Enabling Hyperparameter Tuning of Machine Learning Classifiers in Production

Sandeep Singh Sandha†, Mohit Aggarwal‡, Swapnil Sayan Saha† and Mani Srivastava†
University of California, Los Angeles†; Arm Research, Austin‡
sandha@cs.ucla.edu, mohit.aggarwal@arm.com, swapnilsayan@g.ucla.edu, mbs@ucla.edu

***Abstract*—Machine learning (ML) classifiers are widely adopted in the learning-enabled components of intelligent Cyber-physical Systems (CPS) and tools used in designing integrated circuits. Due to the impact of the choice of hyperparameters on an ML classifier performance, hyperparameter tuning is a crucial step for application success. However, the practical adoption of existing hyperparameter tuning frameworks in production is hindered due to several factors such as inflexible architecture, limitations of search algorithms, software dependencies, or closed source nature. To enable state-of-the-art hyperparameter tuning in production, we propose the design of a lightweight library (1) having a flexible architecture facilitating usage on arbitrary systems, and (2) providing parallel optimization algorithms supporting mixed parameters (continuous, integer, and categorical), handling runtime failures, and allowing combined classifier selection and hyperparameter tuning (CASH). We present *Mango*, a black-box optimization library, to realize the proposed design. Mango is currently used in production at Arm for more than 25 months and is available open-source (https://github.com/ARM-software/mango). Our evaluation shows that Mango outperforms other black-box optimization libraries in tuning hyperparameters of ML classifiers having mixed parameter search spaces. We discuss two use cases of Mango deployed in production at Arm, highlighting its flexible architecture and ease of adoption. The first use case trains ML classifiers on the Dask cluster using Mango to find bugs in Arm's integrated circuits designs. As a second use case, we introduce an AutoML framework deployed on the Kubernetes cluster using Mango. Finally, we present the third use-case of Mango in enabling neural architecture search (NAS) to transfer deep neural networks to TinyML platforms (microcontroller class devices) used by CPS/IoT applications.**

***Index Terms*—Hyperparameter tuning, Machine learning in production, Parallel Bayesian optimization.**

I. Introduction

Enabling Hyperparameter tuning at a production scale is crucial to designing better performing ML classifiers embedded in emerging CPS/IoT applications [1]. However, a typical ML pipeline in production can be too specialized and complex, demanding a trained team of human experts with specific domain knowledge for classifier selection with optimal hyperparameters. The combined classifier selection and hyperparameter optimization in production face the following challenges:

1. Complex deployments: The production ML pipelines are complex and realized using a combination of arbitrary systems (e.g., custom on-premise software, cluster frameworks, cloud infrastructures) decided by several factors, including the nature of application and developer preferences. Therefore, flexible architecture with abstractions allowing *usage on arbitrary systems* is needed.

2. High complexity of the hyperparameter search: Search is becoming increasingly complex, with many choices for classifiers and their rich parameter spaces. It is further exacerbated in production pipelines due to the recurrent nature of tuning tasks triggered by data shifts or process changes. Consequently, searching the space of several classifiers demands *combined algorithm/classifier selection and hyperparameter optimization (CASH) [2]*. Further, to speed up the search, *intelligent parallel algorithms* utilizing parallel computing with abstractions to handle runtime failures are needed.

Further, abstractions offering uniformity in local and cluster usage, including syntax compatibility with the widely used ML libraries like Scikit-learn [3], can reduce the effort needed to integrate with existing deployments. While several hyperparameter tuning software exists, their adoption in an arbitrary production pipeline is hindered due to their dependence on particular compute scheduling extensions [4]–[8], closed source nature [9], search algorithm limitations [10], [11] and significant overhead adopting the entire software dependencies [12]–[15]. For example, the parallel search in Hyperopt [4] is dependent on the MongoWorker processes or Apache Spark [16].

To enable hyperparameters tuning in production, we present Mango, a black-box optimization library. Mango is a research project that provides hyperparameter tuning to ML pipelines at Arm with more than 25 months of production usage. Mango is open-sourced under Apache 2.0 license to contribute and learn from the community. Mango provides the following core features addressing the above challenges:

- Modular design that allows the user to schedule objective function evaluations on arbitrary infrastructure. Furthermore, API provides a unique capability to handle runtime failures crucial for production deployments.
- An efficient realization of Bayesian optimization using the Gaussian process (GP). We incorporate optimal handling of mixed parameters and intelligent batch sampling for parallel search for practical adoption of GP.
- An algorithm to directly solve CASH problem using multiple GP surrogates. To the best of our knowledge, existing GP libraries don't solve the CASH problem.

To highlight the flexible architecture enabling the adoption of Mango in complex ML pipelines, we discuss two production

978-1-6654-1621-4/21/$31.00 ©2021 IEEE
DOI 10.1109/CogMI52975.2021.00041

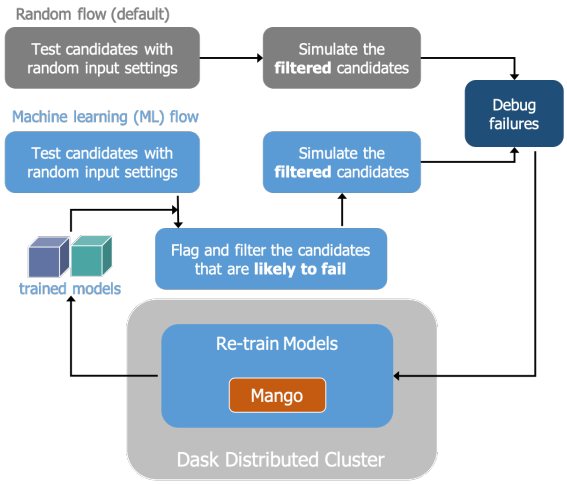


Fig. 1: A *Bug Hunting Workflow* [17] is part of the design verification of integrated circuits at Arm. A machine learning pipeline replaced the default pipeline to predict the preferred input candidates. Mango is deployed on the Dask distributed cluster to automate the hyperparameter tuning of ML models used for design verification.

use cases (1) a bug hunting workflow deployed on the Dask cluster [14] using ML classifiers to optimize the design verification of Arm's integrated circuits, (2) an AutoML framework deployed on the Kubernetes cluster[1]. Figure 1 shows the first use case doing design verification of integrated circuits at Arm. We evaluate the implemented optimization algorithms in Mango on a collection of benchmark functions and classifiers. Finally, we present a third use case of Mango enabling NAS for Cortex-M microcontrollers class devices found in resource constrained IoT and CPS applications.

II. Background and Related Work

A. Hyperparameter Tuning Frameworks

The hyperparameter frameworks can be broadly categorized into two groups (1) software built on distributed frameworks to provide hyperparameter tuning as a feature [12]–[15] and (2) optimization libraries with integrated scheduling extensions [4]–[8]. However, adopting existing frameworks in production ML pipelines faces hindrances primarily due to high overhead in adoption for the former group and dependence on custom-built parallel schedulers for the latter.

For example, Katib [13] and Polyaxon [15] are built on top of Kubernetes. Tune [12] is a python library deployed using the Ray framework [18]. Dask-ml[2] provides hyperparameter tuning using the Dask framework. These systems offer features like auto-scaling, failovers, and rich scheduling abstractions. However, their integration into an arbitrary deployment demands adopting the specific underlying framework and additional software dependencies, creating development and maintenance overhead. For example, using Katib requires Kubernetes and adding components like database, API server, and controller processes. These systems can be a good fit if the application can benefit from other functionalities (a web dashboard, for example) provided by these frameworks, thereby justifying the integration overhead.

The second group includes optimization libraries with custom-built scheduling mechanisms to run parallel workers. The scheduling mechanisms in these libraries are not as stable as production-grade frameworks like Kubernetes. They lack critical features required in production systems like auto-scaling, failover, and the ability to deploy on arbitrary infrastructures. For example, parallel search in Hyperopt [4] is dependent on MongoDB database or Apache Spark. Parallel workers read and update the evaluation history from a shared document. During performance evaluation, we encountered repeated failures of workers due to communication loss with the MongoDB database, and we had to restart the worker after every failure manually. Spearmint [5] is also dependent on MongoDB as a shared document store to enable parallel evaluations. However, the Spearmint repository has not been actively supported since 2015, so we did not evaluate it in our experiments. Parallel search in SMAC [6] relies on a shared file system for multiple workers to collaborate. Further, there is no direct way to combine various workers' results[3]. Optuna [7] also requires a relational database like MySQL to communicate between workers. Auto-sklearn [8] provides a wrapper around the SMAC optimizer to enable hyperparameter tuning of Scikit-learn's ML models. Auto-sklearn's parallel search is dependent on setting up a Dask cluster framework [14] or requires a shared file system.

To summarize, all these libraries use some form of shared storage to communicate between parallel workers. However, this is prone to failures due to communication loss, node failures, and run-time errors. The libraries also do not provide mechanisms to recover when such failures occur. These limitations offer critical hindrances towards adopting these libraries in production deployment. It is important to note that GPyOpt [10] and Skopt [11] expose the sampled batch to be scheduled on an arbitrary framework. However, they have limitations when applied to hyperparameter tuning of ML classifiers, as discussed in the Section II-B.

Mango was borne out of the need to overcome the above limitations. Mango is independent of a specific distributed framework. Moreover, it does not mandate the use of any additional software component like database, API server, or shared file storage enabling classifier evaluations on arbitrary systems.

B. Hyperparameter Tuning Algorithms

Bayesian optimization provides a state-of-the-art approach to optimize expensive objective functions in a few evaluations. Typical surrogate models used in Bayesian optimization libraries are GP (GPyOpt, Skopt, Spearmint), tree-structured Parzen estimators (Hyperopt, Optuna), and random forest

[1]https://kubernetes.io/
[2]https://ml.dask.org/
[3]https://github.com/automl/SMAC3/issues/446

(SMAC, Skopt). GP surrogate is one of the preferred choices due to its ability to provide a tractable assessment of prediction uncertainty incorporating the effect of data scarcity [5], [19]. Further, GP is shown to outperform tree-structured Parzen estimators (TPE) and random forest for functional benchmarks [7]. However, when using GP for hyperparameter tuning, libraries need to address the following shortcomings (1) handling of categorical variables [20], (2) enabling conditional parameter spaces and CASH [21], and (3) realizing parallel search [6], [22], [23].

GpyOpt[4], Skopt[5], and Spearmint[6] doesn't support conditional spaces, so they cannot be used to solve CASH[7] problem. GpyOpt cannot handle non-numerical categorical values[8]. Hyperopt and Optuna provide TPE surrogates. SMAC uses a random forest surrogate. Hyperopt, Optuna, and SMAC support conditional variables that can be applied to solve a CASH problem. However, TPE is designed to be sequential in nature [4], thus suffers performance loss in parallel search. SMAC doesn't support inbuilt parallel search and uses multiple sequential runs to simulate parallel search.

The alternatives to Bayesian optimization are the multi-fidelity optimization algorithms like successive having [24] and Hyperband [25] exploiting partial training. Although these approaches are cheaper to evaluate, they suffer from approximations errors in small-budget evaluations.

Mango algorithms use Bayesian optimization using GP and address the GP's shortcomings. The mixed numerical/categorical search spaces are handled using one-hot encoding combined with Monte-Carlo sampling for acquisition function optimization. For batch sampling to enable parallel search, Mango provides a penalty approach and clustering search [26]. We present an algorithm using multiple GP surrogates motivated by exploiting the structure of hyperparameter space [19], [27] to solve the CASH problem.

A short paper introducing Mango with its early production usage is available [28]. This paper discusses the challenges in enabling hyperparameter tuning in production, Mango features addressing these challenges, hindrances in adopting existing frameworks, implemented algorithms, and the learning experiences from deployed use cases.

III. Mango

Mango has a functional-based API to integrate with a model training pipeline. The four abstractions in Mango are (1) *Parameter Space Definer*, (2) *Objective Specifier*, (3) *Tuner*, and (4) *MetaTuner*. The modular architecture enables the integration of new functionality and the ease of production maintenance.

Parameter Space Definer provides python constructs to easily specify complex search spaces, including mixed numerical/categorical values. The design of *Objective Specifier* allows classifiers' training on local machines using an integrated scheduler and arbitrary systems (e.g., custom-local software, cluster frameworks) by exposing sampled batches. *Tuner* exposes implemented algorithms for serial and parallel search. *MetaTuner* solves a CASH problem. We show the skeleton codes from production use cases to highlight these features. Figure 2 shows an example of Mango for hyperparameter tuning of XGBClassifier on a local machine using the integrated parallel scheduler. The default optimization algorithm and configurations can be modified.

```
from mango import Tuner, scheduler
from scipy.stats import uniform
from xgboost import XGBClassifier
...
param_space = {'learning_rate': uniform(0, 1),
               'gamma': uniform(0, 5),
               'max_depth': range(1, 21),
               'n_estimators': range(1, 11),
               'booster':['gbtree','gblinear',
               'dart']}
@scheduler.parallel(n_jobs=4)
def objective(**params):
    ...
    clf = XGBClassifier(**params)
    accuracy = ...
    return accuracy
tuner = Tuner(param_dict, objective)
Study = tuner.maximize()
```

Fig. 2: An example of Mango to tune the hyperparameters of XGBClassifier from the Xgboost library using a parallel scheduler on the local machine. Parameter space consists of distribution, range, and categorical variables.

A. Mango Abstractions

Parameter Space Definer: Mango uses Python constructs (range and list) to define search spaces with mixed numerical/categorical values. As shown in Figure 2, *param_space* is defined as a python dictionary. Continuous variables use distributions from Scipy[9]. All the 60+ distributions from Scipy are supported, allowing the flexibility to specify preferred regions in the search space. Mango supports user-defined parameter distributions. The parameter space definitions are compatible with the Scikit-learn, allowing replacement for existing applications using Scikit-learn.

Objective Specifier: The objective specifications are available to train the classifier using a local machine or any arbitrary system. The objective function training classifier uses an integrated parallel scheduler on the local machine is shown in the Figure 2. Here, the input to the *objective()* function is a dictionary (*params*) with a single sampled point suggested for evaluation by *Mango*. The *@scheduler* decorator specifies the number of parallel jobs.

For *deployments on arbitrary systems*, a more general skeleton of *Objective Specifier* is available, as shown in Figure 3. It exposes the sampled batch directly to the user-defined objective function to evaluate an application-specific

[4]https://github.com/SheffieldML/GPyOpt/issues/241
[5]https://github.com/scikit-optimize/scikit-optimize/issues/770
[6]https://github.com/HIPS/Spearmint/issues/54
[7]A library needs to allow classifier type as meta-hyperparameter to support CASH, requiring conditional spaces or a specialized approach.
[8]https://github.com/SheffieldML/GPyOpt/issues/161
[9]http://www.scipy.org/

```
from kubernetes import client
...
param_space = ...
def objective(params_batch):
     # train on cluster using the sampled parameters
    jobs = [client.create_job(params, ...)
            for params in params_batch]
    # poll for job completion
    results = []
    while not timeout or not all_done:
        results = [job.result() for job in jobs
                   if job.complete()]
    return results
# control the max number of iterations, batch size,
conf = {'num_iteration':100, 'initial_random':5,
'batch_size':4, 'parallel':'clustering'}
tuner = Tuner(objective, param_space,conf)
Study = tuner.maximize()
```

Fig. 3: Skeleton code of Mango on Kubernetes cluster that is deployed as part of the AutoML framework at Arm. Partial *results* are returned by the *objective* function based on timeout. The *conf* data structure modifies the default behavior of *Tuner*.

scheduler. This scheduler's nature is decided based on the deployment framework. We allows the user-defined objective function to discard the *failed evaluations* as shown in *objective* function in the Figure 3 to make progress even with runtime failures. The specific technique (e.g., timeout in Figure 3) to identify a failure is kept outside of the Mango, as it may depend on the underlying compute platform and tuning task.

The skeleton code shown in Figure 3 is part of an AutoML framework (see Section IV-C) deployed on the Kubernetes cluster at Arm. The objective function can return the result as a list of values specifying the successful evaluations and their respective hyperparameters without waiting for all the evaluations to complete. The batch objective function skeleton is kept independent of the underlying compute infrastructure, with no dependency on the additional databases or a shared file system, enabling its adoption across applications.

Tuner: A parameter space definition and the specified objective function are used by *Tuner* to search optimal hyperparameters. The *config* parameter (Figure 3) is optional. It controls the maximum number of iterations, the initial random iterations, the batch size for parallel search, and the optimization algorithm. *Tuner* exposes sequential and parallel search algorithms.

MetaTuner: *MetaTuner* is designed to solve a CASH problem. The skeleton code of *MetaTuner* deployed in production on a Dask distributed framework [14] is shown in Figure 4. This code is part of the *Bug Hunting Workflow* shown in Figure 1. The *param_space* data structure is a list of search spaces for individual classifiers identified by their *type* during scheduling.

B. Optimization Algorithms in Mango

Mango algorithms are based on Bayesian optimization. Here, we summarize the sequential search, handling of the categorical variables, batch sampling to enable parallel search, and the CASH algorithm.

```
from dask.distributed import Client
...
dask_client = Client()

param_clf_nn = {'type': 'clf_nn',...}
param_clf_svm = {'type': 'clf_svm',...}
param_spaces = [param_clf_nn, param_clf_svm]

def objective(params_batch):
    futures = []
    # Submit Jobs to the Dask cluster
    for params in params_batch:
        #schedule classifier based on type
        clf = params.pop('type')
        future = dask_client.submit(fit_and_score,
                                    clf, **params)
        futures.append(future)
    # Job completion or wait for timeout
    results = [future.result(timeout) for future in
    futures]
    return results
metatuner = MetaTuner(objective, param_spaces)
Study = metatuner.maximize()
```

Fig. 4: Skeleton code deploying *MetaTuner* algorithm on the Dask cluster, which is part of the bug hunting application used for design verification of Arm integrated circuits designs.

Sequential search: The sequential search uses Bayesian optimization with GP as the surrogate model. We use the Matern kernel function and the upper confidence bound (UCB) as the acquisition function [29]. The next sampled hyperparameter is selected based on the predicted mean (exploitation) and the corresponding variance (exploration). The exploration factor is used to decide a trade-off between exploitation and exploration. The exploration factor in Mango is fixed by default to 2.0; however, for expert users, we allow adaptive exploration proposed by Srinivas et al. [29], where the exploration factor is heuristically decided based on the complexity of the search space (domain size) and the current iteration count. The idea is to allow more exploration when the classifier's search space is huge. We do the Monte Carlo optimization of the acquisition function by sampling the parameter space and then selecting the next point to evaluate based on the acquisition function. The total number of samples drawn is decided based on the the complexity of the search space inferred using the definition.

Handling categorical values: The naive GP assumes continuous input variables. Thus, handling categorical and integer values requires careful consideration. We use one-hot encoding for the categorical values. However, naively rounding off the categories or integers during evaluations can result in poor performance as the actual point of objective evaluation may differ from the proposed point [20]. Our approach is motivated by the solution proposed by Garrido-Merchán et al. [20]. We optimize the acquisition function by sampling only the valid points from the search space; thus, there is no mismatch between the proposed and actual evaluation.

Parallel search: Conventionally, Bayesian optimization using GP is a sequential search since new information must update the acquisition function. The challenge in selecting a batch of values is to ensure exploration diversity in the batch. A simple

technique to enforce diversity is that no choice is selected twice in the batch. It can be done by ranking the choices according to the UCB and then selecting top picks until new feedback is available. However, this naive approach has limited exploration [23], demanding intelligent parallel strategies. We provide two algorithms to sample a batch of values in Mango.

The first algorithm, *Clustering search* used by default, is motivated by selecting peaks [26], [30] of acquisition function within a batch. It has the following two steps: (1) First, we select a set of promising domain samples (top 25% by default) based on the acquisition function. (2) Next, these domain samples are clustered based on their distance in the search domain space. We select the hyperparameter choice from each cluster with the highest acquisition function value and add it to the batch. We use K-Means clustering.

The second algorithm is *hallucination search* which is based on the idea of applying penalty [22], [23] to sample a batch using the acquisition function.

Algorithm 1: *MetaTuner* algorithm.

input : list of parameter spaces P_{List}, objective function obj_fxn, and configuration $Conf$
output: type of classifier C_{type}, optimal parameters O_{par}

1 $meta_{xpl} \leftarrow$ 1.0, $min_{xpl} \leftarrow$ exp_value;
2 $decay_{rate} \leftarrow$ decay_value, acc_{max} = 0;
3 $C_{type} \leftarrow$ None, $O_{par} \leftarrow$ None;
4 **for** $i \leftarrow 1$ **to** $Conf$*[max_iterations]* **do**
5 $Curr_{clf} \leftarrow$ None, $Curr_{par} \leftarrow$ None;
6 rand $\leftarrow$ random();
7 **if** *rand*$<meta_{xpl}$ **then**
8 $Curr_{clf} \leftarrow$ randINT(1,no_of_clf)
9 $Curr_{par}$, _ $\leftarrow$ get_gp_acq($P_{List}[Curr_{clf}]$,$Conf$);
10 $meta_{xpl} \leftarrow$ max($meta_{xpl} * decay_{rate}$, min_{xpl});
11 **else**
12 **for** $i \leftarrow 1$ **to** *Size*(P_{List}) **do**
13 $X[i], Y[i] \leftarrow$ get_gp_acq($P_{List}[i]$,$Conf$);
14 **end**
15 $Curr_{clf} \leftarrow$ argmax($Y[i]$);
16 $Curr_{par} \leftarrow X[Curr_{clf}]$;
17 **end**
18 curr_evaluation $\leftarrow$ obj_fxn ([$Curr_{clf}$,$Curr_{par}$]);
19 update_gp([$Curr_{clf}$,$Curr_{par}$,curr_evaluation]);
20 update_gp_exp([$Curr_{clf}$]);
21 **if** *curr_evaluation*$>acc_{max}$ **then**
22 $acc_{max} \leftarrow$ curr_evaluation;
23 $C_{type} \leftarrow Curr_{clf}$, $O_{par} \leftarrow Curr_{par}$;
24 **end**
25 **end**
26 **return** C_{type}, O_{par}

MetaTuner algorithm to solve CASH: Direct addition of an extra algorithm selection parameter in GP assumes that information is shared between the hyperparameters of different classifiers. A regular GP would make an invalid credit assignment in these settings [21], [27]. To address this, we train multiple GP surrogates for each classifier independently. Our approach is motivated by the idea of exploiting the structure of the optimization problem proposed by Bergstra et al. [19] and Jenatton et al. [27]. Algorithm 1 is the serial version of *MetaTuner* algorithm.

Some classifiers can have an exploration bias occurring from the evaluation of good accuracy regions early on. To avoid these issues, we use random exploration ($meta_{xpl}$) with a decay rate ($decay_{rate}$) along with a minimum exploration (min_{xpl}) across classifier. Lines[7-10] do random exploration across classifiers using $meta_{xpl}$. Function *get_gp_acq* suggests the parameter and the respective acquisition function value using the parameter space definition and configuration of the used classifier. Lines[12-16] select a classifier and hyperparameter to evaluate based on its acquisition value. The objective function evaluation for the selected classifier and the Gaussian process surrogate update is done in Lines[18-19]. Line-20 updates the surrogate's exploration for the classifier that is evaluated. The idea is to favor other classifiers for future evaluations by using more exploration factors if they have high uncertainty due to their large search space. Finally the best performing classifier and optimal parameter is maintained in the Lines[21-23]. The default values ($decay_{rate}$=0.9, min_{xpl}=0.1) available in *MetaTuner* are the same that are used for experiments.

A batch version of this algorithm is implemented in Mango, where we initially select a batch $|B|$ of values from individual surrogates (*get_gp_acq*) using parallel search, and then rank these $(N * |B|)$ values, where N is the number of classifiers, to select $|B|$ points to evaluate in parallel. Note that a mix of classifiers may be evaluated in batch based on their acquisition function.

IV. Evaluation and Case Studies

A. Optimization Performance Evaluation

We compare Mango with several black-box optimization libraries using the multiple criteria methodology proposed by Dewancker et al. [31], also used by Akiba et al. [7]. Specifically, we measure performance by the solution's proximity to the optimal point (accuracy) and the number of iterations required to reach the optima (speed). We performed experiments across two classes of optimization tasks: (1) Synthetic test functions and (2) ML classifiers. Each optimization task uses 80 iterations and is repeated 30 times to account for the algorithm's stochastic nature [31]. Results are statistically compared for accuracy and speed criteria using paired Mann-Whitney U test with $\alpha = 0.01$ [31]. Libraries to compare against are chosen to represent different flavors of Bayesian optimization: Hyperopt with TPE surrogate, Optuna with a mixture of TPE and CMA-ES, SMAC with random forest surrogate. GPyOpt with Gaussian process surrogate, and lastly, random search serves as the baseline.

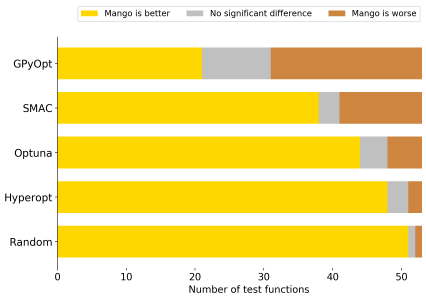
1) Synthetic test functions: We used a collection of 53 functions having continuous search spaces from a benchmark suite of test functions [31]. Figure 5a shows the results where the objective function is evaluated sequentially. Mango is worse than Optuna in 5/53 tests and Hyperopt in 2/53 tests. This is expected because the GP surrogate provides a more accurate representation of the objective function than TPE [7]. Mango performs worse than SMAC in 12/53 tests. The performance of Mango is competitive when compared to the other GP-based optimizer GPyOpt (worse in 22/53, tied in 10/53 tests).

Figure 5b shows the results for parallel search where the objective function is evaluated using four workers. Mango's clustering parallel search performance is compared with GPyOpt's local penalization approach, Optuna's random sampling, and random search. Mango performs worse than Optuna in the 17/53 test and worse than GPyOpt in the 32/53 tests. Hyperopt and SMAC also provide distributed optimization using custom-built scheduling frameworks. However, we could not complete the experiments for them due to repeated failures of their custom scheduling framework.

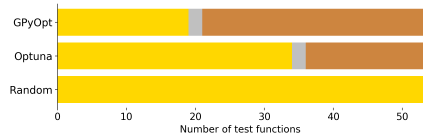
GPyOpt internally uses a gradient-based method to optimize the acquisition function, while Mango uses Monte Carlo sampling. The gradient-based method provides a slight advantage to GPyOpt in continuous search spaces, which is the case for these test functions. However, Mango's sampling approach is more suitable for heterogeneous search spaces that include categorical and integer parameters, which is the case for hyperparameter tuning of ML classifiers, as discussed in the next section.

2) Tuning ML classifiers: We compared the performance for hyperparameter tuning of three ML classifiers: Xgboost, K-Nearest Neighbor (KNN), Support Vector Machines (SVM) to maximize the 3-way cross-validation accuracy for the iris plants dataset, wine recognition dataset, and breast cancer Wisconsin (diagnostic) dataset taken from Scikit-learn, i.e., a total of 9 tuning tasks (three classifiers trained using three datasets). The search space includes continuous, integer, and categorical parameters with the exact definitions available [32]. The experiment setup is the same as before, having 80 iterations and 30 repeated runs. Results are shown in Figure 6. As seen in Figure 6a, Mango performs better than all other libraries in 6 or more tasks out of 9. Figure 6b shows the results for parallel hyperparameter tuning with four workers.' As seen, the *clustering search algorithm* of Mango outperforms GPyOpt's local penalization approach and Optuna's random sampling.

3) Hyperparameter Tuning across Classifiers: We compare the performance of *MetaTuner* to solve the CASH problem with Optuna's TPE+CMA-ES surrogate, Hyperopt's TPE surrogate, SMAC's random forest surrogate, and naive random search. GPyOpt is not included in this evaluation as it doesn't allow conditional search spaces. Sampling for the naive random search is done by uniformly choosing a classifier followed by randomly sampling a hyperparameter from the corresponding search space. The optimization objective is to find the best classifier and corresponding hyperparameters

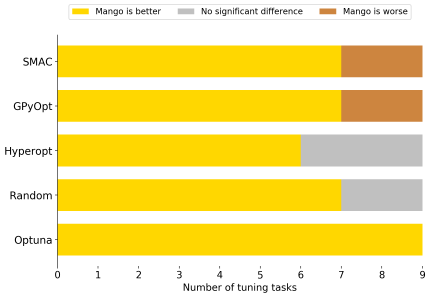


(a) Sequential optimization

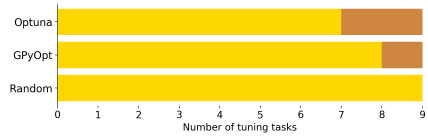


(b) Parallel optimization with 4 workers

Fig. 5: Comparison of Mango to optimize functions.



(a) Sequential optimization



(b) Parallel optimization with 4 workers

Fig. 6: Comparison of Mango to tune hyperparameters.

from the neural network, Xgboost, KNN, SVM, and decision tree. Experiments are done for three datasets taken from Scikit-learn: iris plants dataset, wine recognition dataset, and breast cancer Wisconsin (diagnostic) dataset. The exact parameter search spaces for all the classifiers are listed online [32].

Figure 7 shows the results for 150 serial iterations and an average of 30 runs. Optuna performs better than *MetaTuner* on iris dataset and breast cancer Wisconsin (diagnostic) dataset. *MetaTuner* performs better than Optuna on the wine recognition dataset. *MetaTuner* performs better than the Hyperopt and SMAC on all three datasets The results show that the *MetaTuner* algorithm performs comparably with TPE and random forest surrogates that directly support conditional search spaces.

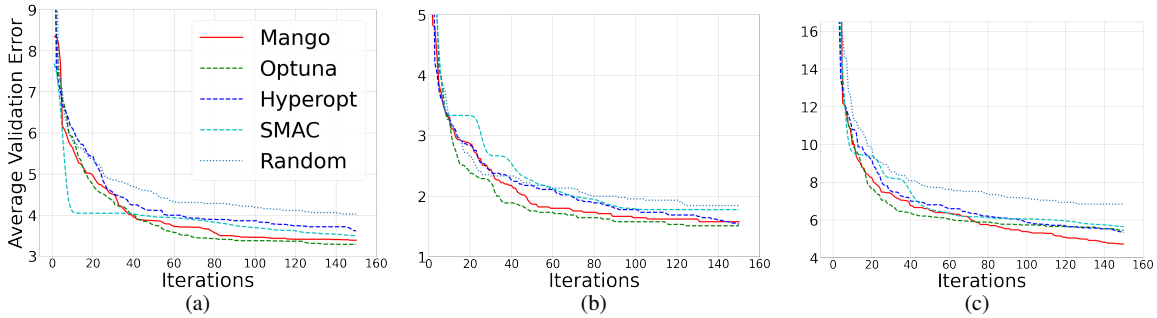


Fig. 7: The comparison of Mango's *MetaTuner* for combined classifier selection and hyperparameter optimization problem with other libraries. The evaluation uses five different classifiers (Xgboost, k-nearest neighbor, Support Vector Machines, decision tree, and neural network). Sub-figure (a) is for the Breast cancer dataset, sub-figure (b) the Iris plants dataset, and sub-figure (c) the Wine recognition dataset. Mango performs better than Hyperopt and SMAC and is competitive with Optuna.

4) Optimizer sampling time: One disadvantage of GP surrogate is that it is computationally expensive due to the cubic complexity in the number of samples evaluated. Comparatively, TPE surrogate used in *Hyperopt* and *Optuna* is very inexpensive. In Mango, we have reduced the computational complexity by using Monte Carlo optimization of acquisition function instead of commonly used gradient-based methods like L-BFGS. We evaluate this feature by comparing various optimizers' sampling times in sequential, parallel, and CASH settings. Results are shown in Table I. We did 30 runs of 80 iterations to calculate the average time taken per iteration. The sampling time depends on the complexity of the parameter space. For serial and parallel, we use the Xgboost's parameter space definition [32]. The CASH sampling time is shown for the Xgboost parameter space definition [32]. As expected TPE based optimizers are the fastest; however, Mango (GP) is significantly faster than the GPyOpt (GP) and SMAC (Random-forest). It is important to note that this comparison is inconsequential for hyperparameter tuning because the time taken to train ML models would dwarf the optimizer sampling time.

Summary: Mango outperforms other libraries in hyperparameter tuning for classifiers with mixed parameter (continuous, integer, and categorical) spaces. When evaluated for CASH problems, Mango's is competitive in performance to Optuna. In the case of functional benchmarks, Mango is competitive with the GpyOpt. However, Optuna performs poorly for functional benchmarks and tuning parameters for a single classifier. Further, GpyOpt performs poorly when tuning ML classifiers. Overall, Mango offers state-of-the-art algorithms having better or at par performance across settings.

TABLE I: Wall clock time (sec) taken by optimizers to sample next evaluation in sequential, parallel, and CASH settings.

Optimizer (Surrogate)	**Sequential**	**Parallel**	**CASH**
Hyperopt (TPE)	0.001 ± 0.005	na	0.02 ± 0.001
Optuna (TPE)	0.07 ± 0.035	0.02 ± 0.006	0.02 ± 0.001
Mango (GP)	0.16 ± 0.008	0.12 ± 0.021	0.11 ± 0.002
GPyOpt (GP)	0.37 ± 0.051	1.76 ± 0.223	na
SMAC (Random forest)	0.70 ± 0.046	na	0.94 ± 0.037

B. Case Study: ML Classifiers doing Bug Hunting in Design Verification of Integrated Circuits

The goal of design verification of integrated circuits (ICs) is to test the functionality correctness by generating input signals and evaluating the resulting output against the expected values. Modern ICs may contain billions of devices, so manual design verification is no longer feasible to verify all possible functionality. Standard practice in design verification is to generate the test signal candidates using constrained-random stimulus [33]. The random input generation is controlled to allow a rich and diverse set covering the desired functionality. These inputs are simulated and monitored for bugs in the design. The bugs are then analyzed, fixed, and the entire process is repeated to verify the updated design. The input space for design verification is astronomically large, using a lot of computing using the random search. It is evident from the fact that the verification process accounts for a large fraction (50 %) of the total compute budget during development [33].

At Arm, we are using ML to increase design verification efficiency. ML models are trained to classify test candidates likely to find bugs. The test candidates are then passed through the ML filter to select the tests with a high probability of failure. This process, called bug hunting ML flow, has been deployed in production and has been shown to increase the efficiency, measured as compute cycles used to find the same number of bugs by 40 %. The overall workflow is shown in Figure 1. The bug hunting workflow requires ML models to be frequently re-trained as the design is updated or the test bench that generates the test candidates is modified. We also prefer training ML classifiers on the *entire dataset to avoid partial training errors* when comparing the hyperparameters. Overall our goal is to ensure that the compute budget for training ML models does not grow and eat into the gains made in

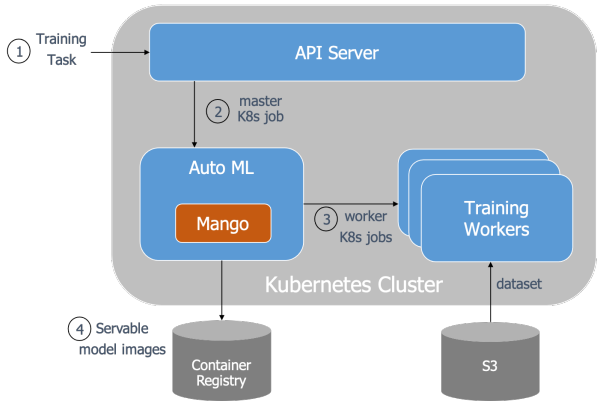


Fig. 8: Workflow of the AutoML framework using Mango for hyperparameter tuning on the Kubernetes (K8s) cluster.

the verification process. Hence our inclination for Bayesian optimization to reduce training iterations. Besides, we required the following features in the tuning library:

Deployment dependencies: The bug hunting workflow is implemented on a computing cluster using Dask distributed framework. Therefore, ideally, the hyperparameter tuning library should have the capability of being integrated with Dask without significant external dependencies. Furthermore, the library's compatibility with Scikit-learn's estimator interface would ease the integration due to the existing usage of a similar interface.

Runtime failures: Due to a cluster deployment, it is required that the library should expose abstractions to discarding the failed evaluations due to failed jobs, communication issues, or incorrect parameter values. This is critical to reducing the manual maintenance/debugging time in deployment.

CASH Problem Multiple ML classifiers are re-trained every time the training event is triggered, and the best model is chosen based on a custom metric.

The *RandomizedSearchCV* from Dask-ML partially supported these features and was used in the past ML production pipeline. However, the key missing features were efficient search and CASH. Mango provided all the required features with flexible, lightweight architecture, allowing scheduling on Dask without additional dependencies. Mango was integrated into the production ML pipeline to tune the ML models used for **Arm's ICs designs'** verification process. An extensive evaluation using Mango on six proprietary design verification benchmark datasets (3 test benches for 2 different designs) in comparison to the RandomizedSearchCV from Dask-ML showed that Mango reduces the model training iterations by an average of **45%** across all experiments, with the range being 23% - 69%.

C. Case Study: AutoML Framework

At Arm, an AutoML platform was developed to provide a simple interface for non-data scientists to train and deploy ML models. The platform is deployed on a Kubernetes cluster. The AutoML framework uses the Kubernetes Jobs API to orchestrate distributed hyperparameter tuning. The hindrances in adopting Katib and Polyaxon hyperparameter tuning frameworks built on top of Kubernetes is their dependencies on components like API server, database, and persistent storage volumes increasing the maintenance and development overhead substantially. Mango provided a lightweight and robust alternative with efficient search algorithms.

Figure 8 shows the process flow of AutoML platform. The process is initiated by a POST request to the RESTful API server with the training task's configuration data. The configuration data includes the dataset reference from S3, training type (classification, forecasting, regression), target column, performance metric, etc. The API server authenticates the request, fetches the relevant metadata from the database, and starts a master AutoML process using the Kubernetes Jobs API (Step 2). The master AutoML process is responsible for orchestrating the training task and invoking Mango for hyperparameter tuning. Mango's flexible scheduler interface is used to create parallel ML training tasks using the Kubernetes Jobs API (Step 3). Once tuning is complete, the master AutoML process saves the best model deployed as a Docker image in a container registry (Step 4). The pseudo-code of Mango used by the AutoML framework is shown in Figure 3. We use a timeout and return the partial results to make progress on the search.

D. Case Study: Network Architecture Search for TinyML Platforms used in CPS/IoT Applications

Modern CPS/IoT applications are bringing ML classifiers to microcontroller class devices. These devices, dubbed TinyML devices, have stringent hardware constraints. As a result, the neural architecture search (NAS) needs to be optimized by target hardware specifications [1] to balance accuracy and efficiency via hardware-aware NAS.

We show the use case of Mango to model the search for limited flash and RAM requirements. The search space Ω consists of neural network weights w, hyperparameters θ, network structure denoted as a directed acyclic graph (DAG) g with edges E and vertices V representing activation maps and common ML operations v (e.g., convolution, batch normalization, pooling, etc.) respectively, which act on V. The goal is to find a neural network that maximizes the hardware SRAM and flash usage within the device capabilities while minimizing the error metric.

$$f_{\text{opt}} = \lambda_1 f_{\text{error}}(\Omega) + \lambda_2 f_{\text{flash}}(\Omega) + \lambda_3 f_{\text{SRAM}}(\Omega) \tag{1}$$

where

$$f_{\text{error}}(\Omega) = \mathcal{L}_{\text{test}}(\Omega), \Omega = \{\{V, E\}, w, \theta, v\} \tag{2}$$

$$f_{\text{flash}}(\Omega) = \begin{cases} -\frac{||h_{\text{FB}}(w,\{V,E\})||_0}{\text{flash}_{\max}} \vee -\frac{\text{HIL information}}{\text{flash}_{\max}} \\ \infty, f_{\text{flash}}(\Omega) > \text{flash}_{\max} \end{cases} \tag{3}$$

$$f_{\text{SRAM}}(\Omega) = \begin{cases} -\frac{\max_{l\in[1,L]}\{||x_l||_0+||a_l||_0\}}{\text{SRAM}_{\max}} \vee -\frac{\text{HIL information}}{\text{SRAM}_{\max}} \\ \infty, f_{\text{SRAM}}(\Omega) > \text{SRAM}_{\max} \end{cases} \tag{4}$$

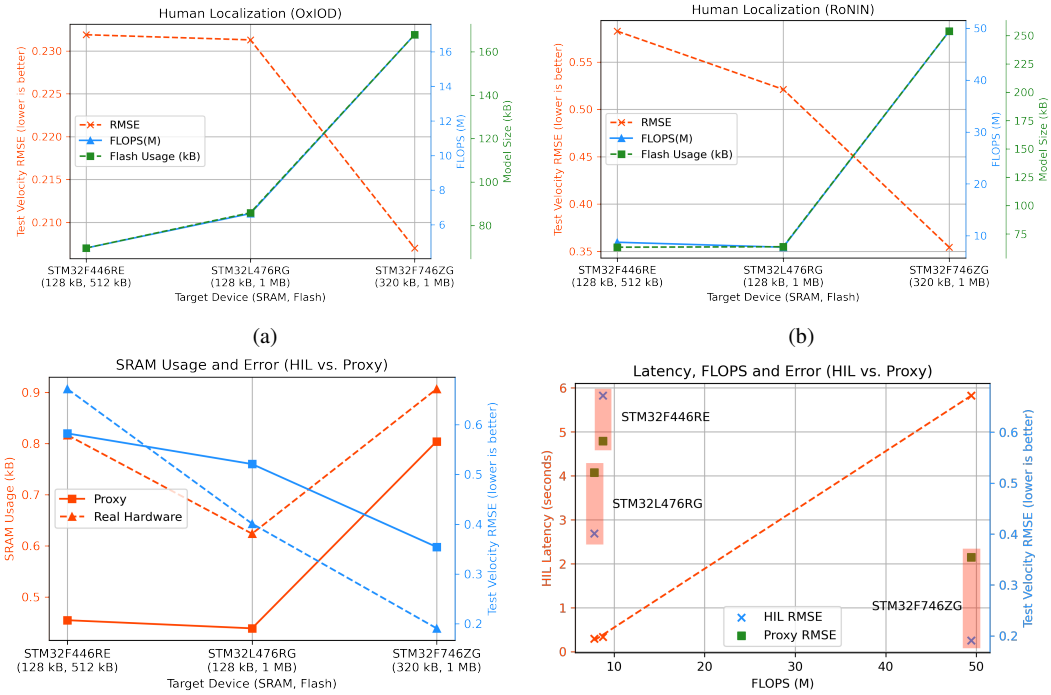


Fig. 9: Performance of Mango for hardware-aware NAS for OxIOD and RoNIN datasets. Subgraphs (a) and (b) illustrate how Mango maximizes resource usage with looser compute and memory constraints to improve error metrics for three different hardware models. Subgraph (c) shows the difference in model size and error metric with and without hardware-in-the-loop (HIL) for the RoNIN dataset on three different hardware models. Subgraph (d) shows the relation between FLOPS and latency for the RoNIN dataset and the difference in error metric with and without HIL.

$$a = w \vee y, \qquad y = \sum_{k=1}^{K} v_k g_k(x, w_k)$$

Error metric (e.g. RMSE or accuracy) serves as a proxy for the error characteristics $f_{\text{error}}(\Omega)$ of the model candidate. When real hardware is absent, we use the size of the flatbuffer model schema $h_{\text{FB}}(\cdot)$ [34] as a proxy for flash usage. Moreover, we use the standard RAM usage model as a proxy for SRAM usage $f_{\text{SRAM}}(\Omega)$, with intermediate layer-wise activation maps and tensors being stored in SRAM [1]. When hardware-in-the-loop (HIL) is available, we obtain the SRAM and flash parameters directly from the target compiler and real-time operating system (RTOS). All hardware parameters are normalized by device capacity or target metrics.

Evaluation We evaluate our NAS formulation on three ARM Cortex-M microcontrollers with different compute and memory constraints. The task is to learn the velocity regression on the Oxford Inertial Odometry (OxIOD) [35] and RoNIN datasets [36] using a temporal convolutional network (TCN) having the parameter search space definition available here [32]. The performance of a classifier is measured by average test root-mean-square error (RMSE). Figure 9 illustrates the performance of Mango in finding optimal TCN networks on the two datasets. From Figure 9a and Figure 9b, it is evident that Mango attempts to exploit the full device capabilities within the resource constraints to minimize the error metric rather than choosing the smallest model every time. Thus, as compute capability improves, the network size for the target hardware also increases. In addition, we compare the performance between using HIL and using proxies to model device constraints and error metric in Figure 9c and Figure 9d. We observe that there is a constant offset between HIL and proxies in SRAM usage, stemming from model runtime interpreter and RTOS overhead on target hardware. However, the error metric can be optimized further through HIL than proxies as compute constraints relax. The evaluation of latency in Figure 9d shows latency is proportional to FLOPS, thereby FLOPS servers a good latency proxy for microcontroller class devices.

V. Conclusion

We presented the limitations of existing hyperparameter tuning frameworks hindering their adoption in production. Mango, a black-box optimization library with flexible architecture and state-of-the-art algorithms, was designed to address these limitations. Mango is evaluated on a set of functions and classifier tuning tasks to benchmark its superior performance. Finally, case studies are examined to highlight the adoption of Mango in production ML pipelines at Arm.

ACKNOWLEDGEMENTS

The research reported in this paper was supported by a summer internship at Arm. All the UCLA authors would also like to acknowledge the support of their research from the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The material reported in Section IV-D is based on UCLA authors' research as part of the the IoBT REIGN Collaborative Research Alliance funded by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

REFERENCES

[1] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[2] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 847–855.

[3] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in science conference*. Citeseer, 2013, pp. 13–20.

[5] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, pp. 2951–2959, 2012.

[6] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, 2011, pp. 507–523.

[7] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2623–2631.

[8] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in neural information processing systems*, 2015, pp. 2962–2970.

[9] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1487–1495.

[10] T. G. authors, "GPyOpt: A bayesian optimization framework in python," http://github.com/SheffieldML/GPyOpt, 2016.

[11] T. S. authors, "Skopt: scikit-optimize," https://scikit-optimize.github.io/, 2016.

[12] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[13] J. Zhou, A. Velichkevich, K. Prosvirov, A. Garg, Y. Oshima, and D. Dutta, "Katib: A distributed general automl platform on kubernetes," in *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 55–57.

[14] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, vol. 126. Citeseer, 2015.

[15] M. Mourafiq, "Polyaxon: Cloud native machine learning automation platform," Web page, 2017. [Online]. Available: https://github.com/polyaxon/polyaxon

[16] Hyperopt, "hyperopt-mongo-worker," https://hyperopt.github.io/hyperopt/, 2019, accessed: 2021-1-29.

[17] H. Shin, "Exploiting while exploring: Effective bug discovery in unit-level verification via supervised learning," http://www2.dac.com/events/eventdetails.aspx?id=295-48, 2020.

[18] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[19] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, pp. 2546–2554, 2011.

[20] E. C. Garrido-Merchán and D. Hernández-Lobato, "Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes," *Neurocomputing*, vol. 380, pp. 20–35, 2020.

[21] J.-C. Lévesque, A. Durand, C. Gagné, and R. Sabourin, "Bayesian optimization for conditional hyperparameter spaces," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 286–293.

[22] J. González, Z. Dai, P. Hennig, and N. Lawrence, "Batch bayesian optimization via local penalization," in *Artificial intelligence and statistics*, 2016, pp. 648–657.

[23] T. Desautels, A. Krause, and J. W. Burdick, "Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3873–3923, 2014.

[24] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial Intelligence and Statistics*, 2016, pp. 240–248.

[25] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[26] M. Groves and E. O. Pyzer-Knapp, "Efficient and scalable batch bayesian optimization using k-means," *arXiv preprint arXiv:1806.01159*, 2018.

[27] R. Jenatton, C. Archambeau, J. González, and M. Seeger, "Bayesian optimization with tree-structured dependencies," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1655–1664.

[28] S. S. Sandha, M. Aggarwal, I. Fedorov, and M. Srivastava, "Mango: A python library for parallel hyperparameter tuning," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 3987–3991.

[29] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[30] V. Nguyen, S. Rana, S. K. Gupta, C. Li, and S. Venkatesh, "Budgeted batch bayesian optimization," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 1107–1112.

[31] I. Dewancker, M. McCourt, S. Clark, P. Hayes, A. Johnson, and G. Ke, "A strategy for ranking optimization methods using multiple criteria," in *Workshop on Automatic Machine Learning*. PMLR, 2016, pp. 11–20.

[32] A. Research, "Parameter search spaces use to evaluate mango on classifiers," https://github.com/ARM-software/mango/blob/master/benchmarking/Parameter_Spaces_Evaluated.ipynb, 2021.

[33] A. B. Mehta, "Constrained random verification (crv)," in *ASIC/SoC Functional Design Verification*. Springer, 2018, pp. 65–74.

[34] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[35] C. Chen, P. Zhao, C. X. Lu, W. Wang, A. Markham, and N. Trigoni, "Deep-learning-based pedestrian inertial navigation: Methods, data set, and on-device inference," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4431–4441, 2020.

[36] S. Herath, H. Yan, and Y. Furukawa, "Ronin: Robust neural inertial navigation in the wild: Benchmark, evaluations, & new methods," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 3146–3152.