HAW
HAMBURG

BACHELOR THESIS
Maximilian Boje

Hyperparameter Optimization for Deep Neural Networks using Reinforcement Learning

Faculty of Engineering and Computer Science
Department Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Maximilian Boje

Hyperparameter Optimization for Deep Neural Networks using Reinforcement Learning

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Regenerative Energiesysteme und Energiemanagement - Elektro- und Informationstechnik*
at the Department Information and Electrical Engineering
at the Faculty of Engineering and Computer Science
at University of Applied Sciences Hamburg

Supervisor: Prof. Dr. Kolja Eger
Supervisor: Prof. Dr. Wolfgang Renz

Submitted on: 28th of June 2025

Maximilian Boje

Thema der Arbeit

Hyperparameteroptimierung für Tiefe Neuronale Netze mit Hilfe von Reinforcement Learning

Stichworte

Hyperparameteroptimierung, Tiefe Neuronale Netzwerke, Reinforcement Learning, Partikelschwarmoptimierung, Multi-Fidelity-Optimierung

Kurzzusammenfassung

In dieser Arbeit wird ein Ansatz einer Partikelschwarmoptimierung mit unterliegendem Q-learning (QLPSO) für die Hyperparameteroptimierung von Machine-Learning Modellen implementiert und ausgewertet. QLPSO verwendet Reinforcement Learning um die zugrundeliegenden Optimierungsparameter des Partikelschwarmoptimierers während des Suchprozesses dynamisch anzupassen. Dieser Ansatz wird über das Ray Tune framework implementiert und mit etablierten Optimierunsmethoden verglichen. Dafür werden drei verschiedene Experimente durchgeführt: ein tiefes neuronales Netwerk für die Prognose von Stromverbrauch, eine Support-Vektor-Maschine für Klassifizierung des MNIST-Datensatzes und zwei mathematische Testfunktionen mit variierender Dimensionalität und unterschiedlicher Topologie. Der QLPSO Ansatz weist konkurrenzfähige Leistungen für die Optimierung des tiefen neuronalen Netzwerkes und bei niederdimensionalen Tesfunktionen auf, aber zeigt Schwierigkeiten bei der Optimierung der Support-Vektor-Maschine und hochdimensionalen Problemen auf. In dieser Arbeit werden die Grenzen der ursprünglichen QLPSO-Implementierung aufgezeigt und mögliche Verbesserungen für die zukünftige Forschung vorgeschlagen.

Maximilian Boje

Title of Thesis

Hyperparameter Optimization for Deep Neural Networks using Reinforcement Learning

Keywords

Hyperparameter Optimization, Deep Neural Networks, Reinforcement Learning, Particle Swarm Optimization, Multi-fidelity Optimization

Abstract

This thesis implements and evaluates a Q-learning-based particle swarm optimization (QLPSO) approach for hyperparameter optimization (HPO) of machine learning (ML) models. QLPSO uses reinforcement learning (RL) to dynamically adjust optimization parameters of the underlying particle swarm optimization (PSO) algorithm during the search process. This approach is implemented in the Ray Tune framework and benchmarked against established methods. Experiments span three problem domains: a deep neural network model for electricity consumption forecasting, support vector machine (SVM) classification on the MNIST dataset, and two mathematical test functions with varying dimensionality and distinct topology. The QLPSO approach displays competitive results for the deep neural network optimization and in lower-dimensional test functions, but struggles with the SVM task and high-dimensional problems. The thesis identifies limitations in the original QLPSO implementation and proposes potential improvements for future research.

Contents

List of Figures

List of Tables

List of Abbreviations

BO Bayesian optimization.

BOHB Bayesian optimization with Hyperband.

CNN Convolutional neural network.

ECDF Empirical cumulative distribution function.

EI Expected improvement.

HB Hyperband.

HPO Hyperparameter optimization.

KDE Kernel density estimator.

LSTM Long short-term memory.

MAPE Mean absolute percentage error.

ML Machine learning.

NN Neural network.

PSO Particle swarm optimization.

QLPSO Q-Learning-based particle swarm optimization.

RL Reinforcement learning.

RS Random search.

SH Successive halving.

SVM Support vector machine.

TPE Tree-structured parzen estimator.

UCB	Upper/lower confidence bound.

1 Introduction

1.1 Background

The past decade has witnessed an unprecedented rise in machine learning (ML) applications, fueled by a significant increase in computational power and the widespread adoption of GPU technology [11]. From transformative language models such as ChatGPT to advanced autonomous driving systems, these technologies have revolutionized both research and industry landscapes, while simultaneously exerting significant societal impacts. This technological leap has enabled the training of increasingly complex neural network (NN) architectures, leading to notable achievements in computer vision, natural language processing, and predictive analytics [23].
As ML models grow in complexity, the importance of hyperparameter optimization (HPO) becomes increasingly apparent. Unlike internal model parameters that are learned during training, hyperparameters are configuration settings specified before the learning process and may strongly affect a model's performance. Despite their critical importance, manual hyperparameter tuning remains commonplace in practice, introducing human bias and inefficiency into the development process [6]. HPO approaches offer a systematic and efficient alternative to manual tuning, allowing a more thorough exploration of the hyperparameter space, while reducing human effort and potentially leading to near-optimal configuration selection [52].
The field of HPO presents several significant challenges. The parameter spaces of ML models are often complex mixtures of continuous, discrete, categorical, and even conditional variables existing in high-dimensional, non-convex topologies. Optimal hyperparameter configurations may depend on the specific dataset and problem domain, e.g., a high learning rate being effective for one problem type may prove detrimental for another one. Although numerous optimization approaches have emerged, each exhibits distinct strengths and weaknesses. Sample efficiency is essential for problems with computationally expensive evaluations (such as training large NNs), but may be less critical

for low-cost optimization problems. As a result of these differences, most optimization algorithms excel only in specific problem types, with few universal methods applicable across all scenarios [52].
Somewhat counterintuitively, many HPO algorithms introduce their own set of parameters that need to be selected, effectively transforming rather than solving the underlying optimization challenge. These meta-parameters often remain problem-specific, possibly requiring expertise or prior knowledge of a problem's search space characteristics [6].
A Q-Learning-based particle swarm optimization (QLPSO) approach seeks to address this challenge by combining particle swarm optimization (PSO) with reinforcement learning (RL) [32]. By dynamically adjusting its parameters during the optimization process, QLPSO aims to adapt to the underlying problem characteristics without requiring parameter tuning itself.

1.2 Objectives

This thesis pursues three primary objectives. Firstly, it provides a comprehensive overview of common HPO algorithms, explaining their theoretical foundations, mechanisms, and respective strengths and weaknesses. Secondly, through implementation in the Ray Tune framework, these algorithms are benchmarked across diverse optimization problems to verify theoretical assumptions through empirical results. While HPO for deep neural networks is a primary focus, the benchmarks intentionally span a broader range of ML models and optimization scenarios: tuning of the SMARDCast model (a deep neural network architecture for electricity consumption forecasting), support vector machine (SVM) optimization on the MNIST dataset, and two standard optimization test functions with varying dimensionality and topological characteristics. This diverse set of problems allows for a more comprehensive evaluation of optimizer performance across different contexts. Finally, this thesis implements the QLPSO approach to evaluate its performance on these benchmarks, extending previous research by comparing it to established methods for HPO. Using the test function, the underlying behavioral characteristics are examined for different optimization scenarios.

2 Theory

This chapter establishes the theoretical foundation necessary to understand the topics discussed and methods used in this thesis. This includes a basic overview on the topic of ML with a more in-depth explanation of RL and an extensive overview of commonly used HPO algorithms including an RL approach.

2.1 Machine Learning

A diverse subfield of artificial intelligence is ML, which encompasses numerous methodologies and applications. ML enables models to learn from data and make predictions without being explicitly programmed [25]. These models range in complexity from straightforward approaches like ordinary least squares regression to sophisticated architectures such as neural networks and transformer models that power large language models such as ChatGPT and Claude [12, 44]. ML approaches typically fall into one of three main categories: supervised, unsupervised and reinforcement learning [47].
For brevity, this section focuses only on concepts relevant to the thesis.

Supervised Learning
Supervised learning involves training models on input data paired with corresponding expected outputs or labels, enabling the model to perform regression or classification tasks [8]. In the energy sector, a common regression application is forecasting renewable energy production. For instance, predicting solar or wind power output relies on labeled historical generation data (actual measured power) as well as corresponding input features such as solar irradiation, wind speed and other meteorological variables [46]. The trained model can then forecast future power generation based on new weather data inputs.

Unsupervised Learning
Unlike supervised learning, unsupervised learning works with unlabeled data, focusing not on direct prediction but on discovering underlying patterns, structures, and relationships within datasets [8]. One application is clustering audio content based on features such tempo, valence, energy levels, and harmonic characteristics to group similar songs together. These clusters might organically reveal musical genres or moods without predefined categories, helping to organize large music libraries or power recommendation systems of music streaming services [1].

The third fundamental category of ML is RL, which is explored in greater depth in section 2.1.3 due to its importance to this thesis.

2.1.1 Neural Networks

Inspired by the human's brain structure, NNs are computational frameworks composed of interconnected processing nodes (neurons or perceptrons) arranged in layers designed to identify patterns and resolve complex problems through data-driven learning and represent a large branch of ML [36].
In a feedforward NN, information flows in one direction from an input layer through hidden layers to an output layer. Each connection between neurons has an associated weight that determines the strength of influence. The learning process involves a forward pass, where input data propagates through the network, with each neuron computing the sum of its weighted inputs in addition to a bias term, then applying an activation function. The network's output is then compared to the desired output using a loss function which quantifies the prediction error (e.g. absolute or squared error). During backpropagation the network's weights are then adjusted using gradient descent to minimize this loss function [43].
NNs with multiple hidden layers (deep neural networks) excel at identifying complex, non-linear relationships between data. Each layer transforms the data representation, with deeper layers capturing increasingly abstract features. Initial layers might detect simple patterns, middle layers combine these into more complex features, and deep layers recognize high-level concepts. This hierarchical feature extraction is the foundation of deep learning [4].
Despite their effectiveness, NNs are often characterized as 'black boxes' because the internal decision-making process is not easily interpretable by humans, the learned representations are distributed across thousands or millions of parameters, and the relationship

between inputs and outputs lacks transparency compared to traditional algorithms [35]. Convolutional neural networks (CNNs) specialize in processing grid-like data by employing convolutional layers that apply learnable filters across the input to detect spatial patterns [45]. They incorporate pooling operations to reduce dimensionality while preserving essential features, making them particularly effective for image recognition tasks [21]. Long short-term memory (LSTM) networks are recurrent NNs designed to process sequential data by maintaining information over extended time intervals through specialized memory cells [38]. Their gating mechanisms control information flow, enabling the network to learn long-term dependencies in tasks such as language modeling, speech recognition and time series forecasting.

2.1.2 Generalization

A fundamental goal in ML is developing models that generalize effectively, such that it is capable of performing well on previously unseen data rather than merely memorizing training examples (overfitting) [49].
To counteract and measure overfitting, the data is typically split into training and validation datasets. The model learns exclusively from the training data but is evaluated on the validation set to estimate its performance on unseen examples. Training usually terminates early when the validation performance reaches a plateau or deteriorates significantly, preventing further overfitting. In some cases, a third subset of the data is implemented called the test data, which is reserved to assess the model once the training process is finished. This can help to determine real world performance since it remains entirely unused during the model's training phase, offering an unbiased evaluation of the model's generalization capabilities.

2.1.3 Reinforcement Learning

RL distinguishes itself from supervised and unsupervised learning by not training on a static dataset, but instead learning through dynamic interaction with an environment. In this learning paradigm, an RL model, or 'agent', explores its environment to gather information through interaction. The core RL concepts outlined in this section are primarily based on information provided by OpenAI [39].
The core components of an RL system can be defined as:

- State s: The observation provided by the environment that represents its current condition
- Action a: The agent's response to a given state
- Reward R: Numerical feedback signal indicating the quality of an action in a given state
- Policy π: The strategy that determines the agent's action

The environment provides the agent with observations in the form of a state s, which the agent then acts upon through an action a and learns through positive and negative feedback in the form of a reward signal R. Environments can be characterized through a state transition function P, which is oftentimes stochastic and describes the likelihood of transitioning to a new state s' given the current state s and action a:

$$s' \sim P(\cdot \mid s, a) \tag{2.1}$$

This transition function can also be deterministic instead of stochastic.

RL algorithms that predict these state transitions through modeling or direct access to environment behavior are known as model-based methods. While these can significantly improve performance, they are often infeasible in practice and may be difficult to deploy if the modeled environment does not exactly match the real-world counterpart. In practice, mostly model-free methods are used due to their easier implementation and ability to be deployed on any kind of environment, such that model-based methods are not discussed further in this chapter.

The agent itself can be described through a stochastic policy π, which describes the likelihood of choosing any action when the environment is in a given state.

$$a \sim \pi(\cdot \mid s) \tag{2.2}$$

The policy may also be expressed deterministically, directly mapping actions to states. Since there is no labeled data available, feedback for the training process is provided through a custom reward function. This reward function returns a numerical value that signifies whether the action chosen in this state was beneficial or detrimental.

$$R(s, a, s') \rightarrow \mathbb{R} \tag{2.3}$$

Based on the reward signal received after taking an action, the agent's behavior is adjusted to maximize for the cumulative sum of rewards over an episode, such that intuitively positive values serve to reinforce behaviors, while negative values penalize chosen actions.

For model-free RL, this learning process falls into one of two approaches: value-based and policy-based. Value-based approaches aim to assess the quality of a given state or a state-action pair in the form of a Value- or Q-function respectively

$$V^{\pi}(s) = \mathbb{E}[R(s, a, s') + \gamma V^{\pi}(s')] \tag{2.4}$$

$$Q^{\pi}(s, a) = \mathbb{E}[R(s, a, s') + \gamma \mathbb{E}[Q^{\pi}(s', a')]] \tag{2.5}$$

where $\gamma \in [0, 1]$ represents a discount factor and $\mathbb{E}$ denotes the expected value, which essentially weighs outcome values with their respective probability. Both equations assume actions drawn according to equation 2.2 from the current policy π (on-policy) and state transitions occurring as per equation 2.1. The discount factor serves two purposes: it ensures mathematical convergence of the sum for infinite-horizon problems, and it captures the intuitive notion that immediate rewards are typically more valuable than distant ones.

Equations 2.4 and 2.5 are based on the Bellman equations, which express a state's value through the expected reward of the current state and the next state's estimated value.

Assuming these state-values $V(s)$ are representative of the underlying truth, an agent may adapt its policy to seek high-value states while avoiding low-value ones. With using Q-values $Q(s, a)$, the agent's decision-making becomes even more direct, as it may simply select the action a that maximizes this value for a given state s.
A simple and popular value-based approach is Q-learning. In its most practical application, this involves using a Q-table, which represents Q-values for all state-action pairs, such that given a state, the policy would be to select the action assigned with the highest Q-value.

$$\pi(s) = \arg\max_a Q(s, a) \tag{2.6}$$

Q-values are typically updated each step by considering the current reward signal and the next state's expected reward, modeled by the maximum Q-value of s'

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left[R(s, a, s') + \gamma \max_{a'} Q(s', a') \right] \tag{2.7}$$

where α is the learning rate [15]. This update rule balances retaining previously learned information with incorporating new experiences through the Bellman equation 2.5 for calculating expected Q-values.
While Q-learning through a Q-table is straightforward to implement, it struggles with high dimensionality and continuous state and action spaces. To address these limitations, deep Q-networks approaches attempt to model the Q-function through a NN, enabling better handling of continuous spaces and higher dimensionalities.
Policy-based methods change a policy function's parameters directly, instead of assigning values to states or state-action pairs, often using gradient methods for updating. Value and policy methods can also be combined into so-called Actor-Critic models, with the actor representing a policy-based method for selecting actions and the critic estimating the Value-function for additional feedback. As only a Q-learning method is used in this thesis, policy-based and Actor-Critic methods fall beyond the scope of this work.

2.2 Hyperparameter Optimization

When training any ML model, some meta parameters must be chosen, which affect both the model's final performance and its training process. These parameters, which do not directly change the model's internal parameters such as the weights in a NN, are conventionally referenced as hyperparameters. Finding optimal hyperparameters is crucial for maximizing model performance, but presents a significant challenge due to the complex, often non-convex nature of the hyperparameter space. Although rule-of-thumb approaches were common in the formative years of ML, more sophisticated algorithms have been developed to systematically optimize model performance [52]. This section covers the most popular HPO methods, explains their mechanisms, and analyzes their respective strengths and weaknesses.

2.2.1 Trial and Error

The trial and error method, colloquially known as 'Grad Student Descent', is the manual method of choosing hyperparameters [52]. This approach involves the student or researcher selecting parameters based on their prior experiences and fine-tuning these until either sufficient performance or a deadline is reached. While domain knowledge can help find a decent configuration, this approach has several major limitations: it is not automated, results exhibit strong human bias, and the final performance will likely be suboptimal. This approach can be used for initial prototyping, but is generally to be avoided due to its poor efficiency and performance issues.

2.2.2 Grid Search

Arguably, the most commonly used HPO method is grid search, which requires a set of predetermined values chosen for each hyperparameter by the user. Parameter configurations are then evaluated systematically until all combinations are tried. This method is straightforward to implement, allows for maximum user control, and is easily parallelizable. However, it has severe limitations. Similar to trial and error, grid search suffers from human bias, especially for continuous parameters, as manually selecting optimal discrete values is essentially impossible. Additionally, its computational complexity increases exponentially with $O(n^k)$, assuming k parameters with n distinct values each, making it inefficient for high-dimensional configuration spaces [52]. Due to the neccessarily involved human bias and efficiency issues laid out in the next section, usage of this approach should be limited to prototyping in low-dimensional configuration spaces.

2.2.3 Random Search

Random search (RS) samples values using probability distributions (typically uniform) within user-defined ranges. This method is more straightforward to implement than grid search, as only parameter ranges need to be specified rather than specific values. This reduces human bias while still allowing for parallelization. A significant advantage of RS over grid search, empirically and theoretically demonstrated by Bergstra and Bengio [5], is its superior efficiency, especially for continuous parameters, as illustrated in Figure 2.1. In this 2-dimensional illustration, both search algorithms evaluate nine parameter combinations for a function $f(x, y) = g(x) + h(y) \approx g(x)$, with $g(x)$ displayed in green and $h(y)$ in yellow. Since parameter y has virtually no impact on the resulting value, grid search evaluates only three distinct values for $g(x)$, while RS explores the space more efficiently, using all nine evaluations [5]. Although RS is typically more efficient than grid search, it still suffers from low sample efficiency, as it does not take previous results into account, making it nearly unviable for expensive-to-evaluate objective functions like HPO of large ML models [52].

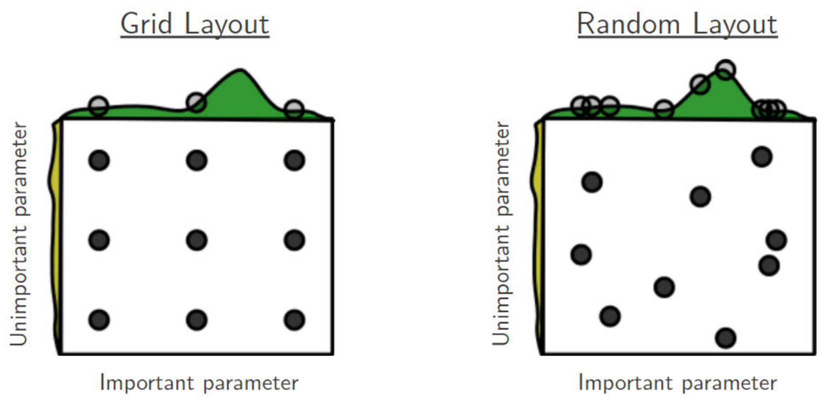


Figure 2.1: Grid search vs. random search illustration. Source: [5]

2.2.4 Hyperband

The Hyperband (HB) algorithm proposed by Li et al. [30] is essentially a highly efficient version of random search for multi-fidelity optimization problems, which can be approached incrementally with increasingly accurate but computationally more expensive evaluations. A typical example for a multi-fidelity task is the HPO of an ML model, as the model can be evaluated after only a few training epochs, saving computation time at the cost of less accurate performance estimates.
At the core of the HB algorithm is the Successive halving (SH) subroutine proposed by Jamieson and Talwalkar [26]. SH uniformly distributes a resource budget across a set of parameter configurations and terminates the worse-performing half of configurations after this budget is exhausted. This process repeats until only one configuration remains, which is then evaluated on the maximum budget, as illustrated in Figure 2.2. For example, in HPO of a NN with ten sampled configurations and a resource budget of 50 epochs, each parameter configuration would initially be trained for five epochs. After this initial training, only the five best-performing configurations would continue with additional training on a larger budget.

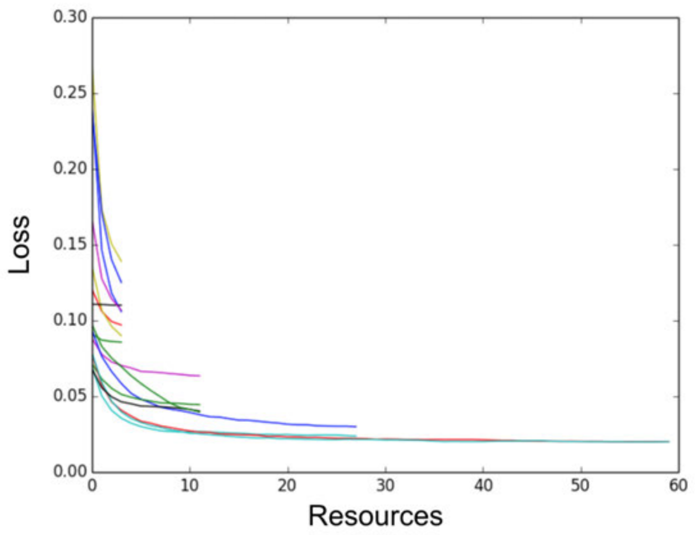


Figure 2.2: Successive halving illustration. Source: [30]

Under the assumption that low-fidelity evaluations correlate with high-fidelity results, SH minimizes resources allocated to unpromising configurations, achieving more extensive exploration of the hyperparameter space than random search. Building upon the

SH subroutine, HB uses brackets s to regulate the number of configurations n_i and their respective resource allocation r_i for each SH round i. This bracketing system requires two inputs: R, representing the maximum resources to be allocated to a single configuration (e.g., maximum number of epochs), and η, the proportion of configurations discarded in each round of SH. The number of brackets is calculated through $s_{max} = \lfloor \log_\eta(R) \rfloor$, and they are sequentially run until the optimization process is complete (see Appendix Figure A.1 for an example).
According to the authors, this is "in essence performing a grid search over feasible values of n" (Li et al. [30], p. 7), which may produce suboptimal results since each HPO problem would have an optimal bracket. However, the practical advantage of using various brackets is ensuring decent performance across tasks without manually determining the optimal resource allocation parameters for each specific HPO problem. While optimizer parameters generally add complexity, HB's two parameters are manageable, as "results are not very sensitive to the choice of η" (Li et al. [30], p. 8), and R typically being predetermined by the task requirements (e.g., how many epochs a model should be trained). Although HB significantly improves resource allocation compared to RS, it still relies on random sampling for initial configurations.

2.2.5 Bayesian Optimization

Bayesian optimization (BO) represents a sophisticated HPO approach grounded in Bayes theorem that constructs a probabilistic model of the objective function through sequential evaluation, using this model to identify the most promising points for sampling new configurations. This surrogate model works in conjunction with an acquisition function that aims to balance between exploring uncertain regions and exploiting areas of potential value. Figure 2.3 illustrates this iterative process specifically for the Gaussian process surrogate model. While Görtler et al. [22] provides an in-depth explanation of Gaussian processes, the fundamental mechanism involves the usage of a kernel function to create a distribution across possible functions that align with observed data. For any new input, it predicts a normal distribution $\mathcal{N}(\mu, \sigma)$ where the mean μ represents the expected function value, and the standard deviation σ captures the model's uncertainty. This uncertainty is typically larger in regions farther from observed data points. This probabilistic representation forms the foundation upon which acquisition functions operate, determining the strategy for selecting the next sampling point based on the current model's predictions and uncertainties.

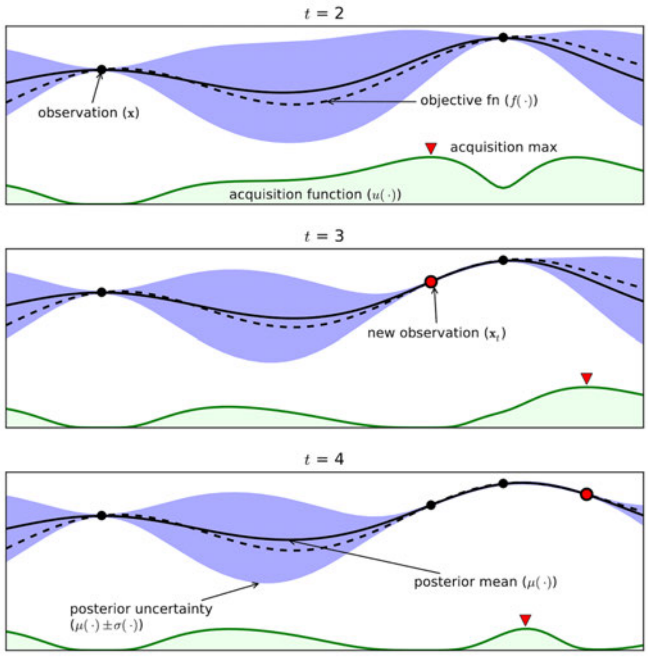


Figure 2.3: Bayesian optimization Gaussian process illustration. Source: [7]

The acquisition function plays a critical role in the BO framework, with several established approaches available for this purpose. Among these, the upper/lower confidence bound (collectively referred to as UCB throughout this thesis) and expected improvement (EI) methods are well-established options.
The UCB method represents the most straightforward acquisition function, simply representing the surrogate model's uncertainty bounds with a scaling factor λ.

$$UCB(x) = \mu(x) \pm \lambda \cdot \sigma(x) \tag{2.8}$$

The operation depends on whether the objective is to maximize (adding the scaled deviation) or minimize (subtracting the scaled deviation) the function. The scaling factor λ serves as a regulator for the exploration-exploitation trade-off, with higher values promoting exploration and lower values incentivizing exploitation. The optimal value for this parameter may change depending on the objective function at hand and while a decay parameter can be added to linearly decrease λ over time for more flexibility, these values still need to be selected manually by the user. As implemented in Nogueira [37], λ defaults to 2.576 without a decay factor.

The other acquisition function used in the experiments is EI, which takes both the probability and the magnitude of improvement into account by calculating the expected value of improvement upon the current best observation by a new configuration. The parameter ξ allows further control over this trade-off, with higher values encouraging broader exploration of the search space. A blog post by ekamperi [16] offers more detailed insights into EI and alternative acquisition functions, including mathematical derivations. The parameter ξ is typically set at 0.01 as noted by Adyatama [2].
Despite its advantages, using a Gaussian process for surrogate function modeling presents significant limitations, as it natively only supports usage of continuous parameters and poses high computational costs with a time complexity of $O(n^3)$ and a space complexity of $O(d^2)$ [52]. Furthermore, its sequential nature of building a surrogate after each sample severely restricts parallelization potential. For low-dimensional, continuous, and computationally expensive optimization problems, it still remains among the superior HPO approaches due to its high sample efficiency.

2.2.6 Bayesian Optimization with Hyperband

Bayesian optimization with Hyperband (BOHB), proposed by Falkner et al. [18], represents an approach that combines the strengths of both BO and HB to more efficiently solve multi-fidelity optimization problems. Unlike traditional BO implementations that use Gaussian processes, BOHB employs a tree-structured parzen estimator (TPE) as its surrogate model [51]. The TPE approach illustrated in Figure 2.4 divides samples into two distinct groups based on a threshold value y^γ (represented by the green line in the illustration). New samples with values $y \leq y^\gamma$ are assigned to the better group $\mathcal{D}^{(l)}$ (shown in red), while values $y > y^\gamma$ are assigned to the worse group $\mathcal{D}^{(g)}$ (shown in blue). This method then utilizes kernel density estimators (KDEs), which essentially apply normal distributions around samples and aggregate these distributions separately for each group to derive two probability density functions:

$$l(x) = p(x \mid \mathcal{D}^{(l)}) \tag{2.9}$$

$$g(x) = p(x \mid \mathcal{D}^{(g)}) \tag{2.10}$$

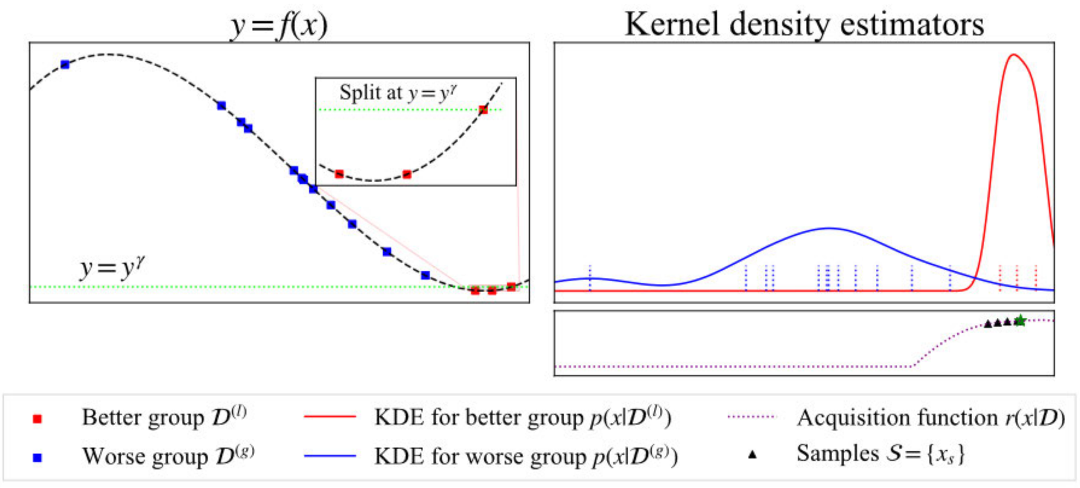


Figure 2.4: Tree parzen estimator illustration. Source: [51]

In Figure 2.4, $l(x)$ and $g(x)$ are represented by the red and blue KDEs respectively. The acquisition function is then described through the ratio of these probability density functions:

$$TPE(x \mid \mathcal{D}) := \frac{p(x \mid \mathcal{D}^{(l)})}{p(x \mid \mathcal{D}^{(g)})} = \frac{l(x)}{g(x)} \tag{2.11}$$

with $\mathcal{D}$ representing the set of all observations thus far. This ratio effectively weighs each point's estimated likelihood of belonging to the better group against its likelihood of belonging to the worse group. Falkner et al. [18] opted for this TPE approach instead of a Gaussian process due to its lower time complexity and capability to handle discrete parameters by employing a probability mass function, i.e., a discrete probability distribution.

The algorithm achieves parallelization by intentionally restricting the number of samples drawn from an adjusted KDE $l'(x)$, the same distribution $l(x)$ but with its bandwidths multiplied to promote exploration [18]. The sample with the highest attached acquisition function value as per Equation 2.2.6 is selected. This results in a deliberately sub-optimal configurations to avoid duplicates, which in turn enables concurrent evaluations. Additionally, the algorithm may with a small probability sample at random instead of through the TPE, "in order to keep the theoretical guarantees of HB" (Falkner et al. [18], p. 4).

Since BOHB functions as a multi-fidelity optimization algorithm, the TPE model aims to represent evaluations with the maximum possible fidelity. For this, it utilizes the set of evaluations with the highest budget/fidelity that contains a sufficient number of evaluations $N_{min} = d + 1$, with d representing the dimensionality of the search space.
Through extensive testing on various optimization problems, BOHB has demonstrated competitive or superior performance in most scenarios [18].

2.2.7 Particle Swarm Optimization

Within the field of optimization algorithms, PSO emerged as an effective nature-inspired contender. Developed by Kennedy and Eberhart [27], this algorithm draws inspiration from the collective behavior of animal groups such as bird flocks, where individuals adjust movements based on both personal experience and social information.
The core mechanism remains consistent across various implementations: multiple particles navigate through a D-dimensional space, each represented by position vectors $\mathbf{x_i} = \left[x_i^1, x_i^2, \ldots, x_i^D\right]$ and velocity vectors $\mathbf{v_i} = \left[v_i^1, v_i^2, \ldots, v_i^D\right]$. Each dimension represents one parameter of the optimization problem, such that each particle's position corresponds to a specific parameter configuration to be evaluated. The algorithm operates by iteratively updating the velocity and position of each particle based on three factors: the particle's current velocity, its personal best position ($\mathbf{b}_i$), and the global best position discovered by any particle in the swarm ($\mathbf{b}_g$). Following evaluation of the entire swarm, updates occur according to the equations:

$$\mathbf{v}_i = \omega \mathbf{v}_i + c_1 r_1 (\mathbf{b}_i - \mathbf{x}_i) + c_2 r_2 (\mathbf{b}_g - \mathbf{x}_i) \tag{2.12}$$

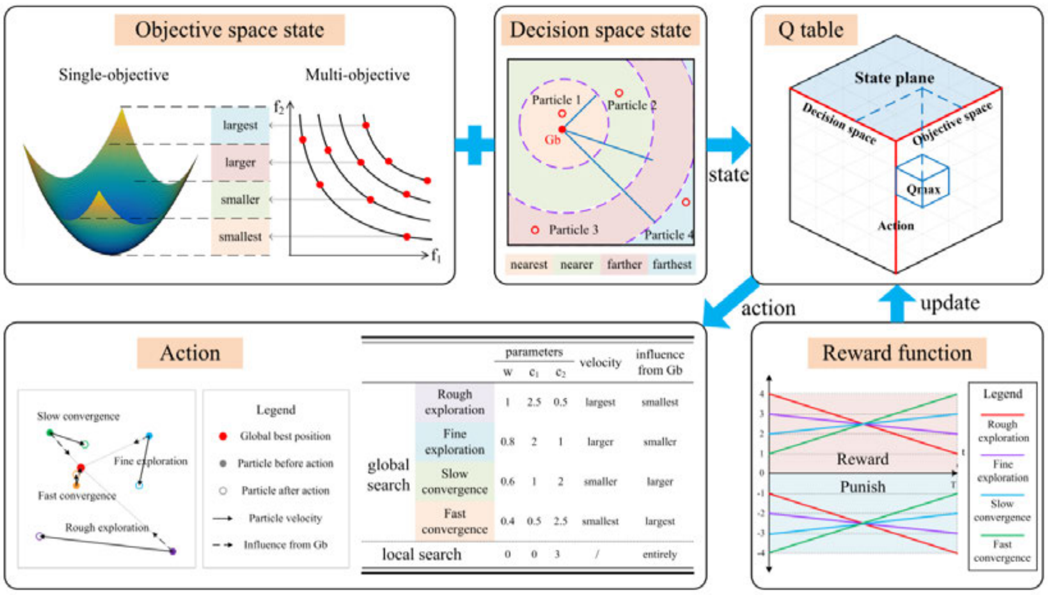
$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{v}_i \tag{2.13}$$

where $i = 1, 2, \ldots, N$ denotes a particle's index in the population size N, ω represents inertia, and r_1 and r_2 are random values drawn from a uniform distribution within the range $[0, 1]$. The cognitive parameter c_1 governs how much a particle trusts its own experiences and discoveries, encouraging exploration of promising areas it has individually found. Complementary, the social parameter c_2 determines the particle's reliance on the swarm's collective knowledge, promoting exploitation of globally promising regions. This balance between personal and collective information creates the characteristic swarm behavior.

Despite the algorithm's conceptual simplicity, several implementation considerations warrant attention. The three key parameters - inertia ω and the cognitive and social parameters c_1 and c_2 require careful selection as they directly impact the algorithm's convergence properties. Inertia is typically set around 0.9 and may incorporate a decay mechanism to facilitate exploration in early iterations while promoting exploitation in later stages [3]. This time-varying approach allows particles to broadly search the solution space initially before focusing on refining promising solutions. The cognitive and social parameters are commonly assigned equal values, usually between 1.5 and 2, with lower values potentially enhancing convergence properties [50]. Higher values for these parameters increase the step sizes particles take, potentially accelerating convergence but risking overshooting optimal solutions. These parameters should be calibrated in relation to each other, as excessive values may result in these undesired acceleration behaviors. Implementing velocity limits through clipping may further mitigate such rapid acceleration issues [50].
Given that optimization typically occurs within constrained search spaces, boundary handling becomes necessary when particles attempt to exceed defined limits. Without appropriate boundary management, particles might waste computational resources exploring infeasible regions or become trapped outside the solution space entirely. The reflection approach with random dampening suggested in Huang and Mohan [24], where particles encountering boundaries are reflected back with a dampening factor d sampled through $d \sim \mathcal{U}(0, 1)$, has demonstrated superior empirical performance compared to alternative methods.
Particle position initialization represents another significant consideration. While uniform distribution across the parameter space serves as the standard approach, the findings in Cazzaniga et al. [10] suggest that alternative distributions such as logarithmic, normal, or lognormal may yield substantial performance improvements depending on the objective function's topology.
Similar to BO, PSO lacks native support for discrete and categorical parameters. Though various approaches to address this limitation have been proposed, most involve considerable complexity, and the issue is still considered a problem due to limited research [50].
PSO offers notable advantages as an optimization technique, as it is easily parallelized and empirically performs well in high dimensional, continuous optimization problems [52]. However, a big limitation is posed by the numerous parameters and behavioral aspects requiring user specification, which often depend heavily on the particular optimization task at hand [50].

2.2.8 Q-Learning-based Particle Swarm Optimization

An approach to dynamically control the three swarm parameters during the optimization process is QLPSO proposed in Liu et al. [32]. This method addresses the fundamental exploration-exploitation dilemma inherent in optimization algorithms: particles must broadly explore the search space to avoid local optima while also exploiting promising regions to find optimal solutions efficiently. Unlike traditional parameter control methods such as a decreasing inertia weight that follow predetermined schedules as examined in Alhussein and Haider [3], QLPSO uses a Q-table to adaptively change the parameters of individual particles based on their current state and performance.



		parameters			velocity	influence from Gb
		w	c_1	c_2		
global search	Rough exploration	1	2.5	0.5	largest	smallest
	Fine exploration	0.8	2	1	larger	smaller
	Slow convergence	0.6	1	2	smaller	larger
	Fast convergence	0.4	0.5	2.5	smallest	largest
local search		0	0	3	/	entirely

Figure 2.5: QLPSO illustration. Source: [32]

The idea is to promote more explorative or exploitative behavior depending on positioning, performance and optimization progress as illustrated in Figure 2.5. To capture these features, two discrete state representations are used - the objective space state and the decision space state. The prior maps the performance of a particle into four equal sized sections based on the difference between global best and worst fitness values. To characterize the positioning of particles, the decision space state divides the parameter space into four regions based on the Euclidean distance from the global best position of the swarm. Depending on the region a particle is positioned in, it is assigned the corresponding decision space state.

Actions are represented by four predefined parameter combinations to promote different expected characteristics, such as stronger exploration or faster convergence to the global best. Values of the Q-table are updated following the standard Q-learning update Equation 2.7 with the reward function from the illustration:

$$R = \begin{cases} m_a t + b_a & \text{for } P_{current} < P_{previous} \\ m_a t + b_a - 5 & \text{for } P_{current} \geq P_{previous} \end{cases} \tag{2.14}$$

where m_a and b_a are action-specific slope and intercept values (detailed in Table 2.1 with action references as per Figure 2.5), $t \in [0, 1]$ represents the normalized optimization progress and $P_{current}$ and $P_{previous}$ are the current and previous function evaluations. The conditions in Equation 2.14 assume the goal of minimizing the objective function, such that performance improvements of particles are rewarded and conversely a loss of performance is punished. The action-specific parameters ensure early-stage exploration is rewarded initially, with convergence behaviors being favored towards the end.

Table 2.1: QLPSO reward function parameters m_a and b_a by action

Action	m_a	b_a
Rough exploration	-3	4
Fine exploration	-1	3
Slow convergence	1	2
Fast convergence	3	1

The algorithm operates in two phases: during the first 90 % of iterations, particles use the Q-table for parameter selection (global search). For the final 10 %, QLPSO switches to a local search configuration ($\omega = c_1 = 0$ and $c_2 = 3$), forcing rapid convergence around the global best, which is recommended by the authors for its robustness [32].
Notably, QLPSO does not employ generalization capabilities described in section 2.1.2. Rather than training across different tasks, QLPSO adopts a purely instance-specific approach. The Q-table is initialized with zeros at the beginning of each new optimization task, with no knowledge transfer from previously encountered problems. This design choice allows the algorithm to adapt exclusively to the current problem's landscape without any assumptions derived from past experiences, effectively treating each optimization task as an independent learning problem.

3 Methodology

This chapter presents the methods used to evaluate and compare the different HPO methods. Three distinct optimization problems are examined: a CNN-LSTM model for electricity consumption forecasting, two optimization test functions with different characteristics, and a multi-fidelity SVM approach from literature applied to the MNIST dataset [13]. The evaluation methodology chapter addresses the challenges of fairly comparing different HPO algorithms across varying problem types. If not stated otherwise, the used HPO algorithms include RS, BO-UCB, BO-EI, HB, BOHB, PSO and QLPSO.

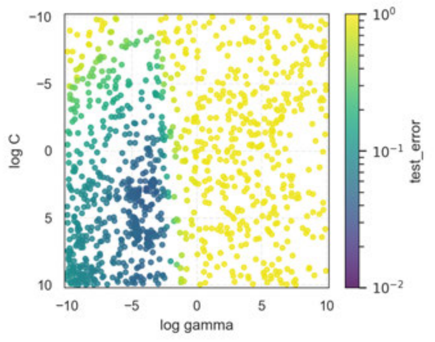
3.1 SMARDCast Model

The first optimization problem chosen for this thesis is a CNN-LSTM model derived from a publication on electricity consumption forecasting called SMARDCast [29]. Forecasts are made daily at 10 a.m. for a 24-hour window with a 15-minute resolution to match the methodology of the forecast data published on the German Federal Network Agency's SMARD platform [9].
For their approach, the authors used several input signals: the previous consumption values, time features such as hours or weeks transformed into sine and cosine signals respectively, the day of the week as a categorical feature and a sparse step signal to represent holidays, where the values are 0 by default, 0.5 the day before a holiday and 1 during the holiday. The CNN-LSTM model used the last 672 values as input, representing one week of data.
A grid search of 2,688 configurations resulted in a model with a final mean absolute percentage error (MAPE) of 2.99 % compared to the 3.72 % error of the SMARD forecasts. The search space for the HPO methods was extended in this thesis and is shown in Appendix Table A.2. Parameters marked with an asterisk (*) indicate layer-specific parameters and are therefore conditional on the number of layers, e.g. if two CNN layers are chosen, the number of filters in layer three would have no effect on the actual model.

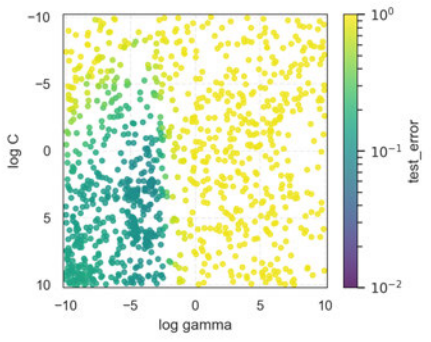
For HPO methods that are unable to handle conditional parameters (BO and PSO) this creates a “dead region” in the hyperparameter space, since the algorithm still samples these values even though they have no influence on the results. These dead parameter regions were maintained across all HPO algorithms for better experimental consistency, although it must be noted that this creates an artificially larger hyperparameter space for RS, HB and BOHB than would be necessary and should be seen as a disadvantage.
Discrete and categorical hyperparameters pose another issue for BO and PSO, as they only support continuous values. As a pragmatic solution, a stochastic rounding function is implemented to map between discrete and continuous space. Details of this rounding function can be found in section 4.4.
Each optimizer is executed three separate times, with each optimization run allocating a time budget of ten hours, resulting in a total optimization time of 210 hours for this task.

3.2 Support Vector Machine on MNIST

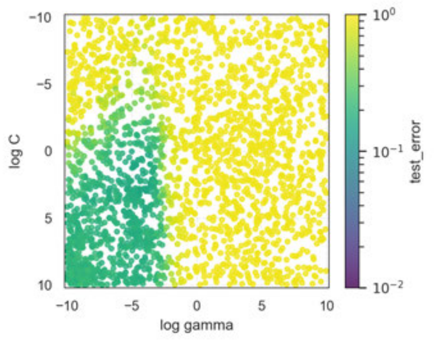
The second optimization problem involves multi-fidelity training of a SVM on the MNIST handwritten digits dataset, as introduced by Klein et al. [28] and later used by Falkner et al. [18] to benchmark the BOHB algorithm. Unlike iterative models, SVMs fit a hyperplane through the entire dataset once, so the multi-fidelity approach is implemented by training on increasingly larger subsets of the original data. Instead of using epochs as in Section 3.1, this approach uses a variable $s = \frac{N_{subset}}{N} \in (0, 1]$ to express the ratio of the subset size to the complete dataset. In the publications, it ranges from $1/512$ to 1, with each step doubling the previous proportion. The two tunable hyperparameters for the SVM are C and γ (gamma), which both are sampled using a Log-Uniform distribution (natural logarithm e.g., base e) with the range $[e^{-10}, e^{10}]$.
Due to computational constraints, a Nystroem kernel approximation was employed instead of the SVM’s radial kernel. The Nystroem approach approximates a kernel matrix by selecting a subset of m data points from a dataset of size n, reducing time complexity for matrix operations from $O(n^3)$ to $O(nm^2)$ [14]. By setting $m = \sqrt{n}$ the time complexity becomes quadratic $O(n^2)$ instead of the original cubic complexity. This approximated non-linear kernel map can then be used in combination of a linear SVM with an approximate time complexity of $O(n)$, significantly reducing overall time complexity [41]. Using this technique, a reproduction of Figure 1 from Klein et al. [28] was made for qualitative verification.



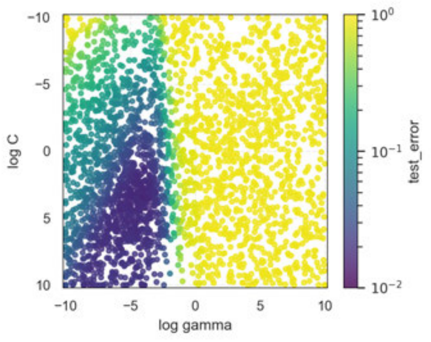
(a) $s = 1/128$



(b) $s = 1/16$



(c) $s = 1/4$



(d) $s = 1$

Figure 3.1: Test error of Nystroem approximated SVM for different values of s

Notably, Figure 3.1 shows that smaller subset sizes ($s = 1/128$, $s = 1/16$) seem to perform as well as or better than larger subsets ($s = 1/4$). This suggests that small, well-distributed subsets may sufficiently capture the essential structure for the Nystroem approximation, and that adding more data may have a negative impact on performance. This non-monotonic performance increase could have significant implications for the multi-fidelity optimizers HB and BOHB, as they typically assume performance improvements with increasing fidelity.

Still, this optimization task should provide valuable results, due to it offering a direct comparison with the findings in the BOHB paper and its established unimodal, convex hyperparameter space topology.

For this task, each optimization algorithm is run five times for two hours each using the test error (percentage of incorrect classifications) as the evaluation metric. This is due to increased variance levels in both evaluation and training time, resulting in a total optimization time of 70 hours.

3.3 Test Functions

The final optimization task performed consists of two test functions, namely the Rastrigin and the Styblinski-Tang (abbreviated as Styblinski throughout this thesis) function. Advantages of using test functions include their topology being known beforehand (see Figure 3.2) and the ability to easily change their dimensionality. The Rastrigin function serves as a highly multimodal function featuring many local optima in close proximity ('bumpy') while the Styblinski-Tang function exhibits a more convex shape with fewer and more spread out local minima ('smooth'). This difference in topology and the ability to compare different dimensionalities serve as a good control for evaluating algorithm performances.

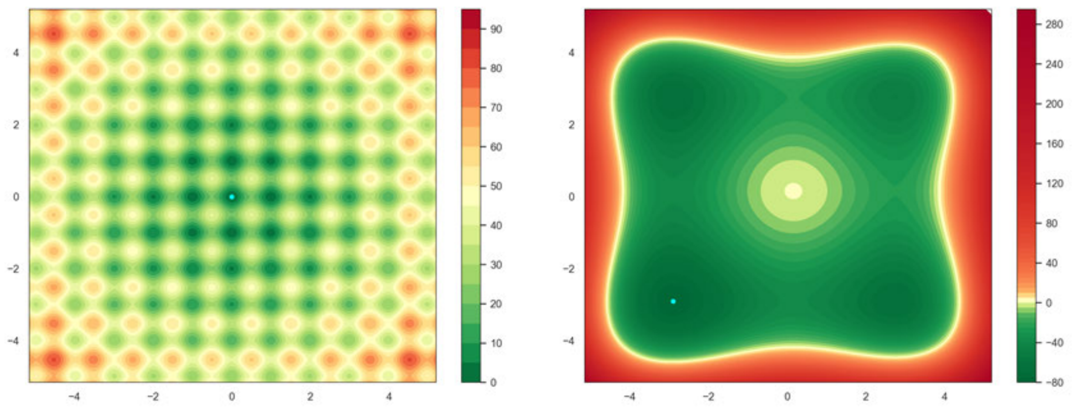


Figure 3.2: Rastrigin (left) and Styblinski-Tang function (right), cyan points represent respective minima

The two functions are described through:

$$f_{Rastrigin}(\mathbf{x}) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)] \tag{3.1}$$

$$f_{Styblinski}(\mathbf{x}) = \frac{\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i}{2} \tag{3.2}$$

where n is the dimensionality (number of parameters) and $A = 10$ represents a scaling factor for the Rastrigin function.

Since these functions can not be transformed into a multi-fidelity problem, both HB and BOHB can not be evaluated on this task. The other optimization algorithms are executed three times on both test functions, once with $n = 10$ and once with $n = 50$, to observe behavior in different dimensionalities. With five optimizers on essentially four different objective functions, this results in a total optimization time of 120 hours.

3.4 Evaluation Methodology

Fair evaluation and comparison of different HPO algorithms is a recognized challenge in the field [52]. Most approaches are not universally robust or usable across diverse optimization problems, but rather tend to be specialized for specific tasks or parameter spaces.

The performance metric to use for evaluation can vary depending on the type of the optimization problem at hand. For expensive optimization problems like training large ML models, minimizing the number of evaluations (sample efficiency) is often more important, while for low-cost test functions, the total runtime (time efficiency) becomes critical as individual evaluations take very little time.

When comparing multi-fidelity approaches like HB and BOHB with traditional methods, evaluating performance based solely on the number of samples can skew results due to low-fidelity evaluations typically performing worse than high-fidelity ones.

For better comparability and a more application-grounded approach, total compute time serves as the limiting factor for all algorithms. Limiting the number of configurations would unfairly advantage BO since each sample would be drawn optimally according to its underlying model, despite their significantly higher time complexity. At the same time, it would be highly disadvantageous for HB and BOHB since only a fraction of their samples are evaluated at maximum fidelity.

The metrics chosen to be optimized are validation loss for SMARDCast and test error for SVM on MNIST. Additionally, GPU usage is tracked as an indicator of how efficiently each optimizer utilizes available resources. However, since GPU parallelization is handled by RayTune, there may be discrepancies between observed and theoretical performance that might be difficult to explain precisely. These implementation details may prevent some algorithms from achieving their theoretically possible performance.

Due to the in parts statistical nature of all used HPO methods, each algorithm is run multiple times with different random seeds, which should help to obtain statistically relevant results.

4 Implementation

This chapter presents the technical implementation of HPO algorithms built upon the Ray Tune framework. It details the custom development of PSO and QLPSO algorithms, addresses implementation challenges such as handling discrete parameters via stochastic rounding, and describes the experimental setup for testing these approaches on the previously described experiments.

4.1 Ray Tune: Hyperparameter Optimization Framework

Ray Tune is a Python framework designed for HPO that supports all major ML libraries, including PyTorch, XGBoost, TensorFlow and Keras [31]. For optimization algorithms, Ray Tune provides access to Gaussian process BO (Bayesian Optimization [37]) and BOHB (HpBandSter [17]) through their respective external implementations. Additionally, optimizers such as GS, RS, HB and even population based training are available through Ray Tune's native internal implementations. However, PSO is not natively supported in Ray Tune, so a custom algorithm had to be created within the framework, which was also used for the QLPSO implementation. A detailed description of this PSO searcher can be found in section 4.2.
This section provides a brief overview of the four main components that comprise the Ray Tune framework and how they interact.

1. **Search Space**
   The search space or configuration space represents the hyperparameter domain to be explored during optimization, defining each parameter's distribution and their respective value ranges or discrete sets of available values. Ray Tune offers a wide variety of distributions, with the notable limitation that BOHB is not compatible with these sampler objects. Instead, HpBandSter's *ConfigSpace* class must be used, which supports most common distribution types.

2. **Trainable**
   Trainables function as the optimization objective to be solved, accepting a configuration of parameters as input and returning a corresponding value, such as the validation loss of an ML model. Since the only requirement is to return results using Ray Tune's *report* method, trainables are highly customizable and allow users to process hyperparameters as needed. This flexibility is valuable when handling algorithm limitations, such as BO and PSO's inability to natively process discrete hyperparameters, which can be addressed explicitly in the trainable by mapping continuous values to corresponding discrete ranges using custom logic.

3. **Search Algorithm**
   The search algorithm is responsible for proposing new parameter configurations for evaluation, potentially leveraging past results, such as BO creating a surrogate model based on all prior observations. Users can utilize existing models within the Ray Tune API or implement custom search logic through Ray Tune's generic *Searcher* class.

4. **Scheduler**
   Schedulers in Ray Tune serve as the interface between search algorithm and trainable, determining when specific configurations are evaluated. The simplest example is the *FIFOScheduler*, which evaluates samples in the order they are suggested. Schedulers can also manage the pausing or premature termination of trials in multi-fidelity problems, such as HB's and BOHB's successive halving approach. In Ray Tune terminology, trials typically represent unique parameter sets for evaluation; for multi-fidelity tasks, each trial encompasses all training iterations using that specific configuration.

These four building blocks make Ray Tune highly customizable while remaining accessible for prototyping through its existing API.
Another advantage is Ray Tune's built-in resource management capabilities. CPU, GPU and memory usage can be specified per trial, and the number of concurrent trials can be limited. Multi-threading and parallel processing are handled entirely by Ray Tune, ensuring seamless deployment for most hardware environments.

4.2 PSO Implementation

As mentioned prior, PSO is not natively supported by Ray Tune, necessitating a custom implementation of this algorithm through the *Searcher* class. This section covers both the technical integration in the Ray Tune framework, as well as details for the exact values and behaviors chosen, regarding the different possible approaches discussed in the theory chapter.
Regarding the previously mentioned *Searcher* class of the Ray Tune API, there are three corresponding methods that need to be utilized, through which the PSO algorithm is implemented in form of the custom *PSO_Searcher* class.

1. **Setting Search Properties**
   The *Searcher.set_search_properties* method is called at the beginning of an experiment to initialize the search algorithm by providing the evaluation metric, the mode (e.g., min or max) and the search space. For PSO implementation, swarm initialization occurs through several steps. First, either linear or logarithmic mappings to the fixed range $[0, 1]$ are created for each parameter based on their respective lower and upper bounds. Next, each parameter is sampled N times using the provided samplers in the configuration dictionary, with N representing the population size of the swarm, ensuring uniform distribution of particles throughout the hyperparameter space. Finally, these positions are used to initialize particles via a *Particle* class described later in this section, which are subsequently added to a list representing the swarm.

2. **On Trial Completion**
   This method (*Searcher.on_trial_complete*) activates upon trial termination and delivers results from the trainable with that trial's hyperparameters. Since PSO requires waiting for all N particle positions to be evaluated before calling a subroutine for updating, Ray Tune's *ConcurrencyLimiter* wrapper is used. This wrapper restricts the number of concurrent trial suggestions for evaluation by setting the *max_concurrent* parameter to the population size N and the *batch* argument to true. Upon evaluation of all trials, this method receives N sequential calls returning each particle's position result, which then updates the global and personal best positions when applicable.

3. **Suggesting new Configurations**
   After all N results are returned and processed through the previous method, the searcher provides a new set of hyperparameters via the *Searcher.suggest* method. This process is also called sequentially as many times as specified in the *ConcurrencyLimiter* each batch, e.g. N times. Due to the sequential nature, an internal counter serves as a proxy for particle ID, enabling accurate tracking for proper updates of personal best positions. During its first call in an experiment run, as no prior results are available, the method simply returns the randomly initialized positions of the particles. On the first call of this method in each batch, all particles' velocities and positions are updated according to equations 2.12 and 2.13.

Regarding the selection of swarm parameters and behaviors discussed in section 2.2.7, particle inertia follows a linear decrease[1] from 0.85 to 0.35 as used in Alhussein and Haider [3]. Social and cognitive parameters were set to $c_1 = c_2 = 1.5$ and since PSO is not too sensitive to population size, $N = 20$ was chosen [50]. Velocity clipping was configured to $\mathbf{v}_{max} = [0.25, 0.25, \ldots, 0.25] \in \mathbb{R}^D$, appearing like the most robust value in Cazzaniga et al. [10]. Out-of-bounds behavior follows the approach suggested in Huang and Mohan [24], whereby particles at the parameter space edge undergo reflection with a dampening factor sampled from a uniform random distribution $d \sim \mathcal{U}(0, 1)$. Regarding the SMARDCast model's noisy objective function characteristics, Parsopoulos and Vrahatis [40] determined that standard PSO adequately handles noise and may even help the swarm to avoid local minima, such that no additional mechanism for noise-handling is required.
Supplementary to the *PSO_Searcher* class, a *Particle* class is used to model the individual particles of the swarm. Each particle instance keeps track of its own position, velocity and personal best position, while the global best position and inertia are stored as class variables shared across the entire swarm. The class has two main methods: *update_velocity* and *update_position*, which apply the equations 2.12 and 2.13 of the PSO algorithm, while handling velocity limits and out-of-bounds corrections as described prior.

[1]Although Algorithm 3 demonstrated superior results in the paper, the parameters *a*-*d* appear highly tuned, such that ultimately algorithm 2 was chosen as a more pragmatic solution, potentially sacrificing some performance

4.3 QLPSO Implementation

This section details the implementation of the QLPSO search algorithm according to Liu et al. [32], building upon the previously outlined *PSO_searcher* class as its foundation. The resulting *QLPSO_Searcher* class integrates Q-learning functionality through a *qlpso_subroutine* method, which is executed after the final trial evaluation in the batched *on_trial_complete* method. This subroutine utilizes all recorded performances to establish objective space state thresholds, since both the SMARDCast and SVM tasks' objective function boundaries are not known a priori, unlike the QLPSO publication's test functions. The implementation employs median absolute deviation (MAD) as a robust outlier handling technique, creating a filtered performance list containing only values within a 3 MAD range from the median (equivalent to a 2σ range in a normal distribution). Thresholds are configured as illustrated in Figure 2.5, dividing the filtered performances into four equal ranges, which are then applied to the current particle performances to determine their objective states. An additional potential benefit of these dynamic thresholds is that as optimization progresses and performances typically improve, thresholds may shift toward lower values with narrowing intervals, potentially leading to better distinction between high-quality results compared to static thresholds. Decision states are subsequently determined using a *get_decision_space_state* method of the *Particle* class that utilizes the squared distances to the global best directly, eliminating square root calculations for computational efficiency.
Following the completion of the first batch iteration, the algorithm calculates rewards and updates the Q-table according to equations 2.14 and 2.7. During global search, each particle's next action is determined through the Q-table; otherwise, local search parameters are chosen and applied. Finally, particle velocities and positions are updated as per the previously described PSO implementation, while performances, states, and actions are preserved for the next iteration's reward calculation and Q-table updates.
This implementation aligns with Algorithm 1 from Liu et al. [32], though the operation sequence was slightly modified for practical implementation purposes. A visual representation of this process is provided through a flowchart with Figure 4.1.
As no values for the parameter selection for the learning rate α and discount factor γ for updating the Q-values per equation 2.7 are defined in Liu et al. [32], values of $\alpha = 0.1$ and $\gamma = 0.99$ were chosen [33]. Notably, α was chosen to be larger than typically is the case in order to promote a faster adaptation due to the online learning approach [34].

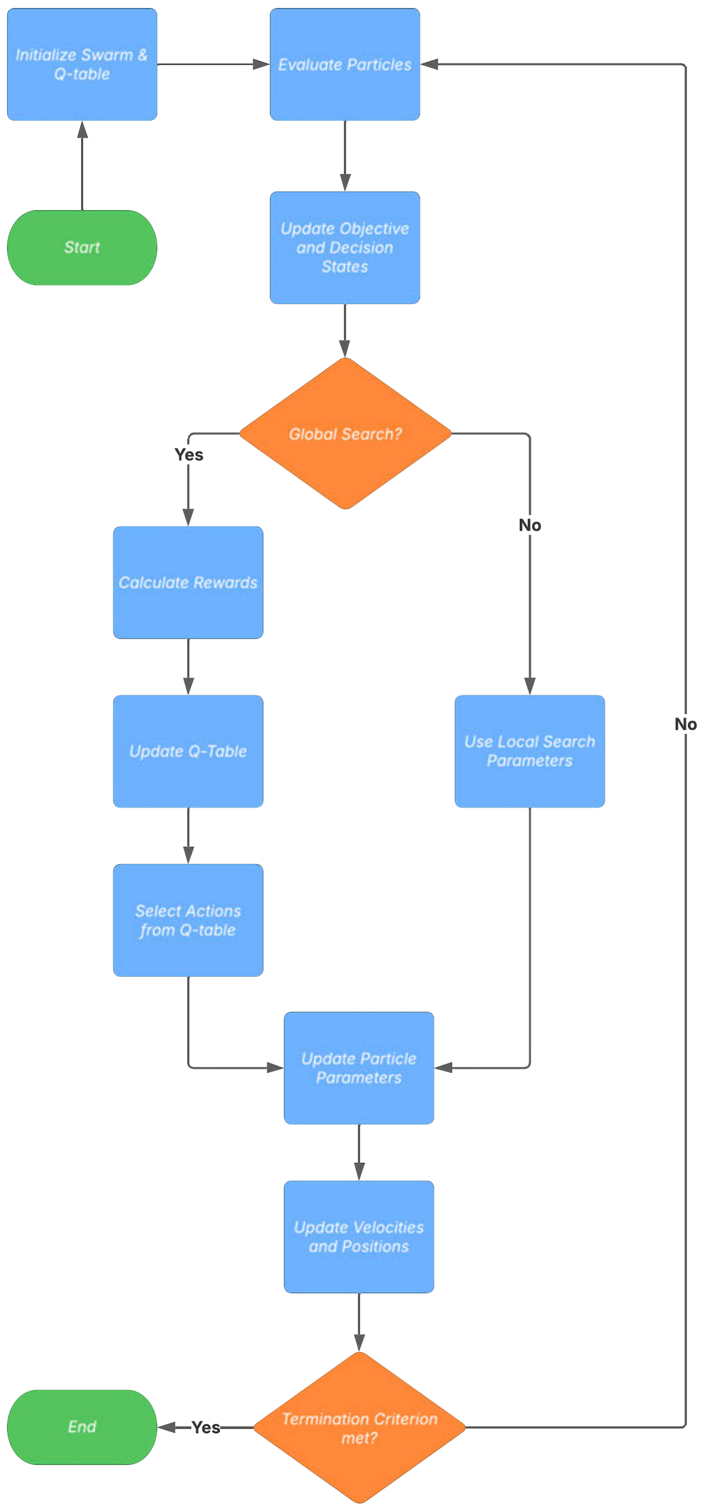

Initialize Swarm & Q-table
Evaluate Particles
Start
Update Objective and Decision States
Global Search?
Yes
No
Calculate Rewards
Update Q-Table
Use Local Search Parameters
No
Select Actions from Q-table
Update Particle Parameters
Update Velocities and Positions
End
Yes
Termination Criterion met?

Figure 4.1: Flowchart showing the QLPSO process

4.4 Stochastic Rounding

Since ways to deal with discrete values can tend to be quite complex as per the methods described in Wang et al. [50], a simple stochastic rounding approach was implemented using a steep sigmoid function

$$p_{sigmoid}(x,k) = \frac{1}{1 + e^{-k(x-\lfloor x \rfloor - 0.5)}} \tag{4.1}$$

that determines the probability of rounding up, with k representing the steepness factor with a chosen value of 20 and $\lfloor x \rfloor$ the rounded down value of x. Using a Bernoulli random variable $\mathcal{B}$ with this corresponding probability, a value will either be rounded up or down:

$$s_{int}(x,k) = \lfloor x \rfloor + \mathcal{B}(p_{sigmoid}(x,k)) \tag{4.2}$$

A visual representation is provided with Appendix Figure A.2.
The controlled randomness introduced with this rounding approach should enable the PSO-based algorithms to more easily escape local optima by providing gradient information, that would be lost with deterministic rounding. This stochastic rounding function was applied for BO and both PSO based methods for consistency, although more sophisticated ways of dealing with discrete values exist for these optimizers, as proposed in Garrido-Merchán and Hernández-Lobato [20], a publication by the same authors of the *Bayesian Optimization* framework implemented by Ray Tune, that deals with discrete parameters when using BO with surrogate modelling through Gaussian process [37]. The probabilistic rounding approach used here resembles their 'Basic' approach of deterministically rounding values to the nearest integer, with the drawback being, that the model must sample the same discrete values multiple times to construct an adequate surrogate model. Although the stochastic nature of this approach may introduce an additional disadvantage for BO by potentially skewing the surrogate model during rare rounding occurrences, e.g, a parameter with a value of 0.01 being rounded up to 1 instead of rounded down. If significant performance differences exist between the rounded values, this could introduce noise into the modeling process, which could potentially have substantial implications for convergence and should be considered when evaluating results.

4.5 Experiment Setup

This section details the complete experimental setup and the implementation of the algorithms used. For conducting experiments, setting files in JSON format were created for the SMARDCast and SVM on MNIST tasks, containing two parameter categories: fixed parameters and hyperparameters. Fixed parameters include the evaluation metric, optimization mode (minimum or maximum), and the number of epochs, which in the SVM optimization problem represents the number of multi-fidelity steps. For the SMARDCast model, additional model-specific runtime parameters were included, such as the number of output time steps, window split, offset, etc.; both settings can be found in the Appendix with Tables A.5 and A.4 displaying the hyperparameters and Tables A.3 and A.2 the fixed parameters. The hyperparameters section in the settings was used to create the search space required by Ray Tune and custom keywords were employed to describe different samplers. Most of these are straightforward, with equivalent samplers existing in the Ray Tune API, such as the uniform sampler. Two additional custom keywords were implemented: *pow2* and *combination*. The *pow2* sampler functions as a discrete uniform distribution, with the distinction that each sampled value represents their respective power of 2 - for example, sampling a 4 would result in $2^4 = 16$. The *combination* sampler takes a list of categorical values and assigns either true or false to each one, thereby generating a random combination of the original values, as the name suggests. This was used to handle the optional time feature inputs as described in section 3.1. For the test function, mode and evaluation metric remained static, and the hyperparameters consisted of D (where D represents the dimensionality of the function) continuous values uniformly sampled from their respective value ranges, which were implemented directly in the code.

Mapping these custom samplers to Ray Tune samplers occurs in a *create_param_space* function (or *create_param_space_bohb* as the BOHB equivalent using the ConfigSpace class as mentioned previously). This function considers whether the optimizer can handle discrete samplers, ensuring that BO, PSO, and QLPSO receive the continuous equivalents. Another distinction arises due to BO's inability to use Ray Tune's *loguniform* distribution; similar to the *pow2* sampler, it instead uses a uniform distribution with values representing powers of 10.

Regarding the implementation of trainables, the test functions do not need to handle these custom samplers and instead take the sampled values as function arguments directly and return values according to equations 3.1 and 3.2.

The SMARDCast trainable maps continuous parameters into discrete equivalents for BO,

PSO, and QLPSO using the stochastic rounding function described in equation 4.2, while also handling loguniform parameters when using the BO algorithm. The seed used for rounding is saved to the configuration dictionary, such that the real parameters used in the model can be reconstructed when evaluating experiment results. The hyperparameters are then used to instantiate the CNN-LSTM model using the framework provided by the author of Krüger et al. [29]. This framework was used as-is, with minor adjustments such as down-casting the SMARD dataset to float-32 and using mixed float-16 precision in the model to avoid memory issues and achieve faster training times. Additionally, the trainable implements a basic failsafe mechanism allowing retries if memory issues occur during training, halving the batch size up to three times before terminating. For proper integration of search algorithms and schedulers, a *CustomCallback* is used alongside the *EarlyStopping* callback - the former calls the Ray Tune *report* function after each completed epoch, while the latter terminates training if performance fails to improve for three consecutive epochs, as specified by the patience parameter in the settings file. Following successful training, the *evaluate* utility method included in the SMARDCast model is used to benchmark the trained model on both validation and test data and saves the performance results to a *.parquet* file with the according *trial_id* assigned by Ray Tune for reference.
In the SVM task, only two hyperparameters with loguniform distribution are used, meaning that for most algorithms, no further processing of the sampled values is needed. Exceptions include the BO algorithm, as in the SMARDCast task, as well as the BOHB searcher, since the *ConfigSpace* class only supports loguniform sampling with base 10, while the example in the supplementary document of Klein et al. [28] uses e as its base. The previously mentioned Nystroem approximation using a linear SVM instead of the radial kernel is realized through the scikit-learn package using the *Nystroem* and *LinearSVC* classes. The MNIST dataset is loaded through Keras with 60,000 images for training and 10,000 for testing and then normalized. The multi-fidelity approaches HB and BOHB are implemented through a loop, which increments the fidelity s each iteration as described in section 3.2 and creates the according subset by randomly sampling data points from the full dataset. The feature map is created using the Nystroem approximation and subsequently used to fit the linear SVM, which is then evaluated on the test data to report the test error. The subset size m scales through $m = 10\sqrt{n}$ with n being the number of samples, e.g., 60,000 for the full dataset or 15,000 with $s = 1/4$ in multi-fidelity approaches. This achieves the reduction from cubic to quadratic complexity, while a scaling value of 10 ensures an adequate number of data points in low-fidelity evaluations for sufficient coverage of all ten digit classes in the MNIST dataset.

The *run_ exp* function serves as the central orchestrator for the entire experiment pipeline. This function first establishes the result path using the experiment ID and algorithm name, then constructs the appropriate search space via *create_ param_ space* or the equivalent *create_ param_ space_ bohb* for BOHB configurations. Resource allocation is managed through Ray Tune's *tune.with_ resources* wrapper, determining the available hardware - for the SMARDCast task half a GPU per trial is allocated, while test functions and the SVM MNIST task each utilize three CPU cores per trial (the latter one is also restricted to 4 gigabytes of memory). In contrast to the SMARDCast task, these two also allow the BO to use 30 cores per trial, which is equivalent to the total number of cores for the other optimizers (there are 32 cores available in total on the hardware). Due to an implementation error, this special resource allocation for BO is not used for the SMARDCast task, such that the one trial running on BO only utilizes half of one GPU instead of both. The implications of this error should not impact the results in a substantial way, since the training time of the SMARDCast model in comparison to its loading time is quite short. The search algorithm and scheduler initialization adapts to the specified algorithm using the *alg* argument: by default a *BasicVariantGenerator* and a *FIFOScheduler* are initialized, while specialized algorithms overwrite these accordingly. After initialization of the search space and the Searcher/Scheduler logic, the optimization process is launched through Ray Tune's *tune.run* method with the according arguments. On completion, the function collects the reported results and saves them to a CSV file.

4.6 Practical Considerations

Using the batched *ConcurrencyLimiter* required for standard PSO presents several practical challenges depending on the specific optimization task. When evaluation time varies significantly across parameter configurations, PSO efficiency may be negatively affected, as the algorithm must wait until the longest-running trials are completed before suggesting new configurations. Such variations can occur when parameters influence training dataset size or model complexity, directly affecting computational requirements. In the case of the SMARDCast model, execution time remains relatively consistent across different network architectures. However, the SVM implementation on MNIST described in section 3.2 displays substantial variance in computation time based on the chosen hyperparameters, as training durations in prior testing ranged from 10 seconds to several minutes, which is likely due to the parameter C being too large as noted in the *scikit-learn's* documentation [42]. This temporal disparity is particularly problematic for the PSO and subsequently QLPSO implementation. Potential solutions may include an asynchronous PSO searcher that does not require batch completion, or implementing a custom *Scheduler* for early termination of trials, although these approaches were not implemented in this thesis.

A secondary consideration when utilizing the *ConcurrencyLimiter* involves the coordination between population size and resource allocation. Ideally, the population size N should be divisible by the maximum number of concurrent trials permitted by the user's allocation of computational resources. For example, with a population size of $N = 20$ and capacity for six concurrent trials (e.g. three GPUs with two parallel trials each), three batch completions would result in 18 finished trials, when assuming similar evaluation times. The remaining two trials would then only be able to use one third of the available resources, leading to suboptimal resource utilization during this phase.

5 Results and Discussion

This chapter presents the experimental findings from applying the previously described HPO methods to the test cases outlined in Chapter 3. Each algorithm's performance characteristics and optimization efficiency is examined across optimization problems, with special attention given to the behavior of the QLPSO approach, which is explored in detail on the mathematical test functions.

5.1 SMARDCast

As mentioned prior, the grid search used in the SMARDCast paper resulted in the best model achieving a MAPE of 2.99 % on the test dataset. This result was improved upon by roughly 0.3 percentage points by a model found using the standard PSO approach, with a final performance of 2.62 % MAPE on the test data. While this approximately 12 % improvement in accuracy is significant, a total of roughly 75,000 models were fully trained during this experiment (approximately 30 times as many configurations as in Krüger et al. [29]). Considering the influence of random initialization on the performance of the model, as characterized in Appendix Table A.1, this improvement may partially stem from statistical factors rather than just from the choice of hyperparameters.

Figure 5.1 illustrates the average performance across runs of each optimizer over time, measured by their respective cumulative minimum validation loss, i.e., the best performance of a run up to a given point. Most optimizers reach a performance plateau relatively quickly, with improvements slowing down rapidly. Notably, the BOHB algorithm requires significantly more time to reach this performance level. Most likely this is due to the bracketing and resource allocation process, i.e., it takes a while to reach high-fidelity evaluations with typically better performance, as pausing and reloading models during a low-fidelity phase takes proportionally more time. A substantial distinction between optimizers cannot be observed, as the deviation corridors are quite large and often overlap, again suggesting that randomness in both the SMARDCast model and the initialization of optimizers in each run may play a substantial role in final performance.

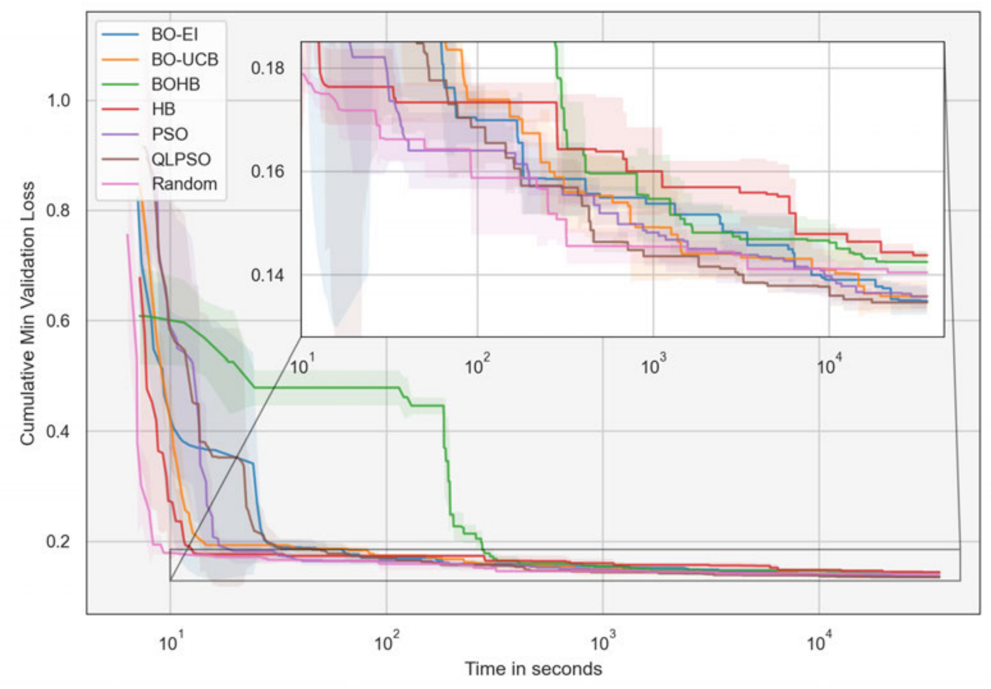


Figure 5.1: SMARDCast: Average performance over time with zoomed inlay, filled areas represent the standard deviation across runs

Figure 5.2 illustrates the average rank across runs of each optimizer at a given time. Ranks calculated by sorting all optimizers across all three runs by their best performance up until a given timestamp (as per Figure 5.1), resulting in each optimizer being assigned three rankings ranging from 1 to 21 in this case. These three rankings are averaged and normalized to the range $[1, 7]$ for each timestamp and a rolling average is applied for better visibility. The same ranking plot using evaluated samples instead of timestamps can be found in Appendix Figure A.3, displaying a better sample efficiency for the used BO algorithms, aligning with theoretical assumptions. The QLPSO algorithm is consistently placed first after approximately ten minutes, while other optimizers display high variance in their rankings. An unexpected result is the poor performance of both multi-fidelity approaches HB and BOHB, as these are consistently placed in the last two ranks, performing even worse than the baseline RS. There are two possible explanations for this contradiction of both empirically and theoretically expected results.

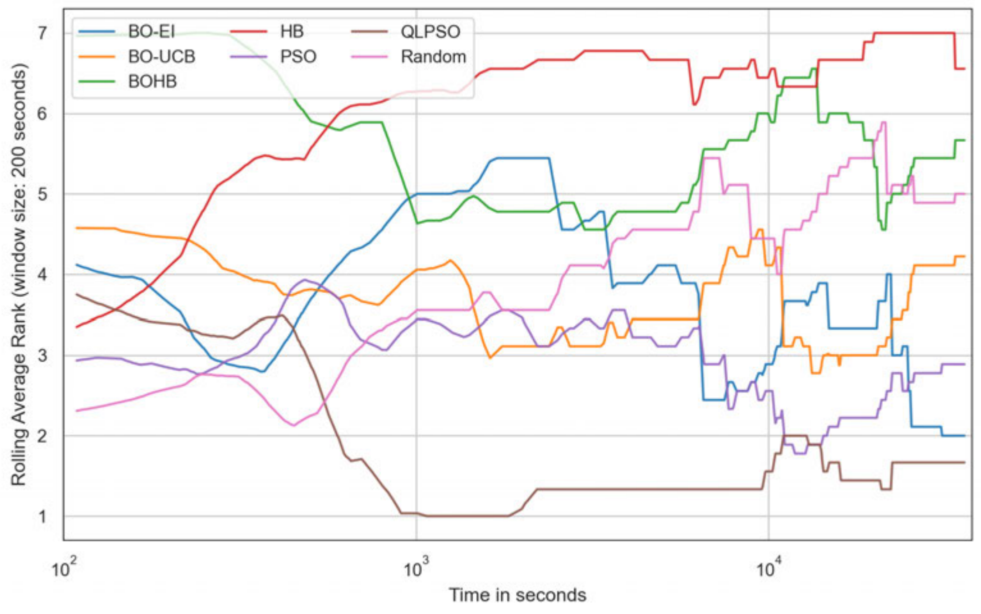


Figure 5.2: SMARDCast: Average rank of optimizers over time (lower is better)

First, the number of suggested configurations is much higher than the number of fully trained models as per the SH algorithm both are implementing, which results in both optimizers evaluating far fewer configurations with maximum fidelity and thus drawing from a smaller pool of fully trained models. However this is naturally the case with both HB and BOHB, such that this should only negatively affect their performance with severe noise in low-fidelity evaluations. Excessive noise may disrupt the correlation between low-fidelity and high-fidelity evaluations, causing the selection process to potentially discard good configurations early. Under such conditions, SH may become counterproductive, as it reduces the number of fully trained models without effectively identifying promising candidates. This flawed early filtering may cause multi-fidelity approaches to potentially perform worse than the baseline RS. This explanation is supported empirically by the number of evaluations by algorithm in Table 5.1; while RS evaluated roughly 7,000 fully trained models on average, BOHB and HB only evaluated 2,241 and 869 on the maximum fidelity respectively. A blog post by Falkner et al. [19] confirms this detrimental influence of misleading low-fidelity evaluation on overall performance.

Table 5.1: Number of evaluations SMARDCast

Algorithm	**Average**	**Run 1**	**Run 2**	**Run 3**
HB (all)	10,011	10,032	10,009	9,994
RS	7,059	7,077	7,040	7,060
QLPSO	5,903	5,816	6,004	5,889
PSO	5,877	5,852	5,841	5,939
BOHB (all)	5,536	5,518	5,612	5,480
BOHB (max-fidelity)	2,241	2,180	2,229	2,316
BO-UCB	1,645	1,692	1,567	1,677
BO-EI	1,461	1,547	1,434	1,403
HB (max-fidelity)	869	856	867	885

Another practical consideration is the previously mentioned low training time of the SMARDCast model. The comparatively longer loading times can significantly impact performance, particularly during early optimization phases when the optimizer must frequently pause and reload models only to train them for a brief amount of time. The issue of computational benefit being lost due to a proportionally large overhead is recognized in Li et al. [30]. Figure 5.3 illustrates this through a descending ECDF (empirical cumulative distribution function) plot, where the proportion indicates how often GPU usage equals or exceeds the corresponding x-value.

Notably, the HB algorithm appears to struggle even more with GPU efficiency, despite theoretically having less downtime than BOHB as it lacks an underlying Bayesian model for new sample suggestions, which represents additional computational overhead. This discrepancy likely stems from implementation differences, as BOHB's optimizer is used directly through the authors' provided package with HpBandster, unlike HB, which Ray Tune implemented based on the original publication. Since the HB paper does not specify certain implementation details, Ray Tune actually recommends an asynchronous alternative *ASHA* to the original HB algorithm, though this was not used to maintain better comparability with BOHB [48].

As to be expected, the BO algorithms exhibit the worst GPU efficiency for two reasons: only half of GPU 0 was allocated per trial as mentioned in section 4.5, and their internal modeling logic creates substantially longer downtime, especially in later phases with $O(n^3)$ complexity. PSO, QLPSO and RS demonstrate the best GPU efficiency, with RS showing slightly more consistency for higher average usage. The somewhat more compu-

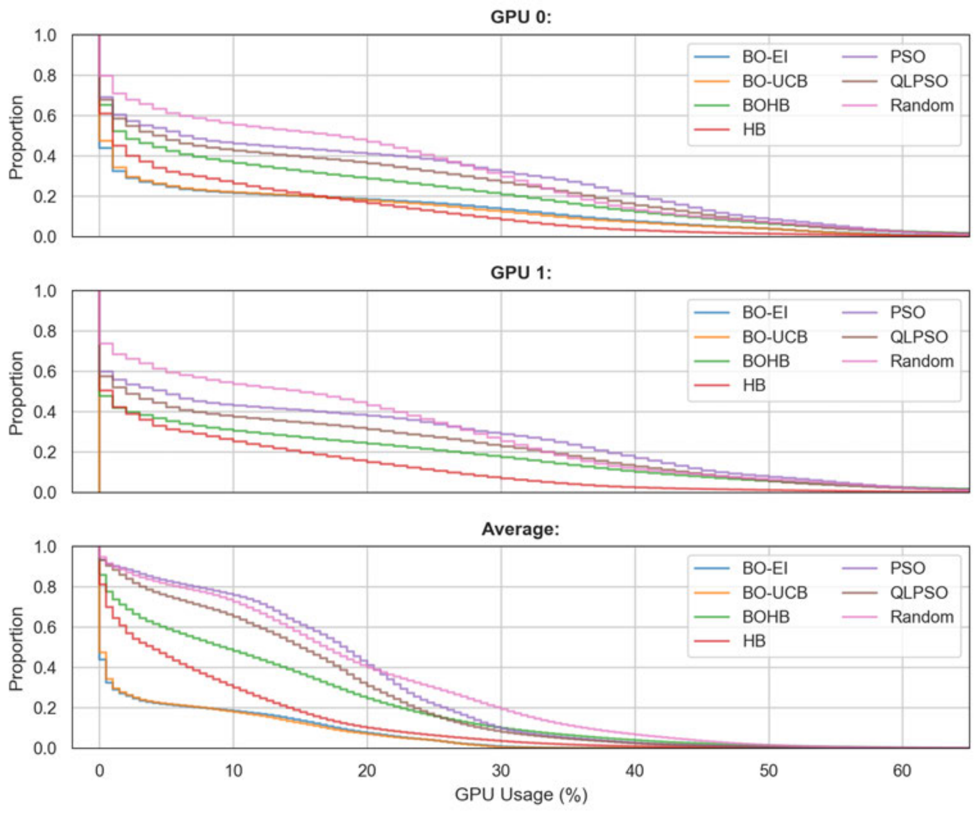


Figure 5.3: GPU usage of optimizers

tationally expensive QLPSO mirrors PSO's GPU usage while experiencing slightly longer downtimes, resulting in marginally lower GPU efficiency.

Upon closer inspection, RS's decent GPU usage may partly be explained by its models' average training duration. Figure 5.4 presents distributions of both epoch counts and trial completion times in form of an ECDF plot, with HB and BOHB expressing their underlying SH subroutine as step functions in the epochs plot. While Bayesian- and PSO-based optimizers show similar epoch counts, models configured by RS disproportionately train to the 50-epoch limit, leading to longer trial completion times and consequently less GPU downtime. This pattern can be explained by examining learning rate parameter sampling, which directly affects model training duration. Lower learning rates lead to slower training processes, necessitating more epochs to be completed before overfitting occurs and triggers the *EarlyStopping* callback, while the opposite holds true for higher learning rates.

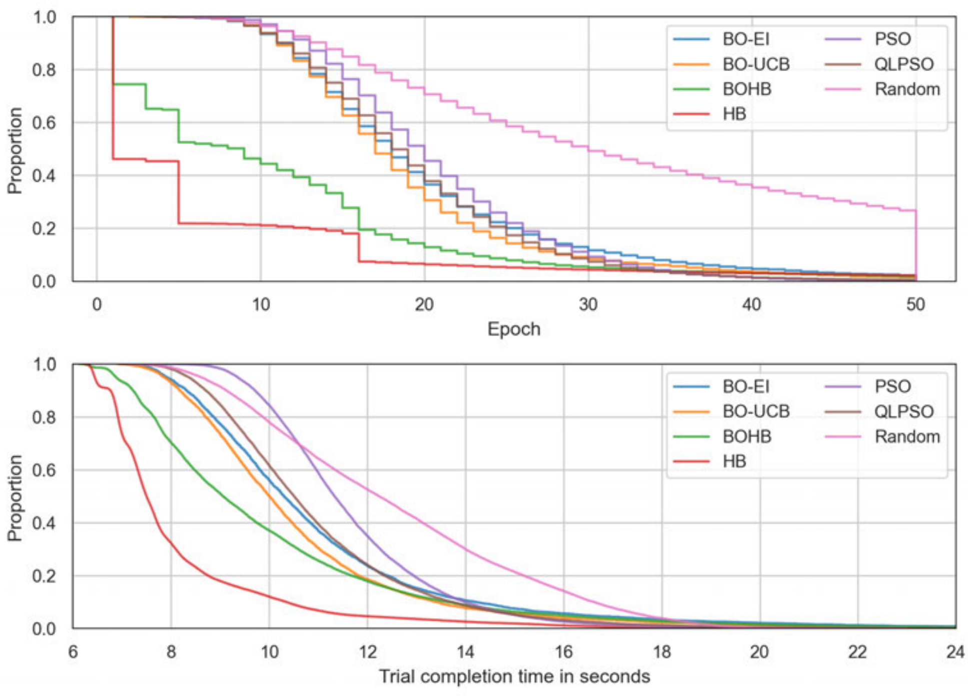


Figure 5.4: Epochs and time per trial for all optimizers

Figure 5.5 displays the learning rate parameter usage by optimizer in a logarithmic ECDF plot, providing insight into the algorithms' optimization processes. As both HB and RS simply draw values from a random loguniform distribution, their sampling is represented by a diagonal line. Due to the implementation of SH by the HB algorithm, only a subset of configurations are not discarded prematurely, resulting in only RS fully training the majority of models with low learning rates. This explains the previously observed high proportion of models trained to the maximum number of epochs. The UCB variant of BO predominantly trains models using the highest possible learning rate, exhibiting strong exploitative behavior. The remaining algorithms display varying curve characteristics, yet all demonstrate a preference for higher learning rates, with approximately 80% of sampled learning rates being greater than or equal to 10^{-3}. This aligns with empirical findings, as the best-performing models typically fall within the range of $[4 \cdot 10^{-4}, 10^{-3}]$ (see Appendix Figure A.4).

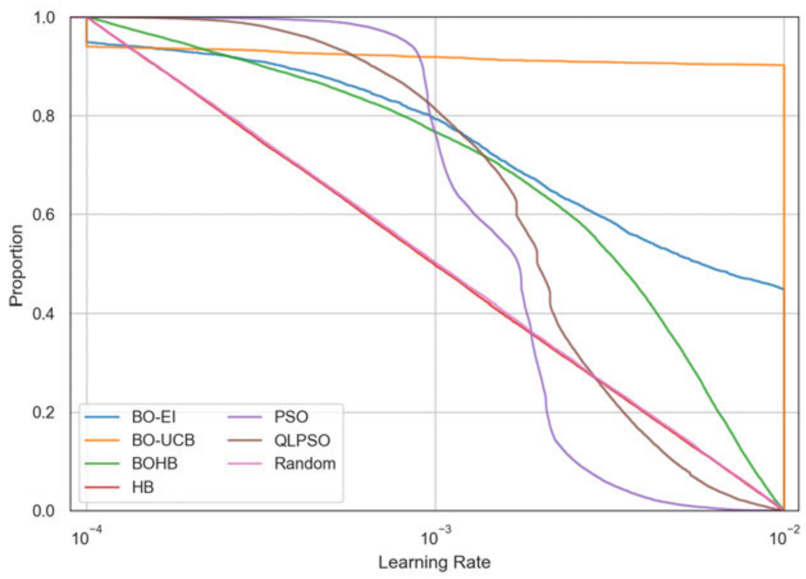


Figure 5.5: Learning rate sampling by optimizer

5.2 SVM on MNIST

The reproduction of the SVM multi-fidelity optimization problem from Klein et al. [28] with an additional Nystroem approximation was done to verify the strong performance of BOHB showcased in Falkner et al. [18]. Figure 5.6 shows the performances of all optimizers in form of the test error. While HB and BOHB show slightly earlier first evaluations, this difference is almost negligible compared to the results displayed in the BOHB publication, as this is likely due to the parameter dependent time variance of the SVM model described in section 4.6 (evaluation time differences are empirically shown in Appendix Figure A.5). Since all optimizers are sampling randomly at first and RS's evaluations start considerably later than those the others, which further points to the significance of this time variance. Additionally, while the Nystroem approximation allowed for this experiment to be run considering the time restrains, the performance implications for multi-fidelity approaches are apparent. This is due the choice of $m = 10\sqrt{n}$ for the Nystroem approximation, the ratio of m/n becomes significantly smaller for large values of n than for smaller subsets as used by HB and BOHB. This results in less significant speedups for low-fidelity evaluations and diminishes the possible performance gains provided by their multi-fidelity approach.

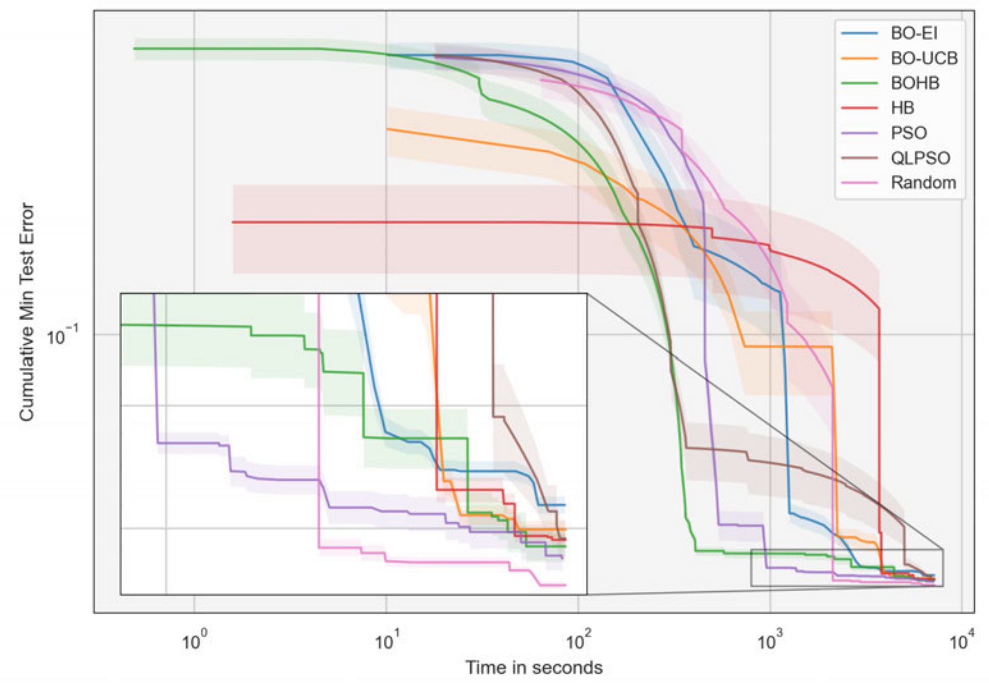


Figure 5.6: SVM on MNIST: Average performance over time with zoomed inlay, filled areas represent the standard deviation across runs (scaled with 1/5 for visibility)

Due to better random configurations, HB still displays the strongest early performance, which again shows the problem of statistical significance, as optimizers were run 30 times in Klein et al. [28] and 512 times (50 for BO) in Falkner et al. [18] respectively, instead of five times in this experiment. This effect combined with the previously mentioned time variance result in strong deviations across runs, such that the standard deviation in Figure 5.6 was scaled down by 1/5. This strong variance is further exemplified by RS quite consistently outperforming the other HPO algorithms, which directly contradicts the empirical results [28, 18].

The practical issues of PSO in respect to high time variance described in section 4.6 can be seen in Table 5.2, as neither PSO based algorithm completed more than 100 evaluations on average. With a population size of 20 the underlying PSO update logic was oftentimes only called four or five times. Despite this, the standard PSO algorithm performs considerably well displaying a strong anytime performance excluding RS, while it seems to have particularly fatal implications for the QLPSO searcher. As it uses online learning through the Q-table, it may not fully adapt to the optimization problem in such

a short amount of time, such the selection of swarm parameters may occur with a high amount of randomness. This effect can be observed regarding QLPSO, as it displays a long phase of stagnation, only converging to competitive results during the final phase. Presumably this is due to its local search effectively promoting convergence near the real optimum, as the objective function is unimodal and convex.
Notably, although the topology, low dimensionality and continuous parameters of this objective function should align well with the modelling approach of the BO methods, both display comparatively poor performances. A likley explanation for this may yet again be the high amount of variance combined with an insufficient number of runs, as results in both Klein et al. [28] and Falkner et al. [18] show an on average superior performance to RS.
Overall the findings of this experiment may not be conclusive, as similar to the SMARD-Cast experiment the optimization problem suffers from high underlying variance and thus may not reach the threshold of statistical significance.

Table 5.2: Number of evaluations SVM on MNIST

Algorithm	**Average**	**Std**	**Relative Std**
HB (all)	689.2	80.9	11.7 %
BOHB (all)	462.0	138.4	29.9 %
Random	186.8	21.2	11.3 %
HB (max-fidelity)	114.6	13.4	11.7 %
QLPSO	96.0	31.4	32.7 %
PSO	77.2	26.3	34.0 %
BOHB (max-fidelity)	55.0	19.1	34.8 %
BO-EI	29.8	8.0	27.0 %
BO-UCB	26.6	6.8	25.4 %

5.3 Test Functions

This section presents the results for the two test functions: Rastrigin and Styblinski. As mentioned prior, this experiment excludes the multi-fidelity optimizers HB and BOHB, with it primarily serving as a controlled analysis of the QLPSO algorithm.

5.3.1 Rastrigin

The Rastrigin function represents a highly multi-modal function with a large number of local optima (see Figure 3.2) and a global optimum with a value of 0 at the origin point. Figure 5.7 illustrates the performances over time for the 10-dimensional Rastrigin function. The BO using the EI acquisition function demonstrates strong anytime performance, with only RS outperforming it during early phases due to faster sampling. BO-UCB exhibits decent performance but struggles with final convergence. Notably, the QLPSO approach slightly outperforms standard PSO, while both achieve similar final performances as BO-UCB.

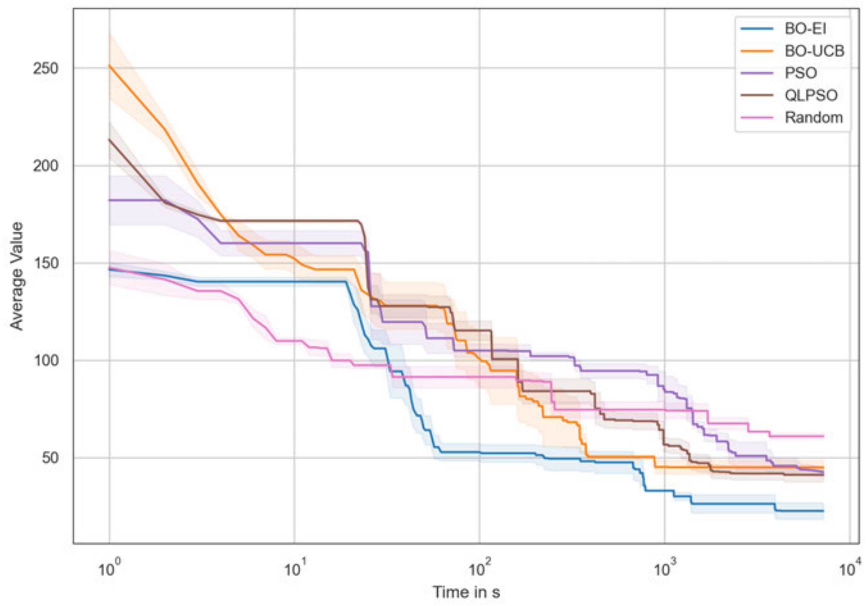


Figure 5.7: Rastrigin-10D: Average performance over time, filled areas represent the standard deviation across runs (scaled with 1/2 for visibility)

In the 50-dimensional case, illustrated in Figure 5.8, the limitations of RS become evident, as pure exploration leads to substantially worse final results compared to other optimizers. The stronger early performance of BO-EI observed in the 10-dimensional case disappears here, as it only begins outperforming other algorithms after approximately 10^3 seconds, showing a short, highly exploitative phase resulting in a performance jump, followed by severe stagnation. Similar behavior appears in BO-UCB at the same time, though it continues improving after the jump, ultimately outperforming its EI counterpart. Unlike the 10-dimensional case, QLPSO is substantially outperformed by standard PSO, which emerges as the best optimizer with a significant lead. This comparatively stronger performance by the PSO-based algorithms aligns with literature, as PSO typically excels at optimizing high-dimensional objective functions [52].

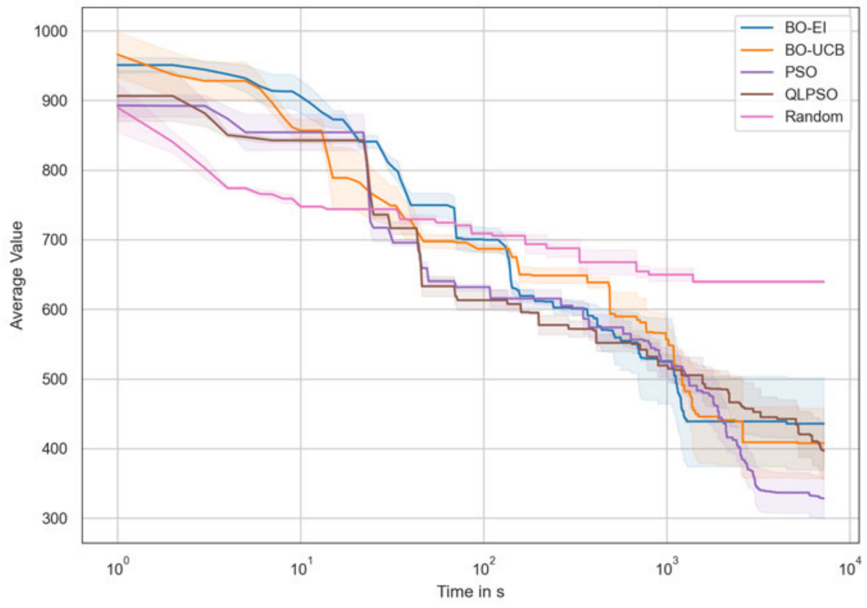


Figure 5.8: Rastrigin-50D: Average performance over time, filled areas represent the standard deviation across runs (scaled with 1/2 for visibility)

5.3.2 Styblinski

The Styblinski function presents another multi-modal function, distinguished by local optima spread toward the edges and a more convex topology compared to Rastrigin. The global minimum occurs where all variables equal approximately -2.9035, with a value of -391.66 for the 10D and -1,958.30 for the 50D case respectively.
Figure 5.9 displays the performance of optimizers in the 10-dimensional case. Performances resemble those of the Rastrigin-10D experiment, with RS showing strong early performance, but producing worse final results. QLPSO outperforms standard PSO again, suggesting an overall improvement in lower dimensionalities. Both Bayesian approaches demonstrate the strongest performance among all optimizers, with BO-UCB outperforming BO-EI. Notably, this experiment was required to be rerun for the BO algorithms, as these terminated their runs early after only around 2,000 seconds. This resulted from optimizers essentially reaching an impasse where acquisition function accuracy proved insufficient to differentiate between similar configurations, interpreting them as duplicates and failing to generate new suggestions. The parameters *patience* and *repeat_float_precision* of the *BayesOptSearch* class were set to a value of 10 each for this experiment specifically, to prevent early termination.

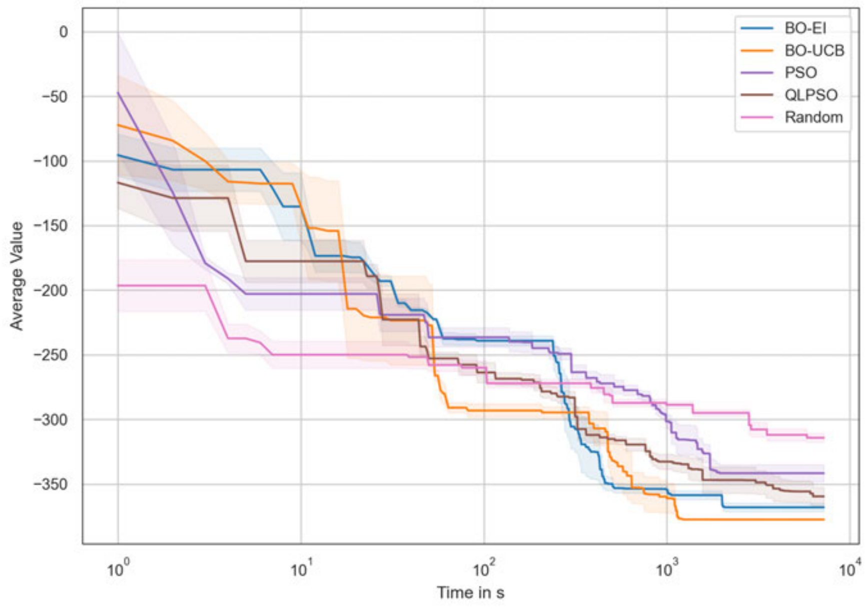


Figure 5.9: Styblinski-10D: Average performance over time, filled areas represent the standard deviation across runs (scaled with 1/2 for visibility)

Similar to the Rastrigin function, standard PSO significantly outperforms QLPSO in the higher-dimensional problem as illustrated in Figure 5.10. This indicates some potential limitations in the QLPSO's ability to cope with high dimensionality, which is further examined and discussed in section 5.4. RS, while remaining the weakest algorithm, displays more competitive results than in Rastrigin-50D. This may partly stem from the overall weaker performance of PSO-based approaches, as indicated by their strong stagnation during the later phases. A possible explanation may be, that the more dispersed local optima of Styblinski present greater challenges for swarm optimization than the clustered optima of Rastrigin, which may allow for better incremental improvements during the exploitation phase. Notably, both BO methods outperform other algorithms, with BO-EI exhibiting better anytime performance than its UCB counterpart. Similar to Rastrigin, both BO algorithms display significant performance jumps toward the end, indicating strong exploitation. Unlike Rastrigin, these jumps occur during different optimization phases and aren't followed by performance stagnation; instead, both continue improving until the conclusion, suggesting potential for further enhancements given longer optimization time. The superior performance of the BO methods likely stems from the Styblinski function's topology, which features smoother contours and a more convex shape that aligns well with Gaussian process modeling capabilities.

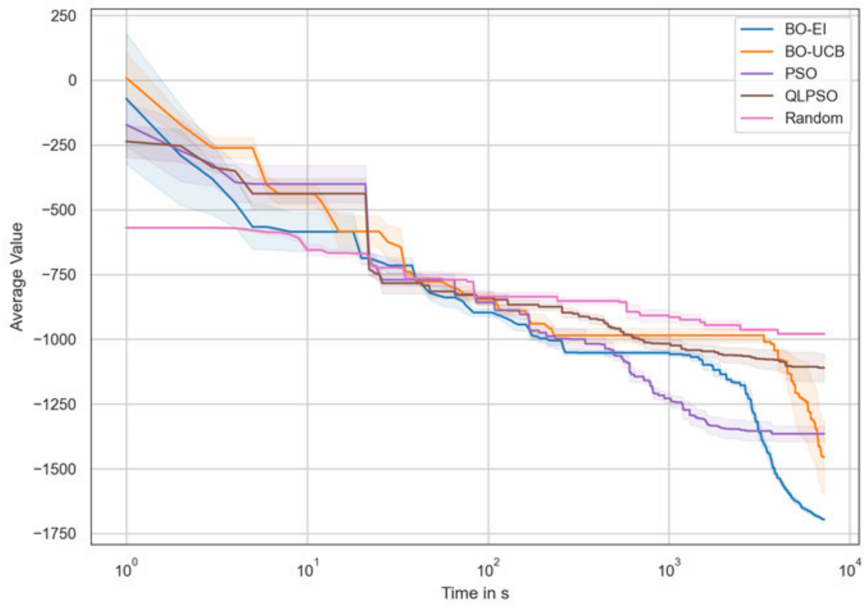


Figure 5.10: Styblinski-50D: Average performance over time, filled areas represent the standard deviation across runs (scaled with 1/2 for visibility)

5.4 QLPSO Behavior

This section examines the optimization behavior and inner workings of the QLPSO approach, with particular attention to the characteristics exhibited by the Q-learning component. As mention prior, the analysis focuses on the test functions, as they allow consideration of both objective function shapes and dimensionalities without evaluation time inconsistencies. Since QLPSO employs an online learning process, the changes and diversity of states and actions over time are of particular interest.

Figure 5.11 illustrates these general behavioral characteristics, with rows representing the respective states and actions, and columns distinguishing between the two test functions. Each combination displays the rolling average (window size of 200 seconds) across runs with a 95% confidence interval ($\mu \pm 1.96\sigma$) for both dimensionalities.

Actions largely remain constant across both test functions and dimensionalities, primarily consisting of high exploration actions (0 and 1) during the global search phase. This pattern was unexpected, as the reward function implemented explicitly promotes a change from exploration to exploitation based on the optimization progress. A possible explanation for this discrepancy may be the choice of α and γ used in the Q-value update function 2.7, but needs be further examined to merit a satisfying conclusion. Only the 10-dimensional Styblinski function shows a slight upward trend, though it's almost negligible. Generally, the QLPSO searcher tends slightly more toward convergent actions with the Rastrigin function compared to Styblinski, while higher dimensionality contributes to more exploratory behavior in both cases. The final spike to action 4 represents the local search initiated after 90 % of runtime.

Interestingly, dimensionality appears to have the most significant impact on decision states, with particles predominantly occupying the 'farther' and 'farthest' categories (expressed with values of 2 and 3) in the 50-dimensional examples, while in lower dimensionality they often inhabit the lower threshold areas. According to the categories proposed by Liu et al. [32], this would suggest greater distances from the global optimum, but this approach may be misleading, as with increasing dimensionality, the 'farthest' region becomes disproportionately large. This effect is illustrated in the extreme example for 2-dimensional space in Figure 5.12, where the global optimum is at the origin point. Although equally sized regions for decision states may not have been the original intention, the strong influence of dimensionality suggests a need for a mechanism to handle the potential skewing of decision states, as particles become increasingly more likely to fall in the 'farthest' regions as dimensionality increases. Additionally, as shown in Figure 2.5, when the global optimum falls toward the center of the hyperparameter space, the

'farthest' region may become completely obsolete, further reducing granularity of the decision state by limiting the Q-table to only use the remaining three states. This may also explain the Styblinski function seemingly promoting more distant decision states than Rastrigin, as the latter features an inherently more centered topology than Styblinski, where local and global optima are distributed more toward the corners of the space.
Overall, the downward trends in decision states display the expected transitioning from exploration to exploitation more pronounced than the chosen actions. Although an interesting observation to be made is the severe stagnation of the decision state occurring in the 50-dimensional Styblinski experiment, as the majority of particles seem to consistently be stuck in the farthest two regions, without converging to the global best. A possible explanation may be the similar stagnation of explorative actions in this example, which are characterized by higher cognitive and lower social parameters. Considering the topology of this function, particles seem to mostly keep exploring in proximity to their respective local optima and are unable to escape, as the attraction toward their personal best exceeds that toward the global best. The fact that local optima occupy large regions in the space may reinforce this stagnation effect, as particles may not be able to approach the global best through incremental improvements, like in the Rastrigin function. The final convergence effect visible across experiments simply reflects the local search parameters, as particles only consider the current global best for velocity updates and thus converge to that point.

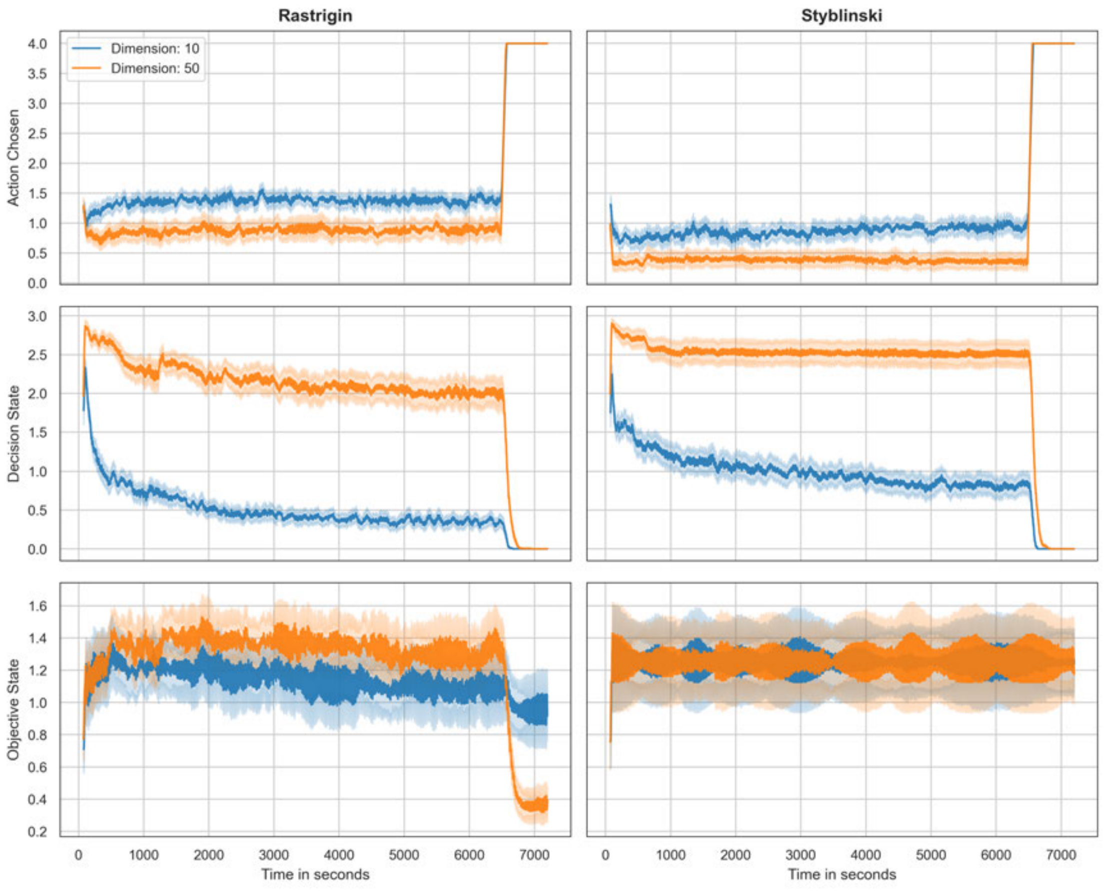


Figure 5.11: Ensemble plot for QLPSO behavior over time

Finally, the objective states illustrated in Figure 5.11 display large characteristical differences between the two test functions, while exhibiting high variance in all cases. The Rastrigin function promotes a slight downward trend with both dimensionalities showing a jump in convergence during the local search. This final convergence behavior is considerably more pronounced in the 50-dimensional case than in the 10-dimensional one. This pattern is not observed for the Styblinski function, where the objective state on average remains relatively constant, displaying significantly higher overall variance with some oscillatory features. The discrepancy may be due to the different topologies yet again, as the global best in the Rastrigin function is likely near the center, such that local search would lead to exploitation close to the actual global optimum or a close local optimum. Although the convergence of the decision states during local search raises another issue regarding the objective states in the Styblinski experiment, as convergence

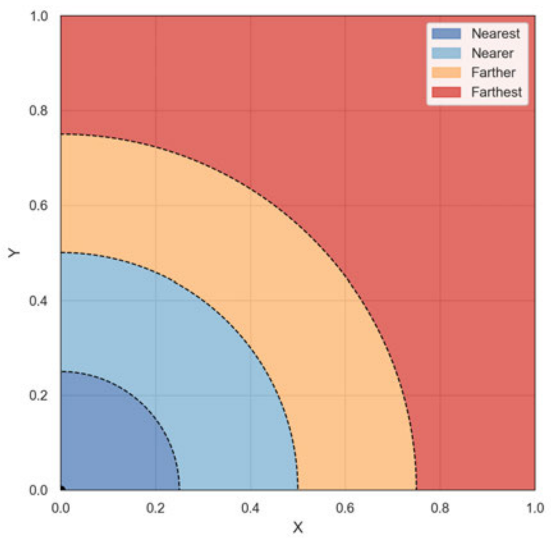


Figure 5.12: Decision state region size illustration

to the region of the global best should in turn lead to results similar to that of the global best and thus objective states of 0 and 1, especially with the more convex Styblinski function. A satisfying explanation remains to be found, although it may be related to the dynamic objective state thresholds used, as the value range of the Styblinski function is quite large, such that the used outlier handling approach might unintentionally distort objective state representation for this topology. States and actions are also visualized using histograms in Appendix Figure A.6, which reveal a consistently strange characteristic regarding objective states in the Styblinski function, irrespective of dimensionality. The vast majority of times only the edge categories are represented, further suggesting a potential issue with the dynamic thresholding approach used.

6 Conclusion

This chapter provides a brief summary of the work performed in this thesis and draws conclusions for the experimental results as a whole. Additionally, some future work ideas are proposed based on the findings of this thesis.

6.1 Thesis Summary

Regarding the in section 1.2 outlined objectives, this thesis largely achieved the intended goals. A comprehensive overview of common HPO techniques and their underlying mechanisms is provided in section 2.2. Using the Ray Tune framework, custom implementations of both the standard PSO algorithm and a QLPSO approach are developed and benchmarked against several other established methods across a variety of optimization tasks.

The results highlight the importance of taking various practical considerations into account when comparing HPO algorithms. All optimization methods exhibit individual strengths and weaknesses, making it crucial to take an array of factors into consideration such as variance, evaluation cost and resource efficiency. This is especially observable for both the SMARDCast model and the SVM on MNIST task, as high variance complicates drawing concrete conclusions, although metrics like GPU efficiency still reveal some underlying characteristics. Notably, the multi-fidelity approaches HB and BOHB especially struggle to compete due to these issues affecting both efficiency and convergence behavior.

Using optimization test functions, a systematic examination of optimizer behavior is provided. Different dimensionality and topology of the objective function impact performances in alignment with theoretical expectations. The PSO-based methods display better comparative performances in high dimensional space, while the BO methods tend to perform better in lower dimensionality. The more centered arrangement of local and global optima in the Rastrigin function seemingly allows for better convergence of the

PSO algorithms, while struggling to escape large local optima regions present in the Styblinski topology. Both BO variations are able to better model the smooth and convex shape of the Styblinski function.

The QLPSO approach shows overall competitive results compared to other HPO techniques and a superior performance to the standard PSO algorithm in some scenarios. Despite this, a strong influence of both dimensionality and topology is revealed when evaluating the test functions. The main issue being the inability to cope with high dimensionality, as the decision space state appears to not scale adequately, resulting in significant performance loss compared to using standard PSO. Actions also seem stagnant in contrast to the intended shift from exploration to exploitation modelled with the reward function. This may be due to the selection of α and γ, possibly suggesting that manual parameter selection is still required if results prove too sensitive to these values. Finally, the objective space state is strongly influence by the objective functions' topology. In the case of the Styblinski function, this state representation does not seem to provide meaningful information, regardless of dimensionality. Although this may stem from the custom implemented outlier handling using the MAD instead of the pre-determined thresholds.

Overall, the most substantial limitation encountered is the statistical significance of the results. The high run-to-run variance in combination with a limited time budget presents a material challenge for deriving a conclusive performance comparison.

6.2 Future Work

Based on the in this thesis outlined limitations of the QLPSO approach, there are several directions for future research:

Reward Function Design

Currently, the reward function considers actions, optimization progress and performance change, but does not incorporate state information, as is typically the case [39]. A more sophisticated approach may include state transitions and/or use the combination of both states to provide meaningful insight. For example, a decision space state of 3 ('farthest') combined with an objective space state of 0 ('smallest') may represent an interesting area for exploration. In this way, the additional reward signal may provide better dynamic guidance for particle behavior rather than the explicit progress based exploration-exploitation transitioning.

Performance Assessment
Step-by-step performance differences can be misleading, as traversing temporarily worse regions might be necessary to reach global optima. A sequential replay buffer storing performance changes over a longer time horizon may be used to bridge this gap, promoting more long-term planning instead of pursuing short-term improvements. Additionally, leveraging population-based improvement metrics instead of individual performance changes may prove beneficial.

State Representation
The decision space state not scaling adequately with higher dimensionality poses a critical limitation and significantly impacts performance. Currently, particles are assigned to the 'farthest' region in a disproportionate manner, such that the inclusion of a dimensionality-aware scaling factor for distance calculation should be examined. Similarly, the threshold values of the objective space state are defined prior to the optimization process in the original QLPSO implementation, which is not applicable to 'black box' objective functions. The in this thesis implemented MAD outlier handling displays shortcoming with the Styblinski topology, such that the development of alternative dynamic thresholding methods presents another research direction.

Evaluation of Parameter Influence
As previously mentioned, the initialization of parameters α and γ is not explicitly stated in the publication. The sensitivity of results due to the choice of these parameters is not quantified, as no extensive testing was done. For further examination of the QLPSO algorithm, the influence of parameter initialization should be characterized across dimensionality, topology and optimization budget.

Learning Approach
Another topic for future work could be exploring the usage of a DQN instead of a Q-table, which would allow for a continuous state space. This may provide a higher resolution of the optimization environment and represent particle states more accurately. Additionally, adjusting the three swarm parameters individually rather than using pre-defined combinations would allow for more precise control and minimize human bias. Using a DQN may also enable generalization and utilize knowledge transfer across optimization problems instead of the instance-specific approach.

Bibliography

[1] Nikita Adarsh. The inner workings of spotify's ai-powered music recommendations: How spotify shapes your playlist, 2023. URL https://medium.com/beyond-the-build/the-inner-workings-of-spotifys-ai-powered-music-recommendations-how-spotify-shapes-your-playlist-a10a9148ee8d.

[2] Arga Adyatama. Tutorial for bayesian optimization in r. https://rpubs.com/Argaadya/bayesian-optimization, 2019. Accessed: 2025-04-22.

[3] Musaed Alhussein and Syed Irtaza Haider. Improved Particle Swarm Optimization Based on Velocity Clamping and Particle Penalization. In *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, pages 61–64, Kota Kinabalu, Malaysia, December 2015. IEEE. doi: 10.1109/aims.2015.20. URL http://ieeexplore.ieee.org/document/7604552/.

[4] Amazon Web Services. What is deep learning? - deep learning explained - aws, 2025. URL https://aws.amazon.com/what-is/deep-learning/. Accessed: April 24, 2025.

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. ISSN 1532-4435.

[6] Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, Difan Deng, and Marius Lindauer. Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges, 2021. URL https://arxiv.org/abs/2107.05847. Version Number: 3.

[7] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning, 2010. URL https://arxiv.org/abs/1012.2599. Version Number: 1.

[8] Jason Brownlee. Supervised and unsupervised machine learning algorithms. Machine Learning Mastery, 2016. Available at: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/.

[9] Bundesnetzagentur. Smard: Electricity market data for germany. https://www.smard.de/en, 2025. Online platform by German Federal Network Agency (Bundesnetzagentur).

[10] Paolo Cazzaniga, Marco S. Nobile, and Daniela Besozzi. The impact of particles initialization in PSO: Parameter estimation as a case in point. In *2015 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 1–8, Niagara Falls, ON, Canada, August 2015. IEEE. doi: 10.1109/cibcb.2015.7300288. URL http://ieeexplore.ieee.org/document/7300288/.

[11] Michael Copeland. What's the difference between artificial intelligence, machine learning, and deep learning?, 2016. URL https://blogs.nvidia.com/blog/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/.

[12] DataCamp. Machine learning models explained. https://www.datacamp.com/blog/machine-learning-models-explained, 2023. Accessed: 2025-04-24.

[13] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[14] Aniket Anand Deshmukh. Kernel approximation. Technical Report 608, University of Michigan, Ann Arbor, 2015. Stats 608.

[15] Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, 1999. URL https://arxiv.org/abs/cs/9905014. Version Number: 1.

[16] ekamperi. Acquisition functions in Bayesian optimization, 6 2021. URL https://ekamperi.github.io/machine%20learning/2021/06/11/acquisition-functions.html. Blog post.

[17] Stefan Falkner. Hpbandster - a distributed hyperband implementation on steroids, 2018. URL https://github.com/automl/HpBandSter.

[18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1437–1446. PMLR, July 2018. URL https://proceedings.mlr.press/v80/falkner18a.html. ISSN: 2640-3498.

[19] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale, 2018. URL https://www.automl.org/blog_bohb/. Blog post on the AutoML.org website.

[20] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes. *Neurocomputing*, 380:20–35, March 2020. ISSN 09252312. doi: 10.1016/j.neucom.2019.11.004. URL https://linkinghub.elsevier.com/retrieve/pii/S0925231219315619.

[21] Google Developers. Convolutional neural networks, 2023. URL https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks. Image Classification.

[22] Jochen Görtler, Rebecca Kehlbeck, and Oliver Deussen. A visual exploration of gaussian processes. Distill.pub, 2020. Available at: https://distill.pub/2019/visual-exploration-gaussian-processes/.

[23] History Tools. Deep learning applications: An expert's guide, 2024. URL https://www.historytools.org/ai/deep-learning-applications. Accessed: 2025-04-25.

[24] T. Huang and A.S. Mohan. A hybrid boundary condition for robust particle swarm optimization. *IEEE Antennas and Wireless Propagation Letters*, 4:112–117, June 2005. ISSN 1536-1225, 1548-5757. doi: 10.1109/lawp.2005.846166. URL http://ieeexplore.ieee.org/document/1425453/. Publisher: Institute of Electrical and Electronics Engineers (IEEE).

[25] IBM. What is machine learning? - ibm think, 2025. URL https://www.ibm.com/think/topics/machine-learning. Accessed: April 25, 2025.

[26] Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 240–248. PMLR, May 2016. URL

https://proceedings.mlr.press/v51/jamieson16.html. ISSN: 1938-7228.

[27] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, WA, Australia, 1995. IEEE. doi: 10.1109/icnn.1995.488968. URL http://ieeexplore.ieee.org/document/488968/.

[28] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 528–536. PMLR, April 2017. URL https://proceedings.mlr.press/v54/klein17a.html. ISSN: 2640-3498.

[29] Nick Krüger, Kolja Eger, and Wolfgang Renz. SMARDcast: Day-Ahead Forecasting of German Electricity Consumption with Deep Learning. In *2024 International Conference on Smart Energy Systems and Technologies (SEST)*, pages 1–6, Torino, Italy, September 2024. IEEE. doi: 10.1109/sest61601.2024.10694018. URL https://ieeexplore.ieee.org/document/10694018/.

[30] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. ISSN 1533-7928. URL http://jmlr.org/papers/v18/16-558.html.

[31] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training, 2018. URL https://arxiv.org/abs/1807.05118. Version Number: 1.

[32] Yaxian Liu, Hui Lu, Shi Cheng, and Yuhui Shi. An Adaptive Online Parameter Control Algorithm for Particle Swarm Optimization Based on Reinforcement Learning. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, Wellington, New Zealand, June 2019. IEEE. doi: 10.1109/cec.2019.8790035. URL https://ieeexplore.ieee.org/document/8790035/.

[33] Milvus. How does the discount factor (gamma) affect rl training?, 2023. URL https://milvus.io/ai-quick-reference/how-does-the-discount-factor-gamma-affect-rl-training.

[34] Milvus. How is the learning rate used in reinforcement learning?, 2023. URL https://milvus.io/ai-quick-reference/how-is-the-learning-rate-used-in-reinforcement-learning.

[35] Christoph Molnar. Interpretable machine learning. Online book, 2022. Available at: https://christophm.github.io/interpretable-ml-book/.

[36] Michael Nielsen. Neural networks and deep learning. Online book, 2019. Available at: http://neuralnetworksanddeeplearning.com/.

[37] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–. URL https://github.com/bayesian-optimization/BayesianOptimization.

[38] Christopher Olah. Understanding lstm networks. Colah's Blog, 2015. Available at: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[39] OpenAI. Introduction to reinforcement learning. OpenAI Spinning Up documentation, 2018. Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[40] Konstantinos Parsopoulos and Michael Vrahatis. Particle swarm optimizer in noisy and continuously changing environments. 2001.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Kernel approximation - mathematical details. scikit-learn documentation, 2011–2024. URL https://scikit-learn.org/stable/modules/kernel_approximation.html#mathematical-details. Accessed: 2025-04-22.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. *Support Vector Machines*. scikit-learn, 2024. URL https://scikit-learn.org/stable/modules/svm.html#complexity. Accessed: 2025-04-22.

[43] Marius-Constantin Popescu, Valentina Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8, 07 2009.

[44] Partha Pratim Ray. ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*, 3:121–154, 2023. ISSN 26673452. doi: 10.1016/j.iotcps.2023.04.003. URL https://linkinghub.elsevier.com/retrieve/pii/S266734522300024X.

[45] Stanford CS231n. Convolutional neural networks. Course notes, 2021. Available at: https://cs231n.github.io/convolutional-networks/.

[46] Conor Sweeney, Ricardo J. Bessa, Jethro Browell, and Pierre Pinson. The future of forecasting for renewable energy. *WIREs Energy and Environment*, 9(2):e365, March 2020. ISSN 2041-8396, 2041-840X. doi: 10.1002/wene.365. URL https://wires.onlinelibrary.wiley.com/doi/10.1002/wene.365.

[47] Tala Talaei Khoei, Hadjar Ould Slimane, and Naima Kaabouch. Deep learning: systematic review, models, challenges, and research directions. *Neural Computing and Applications*, 35(31):23103–23124, November 2023. ISSN 0941-0643, 1433-3058. doi: 10.1007/s00521-023-08957-4. URL https://link.springer.com/10.1007/s00521-023-08957-4.

[48] The Ray Team. Ray tune api: Schedulers, 2024. URL https://docs.ray.io/en/latest/tune/api/schedulers.html. Accessed: 2024-04-25.

[49] Towards Data Science. Understanding the bias-variance tradeoff. Towards Data Science blog, 2018. Available at: https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229.

[50] Dongshu Wang, Dapei Tan, and Lei Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22(2):387–408, January 2018. ISSN 1432-7643, 1433-7479. doi: 10.1007/s00500-016-2474-6. URL http://link.springer.com/10.1007/s00500-016-2474-6. Publisher: Springer Science and Business Media LLC.

[51] Shuhei Watanabe. Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance, 2023. URL https://arxiv.org/abs/2304.11127. Version Number: 3.

[52] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, November 2020. ISSN 0925-2312. doi: 10.1016/j.neucom.2020.07.061. URL https://linkin

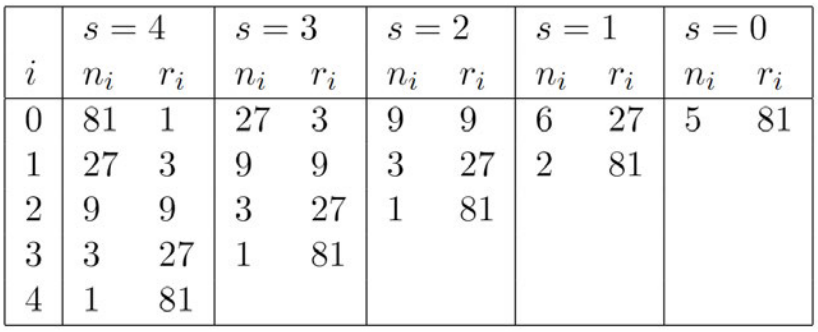
ghub.elsevier.com/retrieve/pii/S0925231220311693. Publisher: Elsevier BV.

A Appendix

Table A.1: SMARDCast model performance variance

Metric	**Validation**		**Test**	
	Mean	**Standard Deviation**	**Mean**	**Standard Deviation**
MSE*	829.34	±56.41(±6.2%)	921.11	±72.57(±7.4%)
MAPE*	4.67	±0.38(±7.4%)	5.60	±0.52(±8.5%)

**MSE: Mean Squared Error, MAPE: Mean Absolute Percentage Error*



	$s=4$		$s=3$		$s=2$		$s=1$		$s=0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

Figure A.1: Hyperband brackets example with $R = 81$ and $\eta = 3$. Source: [30]

Table A.2: Hyperparameter settings for the SMARDCast model

Parameter	**Sampling Method**	**Range/Options**
Learning Rate	loguniform	$[10^{-4}, 10^{-2}]$
Input Timesteps	choice	{96, 192, 288}
CNN Layers	discrete uniform	[1, 3]
CNN Filters*	pow2	$[2^4, 2^8]$
CNN Kernel Size*	pow2	$[2^1, 2^4]$
CNN Batch Normalization*	boolean	{True, False}
CNN Max Pooling	bool	{True, False}
LSTM Layers	discrete uniform	[1, 2]
LSTM Units*	pow2	$[2^5, 2^8]$
LSTM Dropout*	uniform	[0.0, 0.4]
Time Feature Signals	combination	{Day, Hour, Month, Week, Year, Day of the Week, Is Holiday}

* The parameter applies to each layer separately.

Table A.3: Fixed parameter settings for the SMARDCast model

Parameter	**Value**
Input Columns	Total Load [MWh]
Output Timesteps	96
Output Columns	Total Load [MWh]
Offset	0
Window Split	96
Epochs	50
Patience	3
Batch Size	256
Evaluation Metric	Validation Loss
Metrics	MAPE, RMSE, MAE
Mode	Min

Table A.4: Hyperparameter settings for SVM on MNIST

Parameter	**Sampling Method**	**Range/Options**
Gamma	loguniform	$[e^{-10}, e^{10}]$
C	loguniform	$[e^{-10}, e^{10}]$

Table A.5: Fixed parameter settings for SVM on MNIST

Fixed Parameters

Parameter	**Value**
Epochs	10
Evaluation Metric	Test Error
Mode	Min

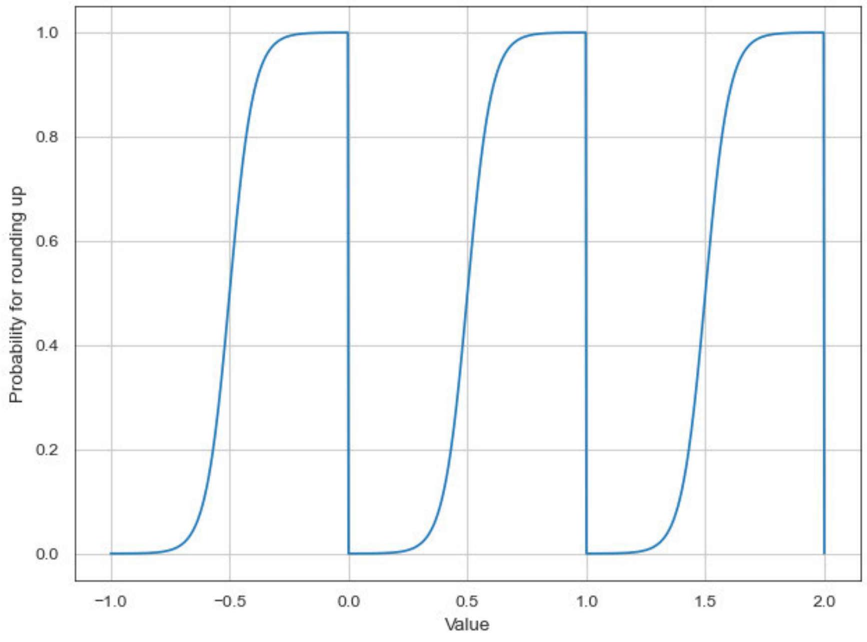


Figure A.2: Sigmoid probability function used for stochastic rounding

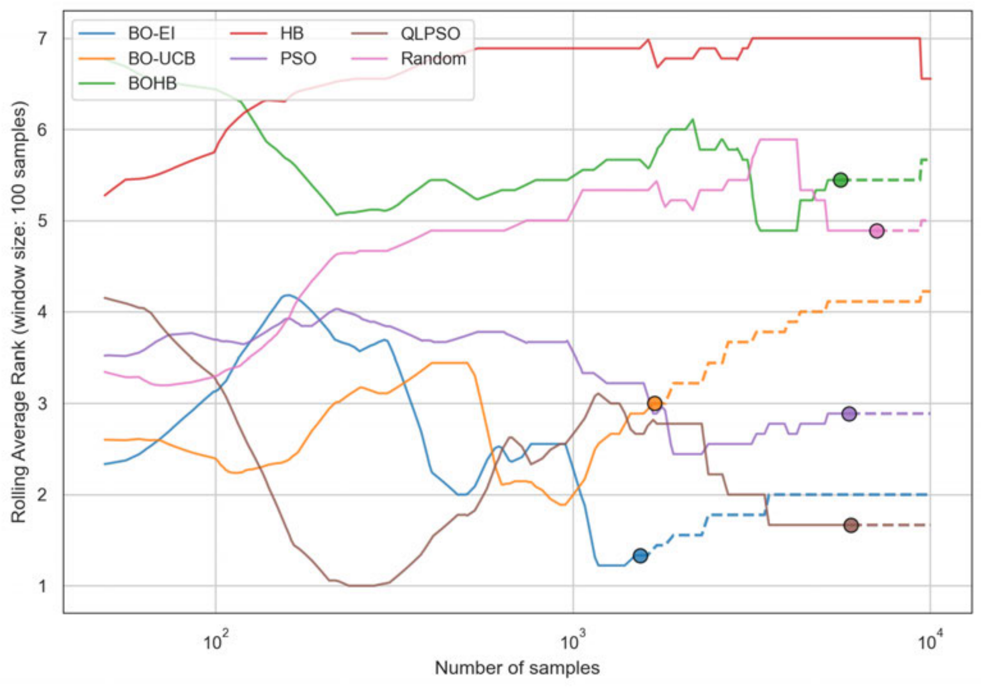


Figure A.3: SMARDCast: Average rank of optimizers over samples (lower is better)

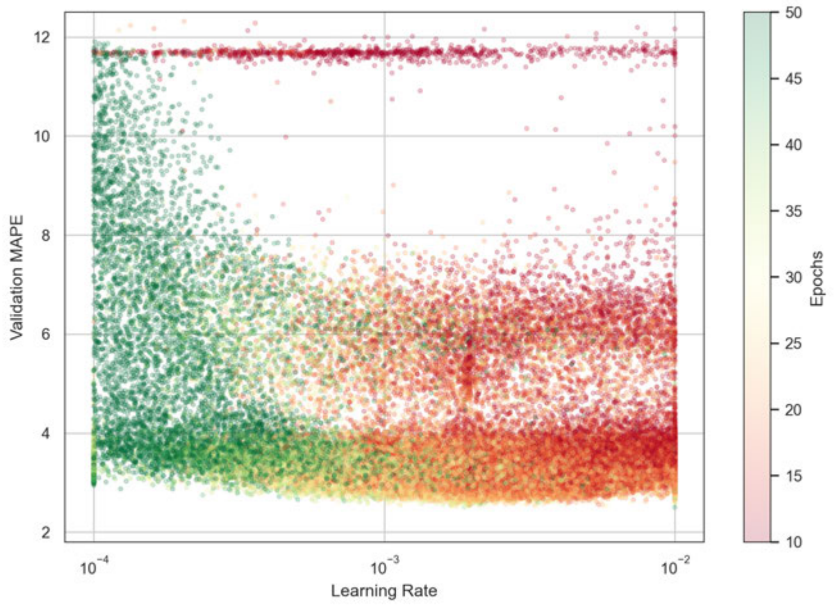


Figure A.4: Validation MAPE over learning rate of fully trained models, hue represents the number of epochs

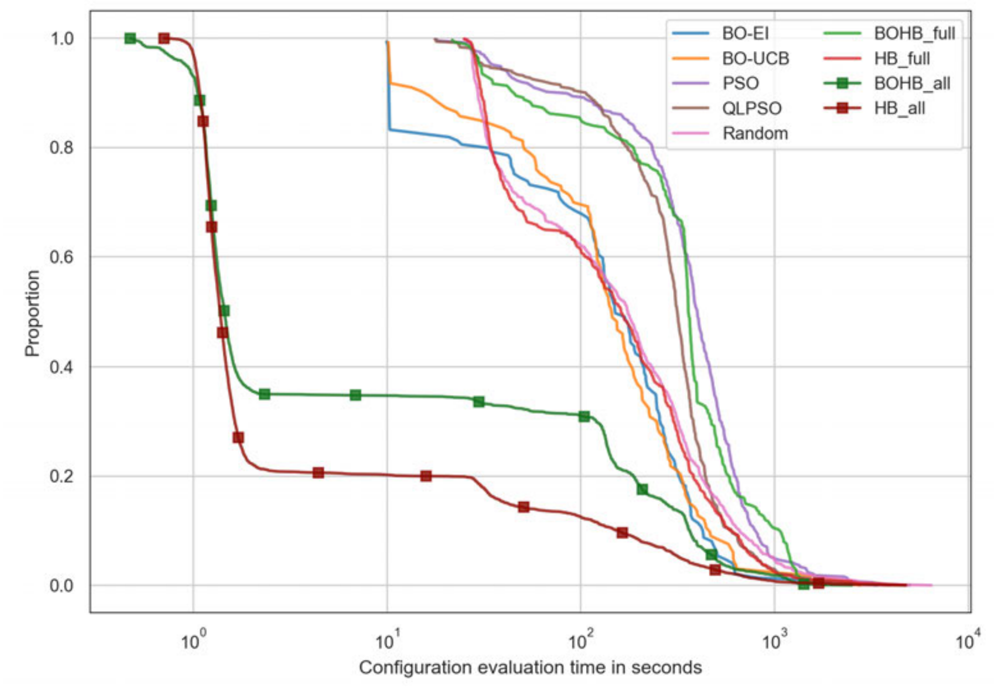


Figure A.5: SVM evaluation time differences displayed as an ECDF plot ('full' suffix represents models trained on the maximum fidelity, 'all' suffix includes low-fidelity evaluations)

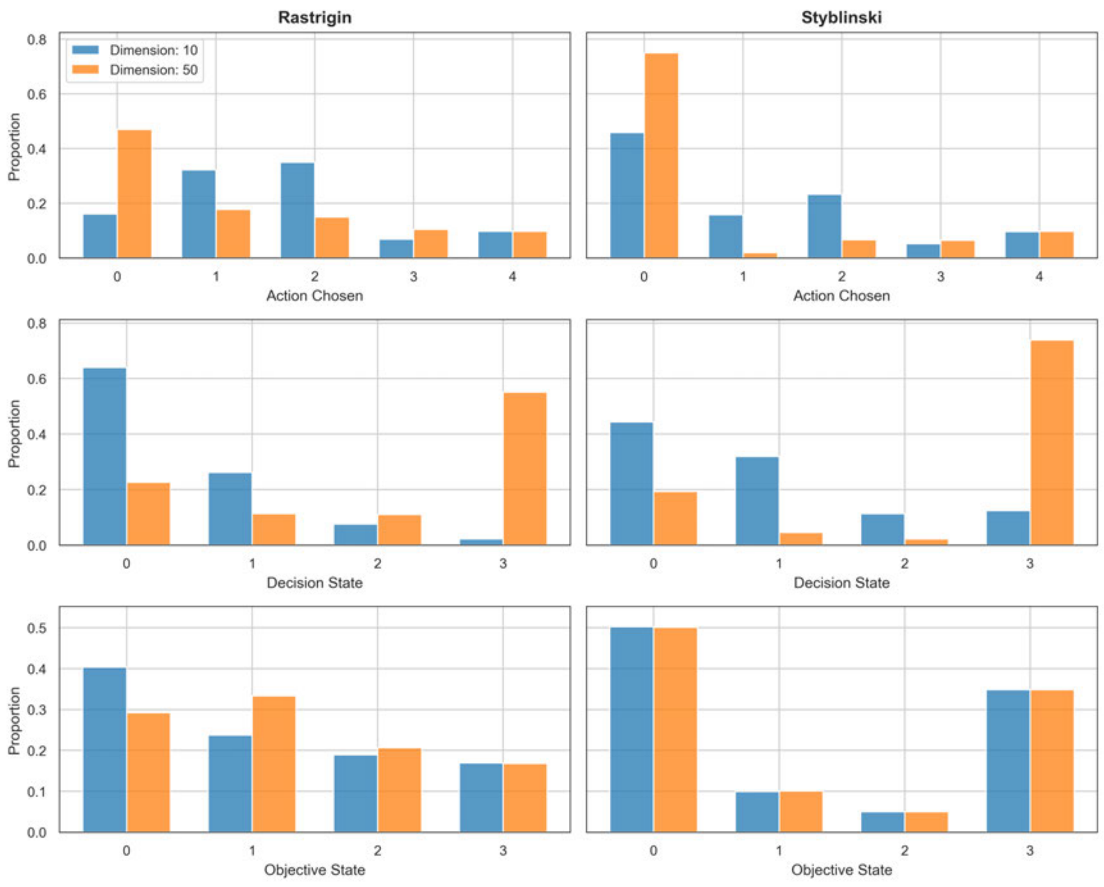


Figure A.6: Histogram ensemble plot for QLPSO behavior

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

[redacted]

Ort	Datum	Unterschrift im Original