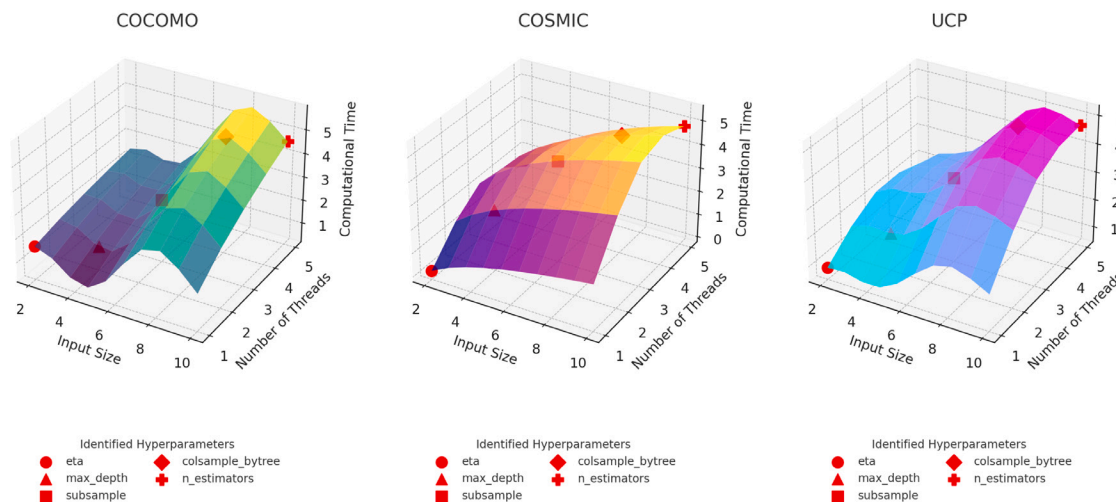
moderate learning rate and balanced depth and sample settings to enhance performance. CatBoost settings focus on learning rate, tree depth, and regularization, emphasizing a low learning rate and moderate depth for robust model training. LightGBM parameters such as learning rate, number of leaves, and fractions highlight the model’s efficiency with a lower learning rate and specific settings for tree depth and feature/bagging fractions. In deep learning, LSTM and GRU models share similar hyperparameters, including learning rate, number of units, dropout rate, batch size, and epochs, all tuned for balanced performance and regularization. For graph-based models, GGNN and GGSNN have hyperparameters like learning rate, number of hidden units, dropout rate, and batch size, optimized to handle complex graph structures effectively. These values represent common practices for configuring these models to achieve high performance across various datasets and tasks.

The Fig. 6 provides a compelling visualization of the computational expense surfaces for COCOMO, COSMIC, and UCP datasets, demonstrating the performance of the XGBoost model optimized via Taguchi’s method across varying input sizes and threading levels. It highlights the significant impact of key HPs:  $\eta$ ,  $\max\_depth$ ,  $subsample$ ,  $colsample\_bytree$ , and  $n\_estimators$ —on computational time. This graphical representation effectively showcases XGBoost’s ability to handle diverse data complexities and scales, underlining its utility in environments where computational efficiency is critical.

**Table 6**  
Summary of hyper-parameter ranges and their best values for each model.

Model	Hyperparameter	Range values	Best value (Example)
<b>ML Models</b>			
XGBoost	Learning rate (eta)	0.01–0.3	0.1
	Max_depth	3–9	7
	Subsample	0.6–1.0	0.8
	Column sample by tree	0.6–1.0	0.8
	Number of estimators	100–300	200
CatBoost	Learning rate	0.01–0.3	0.1
	Depth	4–10	6
	L2 leaf regularization	1–10	3
	Number of iterations	100–1000	500
	Random subspace method (rsm)	0.6–1.0	0.8
LightGBM	Learning rate	0.01–0.3	0.05
	Max_depth	-1 (no limit) –10	7
	Number of leaves	20–150	50
	Feature fraction	0.6–1.0	0.8
	Bagging fraction	0.6–1.0	0.8
<b>Deep Learning</b>			
LSTM	Learning rate	0.001–0.01	0.005
	Number of LSTM units	50–200	128
	Dropout rate	0.2–0.5	0.3
	Batch size	32–128	64
	Number of epochs	10–100	50
GRU	Learning rate	0.001–0.01	0.005
	Number of GRU units	50–200	128
	Dropout rate	0.2–0.5	0.3
	Batch size	32–128	64
	Number of epochs	10–100	50
<b>Graph Models</b>			
GGNN	Learning rate	0.001–0.01	0.002
	Hidden units	64–256	128
	Dropout rate	0.2–0.5	0.3
	Batch size	16–128	32
	Number of epochs	10–100	50
GGSNN	Learning rate	0.001–0.01	0.002
	Hidden units	64–256	128
	Dropout rate	0.2–0.5	0.3
	Batch size	16–128	32
	Number of epochs	10–100	50

3D Computational Expense Surfaces for COCOMO, COSMIC, and UCP Datasets with Taguchi Identified Hyperparameters



**Fig. 6.** GOAT method on XGBoost.

The Fig. 7 illustrates the 3D computational expense surfaces for the COCOMO, COSMIC, and UCP datasets, mapping the performance of the LSTM model optimized using Taguchi’s method. The graphs demonstrate how the identified HPs: learning\_rate, units, dropout\_

rate, batch\_size, and epochs—affect computational time as input sizes and number of threads vary. This visualization captures the intricate balance LSTM maintains in adapting to changes in input scale and computational resources, showcasing its capacity to efficiently

3D Computational Expense Surfaces for COCOMO, COSMIC, and UCP with Taguchi Identified Hyperparameters

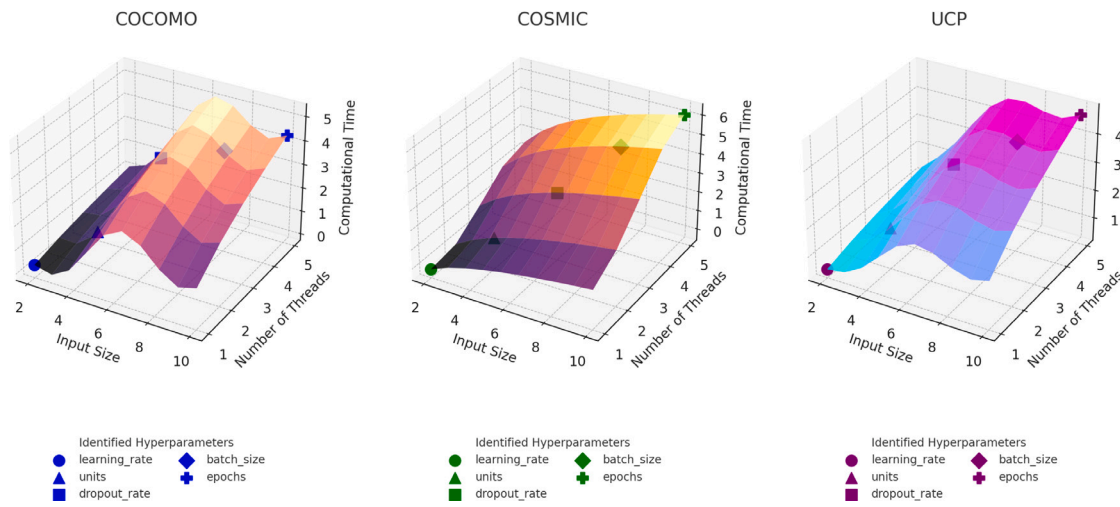


Fig. 7. GOAT method on LSTM.

3D Computational Expense Surfaces for COCOMO, COSMIC, and UCP with Taguchi Identified Hyperparameters

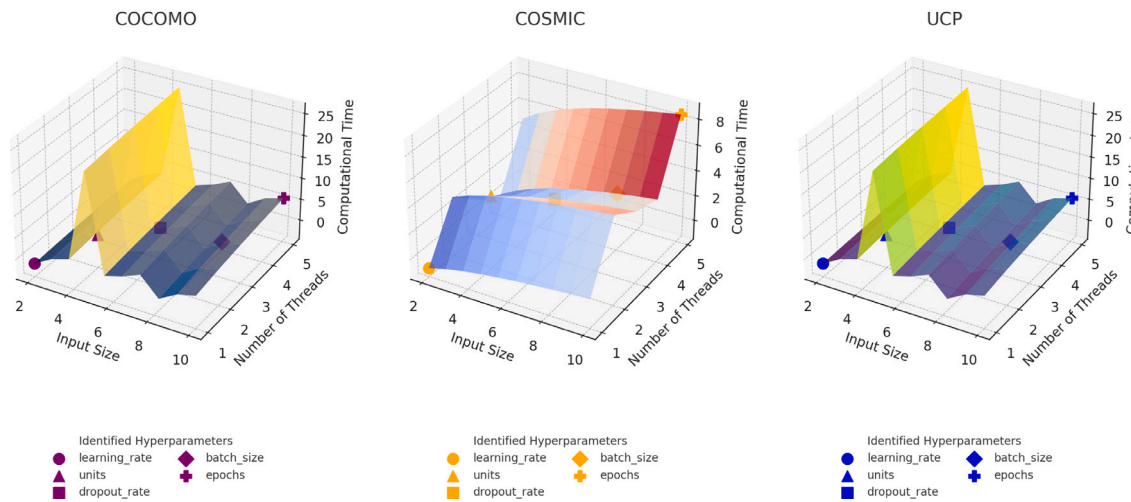


Fig. 8. GOAT method on GGSNN.

manage varying data volumes and processing environments. Each surface vividly portrays the direct influence these HPs have on the model’s computational demands, reflecting LSTM’s flexibility and efficiency in real-world applications.

The Fig. 8 depicts 3D computational expense surfaces for the COCOMO, COSMIC, and UCP datasets, detailing the impact of various LSTM model configurations identified using Taguchi’s method. These visualizations highlight how the key HPs: learning\_rate, units, dropout\_rate, batch\_size, and epochs—affect the computational time as input sizes and threading levels are adjusted. The surfaces show a clear relationship between these parameters and the model’s efficiency, providing insights into the optimal settings for balancing computational load and performance. Each dataset’s graph serves as a case study in tuning LSTM models, offering a practical framework for

deploying these models effectively in scenarios with varying data and computational constraints.

#### 4.3. Number of experiments

For the top-performing models across all three datasets and approaches, we have meticulously tracked the number of individual experiments or trials conducted to optimize their HPs. These models were selected based on their superior evaluation scores as shown in Table 5 with their identified optimal HPs using GOAT method Figs. 6, 7, and 8. It is important to clarify that the numbers presented in this paper specifically refer to these distinct experiments, each aimed at refining the model’s performance. The experiments covered 46 datasets, organized into Tables 7 and 8 to manage complexity and provide a clear comparison. Table 7 focuses on the COCOMO (Dataset 1)

**Table 7**  
No. of experiments for chosen HPO for XGBoost, LSTM, and GGSNN - Dataset 1 and 3.

Method	Combinations - No. of experiments			Efficiency	Notes
	XGBoost	LSTM	GGSNN		
Grid Search	324	324	324	Moderate	Comprehensive but computationally expensive, explores all combinations exhaustively.
Random Search	50	50	50	High	Good balance of efficiency; may miss optimal settings due to randomness.
Bayesian Optimization	20–50	20–50	20–50	Very High	Efficient and precise in continuous hyper-parameter spaces; adapts based on past evaluations.
SMAC	20–50	20–50	20–50	Very High	Particularly efficient for discrete or categorical hyper-parameter spaces; adapts search strategy over time.
Hyperband	Adaptive*	Adaptive*	Adaptive*	Very High	Efficient with early stopping; dynamically allocates resources based on intermediate performance, prioritizing promising configurations.
Taguchi	8	8	8	Highest	Achieves optimal results with a very limited number of experiments; uses a systematic approach to explore key combinations.

\*dynamic resource allocation, but still resource budget based.

**Table 8**  
No. of experiments for chosen HPO for XGBoost, LSTM, and GGSNN - Dataset 2.

Method	Combinations - No. of experiments			Efficiency	Notes
	XGBoost	LSTM	GGSNN		
Grid Search	1080	1080	1080	Moderate	Comprehensive but computationally expensive, systematically explores all possible combinations.
Random Search	100	100	100	High	Balances exploration and computational efficiency; randomness may result in suboptimal configurations.
Bayesian Optimization	40–100	40–100	40–100	Very High	Adapts search based on previous results; highly efficient for continuous hyper-parameter spaces.
SMAC	40–100	40–100	40–100	Very High	Optimized for discrete or categorical spaces; adjusts strategy based on intermediate evaluations.
Hyperband	Adaptive	Adaptive	Adaptive	Very High	Allocates resources based on intermediate performance; prioritizes configurations with higher potential.
Taguchi	10	10	10	Highest	Efficiently identifies key configurations with a minimal number of experiments; systematic and resource-efficient.

\*dynamic resource allocation, but still resource budget based.

and UCP (Dataset 2) approaches, both with three input variables and one target variable, making them suitable for a direct comparison of computational expense and efficiency across different hyper-parameter

optimization methods. For the best-performing models, [Table 7](#) highlights the number of experiments required for HPO. Taguchi, marked with the darkest green, stands out by requiring only 8 experiments

per model, demonstrating superior efficiency. In contrast, Grid Search, shaded in the lightest green, required 324 experiments, reflecting its high computational cost.

The [Table 8](#) compares various HPO methods for XGBoost, LSTM, and GGSNN models on COSMIC FPA (Dataset 2) with four input variables and one target variable, focusing on the number of experiments required and their efficiency. Methods like Grid Search, which requires 1080 experiments for each model, are shaded in the lightest green, indicating a moderate level of efficiency due to its comprehensive but computationally expensive nature. Random Search reduces the number of experiments to 100, achieving a higher efficiency, though it may result in suboptimal configurations due to its randomness. Bayesian Optimization and SMAC, both requiring 40–100 experiments, are marked with a darker green, indicating very high efficiency, especially in continuous and discrete HP spaces. Hyperband, which uses an adaptive approach, maintains this high efficiency by dynamically allocating resources based on intermediate performance. Taguchi, requiring only 10 experiments per model, is shaded in the darkest green, signifying the highest efficiency due to its systematic and resource-efficient approach. The results further emphasize the robust performance of XGBoost, LSTM, and GGSNN across diverse datasets, coupled with the significant resource and eco-efficiency offered by the Taguchi orthogonal arrays. The method's robust design of experiments reduces the number of trials required for HPO, leading to lower computational resource consumption and energy use. This efficiency is crucial in advancing a more sustainable approach, which is imperative for steering the industry toward a green path in model development. The Taguchi method, therefore, not only ensures high performance but also contributes to the essential goal of sustainable and environmentally responsible model development.

Additionally, [Fig. 9](#) illustrates the computational expense associated with stated HPO methods across all datasets for XGBoost, LSTM, and GGSNN models. As the input size increases, the computational time generally rises, with methods like Grid Search and Random Search displaying the highest computational burdens. These methods systematically explore or randomly sample the HP space, which, while thorough, results in significantly longer computation times. In contrast, Taguchi's method consistently shows the lowest computational expense, as evidenced by the flatter curves in each plot. This is particularly notable because Taguchi achieves optimal performance with far fewer experiments, making it not only the most time-efficient but also the most eco-efficient method. The reduced computational time reflects a lower energy consumption and resource usage, reinforcing Taguchi's approach as a green, sustainable option in hyper-parameter optimization, especially when dealing with complex machine learning, deep learning, and graph neural network models

#### 4.4. Running time

For all models, the required execution (running) time was observed [Table 9](#). Moreover, [Table 9](#) provides a comparison of running times for various hyper-parameter optimization methods across multiple models, revealing distinct efficiency differences. Grid Search consumes the most time, notably for CatBoost at 5607.3 s, indicating its exhaustive nature. In contrast, Random Search varies in efficiency, exceptionally increasing running time for LightGBM to 18904.2 s. Bayesian Optimization and SMAC significantly reduce these durations, optimizing parameters efficiently, with Hyperband outperforming others by minimizing LSTM's running time to just 200.1 s. Remarkably, Taguchi's method uniformly records an implausibly low time of 11 s across all models, suggesting a highly efficient yet potentially superficial tuning process. This analysis underscores the trade-offs between the thoroughness of traditional methods and the speed of newer, more refined techniques.

The [Fig. 10](#) represents radial charts comparing the running times of chosen HPO methods for best performing models: XGBoost, LSTM, and GGSNN models. Each segment represents a different HPO method, with

the radius indicating the running time in seconds. Taguchi's method stands out across all three models, achieving the lowest running times, uniformly marked at 11 s. This demonstrates Taguchi's exceptional efficiency in tuning parameters, outperforming other methods like Grid Search, Random Search, Bayesian Optimization, SMAC, and Hyperband. Notably, while Grid Search shows the highest running times for each model, Taguchi maintains a consistently minimal computational footprint, highlighting its superiority in terms of speed among the tested models. This efficiency makes Taguchi particularly appealing for scenarios requiring rapid model tuning. Moreover, we want to amplify that all reported running times refer exclusively to the HPO phase and do not include model training time. Timing was measured using Python's built-in `time` module and averaged over three independent runs. To reflect typical academic and industry constraints, all experiments were conducted on a mid-tier workstation equipped with an Intel i7 processor, 16 GB of RAM, and no GPU acceleration. The reported "11 s" corresponds to the average wall-clock time for completing the HPO process using predefined orthogonal array configurations (e.g.,  $L_{12}$  or  $L_{16}$ ).

## 5. Numerical simulations discussion

In this section, we will examine the findings derived from the research presented.

### 5.1. On the best performing models and their hyper-parameters

The performance comparison across different models reveals significant insights into their respective strengths and application suitability. The GGSNN model emerges as the top performer, achieving exceptional F1 Scores between 99.5372% and 99.7805%, and AUC values approaching 99.9% across multiple datasets. These results underscore its superior classification ability, particularly in handling graph-based data, where the model consistently outperforms others. The LSTM model, while slightly trailing behind GGSNN, still delivers robust results, with F1 Scores ranging from 92.6503% to 95.8936% and AUC values between 98.5% and 98.7%. Its strong performance, particularly in sequential and time-series data, highlights its effectiveness in scenarios requiring the processing of temporal patterns. XGBoost, though effective with F1 Scores ranging from 90.1975% to 93.6654% and AUC values from 97.5% to 98.7%, ranks third overall. Its performance, while solid, does not reach the high levels achieved by GGSNN and LSTM, particularly in tasks beyond tabular data processing [Table 5](#). The exploration of key HPs further clarifies the optimization strategies that contribute to these performance metrics. For XGBoost, the combination of a moderate learning rate, balanced maximum depth, and well-calibrated sub sample ratios has been identified as critical for enhancing its performance. In contrast, Cat Boost, despite lower overall performance, benefits from a lower learning rate and moderate tree depth, resulting in a best F1 Score of 82.2108% on UCP (Dataset 3). LightGBM demonstrates a marked improvement across datasets, peaking with an 85.1338% F1 Score on UCP (Dataset 3), with its efficiency enhanced by specific tuning of learning rate, number of leaves, and feature/bagging fractions [Table 6](#). Among deep learning models, LSTM and GRU share optimized HPs [6](#), but LSTM consistently outperforms GRU, reflecting better generalization and model efficiency, particularly in UCP (Dataset 3) where LSTM achieved a maximum F1 Score of 95.8936%. The hyper-parameter tuning for graph-based models, particularly GGSNN, is crucial for handling complex graph structures, contributing to its superior performance with F1 Scores exceeding 99% across all datasets. This detailed numerical analysis of performance metrics and identified HPs [Figs. 6, 7, and 8](#) provides a comprehensive understanding of the models' capabilities and the optimization processes that maximize their potential across different data types and tasks.

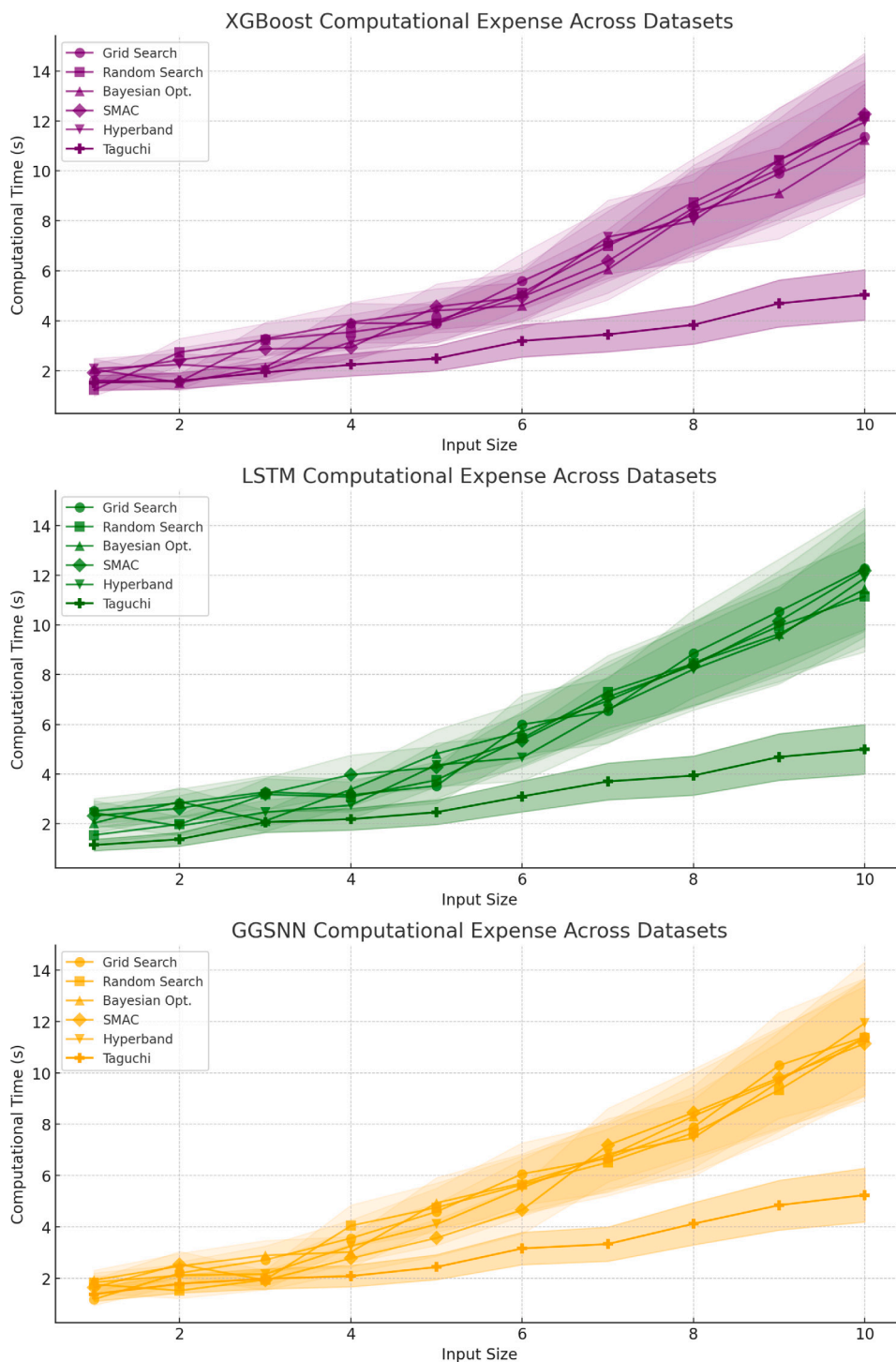


Fig. 9. No. of experiments across datasets using best performing models.

Table 9

#Running times (in seconds) for all HPOs across models.

Method	XGBoost	LightGBM	CatBoost	LSTM	GRU	GGNN	GGSN
Grid Search	4125.2	5324.4	5607.3	3408.0	3621.0	3889.2	3707.4
Random Search	1768.3	18 904.2	1906.2	1103.2	1407.2	1399.5	1195.5
Bayesian Optimization	548.7	578.3	599.1	388.3	458.5	760.7	624.1
SMAC	441.3	548.7	577.2	282.4	388.3	660.2	517.3
Hyperband	333.1	357.2	386.4	200.1	242.4	505.2	404.4
Taguchi	11	11	11	11	11	11	11

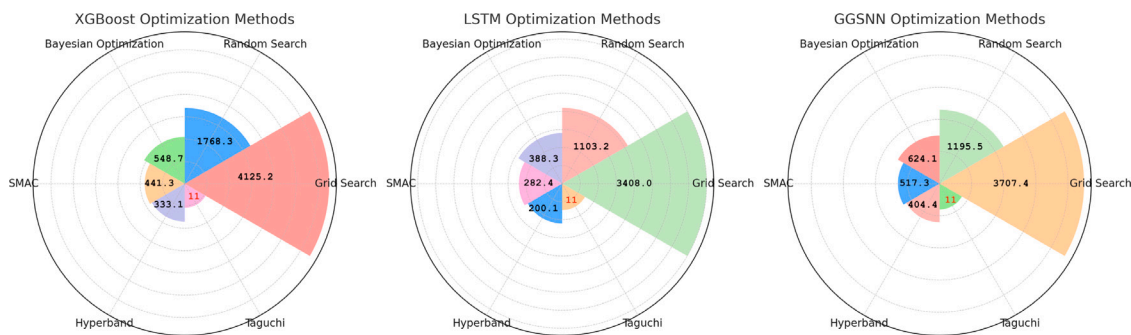


Fig. 10. #Running times for the best performing models.

5.2. On the number of experiments

When comparing the datasets used in Tables 7 and 8, it is clear that not all datasets are equally complex computationally. The first table, covering the COCOMO and UCP approaches (both with three input variables and one target variable), represents a relatively simpler computational task. These datasets allow for a more direct comparison of methods, as they involve a smaller number of variables and thus a less complex HP space. In contrast, COSMIC FP with four input variables and one target variable, reflected in the second table, requires a more extensive exploration of the HP space, as indicated by the higher number of experiments required for Grid Search. This difference in complexity is crucial for understanding why certain methods, like Taguchi, demonstrate such significant efficiency gains across datasets with varying computational demands. Reducing the number of experiments is particularly valuable in more complex datasets, as it directly impacts computational resources, time, and eco-efficiency, making Taguchi’s method a standout choice in scenarios requiring a balance between thoroughness and efficiency. Furthermore, reducing the number of experiments is critical for minimizing computational costs and time, especially in scenarios involving large datasets or intricate models, across multiple approaches. The table underscores the importance of efficient HPO in achieving optimal performance with minimal resources. Moreover, reducing the number of experiments can significantly improve eco-efficiency. Fewer experiments mean less computational power is required, which directly reduces energy consumption. This reduction not only lowers operational costs but also minimizes the environmental impact associated with running large-scale computations. In scenarios where models need to be trained and optimized multiple times, such as in HP tuning, cutting down the number of experiments leads to a more sustainable and environmentally friendly process. Taguchi’s method, which requires fewer experiments, exemplifies how an efficient optimization strategy can contribute to a greener, more resource-conscious approach.

5.3. On the running times

The analysis of computational expense across different models, as shown in the provided graphs Fig. 10, illustrates significant differences in the efficiency of various HPO methods. Notably, GOAT method demonstrates exceptional performance by consistently achieving the lowest running times across all models, recording an implausibly uniform time of just 11 s for models such as XGBoost, LSTM, and GGSNN. This stark contrast highlights the method’s potential in rapidly tuning model parameters with minimal computational overhead. In comparison, traditional methods like Grid Search are markedly more time-consuming, especially for models like CatBoost, which recorded a running time of 5607.3 s Table 9. Random Search also shows variability in its efficiency, with a notably high running time of 18904.2 s for LightGBM. In contrast, more refined techniques like Bayesian Optimization, SMAC, and Hyperband manage to significantly reduce running

times, making them more efficient alternatives. Specifically, Hyperband reduces LSTM’s running time to 200.1 s, illustrating its superior efficiency in balancing thoroughness and speed. These findings underscore the trade-offs between the exhaustive nature of traditional methods and the efficiency of modern HPO techniques, with GOAT method emerging as particularly advantageous in scenarios demanding rapid model tuning.

6. Conclusion

The GOAT method represents a significant advancement in eco-efficient HPO for machine learning, deep learning, and graph neural networks. It reduces the number of required experiments and running times, thereby cutting down on computational resources and energy consumption. Traditional methods like Grid Search and Random Search, which typically require extensive computational power, are outperformed by GOAT’s efficient approach. Although the focus is on efficiency, the method maintains competitive performance, and the minor trade-offs observed in accuracy are generally outweighed by the substantial savings in time and computational cost. This method not only enhances model performance but also contributes to a more sustainable approach to artificial intelligence, addressing the growing need for environmentally responsible practices in the field.

In the context of COCOMO, COSMIC, and UCP models, as applied to 46 datasets, the GOAT method demonstrates its effectiveness in achieving high accuracy while minimizing computational costs. This approach directly reduces the carbon footprint associated with AI processes by lowering energy usage and resource demand. As the field of artificial intelligence continues to expand, the importance of eco-efficient methods like GOAT becomes increasingly clear. This method offers a powerful solution for organizations seeking to balance high-performance model development with sustainable, energy-efficient operations, making it a vital tool in the pursuit of environmentally conscious AI practices.

To conclude, the main contributions of our paper are outlined as follows:

**Scientific Contribution:** This research contributes to the academic base by introducing eco-efficient optimization strategies specifically designed for HPO in advanced models, including machine learning, deep learning, and graph neural networks. By implementing the Taguchi orthogonal array tuning method and proposing as a foundation for GOAT method, the study aims to significantly reduce the number of required experiments, thereby advancing the efficiency and effectiveness of state-of-the-art models such as XGBoost, LightGBM, and others.

**Societal Relevance:** From a societal perspective, this work addresses the growing concern of energy consumption and computational complexity in machine learning processes. By reducing the computational demands and the associated carbon footprint of extensive computational tasks, the research aligns with broader environmental sustainability goals while improving the accuracy and efficiency of predictive models used in various real-world applications.

### 6.1. Limitations

Despite the significant advantages offered by the GOAT method in terms of eco-efficiency and performance, certain limitations persist, particularly concerning the generalization capability of the created orthogonal arrays. One primary challenge lies in the difficulty of constructing sufficiently large orthogonal arrays to cover extensive HP spaces, especially when dealing with complex models or high-dimensional datasets. The construction of these arrays can become progressively impractical as the complexity of factors and their possible values increases, resulting in potential gaps in the exploration of the HP space. This limitation may result in the GOAT method not fully capturing the optimal configurations, particularly in scenarios requiring exhaustive tuning across a broad range of HPs. Consequently, while GOAT is highly effective for smaller, more manageable HP spaces, its applicability might be constrained in more complex settings where a more comprehensive search is needed.

Additionally, while the GOAT method effectively minimizes the number of experiments and execution times, it does face challenges with scalability in certain contexts. For instance, the method’s efficiency could diminish when applied to very large datasets or when integrated with distributed machine learning platforms, where the construction and application of orthogonal arrays may not scale seamlessly. Moreover, the reliance on predefined orthogonal arrays could limit the method’s flexibility in adapting to continuously updated datasets, where HP configurations may need to be dynamically adjusted over time. These limitations suggest that, although the GOAT method is a powerful tool for eco-efficient HPO, it may require further refinement to fully address the challenges of generalization, scalability, and adaptability in more complex and evolving data environments.

Compared to dynamic HPO methods such as Bayesian Optimization or multi-fidelity strategies like Hyperband and BOHB, GOAT does not adapt iteratively based on intermediate performance signals. This makes it less suitable in scenarios where real-time feedback, conditional parameter tuning, or adaptive budget allocation is essential, for instance, in resource-constrained AutoML pipelines or in systems with continuous model retraining. Furthermore, in problems involving complex conditional hyper-parameter hierarchies or where the search space changes over time (e.g., streaming data or online learning setups), techniques with built-in adaptivity and probabilistic exploration are likely to outperform GOAT. While we acknowledge that the dynamic aspect of orthogonal arrays could potentially be extended, this is currently outside the scope of the presented work and will be considered in future developments.

We acknowledge that GOAT does not implement comparisons with advanced dynamic HPO methods such as Hyperband, BOHB, or NAS. Our choice reflects a focus on reproducible, static tuning where runtime feedback and early stopping are not applicable. These advanced techniques may be superior in dynamic or adaptive contexts, but GOAT is deliberately designed for deterministic, eco-efficient deployment with transparent search coverage.

### 6.2. Future directions

Future research should explore the integration of “WHAT-IF” simulations with the GOAT method, particularly in conjunction with specialized Recurrent Neural Networks like Fuzzy Cognitive Maps (FCMs). This combination could enable more sophisticated modeling of complex, real-world scenarios, providing industries with powerful tools for predictive analysis and decision-making. FCMs, known for their ability to model dynamic systems with uncertainty, can be paired with GOAT to optimize HPs in a way that not only enhances model accuracy but also anticipates the outcomes of various strategic decisions under different conditions. This approach could be particularly valuable in sectors like finance, healthcare, and supply chain management, where understanding the potential impact of different actions is critical.

**Table 10**  
Abbreviations and definitions.

Abbreviation	Definition
AI	Artificial Intelligence
AUC	Area Under the Curve
BO	Bayesian Optimization
BO-GP	Bayesian Optimization with Gaussian Processes
BOHB	Bayesian Optimization HyperBand
CatBoost	Categorical Boosting
CIFAR-10	Canadian Institute for Advanced Research Dataset (10 Classes)
CNN	Convolutional Neural Network
COCOMO	Constructive Cost Model
COSMIC	Common Software Measurement International Consortium
CPU	Central Processing Unit
CV	Cross-Validation
DAG	Directed Acyclic Graph
DEEP-BO	Deep Bayesian Optimization
DL	Deep Learning
DNN	Deep Neural Network
EDA	Exploratory Data Analysis
EEAD	Eco-Efficiency Across Domain
FF	Fractional Factorial
FPA	Function Point Analysis
FSpiNN	Fast Spiking Neural Network
GA	Genetic Algorithm
GGNN	Gated Graph Neural Network
GGsNN	Gated Graph Sequence Neural Network
GHO	Gravitational Hardware Optimization
GNN	Graph Neural Network
GOAT	Green Orthogonal Array Tuning
GP	Gaussian Process
GPPT	Graph Pre-Training and Prompt Tuning
GRU	Gated Recurrent Unit
GS	Grid Search
GSD	Grad Student Descent
HB	Hyperband
HP	Hyperparameter
HPC	High-Performance Computing
HPO	Hyperparameter Optimization
KGT	Knowledge Graph Transformer
KNN	k-Nearest Neighbors
LightGMB	Light Gradient Boosting Machine
LSTM	Long Short-Term Memory
ML	Machine Learning
MSE	Mean Squared Error
OA	Orthogonal Array
OATM	Orthogonal Array Tuning Method
POA	Population-based Optimization Algorithm
PSO	Particle Swarm Optimization
RF	Random Forest
RS	Random Search
SMAC	Sequential Model-based Algorithm Configuration
SNN	Spiking Neural Network
SVM	Support Vector Machine
TPE	The Tree-structured Parzen Estimator
UCP	Use Case Points
WRS	Weighted Random Sampling
XGBoost	Extreme Gradient Boosting

Moreover, by applying GOAT in the context of these advanced simulations, it is possible to further refine its energy efficiency. As FCMs require less computational power compared to other deep learning models, the combination with GOAT could reduce the overall energy consumption during HP optimization. This synergy between GOAT and FCMs opens new avenues for developing scalable, eco-efficient AI applications that are not only accurate but also sustainable, making them well-suited for industries looking to minimize their environmental footprint while maximizing operational efficiency.

Finally, we acknowledge that the current implementation of GOAT does not include formal convergence guarantees or a theoretical performance bound, as it is based on deterministic sampling rather than probabilistic modeling. This design choice prioritizes reproducibility and computational simplicity. However, a promising direction for future research involves integrating GOAT with dynamic methods or

surrogate-based modeling techniques to explore hybrid approaches that may enable formal analysis, convergence guarantees, or adaptive behavior under changing optimization constraints.

### List of abbreviations

We provide an alphabetically sorted list of abbreviations used throughout the paper in Table 10, to streamline proofreading for readers.

### CRedit authorship contribution statement

**Nevena Ranković:** Writing – review & editing, Writing – original draft, Visualization, Software, Resources, Methodology, Investigation, Formal analysis, Conceptualization. **Dragica Ranković:** Writing – review & editing, Visualization, Investigation, Formal analysis, Conceptualization.

### Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

### Appendix A. Goat method implementation

Below is an example of the implementation of GOAT method based on Taguchi’s method for HPO, illustrating how the method systematically reduces the number of experiments by leveraging an orthogonal array generated through a *Latin square* extraction, where each element  $L_{ij}$  satisfies  $L_{ij} \neq L_{ik}$  and  $L_{ij} \neq L_{kj}$  for  $i \neq k$ . Although this method offers an effective way to explore the hyper-parameter space and find optimal configurations with reduced computational effort, it should be regarded as a basic starting point. Users are encouraged to customize this implementation to meet their particular requirements, considering the nature of the problem, the dataset characteristics, and the selected model(s). The example provided in Appendix A serves as a practical demonstration, but further customization may be necessary to fully optimize performance in different contexts.

### Appendix B. Formal properties of OA-based GOAT

This appendix formalizes what can be rigorously guaranteed for GOAT as a deterministic Design-of-Experiments (DoE) procedure based on orthogonal arrays (OAs). These guarantees are *design-theoretic* (orthogonality, balance, unbiasedness, concentration) and differ in nature from convergence results for surrogate-based optimizers.

#### B.1. Orthogonal arrays and balance

An orthogonal array  $OA(N, k, s, t)$  is an  $N \times k$  matrix with entries in  $\{1, \dots, s\}$  such that, for any subset of  $t$  columns, every  $t$ -tuple of levels occurs exactly  $N/s^t$  times. Strength  $t = 2$  implies pairwise balance across all factor pairs.

**Proposition 1 (Orthogonality of Main-effect Contrasts).** *Let  $X$  denote the (coded) design matrix formed from an  $OA(N, k, s, 2)$  with orthogonal polynomial (or Helmert) contrasts for main effects. Then the main-effect contrast columns are mutually orthogonal, i.e.,  $X^T X$  is block-diagonal with respect to factors.*

*Consequence.* Estimates of main effects are uncorrelated under the OA design, enabling clean attribution of factor contributions.

#### B.2. Unbiased estimation of main effects

Assume the black-box response on the discretized grid can be decomposed as

$$y = f(x_1, \dots, x_k) + \varepsilon = \mu + \sum_{i=1}^k g_i(x_i) + \sum_{i < j} h_{ij}(x_i, x_j) + r(x) + \varepsilon, \quad (54)$$

with  $\mathbb{E}[\varepsilon] = 0$ , where  $g_i$  are main effects,  $h_{ij}$  interactions, and  $r$  higher-order remainder.

For factor  $i$  and level  $\ell$ , define the OA level-mean as

$$\bar{y}_i(\ell) := \frac{1}{m} \sum_{n: x_i^{(n)} = \ell} y^{(n)}, \quad m = \frac{N}{s}. \quad (55)$$

**Proposition 2 (Unbiasedness Under Strength 2).** *Under an  $OA(N, k, s, 2)$ ,*

$$\mathbb{E}[\bar{y}_i(\ell)] = \mu + g_i(\ell), \quad (56)$$

*since the contributions of pairwise interactions  $h_{ij}$  cancel by balance in expectation.*

*Consequence.* Ranking levels of each factor by  $\bar{y}_i(\ell)$  is an unbiased procedure for ranking main effects.

#### B.3. Optimality under additivity

**Assumption 1 (Additivity on the Grid).** In Eq. (54),  $h_{ij} \equiv 0$  for all  $i \neq j$  and  $r \equiv 0$  (i.e.,  $f$  is additive on the discretized grid).

**Proposition 3 (Global Discrete Optimality Under Additivity).** *Under Assumption 1 and exact (noise-free) evaluation, choosing*

$$x_i^* \in \arg \min_{\ell \in \{1, \dots, s\}} \bar{y}_i(\ell), \quad (57)$$

*for each  $i$  and combining  $x^* = (x_1^*, \dots, x_k^*)$  yields the global minimizer of  $f$  over the discretized grid.*

*Consequence.* In additive settings, GOAT’s “best level per factor” rule attains the discrete optimum with  $N \ll s^k$  runs.

#### B.4. Stability with bounded interactions

Let the minimum main-effect gap for factor  $i$  be

$$\Delta_i := \min_{\ell \neq \ell'} \left| (\mu + g_i(\ell)) - (\mu + g_i(\ell')) \right|. \quad (58)$$

Assume interactions and higher-order terms are uniformly bounded by  $|h_{ij}(x_i, x_j)| \leq \eta$  and  $|r(x)| \leq \eta$  on the grid.

**Proposition 4 ( $\varepsilon$ -optimality Under Bounded Interactions).** *If  $\Delta_i > 2\eta$  for all  $i$ , then the combination of per-factor best levels chosen by GOAT is  $\varepsilon$ -optimal with*

$$\varepsilon \leq k\eta. \quad (59)$$

*Consequence.* Small interactions (relative to main-effect gaps) do not overturn the selected combination; deviation from the true optimum is bounded.

#### B.5. Finite-sample concentration and sample size

Assume sub-Gaussian observation noise with proxy  $\sigma^2$ . Each level mean aggregates  $m = N/s$  i.i.d. samples.

**Proposition 5 (High-probability Uniform Accuracy).** *For any  $\delta \in (0, 1)$ , with probability at least  $1 - \delta$ ,*

$$\max_{i \in \{1, \dots, k\}} \max_{\ell \in \{1, \dots, s\}} |\bar{y}_i(\ell) - \mathbb{E}\bar{y}_i(\ell)| \leq \sqrt{\frac{2\sigma^2 s}{N} \log\left(\frac{2ks}{\delta}\right)}. \quad (60)$$

```

1 # Step 1: Explicitly define the hyper-parameter space with respective discretized levels
2 hyperparameters = {
3     'learning_rate': [0.01, 0.1, 0.2],
4     'n_estimators': [100, 500, 1000],
5     'max_depth': [3, 5, 7]
6 }
7
8 # Step 2: Orchestrate the generation of an orthogonal array via the Latin square principle
9 def latin_square(levels, factors):
10     n = len(levels)
11     square = [[(i + j) % n for i in range(n)] for j in range(n)]
12     orthogonal_array = []
13     for index, row in enumerate(square):
14         experiment = [levels[i][row[i]] for i in range(factors)]
15         orthogonal_array.append({
16             'id': index,
17             'params': experiment,
18             'meta': f'Experiment {index + 1} with factors {experiment}'
19         })
20     return orthogonal_array
21
22 orthogonal_array = latin_square(hyperparameters.values(), len(hyperparameters))
23
24 # Step 3: Systematically execute computational experiments as prescribed by the orthogonal array
25 results = []
26 for experiment in orthogonal_array:
27     # Instantiate the model and train it with the defined parameter set
28     model = train_model(experiment['params'])
29     # Evaluate the model's predictive efficacy and compute associated metrics
30     performance_metrics = evaluate_model(model)
31     # Aggregate results in a structured format, including metadata
32     results.append({
33         'experiment_id': experiment['id'],
34         'params': experiment['params'],
35         'metrics': performance_metrics,
36         'meta': experiment['meta']
37     })
38
39 # Step 4: Perform an exhaustive comparative analysis to determine the most efficacious hyper-
40 parameter configuration
41 def select_best_experiment(results):
42     # Define a complex evaluation criterion, possibly multi-objective
43     best_experiment = max(results, key=lambda x: (x['metrics']['accuracy'], x['metrics']['f1_score']))
44     return best_experiment
45 best_experiment = select_best_experiment(results)

```

Listing 1: An Example of Implementation of GOAT method for HPO space in Python.

*Sample-size condition.* To estimate all level means within  $\varepsilon$  (uniformly) with probability  $\geq 1 - \delta$ , it suffices that

$$N \gtrsim \frac{2\sigma^2 s}{\varepsilon^2} \log\left(\frac{2ks}{\delta}\right). \quad (61)$$

If  $\min_i \Delta_i$  exceeds twice this margin (plus a bound for interaction leakage), GOAT selects the correct level per factor with probability at least  $1 - \delta$ .

### B.6. Computational budget

GOAT evaluates exactly  $N$  configurations (the OA runs), i.e.,  $O(N)$  black-box evaluations, versus  $s^k$  for a full factorial grid. Selection overhead is negligible.

*Scope clarification.* As a deterministic DoE method, GOAT does not employ surrogates or acquisition policies; hence BO-style convergence proofs do not apply by design. The guarantees above are the appropriate, model-agnostic properties for OA-based sampling and selection.

### Data availability

Datasets used for this study are publicly available and link is already given in the main body of the manuscript. Code used in this research will be made available upon reasonable request via email of corresponding author's email address: [n.rankovic@uvt.nl](mailto:n.rankovic@uvt.nl).

### References

- [1] J. Wu, X.Y. Chen, H. Zhang, L.D. Xiong, H. Lei, S.H. Deng, Hyperparameter optimization for machine learning models based on Bayesian optimization, *J. Electron. Sci. Technol.* 17 (1) (2019) 26–40.
- [2] E. Mansouri, M. Manfredi, J.W. Hu, Environmentally friendly concrete compressive strength prediction using hybrid machine learning, *Sustainability* 14 (20) (2022) 12990.
- [3] C. Gambella, B. Ghaddar, J. Naoum-Sawaya, Optimization problems for machine learning: A survey, *European J. Oper. Res.* 290 (3) (2021) 807–828.
- [4] E. Sarker, P. Halder, M. Seyedmahmoudian, E. Jamei, B. Horan, S. Mekhilef, A. Stojcevski, Progress on the demand side management in smart grid and optimization approaches, *Int. J. Energy Res.* 45 (1) (2021) 36–64.

- [5] N. Andrei, et al., *Nonlinear Conjugate Gradient Methods for Unconstrained Optimization*, Springer, 2020.
- [6] N. Tran, J.G. Schneider, I. Weber, A.K. Qin, Hyper-parameter optimization in classification: To-do or not-to-do, *Pattern Recognit.* 103 (2020) 107245.
- [7] S. Holly, T. Hiessl, S.R. Lakani, D. Schall, C. Heitzinger, J. Kemnitz, Evaluation of hyperparameter-optimization approaches in an industrial federated learning system, in: *Data Science-Analytics and Applications: Proceedings of the 4th International Data Science Conference-IDSC2021*, Springer, 2022, pp. 6–13.
- [8] R. Khalid, N. Javaid, A survey on hyperparameters optimization algorithms of forecasting models in smart grid, *Sustain. Cities Soc.* 61 (2020) 102275.
- [9] E. Osaba, E. Villar-Rodríguez, J. Del Ser, A.J. Nebro, D. Molina, A. LaTorre, P.N. Suganthan, C.A.C. Coello, F. Herrera, A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems, *Swarm Evol. Comput.* 64 (2021) 100888.
- [10] X.S. Yang, Nature-inspired optimization algorithms: Challenges and open problems, *J. Comput. Sci.* 46 (2020) 101104.
- [11] L. Yang, A. Shami, On hyperparameter optimization of machine learning algorithms: Theory and practice, *Neurocomputing* 415 (2020) 295–316.
- [12] A. Morales-Hernández, I. Van Nieuwenhuysse, S. Rojas Gonzalez, A survey on multi-objective hyperparameter optimization algorithms for machine learning, *Artif. Intell. Rev.* 56 (8) (2023) 8043–8093.
- [13] M. Wistuba, A. Kadra, J. Grabocka, Supervising the multi-fidelity race of hyperparameter configurations, *Adv. Neural Inf. Process. Syst.* 35 (2022) 13470–13484.
- [14] A. Makarova, H. Shen, V. Perrone, A. Klein, J.B. Faddoul, A. Krause, M. Seeger, C. Archambeau, Overfitting in Bayesian optimization: an empirical study and early-stopping solution, in: *2nd Workshop on Neural Architecture Search, NAS 2021 Collocated with the 9th ICLR 2021*, 2021.
- [15] A. Pal, Y. Wang, L. Zhu, G.G. Zhu, Multi-objective surrogate-assisted stochastic optimization for engine calibration, *J. Dyn. Syst. Meas. Control.* 143 (10) (2021) 101004.
- [16] L. Hertel, J. Collado, P. Sadowski, J. Ott, P. Baldi, Sherpa: Robust hyperparameter optimization for machine learning, *SoftwareX* 12 (2020) 100591.
- [17] H. Cho, Y. Kim, E. Lee, D. Choi, Y. Lee, W. Rhee, Basic enhancement strategies when using Bayesian optimization for hyperparameter tuning of deep neural networks, *IEEE Access* 8 (2020) 52588–52608.
- [18] A.H. Victoria, G. Maragatham, Automatic tuning of hyperparameters using Bayesian optimization, *Evol. Syst.* 12 (1) (2021) 217–223.
- [19] R. Andonie, A.C. Florea, Weighted random search for CNN hyperparameter optimization, 2020, arXiv preprint arXiv:2003.13300.
- [20] J.H. Han, D.J. Choi, S.U. Park, S.K. Hong, Hyperparameter optimization using a genetic algorithm considering verification time in a convolutional neural network, *J. Electr. Eng. Technol.* 15 (2) (2020) 721–726.
- [21] H. Alibrahim, S.A. Ludwig, Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization, in: *2021 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2021*, pp. 1551–1559.
- [22] B. Du, C. Yuan, R.A. Barton, T. Neiman, H. Tong, Hypergraph pre-training with graph neural networks, 2021, arXiv preprint arXiv:2105.10862. URL <https://arxiv.org/abs/2105.10862>.
- [23] Y. Yuan, W. Wang, W. Pang, A genetic algorithm with tree-structured mutation for hyperparameter optimisation of graph neural networks, in: *2021 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2021*, pp. 482–489.
- [24] Y. Zhang, Z. Zhou, Q. Yao, Y. Li, Efficient hyper-parameter search for knowledge graph embedding, in: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, Dublin, Ireland, 2022, pp. 2715–2735, <http://dx.doi.org/10.18653/v1/2022.acl-long.194>.
- [25] M. Sun, K. Zhou, X. He, Y. Wang, X. Wang, GPPT: Graph pre-training and prompt tuning to generalize graph neural networks, in: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, ACM, 2022*, pp. 1717–1727.
- [26] Z. Shen, H. Yang, Y. Li, J. Kwok, Q. Yao, Efficient hyper-parameter optimization with cubic regularization, in: *Advances in Neural Information Processing Systems, 2023*.
- [27] R.V.W. Putra, M. Shafique, FSpiNN: An optimization framework for memory-efficient and energy-efficient spiking neural networks, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 39 (11) (2020) 3601–3613, <http://dx.doi.org/10.1109/TCAD.2020.3013049>.
- [28] A.A. Chowdhury, M.A. Hossen, M.A. Azam, M.H. Rahman, DeepQGO: Quantized greedy hyperparameter optimization in deep neural networks for on-the-fly learning, *IEEE Access* 10 (2022) 6407–6416, <http://dx.doi.org/10.1109/ACCESS.2022.3141781>.
- [29] G. Lemaître, F. Nogueira, C.K. Aridas, Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning, *J. Mach. Learn. Res.* 18 (1) (2017) 559–563, URL <http://jmlr.org/papers/v18/16-365.html>.
- [30] M.J. Islam, S. Ahmad, F. Haque, M.B.I. Reaz, M.A.S. Bhuiyan, M.R. Islam, Application of min-max normalization on subject-invariant EMG pattern recognition, *IEEE Trans. Instrum. Meas.* 71 (2022) 1–12.
- [31] X. Pei, Y.H. Zhao, L. Chen, Q. Guo, Z. Duan, Y. Pan, H. Hou, Robustness of machine learning to color, size change, normalization, and image enhancement on micrograph datasets with large sample differences, *Mater. Des.* 232 (2023) 112086, <http://dx.doi.org/10.1016/j.matdes.2023.112086>.
- [32] S. Abreu, Automated architecture design for deep neural networks, 2019, arXiv preprint arXiv:1908.10714.
- [33] A.A. Chowdhury, A. Das, K.K.S. Hoque, D. Karmaker, A comparative study of hyperparameter optimization techniques for deep learning, in: *Proceedings of International Joint Conference on Advances in Computational Intelligence: IJCACI 2021*, Springer, 2022, pp. 509–521.
- [34] A. Mamun, D. Guerra-Zubiaga, Y. Peng, Smart systems for real-time bearing faults diagnosis by using vibro-acoustics sensor fusion with Bayesian optimised 1-D CNNs, *Nondestruct. Test. Eval.* (2024) 1–25.
- [35] I. Ilievski, T. Akhtar, J. Feng, C. Shoemaker, Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.
- [36] S.S. Olof, A comparative study of black-box optimization algorithms for tuning of hyper-parameters in deep neural networks, 2018.
- [37] M. Adnan, A.A.S. Alarood, M.I. Uddin, I. ur Rehman, Utilizing grid search cross-validation with adaptive boosting for augmenting performance of machine learning models, *PeerJ Comput. Sci.* 8 (2022) e803.
- [38] G. Li, W. Wang, W. Zhang, Z. Wang, H. Tu, W. You, Grid search based multi-population particle swarm optimization algorithm for multimodal multi-objective optimization, *Swarm Evol. Comput.* 62 (2021) 100843.
- [39] P. Ren, Y. Xiao, X. Chang, P.Y. Huang, Z. Li, X. Chen, X. Wang, A comprehensive survey of neural architecture search: Challenges and solutions, *ACM Comput. Surv.* 54 (4) (2021) 1–34.
- [40] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, J. Sun, Single path one-shot neural architecture search with uniform sampling, in: *Computer Vision-ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, Springer, 2020, pp. 544–560.
- [41] W. Zhang, C. Wu, H. Zhong, Y. Li, L. Wang, Prediction of undrained shear strength using extreme gradient boosting and random forest based on Bayesian optimization, *Geosci. Front.* 12 (1) (2021) 469–477.
- [42] Y. Rimal, N. Sharma, A. Alsadoon, The accuracy of machine learning models relies on hyperparameter tuning: student result classification using random forest, randomized search, grid search, Bayesian, genetic, and optuna algorithms, *Multimedia Tools Appl.* (2024) 1–16.
- [43] T.T. Joy, S. Rana, S. Gupta, S. Venkatesh, Batch Bayesian optimization using multi-scale search, *Knowl.-Based Syst.* 187 (2020) 104818.
- [44] G. França, M.I. Jordan, R. Vidal, On dissipative symplectic integration with applications to gradient-based optimization, *J. Stat. Mech. Theory Exp.* 2021 (4) (2021) 043402.
- [45] A.A. Ewees, F.H. Ismail, A.T. Sahlol, Gradient-based optimizer improved by slime mould algorithm for global optimization and feature selection for diverse computation problems, *Expert Syst. Appl.* 213 (2023) 118872.
- [46] L. Du, R. Gao, P.N. Suganthan, D.Z. Wang, Bayesian optimization based dynamic ensemble for time series forecasting, *Inform. Sci.* 591 (2022) 155–175.
- [47] X. Wang, Y. Jin, S. Schmitt, M. Olhofer, Recent advances in Bayesian optimization, *ACM Comput. Surv.* 55 (13s) (2023) 1–36.
- [48] M. Binois, N. Wycoff, A survey on high-dimensional Gaussian process modeling with application to Bayesian optimization, *ACM Trans. Evol. Learn. Optim.* 2 (2) (2022) 1–26.
- [49] M. Anastasio, H.H. Hoos, Combining sequential model-based algorithm configuration with default-guided probabilistic sampling, in: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 301–302.
- [50] Y. Ozaki, Y. Tanigaki, S. Watanabe, M. Nomura, M. Onishi, Multiobjective tree-structured Parzen estimator, *J. Artificial Intelligence Res.* 73 (2022) 1209–1250.
- [51] M. Claesen, B. De Moor, Hyperparameter search in machine learning, 2015, arXiv preprint arXiv:1502.02127.
- [52] L. Bottou, Large-scale machine learning with stochastic gradient descent, in: *Proceedings of COMPSTAT'2010: 19th International Conference on Computational Statistics Paris France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, Springer, 2010, pp. 177–186.
- [53] S. Zhang, J. Xu, E. Huang, C.H. Chen, A new optimal sampling rule for multi-fidelity optimization via ordinal transformation, in: *2016 IEEE International Conference on Automation Science and Engineering, CASE, IEEE, 2016*, pp. 670–674.
- [54] R. Elshawi, M. Maher, S. Sakr, Automated machine learning: State-of-the-art and open challenges, 2019, arXiv preprint arXiv:1906.02287.
- [55] Z. Karnin, T. Koren, O. Somekh, Almost optimal exploration in multi-armed bandits, in: *International Conference on Machine Learning, PMLR, 2013*, pp. 1238–1246.
- [56] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, *J. Mach. Learn. Res.* 18 (185) (2018) 1–52.

- [57] N. Padfield, T. Camilleri, S. Fabri, M. Bugeja, K. Camilleri, A combined EEG motor and speech imagery paradigm with automated successive halving for customizable command selection, *Brain-Computer Interfaces* (2024) 1–18.
- [58] S. Falkner, A. Klein, F. Hutter, BOHB: Robust and efficient hyperparameter optimization at scale, in: *Proceedings of the 35th International Conference on Machine Learning*, in: *Proceedings of Machine Learning Research*, vol. 80, PMLR, 2018, pp. 1437–1446, URL <https://proceedings.mlr.press/v80/falkner18a.html>.
- [59] A. Gogna, A. Tayal, Metaheuristics: review and application, *J. Exp. Theor. Artif. Intell.* 25 (4) (2013) 503–526.
- [60] V.K. Kamboj, A. Nandi, A. Bhadoria, S. Sehgal, An intensify harris hawks optimizer for numerical and engineering optimization problems, *Appl. Soft Comput.* 89 (2020) 106018.
- [61] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, K. Leyton-Brown, et al., Towards an empirical foundation for assessing bayesian optimization of hyperparameters, in: *NIPS Workshop on Bayesian Optimization in Theory and Practice*, vol. 10, no. 3, 2013, pp. 1–5.
- [62] Z. Shen, Y. Zhang, L. Wei, H. Zhao, Q. Yao, Automated machine learning: From principles to practices, 2018, arXiv preprint [arXiv:1810.13306](https://arxiv.org/abs/1810.13306).
- [63] F. Hutter, L. Kotthoff, J. Vanschoren, *Automated Machine Learning: Methods, Systems, Challenges*, Springer Nature, 2019.
- [64] F. Itano, M.A.d.A. de Sousa, E. Del-Moral-Hernandez, Extending MLP ANN hyperparameters optimization by using genetic algorithm, in: *2018 International Joint Conference on Neural Networks, IJCNN, IEEE, 2018*, pp. 1–8.
- [65] F.G. Lobo, D.E. Goldberg, M. Pelikan, Time complexity of genetic algorithms on exponentially scaled problems, in: *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, 2000, pp. 151–158.
- [66] Y. Shi, R.C. Eberhart, Parameter selection in particle swarm optimization, in: *Evolutionary Programming VII: 7th International Conference, EP98 San Diego, California, USA, March 25–27, 1998 Proceedings 7*, Springer, 1998, pp. 591–600.
- [67] M.A. Zöller, M.F. Huber, Benchmark and survey of automated machine learning frameworks, *J. Artificial Intelligence Res.* 70 (2021) 409–472.
- [68] X.H. Yan, F.Z. He, Y.L. Chen, A novel hardware/software partitioning method based on position disturbed particle swarm optimization with invasive weed optimization, *J. Comput. Sci. Tech.* 32 (2017) 340–355.
- [69] M.Y. Cheng, K.Y. Huang, M. Hutomo, Multiobjective dynamic-guiding PSO for optimizing work shift schedules, *J. Constr. Eng. Manag.* 144 (9) (2018) 04018089.
- [70] S. Rahnamayan, H.R. Tizhoosh, M.M. Salama, A novel population initialization method for accelerating evolutionary algorithms, *Comput. Math. Appl.* 53 (10) (2007) 1605–1614.
- [71] H. Wang, Z. Wu, J. Wang, X. Dong, S. Yu, C. Chen, A new population initialization method based on space transformation search, in: *2009 Fifth International Conference on Natural Computation*, vol. 5, IEEE, 2009, pp. 332–336.
- [72] I. Guyon, A. Saffari, G. Dror, G. Cawley, Model selection: beyond the bayesian/frequentist divide, *J. Mach. Learn. Res.* 11 (1) (2010).
- [73] J.M. Kernbach, V.E. Staartjes, Foundations of machine learning-based clinical prediction modeling: Part II—Generalization and overfitting, *Mach. Learn. Clin. Neurosci.: Found. Appl.* (2022) 15–21.
- [74] B. Bischl, O. Mersmann, H. Trautmann, C. Weihs, Resampling methods for meta-model validation with recommendations for evolutionary computation, *Evol. Comput.* 20 (2) (2012) 249–275.
- [75] P. Bayle, A. Bayle, L. Janson, L. Mackey, Cross-validation confidence intervals for test error, *Adv. Neural Inf. Process. Syst.* 33 (2020) 16339–16350.
- [76] K. Stapor, P. Ksieniewicz, S. García, M. Woźniak, How to design the fair experimental classifier evaluation, *Appl. Soft Comput.* 104 (2021) 107219.
- [77] O.A. Montesinos López, A. Montesinos López, J. Crossa, Overfitting, model tuning, and evaluation of prediction performance, in: *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, Springer, 2022, pp. 109–139.
- [78] D. Patel, S. Shrivastava, W. Gifford, S. Siegel, J. Kalaganam, C. Reddy, Smart-ml: A system for machine learning model exploration using pipeline graph, in: *2020 IEEE International Conference on Big Data, Big Data, IEEE, 2020*, pp. 1604–1613.
- [79] X. He, K. Zhao, X. Chu, AutoML: A survey of the state-of-the-art, *Knowl.-Based Syst.* 212 (2021) 106622.
- [80] M. Bahri, F. Salutari, A. Putina, M. Sozio, AutoML: state of the art with a focus on anomaly detection, challenges, and research directions, *Int. J. Data Sci. Anal.* 14 (2) (2022) 113–126.
- [81] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, *Adv. Neural Inf. Process. Syst.* 28 (2015).
- [82] F. Mohr, M. Wever, E. Hüllermeier, ML-plan: Automated machine learning via hierarchical planning, *Mach. Learn.* 107 (2018) 1495–1515.
- [83] R.S. Olson, N. Bartley, R.J. Urbanowicz, J.H. Moore, Evaluation of a tree-based pipeline optimization tool for automating data science, in: *Proceedings of the Genetic and Evolutionary Computation Conference 2016, 2016*, pp. 485–492.
- [84] N. Rankovic, D. Rankovic, M. Ivanovic, L. Ladic, A new approach to software effort estimation using different artificial neural network architectures and Taguchi orthogonal arrays, *IEEE Access* 9 (2021) 26926–26936.
- [85] N. Rankovic, D. Rankovic, M. Ivanovic, L. Ladic, Influence of input values on the prediction model error using artificial neural network based on Taguchi's orthogonal array, *Concurr. Comput.: Pr. Exp.* 34 (20) (2022) e6831.
- [86] N. Rankovic, D. Rankovic, M. Ivanovic, L. Ladic, COSMIC FP method in software development estimation using artificial neural networks based on orthogonal arrays, *Connect. Sci.* 34 (1) (2022) 185–204.
- [87] N. Rankovic, D. Rankovic, M. Ivanovic, J. Kaljevic, Interpretable software estimation with graph neural networks and orthogonal array tuning method, *Inf. Process. Manage.* 61 (5) (2024) 103778.
- [88] M. Li, X. Fu, D. Li, Diabetes prediction based on xgboost algorithm, in: *IOP Conference Series: Materials Science and Engineering*, 768, (7) IOP Publishing, 2020, 072093.
- [89] D. Zhang, Y. Gong, The comparison of LightGBM and XGBoost coupling factor analysis and prediagnosis of acute liver failure, *IEEE Access* 8 (2020) 220990–221003.
- [90] D.D. Rufo, T.G. Debelee, A. Ibenhal, W.G. Negera, Diagnosis of diabetes mellitus using gradient boosting machine (LightGBM), *Diagnostics* 11 (9) (2021) 1714.
- [91] N. Rankovic, D. Rankovic, Delving into human factors through LSTM by navigating environmental complexity factors within use case points for digital enterprises, *J. Theor. Appl. Electron. Commer. Res.* 19 (1) (2024) 381–395.
- [92] A. Al Hamoud, A. Hoernig, K. Roy, Sentence subjectivity analysis of a political and ideological debate dataset using LSTM and BiLSTM with attention and GRU models, *J. King Saud Univ.-Comput. Inf. Sci.* 34 (10) (2022) 7974–7987.
- [93] F. Wang, Y. Laili, L. Zhang, Trust evaluation for service composition in cloud manufacturing using GRU and association analysis, *IEEE Trans. Ind. Inform.* 19 (2) (2022) 1912–1922.
- [94] L. Ruiz, F. Gama, A. Ribeiro, Gated graph convolutional recurrent neural networks, in: *2019 27th European Signal Processing Conference, EUSIPCO, IEEE, 2019*, pp. 1–5.
- [95] Y. Shi, Q. Li, X.X. Zhu, Building segmentation through a gated graph convolutional neural network with deep structured feature embedding, *ISPRS J. Photogramm. Remote Sens.* 159 (2020) 184–197.
- [96] Y. Tao, C. Wang, L. Yao, W. Li, Y. Yu, Item trend learning for sequential recommendation system using gated graph neural network, *Neural Comput. Appl.* (2023) 1–16.
- [97] W.C. Huang, C.T. Chen, C. Lee, F.H. Kuo, S.H. Huang, Attentive gated graph sequence neural network-based time-series information fusion for financial trading, *Inf. Fusion* 91 (2023) 261–276.
- [98] D. Rankovic, N. Rankovic, M. Ivanovic, L. Ladic, The generalization of selection of an appropriate artificial neural network to assess the effort and costs of software projects, in: *IFIP International Conference on Artificial Intelligence Applications and Innovations*, Springer, 2022, pp. 420–431.