DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks

Prasanna Balaprakash, Michael Salim, Thomas D. Uram,
Venkat Vishwanath, and Stefan M. Wild
Mathematics & Computer Science Division and Leadership Computing Facility
Argonne National Laboratory, Lemont, IL 60439
Email: {pbalapra,msalim,turam,venkat,wild}@anl.gov

***Abstract*—Hyperparameters employed by deep learning (DL) methods play a substantial role in the performance and reliability of these methods in practice. Unfortunately, finding performance-optimizing hyperparameter settings is a notoriously difficult task. Hyperparameter search methods typically have limited production-strength implementations or do not target scalability within a highly parallel machine, portability across different machines, experimental comparison between different methods, and tighter integration with workflow systems. In this paper, we present DeepHyper, a Python package that provides a common interface for the implementation and study of scalable hyperparameter search methods. It adopts the Balsam workflow system to hide the complexities of running large numbers of hyperparameter configurations in parallel on high-performance computing (HPC) systems. We implement and study asynchronous model-based search methods that consist of sampling a small number of input hyperparameter configurations and progressively fitting surrogate models over the input-output space until exhausting a user-defined budget of evaluations. We evaluate the efficacy of these methods relative to approaches such as random search, genetic algorithms, Bayesian optimization, and hyperband on DL benchmarks on CPU- and GPU-based HPC systems.**

I. Introduction

Deep learning (DL) algorithms typically require user-specified values for hyperparameters, which strongly influence performance factors such as training time and prediction accuracy [1]–[3]. These hyperparameters include the number of hidden layers, the number of units per layer, sparsity/overfitting regularization parameters, batch size, learning rate, type of initialization, optimizer, and activation function specification. Traditionally, in machine learning (ML) research, finding performance-optimizing hyperparameter settings has been tackled by using a trial-and-error process or by brute-force grid/random search. However, such approaches lead to far-from-optimal performance or are otherwise impractical for addressing large numbers of hyperparameters. Recently, new hyperparameter search methods have emerged to address the issue of finding high-quality hyperparameter configurations in short computation time. These methods typically focus on single-node and small-cluster environments; therefore, methods that scale to large leadership-class supercomputing systems are limited.

The highly empirical nature of research in DL has been criticized recently, and the lack of systematic hyperparameter tuning in several published DL method comparisons has been cited as a key factor [4]. This issue is further exacerbated by the fact that the design and development of hyperparameter search methods are still an evolving research area, and which search method is best suited for a particular problem setting and scale remains unclear. The optimization research community faces high startup costs associated with developing mathematical formulations of the search problems and search infrastructure to test new ideas rapidly. An easy-to-use hyperparameter search infrastructure with well-defined DL hyperparameter search problems can address this issue. Empirical results and insights obtained with the search infrastructure can provide recommendations for search methods based on particular problem characteristics. Given a new DL application, a DL researcher can adopt a search method based on a particular problem setting. Currently, doing so is challenging, however, because existing open-source DL search codes and infrastructures are method-centric and ill-suited for experimental research in DL search methods. Open-source hyperparameter search implementations tend to exist as research prototypes or do not target scalability and portability concerns.

Large high-performance computing (HPC) clusters and leadership-class supercomputing systems pose a number of deployment and portability challenges. For example, the queuing systems, scheduling policies, and scripts needed to run hyperparameter searches can differ significantly from one machine to another. Establishing the appropriate runtime environment requires linking DL backends to properly optimized DL libraries, which are configured differently for GPUs than they are for multicore CPUs. Typically, search parallelism is implemented with message-passing logic (e.g., MPI) built into the search application layer. While flexible, this approach requires search algorithm developers to handle myriad system-level obstacles [5], often related to the shared filesystem or unexpected interactions between Python and the MPI/networking stack. The resulting solutions are usually fragile with low portability; add complexity to the search codebase; and take significant time away from search algorithm development, leading to lower productivity. Moreover, the large volume of training data that needs to be loaded for every hyperparameter evaluation demands methods that can take advantage of specialized compute hardware, such as high-bandwidth memory or flash storage devices, in order to avoid I/O bottlenecks.

Addressing these challenges requires a custom workflow and parallel job execution system, tuned for high-performance, asynchronous parallel task execution. By providing a simple interface for task queueing, HPC system complexities are effectively abstracted away from the hyperparameter search application layer.

In this paper we present DeepHyper, a scalable Python package for deep neural network (DNN) hyperparameter search. The key contributions of the paper are as follows:

- A collection of extensible and portable DNN hyperparameter search problem instances, each consisting of a configurable DNN code and a search space of tunable hyperparameters.
- A generic interface between DNN hyperparameter search methods and parallel task execution engines, enabling rapid implementation and testing on leadership-class supercomputers.
- A scalable, asynchronous model-based search method (AMBS) using Bayesian optimization (BO) ideas.
- Demonstration of AMBS efficacy compared with batch-synchronous methods (genetic algorithm and hyperband) and other asynchronous methods (random search and BO with Gaussian process regression).
- A portable workflow system for parallel, asynchronous evaluations of hyperparameter configurations at HPC scale (up to 1,024 nodes), designed for fault tolerance and parallel efficiency.

II. Hyperparameter search problem

We let $\mathcal{X} = (\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P})$ define a DNN configuration, where $\mathcal{X}_\mathcal{A}$ defines a network topology and $\mathcal{X}_\mathcal{P} = (\mathcal{X}_c, \mathcal{X}_d, \mathcal{X}_n) \in \mathcal{D}$ defines the hyperparameters of the DNN, which we explicitly partition into continuous ($\mathcal{X}_c$), discrete ($\mathcal{X}_d$), and nonordinal parameters ($\mathcal{X}_n$) that belong to a compact decision set $\mathcal{D}$. Let w be the weights of the DNN. The hyperparameter search problem depends on the given training data $\mathcal{T}$ and validation data $\mathcal{V}$.

The problem of finding the hyperparameters of DNNs can then be formulated as a bilevel mathematical optimization problem. For a given topology $\mathcal{X}_\mathcal{A}$ and set of hyperparameters $\mathcal{X}_\mathcal{P}$, the lower-level problem is to train the DNN:

$$\text{solve} \quad \underset{w}{\text{minimize}}\ \text{err}_T\left([\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}]; \mathcal{T}; w\right), \tag{1}$$

where $\text{err}_T\left([\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}]; \mathcal{T}; w\right)$ is the training error obtained on dataset $\mathcal{T}$. In the upper-level problem, we seek $\mathcal{X}_\mathcal{A}$ and $\mathcal{X}_\mathcal{P}$ to minimize the error obtained on the validation set $\mathcal{V}$:

$$\text{solve} \quad \underset{\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}}{\text{minimize}}\ \text{err}_V\left([\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}]; \mathcal{V}; w^*\left[\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}\right]\right), \tag{2}$$

where $w^*\left[\mathcal{X}_\mathcal{A}, \mathcal{X}_\mathcal{P}\right]$ is obtained from (1). Such a formulation allows for different forms of training and validation metrics. In practice, these functions typically are the same, $\text{err}_T = \text{err}_V$, with output differing based only on the data used.

In this paper, we focus on tuning $\mathcal{X}_\mathcal{P} = (\mathcal{X}_c, \mathcal{X}_d, \mathcal{X}_n)$, which are modeled as a vector of decision variables. The continuous hyperparameters $\mathcal{X}_c$ include hyperparameters chosen from a continuum with lower and upper bounds such as dropout and learning rate. The discrete hyperparameters $\mathcal{X}_d$ can model any discrete parameters with a natural ordering. These hyperparameters are expressed with either lower and upper bounds or a list of ordered discrete values. Examples include number of epochs, batch size, number of hidden layers, and number of units per layer in multilayered perceptrons (MLP), convolution filter and pooling sizes in convolutional neural networks (CNNs). The nonordinal, or nonordinal categorical, hyperparameters $\mathcal{X}_n$ do not possess a meaningful ordering. These hyperparameters are expressed as a list of values. Examples include type of activation functions, type of training optimizer, and type of layers (e.g., in recurrent neural nets (RNNs): vanilla RNN, LSTM, GRU).

III. The DeepHyper Package

The DeepHyper package comprises three Python subpackages: `benchmarks`, a collection of DL hyperparameter search problems; `search`, a set of search algorithms (including AMBS) for DL hyperparameter search; and `evaluators`, an infrastructure for evaluating hyperparameter configurations on HPC platforms.

A. Benchmarks

The `benchmarks` subpackage comprises several instances of hyperparameter search problems. Each instance consists of a model and a search space. The model file contains configurable DNN code that receives the hyperparameters via a command line, loads the data, builds the DNN model, runs the training, and returns the validation error. One can perform a standalone run of any benchmark model by providing a single hyperparameter configuration via the command line to the model script. The model file follows the template in source code 1.

Source code 1 Benchmark model template

```
class BenchmarkProblem():
  def __init__(self, param_dict):
  '''parse command line parameters'''
  self.read_params(param_dict)
  '''load the data'''
  self.load_data()
  ''' preprocess data '''
  self.preprocess_data()
  '''training/validation data split'''
  self.split_data()
  '''build the model'''
  self.build_model()
  '''train the model'''
  self.train_model()
  '''evaluate the model performance'''
  self.evaluate_model()
  '''return the objective value'''
  self.return_result()
```

The model code is self-instrumented and logs the Python module import time, which can become significant in shared filesystem environments, in addition to the time spent in each

function in source code 1. Moreover, the code has a graceful termination that stops the training based on a user-defined timeout value and passes control from self.train_model() to self.evaluate_model() and self.return_result(). The benchmark run returns the validation error to the search.

The search space description file follows the template shown in source code 2. The lower and upper bounds for the continuous and discrete integer hyperparameters are given as Python tuples. The discrete noninteger and nonordinal hyperparameters are given as lists of values. To extend the search space by exposing more hyperparameters is straightforward: one simply adds a named hyperparameter to the search space file and ensures that a command-line argument with this name is correctly parsed by the model file. The `starting_point` variable contains the initial hyperparameter configuration that will be evaluated first by the search methods.

The standardized search space description and model run interface provide significant modularity, in the sense that one can easily plug a new benchmark into DeepHyper and immediately apply any of the supported hyperparameter search algorithms without further modification to DeepHyper.

Source code 2 Search space template

```
class Space():
  def __init__(self):
      space = collections.OrderedDict()
      space['epochs'] = (5, 500)
      space['nhidden'] = (1, 2)
      space['nunits'] = (5, 100)
      space['activation'] = ['relu', 'softmax']
      space['batch_size'] = [8, 16, 32, 64, 128]
      space['dropout'] = (0.0, 1.0)
      space['optimizer'] = ['sgd', 'rmsprop']
      space['learning_rate'] = (1e-04, 1e01)
      self.space = space
      self.starting_point = [5, 1, 5,'relu', 8,
                             0.0,'sgd', 1e-04]
```

B. Search

The `search` subpackage contains several modules implementing hyperparameter search methods. These modules share common interfaces for reading the problem search space (described in §III-A) and evaluating the benchmark's objective (see §III-C). The search methods approximately solve (2) by repeating the following steps.

(i) *Sampling:* Promising hyperparameter configurations are generated.

(ii) *Evaluating:* Each configuration is evaluated by training the model and then computing the error on the validation dataset.

(ii) *Updating:* The resulting validation errors are used to bias future hyperparameter configuration samples to minimize the validation error.

Given the expensive nature of training each DNN model, parallelizing hyperparameter configuration evaluations on multiple compute nodes is critical for speeding the search. The search methods that parallelize evaluations can be classified as either *batch synchronous* or *asynchronous* methods. In the former, the search method proceeds to the next iteration only after completely evaluating the set of hyperparameter configurations sampled in the current iteration. In the latter, new configurations are selected before all previous configurations have been evaluated. The `evaluators` subpackage provides a simple, three-function interface for both batch synchronous and asynchronous evaluations. We describe the interface here but leave implementation details to §III-C.

Algorithm 1 Asynchronous parallel model-based search

$\mathcal{X}_s \leftarrow$ `random_sample_configs`$(\mathcal{D})$
`add_eval_batch`$(\mathcal{X}_s)$
while `stopping criterion not met` **do**
  $(\mathcal{X}_r, \mathcal{Y}_r) \leftarrow$ `get_finished_evals()`
  $s \leftarrow |\mathcal{Y}_r|$
  if $s > 0$ **then**
    $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup \mathcal{X}_r$; $\mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup \mathcal{Y}_r$
    $\mathcal{M} \leftarrow$ `Fit`$(\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}})$
    $\mathcal{D} \leftarrow \mathcal{D} - \mathcal{X}_r$
    $\mathcal{X}_s \leftarrow$ `sample_configs`$(\mathcal{M}, \mathcal{D})$
    `add_eval_batch`$(\mathcal{X}_s)$
  end if
end while
Output: Best hyperparameter configuration(s) from $\mathcal{X}_{\text{out}}$

The `evaluator` interface provides a key benefit in modularity to the `search` methods. New hyperparameter search algorithms can be implemented in DeepHyper without consideration of the underlying parallelization details, which are tightly encapsulated. The resulting hyperparameter search method is then immediately portable from personal computers (for debugging and small-scale experimentation) to HPC systems supporting massively-parallel hyperparameter searches. Moreover, the search code is immediately applicable to any of the problems defined in the `benchmarks` subpackage.

1) Asynchronous model-based search (AMBS): AMBS relies on fitting a dynamically updated surrogate model that tries to learn the relationship between the hyperparameter configurations (input) and their validation errors (output). Key properties of the surrogate model are that it is cheap to evaluate and can be used to prune the search space and identify promising regions. The surrogate model is iteratively refined in the promising regions of the search space by obtaining new outputs at inputs that are predicted by the model to be high performing.

The AMBS framework is outlined in Algorithm 1. The search is designed to operate in a manager-worker paradigm, whereby a manager runs the search and generates configurations and workers perform the computationally expensive evaluations and return the validation error to the manager.

The search is initialized with a number of hyperparameter configurations equal to the number of available workers, obtained by random sampling of the search space $\mathcal{D}$. The `evaluators` interface provides a single function

`add_eval_batch` for submitting new hyperparameter configurations for evaluation. Each worker is busy only with one evaluation at a time.

At each iteration, AMBS queries the workers for the most recent evaluation results ($\mathcal{X}_r$, $\mathcal{Y}_r$). The number of results, $|\mathcal{Y}_r| = s$, is therefore also equal to the number of newly idle workers. When s is nonzero, the new results are appended to the evaluated list ($\mathcal{X}_{\text{out}}$, $\mathcal{Y}_{\text{out}}$), and the surrogate model $\mathcal{M}$ is retrained on the latest information (line 8). The asynchronous aspect of this algorithm lies in `get_finished_evals`, which is nonblocking and allows the search to avoid waiting for all the evaluation results before proceeding to the next iteration. As soon as an evaluation is finished, the validation error is used to bias the search toward more promising regions of the search space.

Using the surrogate model $\mathcal{M}$, AMBS samples a set $\mathcal{X}_s$ of s configurations from the search space $\mathcal{D}$ using the function `sample_configs`. This new set of promising configurations is immediately submitted to the workers through `add_eval_batch`, thereby restoring the worker utilization to 100%. Crucial to the effectiveness of AMBS are the method (`sample_configs`) used for selecting hyperparameter configurations and the choice of model ($\mathcal{M}$).

For `sample_configs` we adopt acquisition function ideas from the BO literature [6]. First, we sample a large number of unevaluated hyperparameter configurations. For each sampled configuration x_i, we use the model $\mathcal{M}$ to predict a value $\mu(x_i)$ and standard deviation $\sigma(x_i)$. Evaluating points with small values of $\mu(x_i)$ indicates that the x_i can potentially result in reduction of validation error subject to the accuracy of $\mathcal{M}$, a process called exploitation. Large values of $\sigma(x_i)$ indicate that $\mathcal{M}$ has not learned the regions of the search space well at x_i. Evaluating points with large $\sigma(x_i)$ improves the model $\mathcal{M}$, a process called exploration. Several acquisition functions have been developed in the BO literature that attempt to balance exploration and exploitation for various settings. In this paper we focus on the lower confidence bound, a simple and robust acquisition function defined by

$$\mathcal{A}_{\text{LCB}}(x) = \mu(x) - \lambda * \sigma(x). \tag{3}$$

When the parameter $\lambda \geq 0$ is set to 0, the search performs pure exploitation (i.e., points that have lower predicted values are selected irrespective of model uncertainty). As λ increases, the search tends toward pure exploration, and points are selected based on their potential to improve the accuracy of $\mathcal{M}$.

In AMBS, multiple hyperparameter configurations can finish evaluation simultaneously; this situation must be addressed in order to minimize worker idling. To select s configurations at a time, `sample_configs` adopts a multipoint acquisition function based on a "constant liar" strategy. This approach starts by selecting a point that maximizes $\mathcal{A}_{\text{LCB}}$. A dummy output is assigned to the first configuration and the model is updated with this configuration and the dummy output (a lie). The second configuration in the batch is then obtained by maximizing $\mathcal{A}_{\text{LCB}}$ using the updated model. This process is repeated until a set of s configurations is selected. The dummy output has the same value over the $s-1$ configuration selections. Typical dummy choices include the maximum, minimum, or mean of the validation errors that the search has found up to that point. In AMBS, a distinguishing feature of the `sample_configs` constant liar strategy is that the model trained on dummy values is *not* discarded after s configurations are generated. Instead, the same model containing a mixture of true results and "lies" is maintained, and the lie values are replaced by evaluated validation errors as soon as they become available.

The search space characteristics determine the type of model. Any ML regression method can be used for building the surrogate model by using a bagging approach that builds a number of models, each from a subset of training data, and aggregates them to compute a mean $\mu(x)$ and standard deviation $\sigma(x)$ of a configuration x. However, there is a class of ML methods that are better suited for model-based search because they provide standard deviations natively. The most widely used method in BO is Gaussian process regression, which requires explicit encoding to convert the nonordinal parameters to numeric ones. Depending on the sensitivity to the nonordinal hyperparameters, this conversion can reduce the effectiveness of the method. Random forests is an effective alternative model class because it can handle discrete and nonordinal parameters directly without the need for encoding. DeepHyper allows the use of any of the regression models available in the scikit-learn package [7], a popular ML package implemented in Python.

2) Other search methods: Perhaps the simplest asynchronous search is random search (RS), in which configurations are sampled at random (without replacement) as soon as an evaluation is finished and a worker becomes free. RS has been shown to be more effective than grid search for several ML and DL algorithms. In DeepHyper, RS is implemented by using random sampling instead of a multipoint acquisition function in `sample_configs`.

Although we focus on asynchronous search methods, DeepHyper provides an interface for the implementation of batch-synchronous search methods. These methods still use `add_eval_batch` to submit the evaluation of new configurations, but `get_finished_evals` is replaced by `await_evals`, which blocks (up to a specified timeout period) until all s of the requested configurations have been evaluated.

Using the same interface, any batch-synchronous search method can be implemented. A widely used batch-synchronous search method for hyperparameter search is evolutionary computation (EC), which repeatedly improves a population of configurations by applying a series of genetic operations. In DeepHyper we integrate the Distributed Evolutionary Algorithms in Python (DEAP) [8] framework, which provides transparent and modular EC components that facilitate rapid prototyping and testing of new EC ideas. DEAP contains implementations such as genetic, particle swarm optimization, differential evolution, and estimation of distribution algorithms. DEAP naturally addresses search spaces with real

parameters; to handle mixed-integer spaces, we implemented an encoder-decoder interface for DEAP. The encoder transforms continuous and discrete hyperparameter values such that the range is between 0 and 1. For each nonordinal (or discrete) hyperparameter with k distinct values, we split 0 and 1 into k intervals and assign each value of the hyperparameter to that interval. DeepHyper uses a decoder to transform the values into the original values, evaluates them, and returns their validation errors.

While several search methods offer efficiencies by adaptively choosing new configurations to train, an alternative strategy is to adaptively allocate resources across the selected configurations. Hyperband [3] is a hyperparameter search based on the adaptive allocation approach. It formulates hyperparameter search as a pure-exploration, deterministic, infinitely many-armed bandit problem. It adopts a principled early stopping strategy to preemptively stop configuration evaluations with poor validation error; given a fixed time budget, this allows the search to evaluate more configurations than can other BO search algorithms. The search starts from randomly sampled configurations, runs each for a certain number of epochs, and removes the configurations in the lowest half in terms of validation accuracy. It then increases the number of epochs for the surviving configurations and continues running them. The number of epochs at each iteration is increased geometrically according to a fixed schedule. Hyperband requires batch synchronization and uses `await_evals` because the search cannot proceed to the next iteration before eliminating hyperparameter configurations from the current iteration.

C. Evaluator

As illustrated in source code 3, the `evaluators` subpackage defines the `Evaluator` interface, which is used by the asynchronous and batch-synchronous search methods described in §III-B.

Source code 3 Evaluator interface

```
class Evaluator():
  def add_eval_batch(self, XX):
  '''Submit list of new cfgs XX'''
  def get_finished_evals(self):
  '''Query for newly completed evaluations
  and return results list'''
  def await_evals(self, XX):
  '''Block until evaluations are completed
  for all cfgs in XX and return results list'''
```

This simple interface could have numerous and complex implementations in order to enable features such as

- intranode parallelism for small-scale experiments.
- various types of internode parallelism for large-scale experiments.
- checkpointing of evaluation results.
- fault tolerance for systematic model errors, related to the sampled hyperparameter configuration.
- fault tolerance for unexpected filesystem or network failures.
- reliable caching of evaluations for search algorithms that repeatedly visit a configuration, and
- graceful termination of slow or halted evaluations.

DeepHyper is packaged with a `LocalEvaluator` to support the rapid development and testing of new search algorithms and benchmarks. The Python `concurrent.futures` module is used to dispatch benchmark evaluations as independent processes running concurrently in a multiprocessor node or personal machine. The last three points of the above feature list become significant challenges when one tries to effectively leverage HPC resources. Toward this goal, the `BalsamEvaluator` was implemented to interface DeepHyper with Balsam [9], a general-purpose framework for managing workflows in HPC environments.

Users interact with a local Balsam job database to define task graphs and dynamic workflows. In the DeepHyper context, the `BalsamEvaluator` uses the Python API provided by Balsam to interact with the `BalsamJob` database. Each `BalsamJob` corresponds to a single hyperparameter configuration evaluation and contains fields pointing to the task executable (Python interpreter and benchmark model path) and the command-line arguments used to specify the configuration.

The `BalsamEvaluator` maintains two dictionaries: `pending_evals`, which maps configurations onto the corresponding `BalsamJob` IDs, and `evals`, which maps the same configurations to the stored objective value. As a search proceeds synchronously or asynchronously, receiving data from `BalsamEvaluator`, these data structures are updated accordingly. The `Evaluator` takes advantage of the Balsam Django API to filter jobs according to their state (e.g., process return code) and leverages functionality such as monitoring job output, logging error tracebacks, and generating compute node utilization profiles.

Balsam provides a launcher that asynchronously pulls and packages these tasks for execution while tracking the state of the entire workflow. All details of concurrent execution, load balancing, and machine-specific scheduling/MPI issues are encapsulated within the Balsam layer. DeepHyper leverages the dynamic launcher's ability to kill long-running tasks and tolerate failed ones. In either event, the `BalsamEvaluator` dispatches new hyperparameter evaluation tasks, while the AMBS model retains the dummy objective value. Since specific tasks can be killed and replaced without interrupting the rest of the ensemble, sophisticated early-stopping methods can be leveraged in future search strategies. The launcher abstraction also enhances portability; the same workflow runs on small to large CPU and GPU clusters, insofar as the appropriate DL backends (e.g., TensorFlow) are preloaded. Internally, the launcher wraps single-node hyperparameter evaluation tasks in an MPI manager-worker program, where each worker rank oversees a forked task process.

The `BalsamJob` database automatically records useful provenance data such as task execution time and traceback messages in the event of task runtime errors. Because the Balsam launcher execution is so loosely coupled to the

DeepHyper application through the `BalsamJob` database, the workflow becomes resilient to task failures or walltime expirations. On interruption, the DeepHyper search state is quickly checkpointed, and the pending evaluations are marked for retry in the BalsamJob database.

Balsam users use the `balsam init` command-line interface to establish a new job database for their workflow. In the case of DeepHyper search experiments, a separate database is instantiated for each experiment; users also can tag jobs falling under one experiment and to store all experiments together. Internally, Balsam uses the Django ORM to interface a database driver with a PostgresSQL server instance running on a login or head node.

IV. Experimental Results

We next evaluate the efficacy of DeepHyper using a diverse set of benchmarks using asynchronous and batch synchronous search methods on two large-scale HPC systems. We compare these using metrics such as accuracy and utilization to demonstrate the scalability and ease of portability of DeepHyper on HPC systems.

A. Setup

We used two large-scale and diverse HPC systems for our evaluations: Theta is a 11.69-petaflops Cray XC40-based leadership-class supercomputing system at the Argonne Leadership Computing Facility (ALCF). It consists of 4,392 nodes, each containing a 64-core Intel Xeon Phi processor with 16 gigabytes of high-bandwidth in-package memory, 192 GB of DDR4 memory, and a 128 GB SSD. Theta's file system has a capacity of 10 petabytes and the compute nodes are interconnected using an Aries fabric. Cooley is a GPU-based cluster at the ALCF. It has a total of 126 compute nodes; each node has 12 CPU cores, one NVIDIA Tesla K80 dual-GPU card, with 24 GB of GPU memory and 384 GB of DDR3 CPU memory. The compute nodes are interconnected via an Infini Band fabric.

The Theta environment consisted of Intel Python 3.6.3, Tensorflow 1.3.1 [10], Keras 2.0.9 [11], and scikit-optimize 0.4. On Cooley, TensorFlow 1.3.0 and Keras 2.1.5 were used instead. The lightweight Balsam launcher process ran on either a MOM node of Theta or compute head node of the Cooley cluster.

We selected a set of six benchmarks to span a diverse set of benchmark models commonly used by applications. We used these benchmarks to evaluate DeepHyper and the efficacy of its search methods. The hyperparameters for these benchmarks are summarized in Table I. The baseline codes for these benchmarks (except gcn) have been obtained from the examples directory of the Keras github repo.

mnistmlp is a fully connected hidden DNN for classifying images in the MNIST dataset. **mnistcnn** is a CNN benchmark for classifying images in the MNIST dataset. **cifar10cnn** is a CNN benchmark for classifying images in the cifar10 dataset. **gcn** is an implementation of graph convolution networks [12] for semi-supervised classification on the Cora dataset. **rnn1** is a recurrent neural network (RNN) for sequence-to-sequence learning for performing addition. **rnn2** is a memory network obtained using an RNN trained on the bAbI dataset for question-and-answer systems.

TABLE I
Benchmarks and their associated ranges; for the parameters wherein a range is not listed, we employ the same range for all benchmarks

Benchmark	Hyperparameters
mnistmlp	epochs (nepochs) ∈ [5, 500], number of hidden layers (nhidden) ∈ [1, 100], number of units per layer (nunits) ∈ [1, 1000], activation ∈ [relu, elu, selu, tanh], batch size (bsize) ∈ [8, 1024], dropout ∈ [0.0, 1.0], optimizer ∈ [sgd, rmsprop, adagrad, adadelta, adam, adamax, nadam], learning rate (lrate) ∈ [1e-04, 1e01]
mnistcnn	nepochs, nunits, activation, bsize, dropout, optimizer, lrate, filter size for first and convolution 2D layer ∈ [1, 3, 5, 7], number of filters for first and convolution 2D layer ∈ [8, 16, 32, 64], pooling size (psize) ∈ [2, 3, 4]
cifar10cnn	same as mnistcnn
gcn	nepochs, activation, bsize, dropout, optimizer, lrate, normalization ∈ [False, True], number of graph convolutional units ∈ [2, 4, 6, 8, 16, 32, 64], filter type ∈ [localpool, chebyshev], maximum polynomial degree ∈ [1, 10]
rnn1	nepochs, nunits, nlayers, activation, bsize, dropout, optimizer, lrate, type of rnn (rnntype) ∈ [LSTM, GRU, SimpleRNN]
rnn2	nepochs, nunits, nlayers, activation, bsize, dropout, optimizer, lrate, rnntype

B. Comparison of search methods

We compared AMBS (using a random forest model) with random search (RS), a genetic algorithm (GA), and hyperband (HB). In this case, we conducted our experiments using 128 Theta nodes. We used a fixed budget of 2 hours of wall clock time for the evaluation and compare the performance with respect to the number of evaluations performed as well as the accuracy achieved. Figure 1 depicts the number of hyperparameter evaluations performed by each search method within this budget. From the results, we observe that AMBS significantly outperforms other search methods with respect to the number of evaluations. In particular, AMBS performs more than twice the number of evaluations in comparison with the two batch synchronous methods GA and HB on all six benchmarks. This result can be attributed to the training time variability associated with hyperparameter configurations. A key factor here is the difference in the number of epochs used in the evaluations—different values for this hyperparameter result in significant training time differences. Consequently, at each iteration, the batch synchronous methods need to wait for the slowest evaluation to finish in order to proceed to the next iteration. We observe that RS does not achieve a large number of evaluations. For most benchmarks, fewer epochs are sufficient to obtain higher accuracy, and a larger number of epochs result in overfitting. Despite being asynchronous, RS did not have a feedback mechanism to learn this fact, and therefore, it sampled hyperparameter configurations with a large number of epochs.

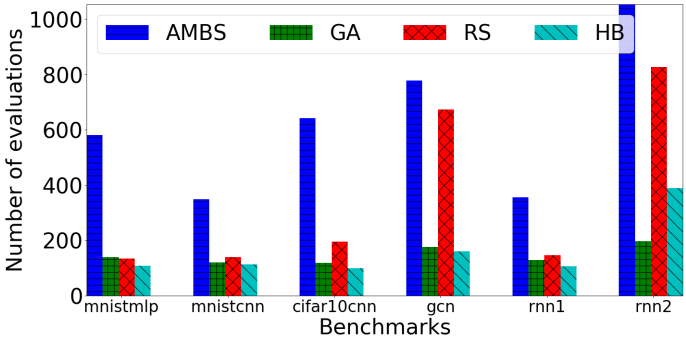

Fig. 1. Comparison of different search methods in DeepHyper on 128 nodes with respect to the number of evaluations. The results show that AMBS with the random forest model (rf) performs more evaluations than do the other search methods.

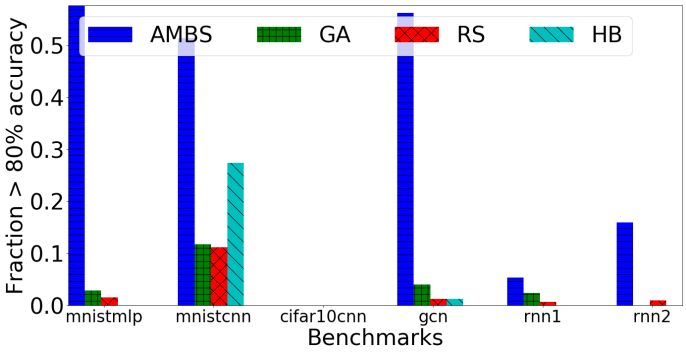

Fig. 2. Comparison of different search methods in DeepHyper on 128 Theta nodes with respect to the accuracy (fraction of number of evaluations). The results show that AMBS with the random forest model (rf) samples and evaluates higher-quality hyperparameter configurations compared with other methods.

Next, we compared the search methods with respect to their achieved accuracy. Here, for each method, we computed the fraction of hyperparameter configurations that obtained more than 80% accuracy. The results are shown in Figure 2. From the results, we observe that AMBS outperforms the other methods. For the mnistmlp, mnistcnn, and gcn, AMBS obtains high-quality hyperparameter configurations—more than 50% of the configurations obtain accuracy greater than 80%. For rnn1 and rnn2, the fractions are 0.08 and 0.15, respectively. An exception is that cifar10cnn benchmark, where none of the search methods obtain hyperparameter configurations with more than 80% accuracy. The effectiveness of AMBS can be attributed to the asynchronous search mechanism that results in learning the search space more effectively and sampling a large number of evaluations from the promising regions of the search space.

C. Comparison of regression methods in AMBS

We study the impact of different regression methods in AMBS with respect to the accuracy. For this purpose, we compare the default random forest model (RF) with extra trees (ET), gradient boosting regression trees (GBRT), and Gaussian process (GP) regression methods. Figure 3 shows

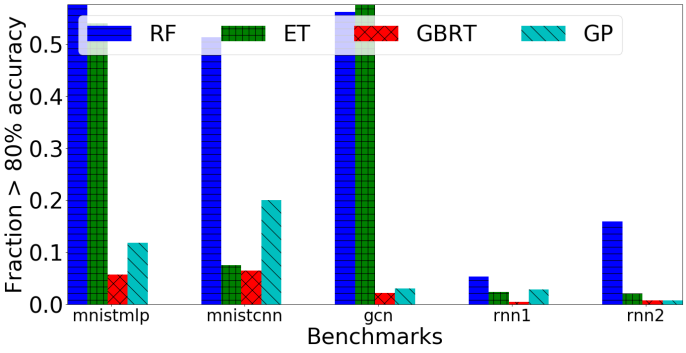

Fig. 3. Comparison of different regression methods in AMBS on 128 Theta nodes with respect to the accuracy (fraction of number of evaluations).

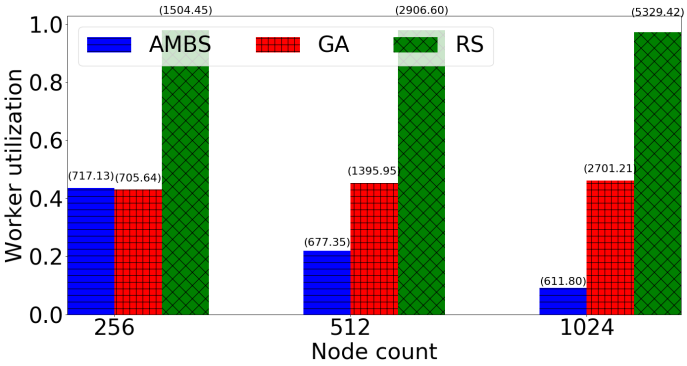

Fig. 4. Time-averaged Theta compute node utilization plotted for AMBS, GA, and RS search methods on the rnn2 benchmark. A temporal utilization profile is generated from Balsam, giving the fraction of worker nodes actively running a hyperparameter evaluation task at any given time. The time average of this value, starting from the first evaluation, is plotted for each search method as we scale from 256 to 1,024 nodes. In parentheses above each bar is the number of completed evaluations per hour (each experiment was run under a user-specified time constraint of 2 hours).

the fraction of hyperparameters that obtain accuracy above 80% for different regression methods. We observe that RF outperforms ET, GBRT, and GP on all but one benchmark (gcn), where ET is slightly better than RF. From a functional point of view, RF produces an ensemble of piecewise constant approximations compared with the ensemble of ET's piecewise multilinear approximations. For the benchmarks considered, the ensemble of piecewise constant models results in better accuracy. The poor performance of GP can be attributed to the mixed-integer search space, in particular, the presence of nonordinal hyperparameters such as activation function and optimizer. While GP can handle the real-valued hyperparameters well, the nonsmoothness introduced by nonordinal hyperparameters can pose significant difficulties and requires special or custom kernels. GBRT differs from RF and ET by adopting a boosting approach for regression, where a number of tree models are built sequentially. From the results, we observe that such sequential tree models do not result in high-quality hyperparameter configurations.

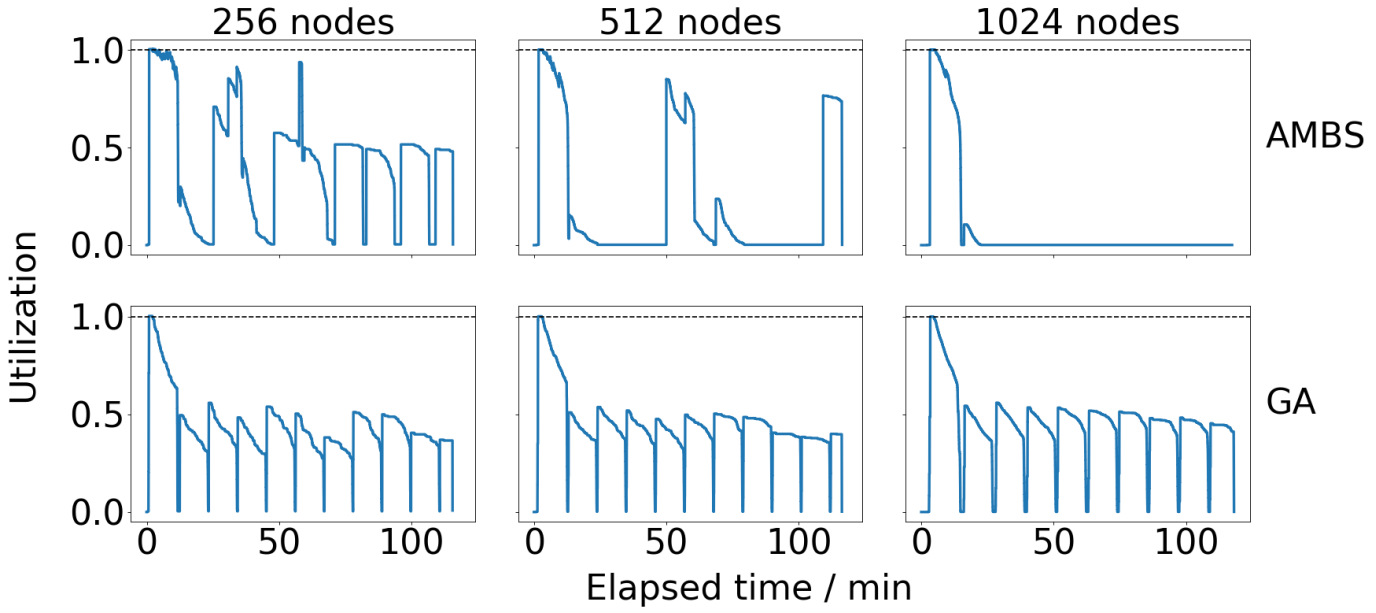

Fig. 5. Theta compute node utilization plotted for AMBS and GA search methods on the rnn2 benchmark on 256, 512, and 1024 nodes.

D. Scaling DeepHyper on the Theta supercomputer

Figure 4 compares the time-averaged utilization of Theta compute nodes between AMBS, GA, and RS search methods for the rnn2 benchmark. This utilization is defined as the fraction of compute nodes actively running a hyperparameter evaluation task at any given time and can be directly inferred from the `BalsamJob` database postrun. We present the time-average utilization, which takes into account the utilization for the duration from when we initially dispatch of hyperparameter evaluation tasks until the end of the experiment. We notice that for RS, the compute nodes are busy with hyperparameter evaluation tasks nearly 100% of the time. Thus, the random sampling and Balsam execution layers pose no bottleneck to the search and scale effectively.

GA sustains only about 50% worker utilization owing to the default mutation and crossover parameters, which are set such that about half of each generation produced during the search remains unchanged from its parent generation. From the temporal utilization profile (bottom row of Figure 5), we notice a periodic sawtooth profile, where the utilization spikes down to 0 at the end of each call to `await_evals`, which is characteristic of all batch-synchronous search algorithms.

In contrast, AMBS fails to effectively scale beyond 256 nodes for the rnn2 benchmark. This can be attributed to the current design consisting of a single search process invoking `sample_configs`. This result can be viewed as a single-server bottleneck, and the utilization suffers as the search fails to generate sufficient hyperparameter configurations quickly enough once a large batch of evaluations finishes concurrently. The prolonged stall in sampling is evident in the top row of Figure 5; at 1,024 nodes, the search stalls completely on `sample_configs` after the majority of initial configurations have been evaluated. We compare this observation for the

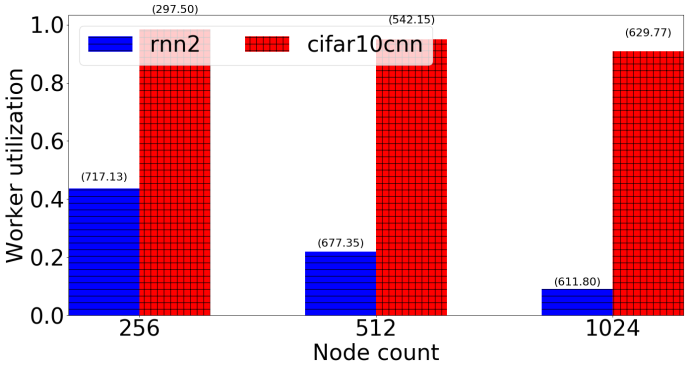

Fig. 6. Time-averaged Theta compute node utilization compared for the AMBS search on rnn2 and cifar10cnn benchmarks. The values in parentheses give the number of completed evaluations-per-hour (cifar10cnn and rnn2 experiments were run for approximately 1 and 2 hours, respectively.)

rnn2 benchmark with the behavior of the same search on the cifar10cnn benchmark (Figure 6). Since the runtimes of the cifar10cnn hyperparameter evaluations have a larger variance, with many runs taking significantly longer than the average, the arrival of evaluation results is both slower and more staggered than the results for rnn2. This reduces the load on the hyperparameter configuration generation process. Thus, AMBS is able to generate sufficient configurations and sustain the worker utilization to over 90% for the cifar10cnn hyperparameter searches.

As a partial remedy to improve the performance of AMBS at scale, we evaluated the same search at 1,024 nodes with a batch generator function. The generator divides a large request for new hyperparameter configurations down into batches of a given size (default 20). The constant liar sampling yields execution for Balsam job injection once each sampled

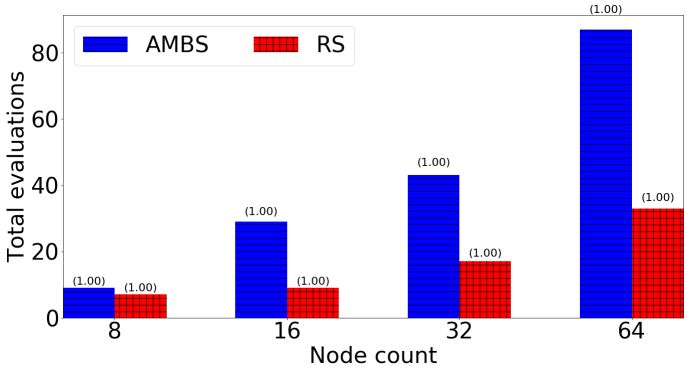

Fig. 7. Total number of completed cifar10cnn evaluations after a 1 hour run are compared between AMBS and RS searches on 8 to 64 Cooley nodes. For an n node run, up to $2n-1$ evaluations would run simultaneously (1 per GPU). The values in parenthesis give the time-averaged Cooley node utilization, which is essentially constant at 100%.

batch is completed. This mitigates some of the performance bottlenecks associated with AMBS and provides an improvement over the blocking call to the hyperparameter generation scheme. We observe an improved utilization, hovering between 10 and 20%, instead of 0% previously. We plan to address this issue using more scalable BO approaches [13].

E. Scaling AMBS on the Cooley GPU cluster

Next, we demonstrate the portability of the DeepHyper package via experiments on Cooley, a 126-node GPU cluster.

Figure 7 depicts the total number of evaluations for the cifar10cnn hyperparameter configurations generated by AMBS and RS as we scale from 8 to 64 Cooley nodes (using two independent GPU workers per node given the dual GPU per node) given a user-specified wall clock constraint of 1 hour. We observe that the utilization is essentially constant at 100%, even for the largest Cooley runs with AMBS. Consequently, as one would hope, the number of completed evaluations increases as we scale the number of compute nodes. This increase is more pronounced with AMBS, because the search converges toward configurations with few tens of training epochs that finish faster, while the RS method uniformly samples configurations over a wide range and thus take longer to finish. Even with 128 GPU workers (64 nodes), the RS produces only 2 cifar10cnn models with validation accuracy better than 50%. In contrast, the number of models with greater than 50% accuracy increases from 1 to 19 as we scale with AMBS from 8 to 64 nodes.

V. Related Work

Traditionally, in DL research, hyperparameter search is limited to tuning the hyperparameters of a DL algorithm by trial and error or enumeration of hyperparameters on a grid. Recently, new algorithmic methods have been developed. They can be grouped into neural architecture and hyperparameter search methods. Neural architecture search methods search over model descriptions of neural network specifications. They are grouped into discrete search-space traversal [14], [15], reinforcement learning [16]–[18], and evolutionary algorithms [19]–[22]. Hyperparameter search approaches try to find best values for the hyperparameters for a fixed neural architecture. Examples include random search [23], Bayesian optimization [2], [24], [25], Bandit-based methods [3], [24], metaheuristics [26], [27], and population-based training [28], [29] approaches. For a comprehensive overview of hyperparameter search methods, we refer the reader to [1]. Most of the existing open-source DL search codes are method-centric and ill-suited for experimental research in DL search methods.

mlrMBO [30] is a R toolbox for BO that addresses the problem of expensive black-box optimization using surrogate regression model. It provides interface for batch synchronous parallel evaluations for single node multicore parallelization, which is not particularly suitable for DL hyperparameter search on large multinode HPC systems. Scikit-optimize [31] is a Python library to minimize expensive and noisy black-box functions. It uses regression methods in scikit-learn [7] for surrogate modeling. While it has functions for asynchronous interface, it does not provide large-scale manager-worker interface for parallelizing hyperparameter evaluations. We used scikit-optimize codebase for implementing DeepHyper. Hyperopt [25] is a Python package that provides distributed asynchronous search interface and contains random and BO search methods. They can be run either serially, or in parallel by communicating via MongoDB. ROBO [2] is a new BO framework that offers an easy-to-use Python interface inspired by the API of SciPy. RoBO offers implementations of BO with Bayesian neural networks, multi-task optimization, and fast Bayesian hyperparameter optimization on large datasets. It performs one hyperparameter evaluation at a time and does not provide approaches and interface for parallel evaluations. A key limitation across all of the existing packages is that they tend to exist as research prototypes or do not target scalability and portability concerns of HPC systems. A closely related package is Ray Tune [32], a scalable hyperparameter optimization framework based on Ray, a general purpose Python-based distributed execution engine. It contains several search methods and provides evaluator interface similar to DeepHyper. However, it does not have workflow system integration to hide the complexities of HPC platforms.

VI. Conclusion and Future Work

Motivated by the lack of production-ready DL hyperparameter search packages for large HPC systems, we developed DeepHyper, a Python package and infrastructure that targets experimental research in DL search methods, scalability, and portability across HPC systems. It comprises three modules: `benchmarks`, a collection of extensible and diverse set of DL hyperparameter search problems; `search`, a set of search algorithms for DL hyperparameter search; and `evaluators`, a common interface for evaluating hyperparameter configurations on HPC platforms. We implemented an asynchronous model-based search method, random search, a batch-synchronous genetic algorithm, and hyperband. We evaluated the efficacy of these methods on CPU- and GPU-based

HPC systems and studied their scaling limits on a diverse set of benchmarks. By using the Balsam workflow system, we isolate search method implementation and deployment from the complexities of running large numbers of hyperparameter configurations in parallel on HPC systems.

Our future work includes scalable BO methods to address the scaling limitations of AMBS, multiple-manager-worker approaches, extension of benchmarks with respect to the number of parameters and type of architecture, architecture search, deployment of DeepHyper on scientific DL applications, and portability on other HPC systems.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

REFERENCES

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *Preprint arXiv:1802.09941*, 2018.

[2] A. Klein, S. Falkner, N. Mansur, and F. Hutter, "RoBO: A flexible and robust Bayesian optimization framework in python," in *NIPS 2017 Bayesian Optimization Workshop*, 2017.

[3] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: Bandit-based configuration evaluation for hyperparameter optimization," in *ICLR*, 2017.

[4] D. Sculley, J. Snoek, A. Wiltschko, and A. Rahimi, "Winner's curse? on pace, progress, and empirical rigor," in *ICLR Workshop*, 2018.

[5] Z. Ronaghi, R. Thomas, J. Deslippe, S. Bailey, D. Gursoy, T. Kisner, R. Keskitalo, and J. Borrill, "Python in the NERSC exascale science applications program for data," in *PyHPC17*. ACM, 2017.

[6] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[7] 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," *J. Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Machine Learning Research*, vol. 13, pp. 2171–2175, 2012.

[9] J. T. Childers, T. D. Uram, D. Benjamin, T. J. LeCompte, and M. E. Papka, "An edge service for managing HPC workflows," in *HUST'17*. ACM, 2017, pp. 1:1–1:8.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[11] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[13] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable Bayesian optimization using deep neural networks," in *International conference on machine learning*, 2015, pp. 2171–2180.

[14] R. Negrinho and G. Gordon, "DeepArchitect: Automatically designing and training deep architectures," *Preprint arXiv:1704.08792*, 2017.

[15] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," *Preprint arXiv:1712.00559*, 2017.

[16] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *ICLR*, 2017.

[17] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.

[18] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *Preprint arXiv:1802.03268*, 2018.

[19] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evolutionary Intelligence*, vol. 1, pp. 47–62, 2008.

[20] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, 2009.

[21] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *GECCO*. ACM, 2017, pp. 497–504.

[22] D. Wierstra, F. J. Gomez, and J. Schmidhuber, "Modeling systems with internal state using Evolino," in *GECCO*. ACM, 2005, pp. 1795–1802.

[23] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[24] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *NIPS*, 2012, pp. 2951–2959.

[25] J. Bergstra, D. Yamins, and D. D. Cox, "HyperOpt: A python library for optimizing the hyperparameters of machine learning algorithms," in *12th Python in Science Conf.*, 2013, pp. 13–20.

[26] P. R. Lorenzo, J. Nalepa, L. S. Ramos, and J. R. Pastor, "Hyper-parameter selection in deep neural networks using parallel particle swarm optimization," in *GECCO*. ACM, 2017, pp. 1864–1871.

[27] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, R. Kozma, C. Alippi, Y. Choe, and F. C. Morabito, Eds., 2018.

[28] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan *et al.*, "Population based training of neural networks," *Preprint arXiv:1711.09846*, 2017.

[29] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015, p. 4.

[30] B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang, "mlrMBO: A modular framework for model-based optimization of expensive black-box functions," *Preprint arXiv:1703.03373*, 2017.

[31] scikit-optimize. [Online]. Available: https://scikit-optimize.github.io/

[32] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," *Preprint arXiv:1712.05889*, 2017.