SHADHO: Massively Scalable Hardware-Aware Distributed Hyperparameter Optimization

Jeff Kinnison  
University of Notre Dame  
University of Notre Dame  
Notre Dame, Indiana 46556  
jkinniso@nd.edu

Douglas Thain  
University of Notre Dame  
University of Notre Dame  
Notre Dame, Indiana 46556  
dthain@nd.edu

Nathaniel Kremer-Herman  
University of Notre Dame  
University of Notre Dame  
Notre Dame, Indiana 46556  
nkremerh@nd.edu

Walter Scheirer  
University of Notre Dame  
University of Notre Dame  
Notre Dame, Indiana 46556  
walter.scheirer@nd.edu

ABSTRACT

Computer science is experiencing an AI renaissance, in which machine learning models are expediting important breakthroughs in academic research and commercial applications. Effective training of these models, however, is not trivial due in part to hyperparameters: user-configured values that parametrize learning models and control their ability to learn from data. Existing hyperparameter optimization methods are highly parallel but make no effort to balance the search across heterogeneous hardware or to prioritize searching high-impact spaces. In this paper, we introduce a framework for massively Scalable Hardware-Aware Distributed Hyperparameter Optimization (SHADHO). Our framework calculates the relative complexity of each search space and monitors performance on the learning task over all trials. These metrics are then used as heuristics to assign hyperparameters to distributed workers based on their hardware. We demonstrate that our framework scales to 1400 heterogeneous cores and that it achieves a factor of 1.6 speedup in the necessary time to find an optimal set of hyperparameters over a standard distributed hyperparameter optimization framework.

KEYWORDS

Machine Learning, Hyperparameter Optimization, Distributed Systems, Parallel Task Management

1 INTRODUCTION

Without question, advances in high-performance computing architectures have fueled the innovation and success of computationally expensive machine learning methods. New many-core architectures allow data scientists and machine learning researchers to make full use of the large labeled datasets that are necessary for training effective models (i.e., solutions) for various learning problems. Modern machine learning tools backed by GPUs, clusters of traditional CPUs, and custom parallel hardware have enabled data-driven approaches in numerous domains, including healthcare [14], finance [8], marketing [22], autonomous vehicles [7], human biometrics [26], and many more.

Despite these successes, selecting the correct model for particular data remains a difficult problem. Model performance is highly data-specific, with different models producing wildly different results on the same dataset. However, model selection is not simply an algorithmic choice. Model searches must also account for hyperparameters: free parameters associated with a particular machine learning model that govern its ability to learn. These parameters are separate from the elementary parameters (i.e., weights) that are learned from the data, and are set before training takes place. Hyperparameters are often defined over nonlinear, non-convex spaces with many local minima, making optimization non-trivial. Choosing the best model for a particular learning task boils down to choosing the parameterized model that can accurately make predictions from new data. This is known as the hyperparameter optimization problem.

Steps have been made toward local [20, 27] and distributed [3, 18, 28] hyperparameter search. Curiously, though, these strategies do not take advantage of potential systems-based optimizations. Existing solutions do not account for the hardware being used, instead enforcing (in the case of cloud-based platforms [11, 29]) or assuming that all connected hardware is identical. A key problem that we and many other machine learning researchers have encountered is that available computing resources for distributed hyperparameter optimization are typically heterogeneous and often spread across different networks. This is the state of affairs for nearly all users outside of a handful of cloud-based providers, which have internal on-demand access to large homogeneous collections of fast hardware. In the former case, efficient hyperparameter searches must adjust the search strategy to make the best use of available hardware. Even in the latter case, though, for collections of homogeneous multi- and manycore systems, decisions may be made about the level of parallelism to exploit during the search.

While adapting distributed hardware to the search is not possible, information about the search can be used to adapt it to the
Hyperparameters are ubiquitous across different classes of machine learning algorithms, and each algorithm can have a different number of hyperparameters distributed over different numeric and categorical spaces. For example, a linear Support Vector Machine (SVM) model has a single hyperparameter (the soft-margin constant) that is incremented logarithmically within a fixed range, a Radial Basis Function SVM has an additional hyperparameter (the kernel coefficient) defined over an infinite real-valued space, a sigmoid SVM has yet another infinitely ranged hyperparameter, and a polynomial SVM has a fourth that controls the degree of the polynomial. In these examples, the search over polynomial SVM hyperparameters is combinatorially larger than that of the linear SVM, with more possible search values. The performance of each parameterized model will depend heavily on the dataset, as demonstrated by Bergstra and Bengio [1]. Moreover, the size of each search space is different by virtue of having more or fewer degrees of freedom. This information can be used to direct the search, emphasizing more complex and higher-impact search spaces to increase search throughput of those spaces.

A number of hyperparameter search strategies have been proposed to solve this optimization problem, ranging from na"ive approaches like random search [3], to more rigorous approaches like Bayesian optimization [5, 28], gradient-based learning [25], and bandit-based searches [24]. These strategies address the problem of choosing the next parameterization to test, but they operate under two major simplifying assumptions: 1) that hyperparameters have equal priority in the search, and 2) that hyperparameter search spaces are equally complex. In practice, neither of these assumptions is generally true.

In this paper, we present the Scalable Hardware-Aware Distributed Hyperparameter Optimization framework (SHADHO), a general-purpose hyperparameter optimization framework that uses approximations of model complexity and search priority as heuristics for proportionally assigning searches to heterogeneous distributed hardware. Such assignments are desirable to account for the differences between hyperparameter spaces, and moving larger, higher-impact spaces to more performant hardware increases the likelihood that such spaces will be searched more quickly. SHADHO initially directs hyperparameters to hardware by ranking structured search spaces in terms of their complexity. Then, during the search, it evaluates the performance of each space to determine impact and reassigns hardware to emphasize higher-impact spaces. SHADHO uses a search space specification language [4] to construct search trees that incorporate domain knowledge about the models being optimized. Resources are specified using the Work Queue framework [33], allowing SHADHO to scale to available distributed hardware. In summary, the contributions of this paper are:

1. A concise explanation of the hyperparameter optimization problem for the high performance computing community (Sec. 2).
2. A comprehensive survey of existing hyperparameter optimization frameworks, including an assessment of their strengths and weaknesses (Sec. 2).
3. The presentation of a new hyperparameter optimization framework (SHADHO) and its implementation details, including the structure of the search and the complexity and priority metrics used to assign hyperparameters to hardware (Secs. 3 and 4).
4. Experiments showing that SHADHO increases the throughput of high-complexity, high-impact search spaces when randomly searching hyperparameters, with comparisons made to a popular distributed random search framework [3] over 1400 heterogeneous cores (Sec. 5).
5. An analysis of SHADHO’s performance and future work to be completed (Sec. 6).
2 MACHINE LEARNING AND HYPERPARAMETERS

In general, the problem of learning from data can be described as the problem of fitting the weights of a model $M$ to a set of input data $D$ such that $M(D)$ accurately approximates the desired pattern in $D$. $M$ can either learn from $D$ by explicit examples of the pattern (supervised learning) or by finding implicit structure (unsupervised learning). In practice, this can apply to many different problems. As a case study, we will consider the problem of learning distinct classes (i.e., categories) from visual data throughout the rest of this paper.

Image classification is accomplished by selecting $M$ and training it on images containing examples of the desired object classes along with the information about the correct classification of each image. Simply selecting $M$ is not sufficient, however, because each $M$ is parameterized by a different set of free values defined over discrete and continuous domains, called hyperparameters. Applying a broad definition of the term hyperparameter, we define the set $\Lambda$ to include the choice of training, validation, and testing subsets of $D$; preprocessing steps such as normalization, feature selection, and image processing; the choice of $M$; and any other non-elementary weight aspects of learning from $D$. Thus, we pose the problem of learning from data as one of choosing the optimal set of hyperparameter values $\lambda \in \Lambda_D$, where $\Lambda_D$ is the set of all possible combinations of configurations (dataset partitions, preprocessing, model selection, and model parameterization) that can be applied to learn from $D$.

2.1 The Hyperparameter Optimization Problem.

Hyperparameter optimization is the process of selecting the set of hyperparameters that learn the most accurate approximations of patterns in $D$, in the shortest amount of time. A detailed treatment of the optimization problem is out of the scope of this work; see [1] for an in-depth description. Recent research into hyperparameter optimization has focused on the methodologies for efficiently choosing search values $\lambda$ from $\Lambda$. Traditionally, hyperparameters have been tuned by hand [21] or by grid search [13] over hyperparameter spaces. Bergstra and Bengio [1] demonstrated that randomly searching hyperparameter spaces outperforms manual and grid search by virtue of the fact that only a handful of hyperparameters have a high impact on the accuracy of $M(D)$. Subsequent work explored Bayesian optimization strategies [5, 16, 28], learning curves [12], gradient-based methods [25], and bandit-based approaches [24] to selecting search values. All of these methods are highly parallelizable due to the fact that different choices of $\lambda$ are independent of each other.

While current tools for distributed hyperparameter search and optimization implement a large number of these strategies for choosing hyperparameters, little work has attempted to optimize distributed search patterns based on the structure and performance of the hyperparameters being searched. Current distributed hyperparameter search tools tend to assume that all search spaces are equally complex and equally important. Moreover, they operate under the assumption that all connected hardware is appropriate for all parts of the search.

Based on the results of [1] and [17], it is evident that different hyperparameters are important for different algorithm choices and datasets. Moreover, different algorithms have a different number of hyperparameters to tune, meaning that different models have different search complexities. This complexity can be approximated, however, based on domain knowledge of the model. Moreover, the performance of different parameterizations over any number of searches can also be known by keeping track of search results over time. Thus, it is possible to develop heuristics for allocating hyperparameter search spaces to distributed hardware in terms of search complexity and search space impact.

2.2 Related Work in Software Frameworks for Hyperparameter Optimization.

Hyperparameter optimization is recognized as one of the major hurdles to effective machine learning, and a number of software platforms have been created to handle distributing the search. These platforms provide tools for performing random search or Bayesian optimization. One of the most prominent software packages, Hyperopt [2, 3], is built on the scientific Python stack and uses MongoDB to coordinate distributed search on arbitrary compute nodes. Hyperopt comes with the random and tree-structured Parzen estimator search strategies out-of-the-box, however it is limited to running Python programs. Optunity [9] is another Python-based platform focused on compatibility with general machine learning models, regardless of implementation language. Optunity comes with wrappers to numerous languages and a number of dependency-free search strategies, with all data transferred as JSON. Spearmint [28] performs Bayesian optimization and hooks into Sun Grid Engine, making it appropriate for a large number of cluster environments. h2o.ai [30] is a platform for general distributed machine learning that also comes with Bayesian optimization tools. All of these platforms offer distributed search strategies, but they make no decisions based on the structure of the search over different models.

Several cloud-based commercial offerings are also available. Aetros [29] provides a platform for collaborative machine learning geared toward both academic and business interests. Users may perform distributed hyperparameter optimization on Aetros servers and on their own machines. In a different vein, SigOpt [11] is a general-purpose tool for research pipelines that includes machine learning and hyperparameter optimization features. Both Aetros and SigOpt provide access to cloud-based compute nodes and allow users to deploy their own servers and workers.

While all of these platforms offer distributed search strategies, they make no decisions based on the structure of the search over or the performance of different models. Instead, they use a naive distribution strategy in which connected workers take any available job. As we will show, the naive strategy can limit throughput when searching over models with different running times or different levels of performance consistency.

Under our definition, neural network architecture search is also a hyperparameter optimization problem, with each network layer serving as a distinct hyperparameter. Zoph and Le [34] proposed a method for distributed architecture optimization, however their method requires access to 800 individual GPUs. Such a hardware requirement is infeasible outside of a relatively small number of
corporate research environments. In nearly all cases, users will have access to several smaller clusters of heterogeneous hardware.

3 STRUCTURED HYPERPARAMETER SEARCH

Hyperparameter search spaces are highly structured due to the fact that different models necessarily require different parameter sets. In the case of optimizing a single model, the search is flat. When searching across multiple models, however, the search can be structured as a tree containing a subtree for each model. The tree structure can be applied to searches over different learning algorithms, different model architectures, different kernel functions applied to the same algorithm, and other disjoint spaces. Regardless of the model, the underlying tree structure can be exploited to approximate the combinatorial size of the models and spaces being searched.

### Algorithm 1: Create Ordered Search Forest

**Algorithm:** Split

**Input:** $T$: a specification tree

**Output:** $F$: the OSF generated by splitting $T$

1. **if** $T$ is a leaf node **then**
   1. **if** $T$ is optional **then**
      1. return $\{T, \emptyset\}$
   2. return $\{T\}$

2. $F := \{\}$

3. foreach $c \in T.\text{children}$ do
   1. $F_c := \text{Split}(c)$

4. foreach $t \in F_c$ do
   1. **if** $T$ is exclusive **then**
      1. $T_{\text{copy}} := T.\text{copy\_without\_children}()$;
      2. $T_{\text{copy\_add\_child}}(t)$;
      3. $F = F \cup \{T_{\text{copy}}\}$
   2. **else**
      1. $F_{\text{new}} := \{\}$
      2. foreach $f \in F$ do
         1. $f_{\text{new}} := f.\text{copy\_with\_children}()$
         2. $f_{\text{new\_add\_child}}(t)$
         3. $F_{\text{new}} = F_{\text{new}} \cup \{f_{\text{new}}\}$
      3. $F = F_{\text{new}}$

5. **if** $T$ is optional **then**
   1. $F = F \cup \{T.\text{copy\_without\_children}()\}$

6. return $F$

To simplify searching disjoint subtrees, the search is structured as an ordered search forest (OSF). Each tree in the forest corresponds to one of the disjoint spaces being searched, and trees are ordered according to approximations of their search complexity and priority. Ordering search trees by these heuristics enables decision-making based on both intrinsic and observed features of the search, allowing the search to be directed proportionately.

3.1 Splitting Search Trees

The OSF is specified by a single $n$-ary tree, where $n$ is variable across subtrees, with only minor additions to standard tree nodes. Within the specification tree, subtrees may be marked as exclusive or optional. An exclusive subtree defines spaces that are only searched individually; an optional space may be excluded from the search entirely. Both of these flags indicate a split point at which multiple disjoint trees are defined. The method for creating the OSF is presented in Algorithm 1. While our method is similar to the method presented in [17] for partitioning random forests of individual hyperparameters, we generalize beyond binary trees and focus on partitioning distinct models.

Figure 2 shows an example of the splitting process. In the example, subtree $A$ is included in its entirety or excluded completely, and at most one child of subtree $B$ is included in any tree. The splitting method enables arbitrary search spaces to be defined: subtree $A$ could correspond to a preprocessing step that should either be included or excluded entirely, and subtree $B$ can represent independent models being optimized.

Each tree in the OSF is uniquely defined by its set of leaf nodes. The leaves contain the set of discrete (finite) and continuous (infinite) search spaces from which hyperparameter values are drawn. Discrete spaces allow searches to include categorical hyperparameter values such as coefficients, optimizers, weight initialization functions, etc. Continuous spaces are real-valued intervals fit to probability distributions, enabling searches over train/test split ratios, learning rate, similarity measures for kernels, etc. Generating a parameterization involves drawing values from the spaces contained within the leaf nodes.

3.2 Complexity

Splitting the specification tree into an OSF ideally creates a set of differently-sized search spaces in terms of both degrees of freedom and individual search space domain sizes. Recognizing that different models are parameterized by different hyperparameters, search trees can be ranked by the complexity of the searches they define. The size of a search is dependent on the individual spaces from which values are drawn. We define the complexity of a search space $s$ as

$$C(s) = \begin{cases} 1 + \left( 1 - \frac{1}{n} \right) & \text{if } s \text{ is discrete} \\ 2 + \|b - a\| & \text{if } s \text{ is continuous} \end{cases}$$

where $[a, b]$ is the interval containing 99% of the probability distribution governing $s$. $C(s)$ ensures that discrete spaces are always smaller than continuous spaces while maintaining the order of discrete spaces relative to one another. Moreover, $C(s)$ approximates the size of infinite continuous spaces while also maintaining their relative order. Once $C(s)$ is calculated for each leaf in the tree, the complexity of the entire search tree is the product of its leaf node complexities.

3.3 Priority

The performance of a set of hyperparameter values applied to a model varies with training data and cannot be known a priori. Instead, priority must be learned as hyperparameter values are tested by determining the covariance between values. Algorithm 2
Figure 2: Splitting a specification tree into a set of disjoint trees. (a) The specification tree contains two subtrees: an optional subtree $A$ and an exclusive subtree $B$. The root level indicates that all subtrees should be included if possible. (b) The trees created by splitting the specification tree in (a). Because $A$ is flagged as optional, trees (1) and (2) include both children of $A$, while $A$ is excluded entirely from trees (3) and (4). Additionally, each of the four trees contain only one child of $B$. In terms of hyperparameter search, $A$ corresponds to an operation that is global across models, such as a preprocessing step, and $B$ corresponds to the disjoint models being searched.

Algorithm 2: Priority of a Search Tree

| Input: $T$: a hyperparameter search tree |
|------|
| Input: $n$: the number of times to fit to $T$ |
| Output: $p$: the priority of $T$ |

$scales := \{\}$;  
for $i$ in $1..n$ do  
  $l := uniform(0.1, 2.0)$;  
  $gp := RBFGaussianProcess(l)$;  
  $gp.fit(T.values, T.results)$;  
  $l_{opt} := gp.length_scale$;  
  $scales = scales \cup \{l_{opt}\}$;  
$p := ||\text{max}(scales) - \text{min}(scales)||$;  
return $p$

presents the procedure for calculating priority, inspired by the process of identifying high-impact hyperparameters in [1]. This process is similar to the tree-structured Parzen estimators approach presented by Bergstra et al. [5], however we use the result to shift search proportions rather than generate hyperparameter values. A similar ANOVA-based importance metric is reported in [17], however this metric is used to partition individual hyperparameter spaces in a random forest search, whereas we use a measure of model parameterization covariance to adjust search proportions at the model level.

For a given tree, a Gaussian Process (GP) with radial basis function (RBF) kernel is fit $n$ times to the set of tested hyperparameters and the corresponding model performance. During fitting, the length scale $l$ of the RBF kernel is tuned to maximize the log marginal likelihood of the GP. From the set of $n l$-values, the priority is defined as the magnitude of the range of tuned length scales encountered. Rather than being a measure of the best or average performance of a model across tested parameterizations, priority approximates the effect of changing hyperparameters on model performance. If the model performs consistently, regardless of the quality of results, then its priority is lower. Increasing variation in performance leads to a higher priority.

While it may be desirable from a practical standpoint to prioritize searching high-performing models, a consistently high-performing model is not a good search target. Consistent performance indicates some intrinsic fitness of the model for the learning problem. Rather, a model with high variation in performance across parameterizations should be searched more thoroughly because its fitness for the learning task is not immediately apparent from its performance. Priority identifies the trees that define models with varying performance so that they can be searched proportionally more than those with consistent performance.

4 SHADHO FRAMEWORK

SHADHO is a Python framework for distributed hyperparameter optimization structured as an OSF. Internally, SHADHO orders search trees by their complexity and priority heuristics and then assigns each a preferred type of hardware based on the ordering. Higher-complexity, higher-priority trees are scheduled to prefer more performant hardware, and each tree is limited to a subset of available hardware to ensure that low-complexity, low-priority spaces are not ignored. In this way, SHADHO adjusts to the search and scales to available hardware, enabling searches over heterogeneous clusters and environments — a scenario that is familiar to most users of machine learning algorithms.
Work Queue workers are pilot jobs which are persistent processes worker processes. The master coordinates the workers which may work which consists of a centralized master process and distributed with a higher scheduling priority than other spaces. The key differ-

A critical component of SHADHO is the capability to prioritize be running on a variety of machines in clusters, clouds, or grids. Work Queue uses a master-worker frame-

4.2 Work Queue for Task Management

A critical component of SHADHO is the capability to prioritize certain high-impact search spaces over those with a lower impact. A similar concept has been explored in the distributed task sched-

4.1 Complexity-Centric Scheduling

Though prioritizing high-impact search spaces is a core component of our work, the impact of search spaces we consider will also affect what spaces will be viable for search later on. In [6], the authors built upon the HEFT algorithm presented in [31] to more explicitly consider the effect scheduling certain tasks before others will have on the performance of the DAG workflow. SHADHO makes decisions in a similar fashion since we must consider how choosing to process one high-impact search space may alter the order in which we look into other spaces and the overall quality of our optimization.

Work Queue has several performance optimizations that other hyperparameter optimization applications may not be able to support. First, each worker has a local cache which persists for the worker’s lifetime. The cache prevents common input files from being transferred excessively, which in turn will reduce the time it takes for a task to complete. Work Queue also tries to match tasks to workers which already have some or all relevant input files present in their respective caches. Workers and tasks also support specifying hardware resource requirements. For the worker process, the resource requirements may consist of any or all of: cores, memory, disk, and GPUs. This ensures that the batch system managing the workers will only land each worker process on a machine with at least the minimum requested resources. A worker can also advertise an arbitrary feature to the master (i.e., the CPU model). For a task, the resource specifications ensure the master only dispatches tasks to workers that can handle them. Task resource specification includes the same resources as worker (cores, memory, disk, and GPUs), but they may also request any arbitrary feature. By requesting an arbitrary feature, we provide a soft guarantee that the master will dispatch tasks to workers with the matching feature.

SHADHO uses Work Queue’s hardware requirements specification to direct searches to hardware based on complexity and priority. Connected hardware is grouped into compute classes, sets of workers that broadcast a common resource. Compute classes are specified at runtime by the name and value of the resource they share. For example, workers can be grouped into classes containing 4, 8, and 16 cores, or a specific CPU or GPU model. Each search tree is then assigned to one of the specified classes, with assignments updated over the course of the search.

4.3 Implementation Details

SHADHO is free and open-source, and available for download from GitHub. It is written in Python with compatibility between language versions 2.7 and 3.x. The package contains the following modules:

- shadho.shadho: driver for the distributed hyperparameter search; manages task submission, the OSF, and Compute Classes.
- shadho: forest: an implementation of the OSF that takes a specification (Listing 1) and splits it into a forest following Algorithm 1.
- shadho.tree: individual tree node structures for managing complexity, priority, and search spaces.
- shadho.hardware: object interface for specifying compute classes as key/value pairs (Listing 2).

SHADHO Repository: https://github.com/jeffkinnison/shadho
With defined specification trees and compute classes, SHADHO can begin distributing tasks (searches) to connected workers. The user can create helpers on a case-by-case basis to aid in reuse and ing these general specifications, domain- and environment-specific knowledge about both the search and computing environments. Using key/value pairs to represent the desired hardware resource, as used in the same way to define subtrees that may be excluded.

Listing 1: Example SHADHO OSF Specification for SVM

```python
from shadho import randint, ln_uniform
from shadho import uniform
from shadho import SearchForest

C = ln_uniform(0, 15)
gamma = uniform(0, 1000)
coef0 = uniform(-1000, 1000)
degree = randint(0, 15)

spec = {
    'exclusive': True,
    'linear': {
        'C': C
    },
    'rbf': {
        'C': C,
        'gamma': gamma
    },
    'sigmoid': {
        'C': C,
        'gamma': gamma,
        'coef0': coef0
    },
    'poly': {
        'C': C,
        'gamma': gamma,
        'coef0': coef0,
        'degree': degree
    }
}

SearchForest(spec)
```

Listing 2: Example SHADHO Compute Class Specification

```python
from shadho import ComputeClass

ccs = [
    # name, key, val, no. expected workers
    ComputeClass('smp16', 'cores', 16, 50),
    ComputeClass('smp8', 'cores', 8, 50),
    ComputeClass('smp4', 'cores', 4, 50)
]
```

Ultimately, SHADHO must be able to adapt to the search problem and available hardware to be useful in general. We allow the user to specify both of these aspects of the search to leverage domain knowledge about both the search and computing environments. Using these general specifications, domain- and environment-specific helpers can be created on a case-by-case basis to aid in reuse and experiment replication.

4.5 Running SHADHO

With defined specification tree and compute classes, SHADHO can begin distributing tasks (searches) to connected workers. The user may specify that SHADHO use both heuristics, priority only, complexity only, or no heuristics to affect scheduling; the last case is undirected random search, equivalent to random search in Hyperopt [3]. Workers are instantiated by starting a Work Queue Worker on an individual compute node. Work Queue Workers are long-running processes that receive files and commands from the master and have the ability to cache files between assigned tasks to reduce network communication overhead. These workers may be managed en masse by using a Work Queue Factory to interface with a batch
system and specify a desired number of workers and hardware requirements; the Work Queue Factory will then attempt to maintain a pool of Work Queue Workers that meet the specification. Workers will automatically connect to SHADHO and receive tasks.

As SHADHO receives results from workers, it recalculates search tree priorities and adjusts search proportions accordingly. SHADHO will continue to distribute tasks until either 1) a user-specified amount of time has passed, or 2) a user-specified number of searches have completed. Once either of these conditions is met, SHADHO shuts down all existing workers and reports the optimal observed hyperparameterization.

5 EXPERIMENTAL EVALUATION

To demonstrate the advantages of adjusting search proportions to priority and complexity, we compared SHADHO to Hyperopt [3], a widely-used framework for distributed hyperparameter optimization. Both frameworks applied random search to the hyperparameter optimization task: choosing the optimal SVM [10] kernel and parameterization for classifying handwritten digits. We chose SVM as a relevant case study for this paper because it is used extensively across numerous problem domains, and is a representative example of a machine learning algorithm that learns a set of weights from data. It also serves as the most popular readout layer for convolutional neural networks [15], the current dominant approach to visual recognition. The experiments presented in this section not only establish feasibility for the proposed approach, but also lay the groundwork for an important real-world use-case.

SVM classifiers are machine learning algorithms that learn decision boundaries between different classes of data. The shape of this decision boundary is determined by the kernel function used to evaluate the data and the hyperparameterization of the chosen kernel. SHADHO and Hyperopt were applied to the problem of selecting the optimal SVM model for classifying handwritten digits from the MNIST dataset [23]. MNIST is a standard dataset for testing machine learning models, consisting of 60,000 28x28 pixel images of handwritten digits. We search over four kernel functions — linear, RBF, sigmoid, and polynomial — and their respective parameterizations.

The linear kernel has the fewest hyperparameters, requiring only the soft-margin constant ($C$). $C$ was defined over a uniform distribution from 0 to 15 and scaled logarithmically. The RBF kernel is also parameterized by $C$, but adds a second hyperparameter, the kernel coefficient ($\gamma$), which we defined uniformly between 0 and 1000. The sigmoid kernel adds an additional coefficient ($r$), and the polynomial kernel uses a fourth hyperparameter to define the degree of the polynomial ($d$). $r$ is defined uniformly from $-1000$ to $1000$, and $d$ takes on integer values from 1 to 15. Figure 4 demonstrates how the kernels are split within the OSF.

To prepare MNIST for classification, the images were flattened, resulting in an array of 784 integer values for each image. These values were then normalized so that all pixel values were in the range [0, 1]. The set of labels for the images consists of an array of 70,000 digits 0-9, corresponding to the digit each image portrays. After classification, the loss (i.e., error), accuracy, precision, and recall statistics were calculated. SHADHO minimized this loss value and used it to calculate precision. Moreover, timing and usage statistics were recorded by Work Queue.

The search was conducted using five different platforms: Hyperopt’s random search, SHADHO in heuristic-free mode (equivalent to Hyperopt’s random search), SHADHO using complexity only, SHADHO using priority only, and SHADHO using both complexity and priority. Each of these configurations conducted distributed SVM kernel optimization five times to in order to calculate error statistics. All five searches were conducted over one hour using 50 4-core, 50 8-core, and 50 16-core machines in a production network environment.

5.1 Results & Discussion

Our SVM experiment demonstrates that using the complexity and priority heuristics to adjust search proportions has a large impact on the search. Table 1 shows the average performance of the four SHADHO platforms and Hyperopt over five trials. In terms of the quality of the optimized model, SHADHO matched or exceeded Hyperopt in every case. The major advantage of SHADHO lies in the time needed to find an optimal model: every platform except one outperformed Hyperopt on average. Moreover, SHADHO consistently used more parallel CPU time while finding an optimal model in less wall-clock time on average, indicating that its heuristic decision-making exploits parallelism more effectively than naïve scheduling.

On average, SHADHO using both heuristics achieved a speedup of 1.6 over Hyperopt in the time taken to find an optimal set of hyperparameters. This speedup was made possible by proportionally searching more high-complexity, high-priority kernels on hardware that can exploit more parallelism. The proportional split for each platform is shown in Figure 5. While Hyperopt searches the kernels relatively evenly, SHADHO using both heuristics clearly emphasizes searching the more complex kernels. In these short tests, complexity is the dominant heuristic, as evidenced by similarity of plots (c) and (e) in Figure 5. Priority had a minimal individual effect on proportions in this experiment, however when combined
| Platform          | Accuracy (%) | Optimization Time (s)   | CPU Time (s) | Tasks Per Compute Class | Max Workers |
|------------------|--------------|-------------------------|--------------|-------------------------|-------------|
| Hyperopt [3]     | 96.95        | 1505.6 ± 383.1          | 50401        | 44, 51, 50              | 50, 50, 50  |
| Heuristic-Free   | 96.96        | 1165.6 ± 599.8          | 122091       | 51, 78, 87              | 50, 50, 50  |
| Complexity-Only  | 97.05        | 2247.5 ± 187.1          | 129717       | 22, 83, 137             | 50, 50, 50  |
| Priority-Only    | 96.98        | 1155.6 ± 185.1          | 142397       | 52, 80, 77              | 50, 50, 50  |
| Both Heuristics  | 97.03        | 956.4 ± 334.4           | 162765       | 21, 123, 165            | 50, 50, 50  |

Table 1: Average accuracy and average time to finding the optimal model over five trials for each platform. Times are listed with their standard error, and all accuracies have a standard error < 10^{-3}. The average total CPU time is the aggregate time of all tasks run during a search. Finally, the average number of tasks run on and the max number of workers connected from each compute class (4-, 8-, and 16-core, respectively) per trial are shown.

Figure 5: The proportion of searches allocated to each kernel function. (a) Hyperopt searched roughly evenly between linear and RBF, and between sigmoid and polynomial. (b) SHADHO’s heuristic-free mode searched in a pattern similar to Hyperopt’s by virtue of also being undirected random search. (c) The complexity metric on its own created a strong proportional ordering that emphasized kernels with larger search spaces over smaller ones. (d) The priority-only search learns its ordering over time, and so resembled heuristic-free due to the relatively short duration of the experiment. A greater proportion of searches were conducted on the sigmoid and linear kernels, however, indicating their high priority. (e) When using both heuristics, the kernels were initially ordered by their complexity. As priorities were learned, kernels were reordered based on both metrics. Because the polynomial kernel had both high complexity and high priority, it was searched much more than the other three.
with complexity it emphasized the kernels with the larger search space.

In terms of the number of searches conducted, SHADHO far exceeded Hyperopt. Figure 6 shows the distribution of running times for Hyperopt and SHADHO using both heuristics. Here and in Figure 5, SHADHO clearly showed a preference for the kernels with the larger search spaces. However, the distributions make it clear that the increased emphasis on larger search spaces was not at the expense of smaller ones. SHADHO searched these spaces as frequently as Hyperopt over all five trials, meaning that SHADHO incurs minimal loss to small, low-impact spaces by shifting search proportions using both complexity and priority. Moreover, based on the counts of tasks run on each compute class, this compensation occurs because of the shift in priority: SHADHO completed far fewer tasks on 4-core machines, but made up for the difference by allowing the lower priority tasks to run on the 8-core machines at a lower proportion.

With these experiments, we have shown that managing parallel tasks using heuristic measures of the size of a task and its relative importance can increase the throughput of parallel and distributed workflows. Moreover, by applying domain knowledge to structure both the set of tasks and the computing environment on which they will run, it is possible to make use of a wide range of hardware with less risk that a task will run on unsuitable hardware. Minimizing this risk allows us to make full use of available computing hardware. Even in the case of homogeneous multi-core distributed hardware (e.g., cloud environments, single clusters), it is possible to use SHADHO to control the level of parallelism exposed to particular tasks through OSF and Compute Class specifications.

6 CONCLUSION

SHADHO has met and exceeded the performance of the state-of-the-art in distributed hyperparameter optimization. By taking advantage of information about the structure and performance of the search, we are able to increase the throughput of high-complexity, high-impact spaces without decreasing the throughput of other spaces. This opens the door to efficiently searching more complex hyperparameter optimization problems by virtue of scaling the search to available hardware. Specifying arbitrary hardware requirements with Work Queue allows such searches to account for different CPU counts and architectures, different GPU models, and even non-GPU accelerators and custom hardware. While our case study was focused on hyperparameter optimization, SHADHO can potentially be applied to other domains with parallel tasks of differing complexity and priority. This particular type of distributed task management is well-suited to the academic domain, in which researchers will most likely have access to heterogeneous hardware.

In terms of next steps for the platform, the current implementation of SHADHO only makes use of a random hyperparameter search strategy. Recent studies have introduced promising search strategies involving Bayesian optimization and bandit-based search, and future versions of SHADHO will incorporate these and forthcoming search strategy developments. Additionally, the problem of automated neural network architecture construction has been broached in recent deep learning work [32, 34]. One of the prime benefits of SHADHO is the ability to dynamically reallocate work to

Figure 6: Running time distribution for each of the four kernel functions from the SHADHO (blue) and Hyperopt (red) runs. These distributions contain the running times of all searches from all five trials. In every case, SHADHO (using both heuristics) conducts more total searches than Hyperopt, and in the case of the polynomial and linear kernels the searches tend to lower running times.
appropriate hardware. Neural network architecture search involves testing neural networks of varying complexities and running times, making SHADHO a suitable framework for exploring and scaling their architectures.

The ability to effectively allocate hyperparameters to hardware is central to SHADHO’s work. We will continue to explore other methods for approximating complexity and priority, including methods for understanding running time across heterogeneous hardware (complexity) and comparative performance metrics between models and search spaces (priority). SHADHO will allow us to explore these methods at a massive scale and advance both machine learning and distributed computing.

ACKNOWLEDGMENTS

This research was supported in part by the Notre Dame Center for Research Computing through access to distributed computing resources. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. Funding was provided under IARPA contract #D16PC00002.

REFERENCES

[1] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research 13, Feb (2012), 281–305.

[2] James Bergstra, Brent Komer, Chris Ellismith, Dan Yamins, and David D Cox. 2015. Hyperopt: a python library for model selection and hyperparameter optimization. Computational Science & Discovery 8, 1 (2015), 014008.

[3] James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In Proceedings of the 12th Python in Science Conference. Citeseer, 13–20.

[4] James Bergstra, Dan Yamins, and David D Cox. 2013. Making a science of model search. In International Conference on Machine Learning.

[5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Al-gorithms for hyper-parameter optimization. In Advances in Neural Information Processing Systems. 2546–2554.

[6] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. 2010. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. 27–34. DOI: http://dx.doi.org/10.1109/PDP.2010.56

[7] Mariusz Bojaranski, Davide Del Testa, Danila Dworakowski, Bernhard Firner, Bet Plepp, Prasoon Goyal, Lawrence D. Jacket, Matthew Monfort, Ulf Muller, Jiaqiu Zheng, Yan Zhang, Jake Zhao, and Karol Zarka. 2016. End to End Learning for Self-Driving Cars. CoRR abs/1604.07316 (2016). http://arxiv.org/abs/1604.07316

[8] L. J. Cao and F. E. H. Taylor. 2003. Support vector machine with adaptive parameters for time series forecasting. IEEE Transactions on Neural Networks 14, 6 (Nov 2003), 1506–1518. DOI: http://dx.doi.org/10.1109/TNN.2003.820556

[9] Marc Claesen, Jaak Simm, Dusan Popovic, and BD Moor. 2014. Hyperparameter tuning in Python using Optunity. In Proceedings of the International Workshop on Technical Computing for Machine Learning and Mathematical Engineering.

[10] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. Machine learning 20, 3 (1995), 273–297.

[11] Ian Dewancker, Michael McCourt, Scott Clark, Patrick Hayes, Alexandra Johnson, and George Ke. 2016. Evaluation System for a Bayesian Optimization Service. arXiv preprint arXiv:1605.06170 (2016).

[12] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In IJCAI 3460–3468.

[13] Kao-Bi Duan and S Sathiya Keerthi. 2005. Which is the best multiclass SVM method? An empirical study. In International Workshop on Multiple Classifier Systems. Springer, 278–285.

[14] Andre Esteve, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. Nature 542, 7639 (2017), 115–118.

[15] Rob Fergus. 2013. Deep Learning for Computer Vision. (2013). http://cs.nyu.edu/~rgarcia/papers/icip2013_final.pdf Tutorial Presented at NIPS 2013.

[16] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. 2015. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In AAM. 1128–1135.

[17] Holger Hoos and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. (2014).

[18] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2013. An evaluation of sequential model-based optimization for expensive blackbox functions. In Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. ACM, 1209–1216.

[19] Eric Jones, Travis Oliphant, Pearu Peterson, and others. 2001.– SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/ [Online; accessed 2017-02-23].

[20] Lars Kotheoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2016. Auto-Weka 2.0: Automatic model selection and hyperparameter optimization in WEKA. Journal of Machine Learning Research 17 (2016), 1–5.

[21] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In Proceedings of the 24th international conference on Machine learning. ACM, 473–480.

[22] Steve LaValle, Eric Lesser, Rebecca Shockley, Michael S Hopkins, and Nina Kruschwitz. 2011. Big data, analytics and the path from insights to value. MIT Sloan management review 52, 2 (2011), 21.

[23] Yann LeCun, Corinna Cortes, and Christopher J BGorges. 1998. The MNIST database of handwritten digits. (1998).

[24] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient hyperparameter optimization and infinitely many armed bandits. In ICML 2016 workshop on AutoML (AutoML 2016).

[25] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016).