A SYSTEM FOR MASSIVELY PARALLEL HYPERPARAMETER TUNING

Anonymous Authors[1]

ABSTRACT

Modern learning models are characterized by large hyperparameter spaces and long training times. These properties, coupled with the rise of parallel computing and the growing demand to productionize machine learning workloads, motivate the need to develop mature hyperparameter optimization functionality in distributed computing settings. We address this challenge by first introducing a simple and robust hyperparameter optimization algorithm called ASHA, which exploits parallelism and aggressive early-stopping to tackle large-scale hyperparameter optimization problems. Our extensive empirical results show that ASHA outperforms existing state-of-the-art hyperparameter optimization methods; scales linearly with the number of workers in distributed settings; and is suitable for massive parallelism, as demonstrated on a task with 500 workers. We then describe several design decisions we encountered, along with our associated solutions, when integrating ASHA in SystemX, an end-to-end production-quality machine learning system that offers hyperparameter tuning as a service.

1 INTRODUCTION

Although machine learning (ML) models have recently achieved dramatic successes in a variety of practical applications, these models are highly sensitive to internal parameters, i.e., *hyperparameters*. In these modern regimes, four trends motivate the need for production-quality systems that support massively parallel for hyperparameter tuning:

1. **High-dimensional search spaces**. Models are becoming increasingly complex, as evidenced by modern neural networks with dozens of hyperparameters. For such complex models with hyperparameters that interact in unknown ways, a practitioner is forced to evaluate potentially thousands of different hyperparameter settings.
2. **Increasing training times**. As datasets grow larger and models become more complex, training a model has become dramatically more expensive, often taking days or weeks on specialized high-performance hardware. This trend is particularly onerous in the context of hyperparameter optimization, as a new model must be trained to evaluate each candidate hyperparameter configuration.
3. **Rise of parallel computing**. The combination of a growing number of hyperparameters and longer training time per model precludes evaluating configurations sequentially; we simply cannot wait years to find a suitable hyperparameter setting. Leveraging distributed computational resources presents a solution to the increasingly challenging problem of hyperparameter optimization.
4. **Productionization of ML**. ML is increasingly driving innovations in industries ranging from autonomous vehicles to scientific discovery to quantitative finance. As ML moves from R&D to production, ML infrastructure must mature accordingly, with hyperparameter optimization as one of the core supported workloads.

In this work, we address the problem of developing production-quality hyperparameter tuning functionality in a distributed computing setting. Support for massive parallelism is a cornerstone design criteria of such a system and thus a main focus of our work.

To this end, and motivated by the shortfalls of existing methods, we first introduce **A**synchronous **S**uccessive **H**alving **A**lgorithm (ASHA), a simple and practical hyperparameter optimization method suitable for massive parallelism that exploits aggressive early stopping. Our algorithm is inspired by the Successive Halving algorithm (SHA) (Karnin et al., 2013; Jamieson & Talwalkar, 2015), a theoretically principled early stopping method that allocates more resources to promising configurations. ASHA is designed for what we refer to as the 'large-scale regime,' where to find a good hyperparameter setting, we must evaluate orders of magnitude more hyperparameter configurations than available parallel workers in a small multiple of the wall-clock time needed to train a single model.

We next perform a thorough comparison of several hyperparameter tuning methods in both the sequential and par-

[1] Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

allel settings. We focus on 'mature' methods, i.e., well-established techniques that have been empirically and/or theoretically studied to an extent that they could be considered for adoption in a production-grade system. In the sequential setting, we compare SHA with Fabolas (Klein et al., 2017a), Population Based Tuning (PBT) (Jaderberg et al., 2017), and BOHB (Falkner et al., 2018), state-of-the-art methods that exploit partial training. Our results show that SHA outperforms these methods, which when coupled with SHA's simplicity and theoretical grounding, motivate the use of a SHA-based method in production. We further verify that SHA and ASHA achieve similar results. In the parallel setting, our experiments demonstrate that ASHA addresses the intrinsic issues of parallelizing SHA, scales linearly with the number of workers, and exceeds the performance of PBT, BOHB, and Vizier (Golovin et al., 2017), Google's internal hyperparameter optimization service.

Finally, based on our experience developing SystemX, a production-quality machine learning system that offers hyperparameter tuning as a service, we describe several systems design decisions and optimizations that we explored while integrating ASHA into SystemX. We focus on four key considerations: (1) streamlining the user interface to enhance usability; (2) autoscaling parallel training to systematically balance the tradeoff between lower latency in individual model training and higher throughput in total configuration evaluation; (3) efficiently scheduling ML jobs to optimize multi-tenant cluster utilization; and (4) tracking parallel hyperparameter tuning jobs for reproducibility.

2 Related Work

We will first discuss related work that motivated our focus on parallelizing SHA for the large-scale regime. We then provide an overview of methods for parallel hyperparameter tuning, from which we identify a mature subset to compare to in our empirical studies (Section 4). Finally, we discuss related work on systems for hyperparameter optimization.

Sequential Methods. Existing hyperparameter tuning methods attempt to speed up the search for a good configuration by either adaptively selecting configurations or adaptively evaluating configurations. Adaptive configuration selection approaches attempt to identify promising regions of the hyperparameter search space from which to sample new configurations to evaluate (Hutter et al., 2011; Snoek et al., 2012; Bergstra et al., 2011; Srinivas et al., 2010). However, by relying on previous observations to inform which configuration to evaluate next, these algorithms are inherently sequential and thus not suitable for the large-scale regime, where the number of updates to the posterior is limited. In contrast, adaptive configuration evaluation approaches attempt to early-stop poor configurations and allocate more training "resources" to promising configurations. Previous methods like György & Kocsis (2011); Agarwal et al. (2011); Sabharwal et al. (2016) provide theoretical guarantees under strong assumptions on the convergence behavior of intermediate losses. (Krueger et al., 2015) relies on a heuristic early-stopping rule based on sequential analysis to terminate poor configurations.

In contrast, SHA (Jamieson & Talwalkar, 2015) and Hyperband (Li et al., 2018) are adaptive configuration evaluation approaches which do not have the aforementioned drawbacks and have achieved state-of-the-art performance on several empirical tasks. SHA serves as the inner loop for Hyperband, with Hyperband automating the choice of the early-stopping rate by running different variants of SHA. While the appropriate choice of early stopping rate is problem dependent, Li et al. (2018)'s empirical results show that aggressive early-stopping works well for a wide variety of tasks. Hence, we focus on adapting SHA to the parallel setting in Section 3, though we also evaluate the corresponding asynchronous Hyperband method.

Hybrid approaches combine adaptive configuration selection and evaluation (Swersky et al., 2013; 2014; Domhan et al., 2015; Klein et al., 2017a). Li et al. (2018) showed that SHA/Hyperband outperforms SMAC with the learning curve based early-stopping method introduced by Domhan et al. (2015). In contrast, Klein et al. (2017a) reported state-of-the-art performance for Fabolas on several tasks in comparison to Hyperband and other leading methods. However, our results in Section 4.1 demonstrate that under an appropriate experimental setup, SHA and Hyperband in fact outperform Fabolas. Moreover, we note that Fabolas, along with most other Bayesian optimization approaches, can be parallelized using a constant liar (CL) type heuristic (Ginsbourger et al., 2010; González et al., 2016). However, the parallel version will underperform the sequential version, since the latter uses a more accurate posterior to propose new points. Hence, our comparisons to these methods are restricted to the sequential setting.

Other hybrid approaches combine Hyperband with adaptive sampling. For example, Klein et al. (2017b) combined Bayesian neural networks with Hyperband by first training a Bayesian neural network to predict learning curves and then using the model to select promising configurations to use as inputs to Hyperband. More recently, Falkner et al. (2018) introduced BOHB, a hybrid method combining Bayesian optimization with Hyperband. They also propose a parallelization scheme for SHA that retains synchronized eliminations of underperforming configurations. We discuss the drawbacks of this parallelization scheme in Section 3 and demonstrate that ASHA outperforms this version of parallel SHA as well as BOHB in Section 4.2. We note that similar to SHA/Hyperband, ASHA can be combined with adaptive sampling for more robustness to certain challenges of parallel computing that we discuss in Section 3.

Parallel Methods. Several parallel versions of sequential methods also exist (Hutter et al., 2011; Snoek et al., 2012; Bergstra et al., 2011). These methods either act greedily or rely on randomness to provide additional configurations to evaluate. Bayesian optimization methods tailored for the parallel setting include a Thompson sampling approach (Kandasamy et al., 2018) and batch-mode methods that select configurations by optimizing a joint objective (Shah & Ghahramani, 2015; González et al., 2016; Wu & Frazier, 2016). However, these methods fair poorly in the large-scale regime since all configurations are trained to completion.

Established parallel methods for hyperparameter tuning include PBT (Jaderberg et al., 2017; Li et al., 2019) and Vizier (Golovin et al., 2017). PBT is a state-of-the-art hybrid evolutionary approach that exploits partial training to iteratively increase the fitness of a population of models. In contrast to Hyperband, PBT lacks any theoretical guarantees. Additionally, PBT is primarily designed for neural networks and is not a general approach for hyperparameter tuning. We further note that PBT is more comparable to SHA than to Hyperband since both PBT and SHA require the user to set the early-stopping rate via internal hyperparameters.

Vizier is Google's black-box optimization service with support for multiple hyperparameter optimization methods and early-stopping options. For succinctness, we will refer to Vizier's default algorithm as "Vizier" although it is simply one of methods available on the Vizier platform. While Vizier provides early-stopping rules, the strategies only offer approximately 3× speedup in contrast to the order of magnitude speedups observed for SHA. We compare to PBT and Vizier in Section 4.2 and Section 4.3, respectively.

Hyperparameter Optimization Systems. While there is a large body of work on systems for machine learning, we narrow our focus to systems for hyperparameter optimization. AutoWEKA (Kotthoff et al., 2017) and AutoSklearn (Feurer et al., 2015) are two established single-machine, single-user systems for hyperparameter optimization. Existing systems for distributed hyperparameter optimization include Vizier (Golovin et al., 2017), RayTune (Liaw et al., 2018), CHOPT (Kim et al., 2018) and Optuna (Akiba et al.). These existing systems provide generic support for a wide range of hyperparameter tuning algorithms; both RayTune and Optuna in fact have support for ASHA. In contrast, our work focuses on a specific algorithm—ASHA—that we argue is particularly well-suited for massively parallel hyperparameter optimization. We further introduce a variety of systems optimizations designed specifically to improve the performance, usability, and robustness of ASHA in production environments. We believe that these optimizations would directly benefit existing systems to effectively support ASHA, and generalizations of these optimizations could also be beneficial in supporting other hyperparameter tuning algorithms.

Similarly, we note that Kim et al. (2018) address the problem of resource management for generic hyperparameter optimization methods in a shared compute environment, while we focus on efficient resource allocation with adaptive scheduling specifically for ASHA in Section 5.3. Additionally, in contrast to the user-specified automated scaling capability for parallel training presented in Xiao et al. (2018), we propose to automate appropriate autoscaling limits by using the performance prediction framework by Qi et al. (2017).

3 ASHA Algorithm

We start with an overview of SHA (Karnin et al., 2013; Jamieson & Talwalkar, 2015) and motivate the need to adapt it to the parallel setting. Then we present ASHA and discuss how it addresses issues with synchronous SHA and improves upon the original algorithm.

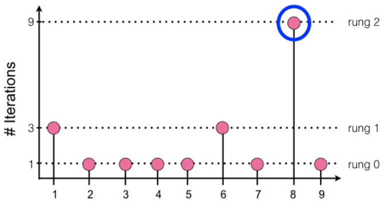
3.1 Successive Halving (SHA)

The idea behind SHA (Algorithm 1) is simple: allocate a small budget to each configuration, evaluate all configurations and keep the top $1/\eta$, increase the budget per configuration by a factor of η, and repeat until the maximum per-configuration budget of R is reached (lines 5–11). The resource allocated by SHA can be iterations of stochastic gradient descent, number of training examples, number of random features, etc.

Algorithm 1 Successive Halving Algorithm.

```
input number of configurations $n$, minimum resource $r$, maximum resource $R$, reduction factor $\eta$, minimum early-stopping rate $s$
$s_{\max} = \lfloor \log_\eta(R/r) \rfloor$
assert $n \geq \eta^{s_{\max}-s}$ so that at least one configuration will be allocated $R$.
$T = \texttt{get\_hyperparameter\_configuration}(n)$
// All configurations trained for a given $i$ constitute a ``rung.''
for $i \in \{0, \ldots, s_{\max} - s\}$ do
  $n_i = \lfloor n\eta^{-i} \rfloor$
  $r_i = r\eta^{i+s}$
  $L = \texttt{run\_then\_return\_val\_loss}(\theta, r_i) : \theta \in T$
  $T = \texttt{top\_k}(T, L, n_i/\eta)$
end for
return best configuration in $T$
```

SHA requires the number of configurations n, a minimum resource r, a maximum resource R, a reduction factor $\eta \geq 2$, and a minimum early-stopping rate s. Additionally, the $\texttt{get\_hyperparameter\_configuration}(n)$ subroutine returns n configurations sampled randomly



(a) Visual depiction of the promotion scheme for bracket $s = 0$.

bracket s	rung i	n_i	r_i	total budget
0	0	9	1	9
	1	3	3	9
	2	1	9	9
1	0	9	3	27
	1	3	9	27
2	0	9	9	81

(b) Promotion scheme for different brackets s.

Figure 1. **Promotion scheme for SHA** with $n = 9$, $r = 1$, $R = 9$, and $\eta = 3$.

from a given search space; and the `run_then_return_val_loss`(θ, r) subroutine returns the validation loss after training the model with the hyperparameter setting θ and for r resources. For a given early-stopping rate s, a minimum resource of $r_0 = r\eta^s$ will be allocated to each configuration. Hence, lower s corresponds to more aggressive early-stopping, with $s = 0$ prescribing a minimum resource of r. We will refer to SHA with different values of s as *brackets* and, within a bracket, we will refer to each round of promotion as a *rung* with the base rung numbered 0 and increasing. Figure 1(a) shows the rungs for bracket 0 for an example setting with $n = 9, r = 1, R = 9$, and $\eta = 3$, while Figure 1(b) shows how resource allocations change for different brackets s. Namely, the starting budget per configuration $r_0 \leq R$ increases by a factor of η per increment of s. Hence, it takes more resources to explore the same number of configurations for higher s. Note that for a given s, the same budget is allocated to each rung but is split between fewer configurations in higher rungs.

Straightforward ways of parallelizing SHA are not well suited for the parallel regime. We could consider the embarrassingly parallel approach of running multiple instances of SHA, one on each worker. However, this strategy is not well suited for the large-scale regime, where we would like results in little more than the time to train one configuration. To see this, assume that training time for a configuration scales linearly with the allocated resource and $time(R)$ represents the time required to train a configuration for the maximum resource R. In general, for a given bracket s, the minimum time to return a configuration trained to completion is $(\log_\eta(R/r) - s + 1) \times time(R)$, where $\log_\eta(R/r) - s + 1$ counts the number of rungs. For example, consider Bracket 0 in the toy example in Figure 1. The time needed to return a fully trained configuration is $3 \times time(R)$, since there are three rungs and each rung is allocated R resource. In contrast, as we will see in the next section, our parallelization scheme for SHA can return an answer in just $time(R)$.

Another naive way of parallelizing SHA is to distribute the training of the n/η^k surviving configurations on each rung k as is done by Falkner et al. (2018) and add brackets when there are no jobs available in existing brackets. We will refer to this method as "synchronous" SHA. The efficacy of this strategy is severely hampered by two issues: (1) SHA's synchronous nature is sensitive to stragglers and dropped jobs as every configuration within a rung must complete before proceeding to the next rung, and (2) the estimate of the top $1/\eta$ configurations for a given early-stopping rate does not improve as more brackets are run since promotions are performed independently for each bracket. We demonstrate the susceptibility of synchronous SHA to stragglers and dropped jobs on simulated workloads in Appendix A.1.

3.2 Asynchronous SHA (ASHA)

We now introduce ASHA as an effective technique to parallelize SHA, leveraging asynchrony to mitigate stragglers and maximize parallelism. Intuitively, ASHA promotes configurations to the next rung whenever possible instead of waiting for a rung to complete before proceeding to the next rung. Additionally, if no promotions are possible, ASHA simply adds a configuration to the base rung, so that more configurations can be promoted to the upper rungs. ASHA is formally defined in Algorithm 2. Given its asynchronous nature it does not require the user to pre-specify the number of configurations to evaluate, but it otherwise requires the same inputs as SHA. Note that the `run_then_return_val_loss` subroutine in `ASHA` is asynchronous and the code execution continues after the job is passed to the worker. ASHA's promotion scheme is laid out in the `get_job` subroutine.

ASHA is well-suited for the large-scale regime, where wall-clock time is constrained to a small multiple of the time needed to train a single model. For ease of comparison with SHA, assume training time scales linearly with the resource. Consider the example of Bracket 0 shown in Figure 1, and assume we can run ASHA with 9 machines. Then ASHA returns a fully trained configuration in $13/9 \times time(R)$, since 9 machines are sufficient to promote configurations to the next rung in the same time it takes to train a single configuration in the rung. Hence, the training time for a configuration in rung 0 is $1/9 \times time(R)$, for rung 1 it is $1/3 \times time(R)$, and for rung 2 it is $time(R)$. In general, $\eta^{\log_\eta(R)-s}$ machines are needed to advance a configuration to the next rung in the same time it takes to train a single configuration in the rung,

Algorithm 2 Asynchronous Successive Halving (ASHA)

```
input minimum resource $r$, maximum resource $R$, reduction factor $\eta$, minimum early-stopping rate $s$
function ASHA()
  repeat
    for for each free worker do
      $(\theta, k)$ = get_job()
      run_then_return_val_loss($\theta, r\eta^{s+k}$)
    end for
    for completed job $(\theta, k)$ with loss $l$ do
      Update configuration $\theta$ in rung $k$ with loss $l$.
    end for
  until desired
end function
function get_job()
  // Check if there is a promotable config.
  for $k = \lfloor \log_\eta(R/r) \rfloor - s - 1, \ldots, 1, 0$ do
    candidates = top_k(rung $k$, |rung $k$|/$\eta$)
    promotable = $\{t \in$ candidates : $t$ not promoted$\}$
    if |promotable| $> 0$ then
      return promotable$[0], k + 1$
    end if
    // If not, grow bottom rung.
    Draw random configuration $\theta$.
    return $\theta, 0$
  end for
end function
```

and it takes $\eta^{s+i-\log_\eta(R)} \times time(R)$ to train a configuration in rung i. Hence, ASHA can return a configuration trained to completion in time

$$\left(\sum_{i=s}^{\log_\eta(R)} \eta^{i-\log_\eta(R)}\right) \times time(R) \leq 2\, time(R).$$

Moreover, when training is iterative, ASHA can return an answer in $time(R)$, since incrementally trained configurations can be checkpointed and resumed.

Finally, since Hyperband simply runs multiple SHA brackets, we can asynchronously parallelize Hyperband by either running multiple brackets of ASHA or looping through brackets of ASHA sequentially as is done in the original Hyperband. We employ the latter looping scheme for asynchronous Hyperband in the next section.

3.3 Algorithm Discussion

ASHA is able to remove the bottleneck associated with synchronous promotions by incurring a small number of incorrect promotions, i.e. configurations that were promoted early on but are not in the top $1/\eta$ of configurations in hindsight. By the law of large numbers, we expect to erroneously promote a vanishing fraction of configurations in each rung as the number of configurations grows. Intuitively, in the first rung with n evaluated configurations, the number of mispromoted configurations is roughly $\sqrt{n}$, since the process resembles the convergence of an empirical cumulative distribution function (CDF) to its expected value (Dvoretzky et al., 1956). For later rungs, although the configurations are no longer i.i.d. since they were advanced based on the empirical CDF from the rung below, we expect this dependence to be weak.

We further note that ASHA improves upon SHA in two ways. First, Li et al. (2018) discusses two SHA variants: finite horizon (bounded resource R per configuration) and infinite horizon (unbounded resources R per configuration). ASHA consolidates these settings into one algorithm. In Algorithm 2, we do not promote configurations that have been trained for R, thereby restricting the number of rungs. However, this algorithm trivially generalizes to the infinite horizon; we can remove this restriction so that the maximum resource per configuration increases naturally as configurations are promoted to higher rungs. In contrast, SHA does not naturally extend to the infinite horizon setting, as it relies on the doubling trick and must rerun brackets with larger budgets to increase the maximum resource.

Additionally, SHA does not return an output until a single bracket completes. In the finite horizon this means that there is a constant interval of (# of rungs $\times$ $time(R)$) between receiving outputs from SHA. In the infinite horizon this interval doubles between outputs. In contrast, ASHA grows the bracket incrementally instead of in fixed budget intervals. To further reduce latency, ASHA uses intermediate losses to determine the current best performing configuration, as opposed to only considering the final SHA outputs.

4 Empirical Evaluation

We first present results in the sequential setting to justify our choice of focusing on SHA and to compare SHA to ASHA. We next evaluate ASHA in parallel environments on three benchmark tasks.

4.1 Sequential Experiments

We benchmark Hyperband and SHA against PBT, BOHB (synchronous SHA with Bayesian optimization as introduced by Falkner et al. (2018)), and Fabolas, and examine the relative performance of SHA versus ASHA and Hyperband versus asynchronous Hyperband. As mentioned previously, asynchronous Hyperband loops through brackets of ASHA with different early-stopping rates.

We compare ASHA against PBT, BOHB, and synchronous SHA on two benchmarks for CIFAR-10: (1) tuning a convolutional neural network (CNN) with the cuda-convnet architecture and the same search space as (Li et al., 2017);

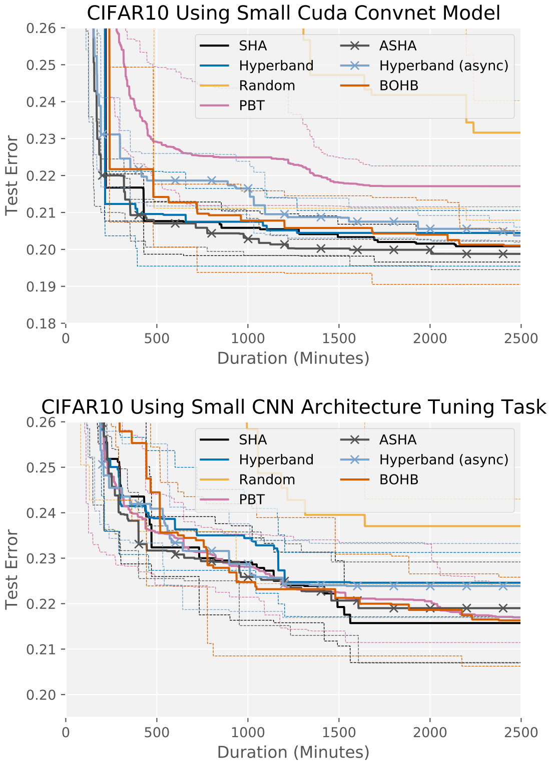


Figure 2. **Sequential experiments** (1 worker). Average across 10 trials is shown for each hyperparameter optimization method. Gridded lines represent top and bottom quartiles of trials.

and (2) tuning a CNN architecture with varying number of layers, batch size, and number of filters. The details for the search spaces considered and the settings we used for each search method can be found in Appendix A.3. Note that BOHB uses SHA to perform early-stopping and differs only in how configurations are sampled; while SHA uses random sampling, BOHB uses Bayesian optimization to adaptively sample new configurations. In the following experiments, we run BOHB using the same early-stopping rate as SHA and ASHA instead of looping through brackets with different early-stopping rates as is done by Hyperband.

The results on these two benchmarks are shown in Figure 2. On benchmark 1, Hyperband and all variants of SHA (i.e., SHA, ASHA, and BOHB) outperform PBT by 3×. On benchmark 2, while all methods comfortably beat random search, SHA, ASHA, BOHB and PBT performed similarly and slightly outperform Hyperband and asynchronous Hyperband. This last observation (i) corroborates the results in Li et al. (2017), which found that the brackets with the most aggressive early-stopping rates performed the best; and (ii) follows from the discussion in Section 2 noting that PBT is more similar in spirit to SHA than Hyperband, as PBT / SHA both require user-specified early-stopping rates (and are more aggressive in their early-stopping behavior in these experiments). We observe that SHA and ASHA are competitive with BOHB, despite the adaptive sampling scheme used by BOHB. Additionally, for both tasks, introducing asynchrony does not consequentially impact the performance of ASHA (relative to SHA) or asynchronous Hyperband (relative to Hyperband). This not surprising; as discussed in Section 3.3, we expect the number of ASHA mispromotions to be square root in the number of configurations.

Finally, due to the nuanced nature of the evaluation framework used by Klein et al. (2017a), we present our results on 4 different benchmarks comparing Hyperband to Fabolas in Appendix A.2. In summary, our results show that Hyperband, specifically the first bracket of SHA, tends to outperform Fabolas while also exhibiting lower variance across experimental trials.

4.2 Limited-Scale Distributed Experiments

We next compare ASHA to synchronous SHA, the parallelization scheme discussed in Section 3.1; BOHB; and PBT on the same two tasks. For each experiment, we run each search method with 25 workers for 150 minutes. We use the same setups for ASHA and PBT as in the previous section. We run synchronous SHA and BOHB with default settings and the same η and early-stopping rate as ASHA.

Figure 3 shows the average test error across 5 trials for each search method. On benchmark 1, ASHA evaluated over 1000 configurations in just over 40 minutes with 25 workers and found a good configuration (error rate below 0.21) in approximately the time needed to train a single model, whereas it took ASHA nearly 400 minutes to do so in the sequential setting (Figure 2). Notably, we only achieve a 10× speedup on 25 workers due to the relative simplicity of this task, i.e., it only required evaluating a few hundred configurations to identify a good one in the sequential setting. In contrast, for the more difficult search space used in benchmark 2, we observe linear speedups with ASHA, as the $\sim$ 700 minutes needed in the sequential setting (Figure 2) to reach a test error below 0.23 is reduced to under 25 minutes in the distributed setting.

Compared to synchronous SHA and BOHB, ASHA finds a good configuration $1.5\times$ as fast on benchmark 1 while BOHB finds a slightly better final configuration. On benchmark 2, ASHA performs significantly better than synchronous SHA and BOHB due to the higher variance in training times between configurations (the average time required to train a configuration on the maximum resource R is 30 minutes with a standard deviation of 27 minutes), which exacerbates the sensitivity of synchronous SHA to stragglers (see Appendix A.1). BOHB actually underperforms synchronous SHA on benchmark 2 due to its bias towards more computationally expensive configurations, reducing the number of configurations trained to completion

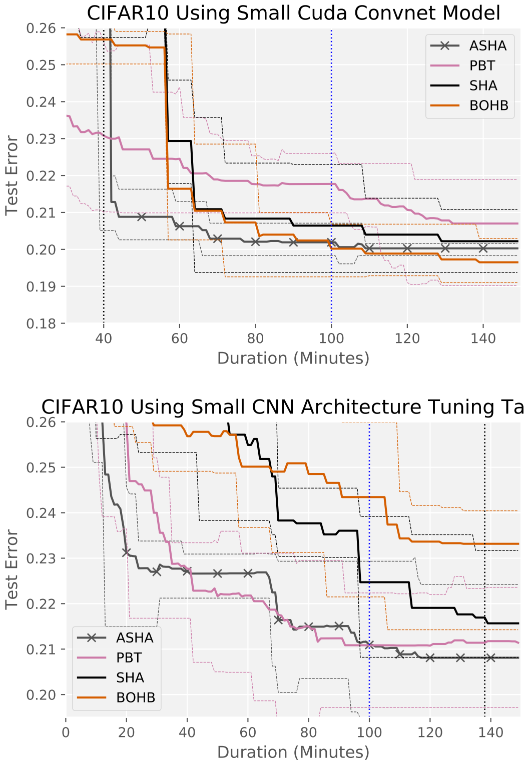


Figure 3. **Limited-scale distributed experiments** with 25 workers. For each searcher, the average test error across 5 trials is shown in each plot. The light dashed lines indicate the min/max ranges. The dotted black line represents the time needed to train the most expensive model in the search space for the maximum resource R. The dotted blue line represents the point at which 25 workers in parallel have performed as much work as a single machine in the sequential experiments (Figure 2).

within the given time frame.

We further note that ASHA outperforms PBT on benchmark 1; in fact the minimum and maximum range for ASHA across 5 trials does not overlap with the average for PBT. On benchmark 2, PBT slightly outperforms asynchronous Hyperband and performs comparably to ASHA. However, note that the ranges for the searchers share large overlap and the result is likely not significant. Overall, ASHA outperforms PBT, BOHB and SHA on these two tasks. This improved performance, coupled with the fact that it is a more principled and general approach than either BOHB or PBT (e.g., agnostic to resource type and robust to hyperparameters that change the size of the model), further motivates its use for the large-scale regime.

4.3 Tuning Large-Scale Language Models

We tune a one layer LSTM language model for next word prediction on the Penn Treebank dataset (Marcus et al., 1993). Our search space is constructed based off of the LSTMs considered in Zaremba et al. (2014), with the largest model in our search space matching their large LSTM (see Appendix A.5 for more details). Each tuner is given 500 workers and $6 \times time(R)$, i.e., $6\times$ the average time needed to train a single model. For ASHA, we set $\eta = 4$, $r = {}^{R}\!/_{64}$, and $s = 0$; asynchronous Hyperband loops through brackets $s = 0, 1, 2, 3$. We compare to Vizier without the performance curve early-stopping rule (Golovin et al., 2017).[1]

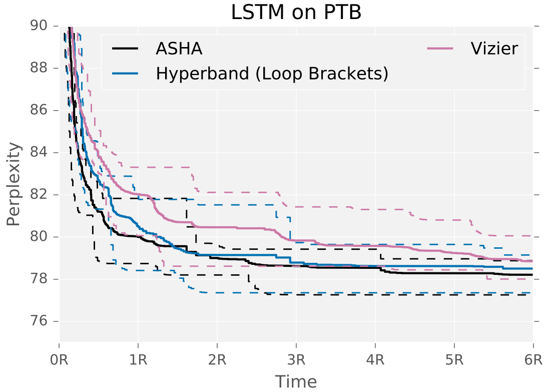


Figure 4. **Large-scale ASHA benchmark** requiring weeks to run with 500 workers. The x-axis is measured in units of average time to train a single configuration for R resource. The average across 5 trials is shown, with dashed lines indicating min/max ranges.

The results in Figure 4 show that ASHA and asynchronous Hyperband found good configurations for this task in $1 \times time(R)$. Additionally, ASHA and asynchronous Hyperband are both about $3\times$ faster than Vizier at finding a configuration with test perplexity below 80, despite being much simpler and easier to implement. Furthermore, the best model found by ASHA achieved a test perplexity of 76.6, which is significantly better than 78.4 reported for the large LSTM in Zaremba et al. (2014). We also note that asynchronous Hyperband initially lags behind ASHA, but eventually catches up at around $1.5 \times time(R)$.

Notably, we observe that certain hyperparameter configurations in this benchmark induce perplexities that are orders of magnitude larger than the average case perplexity. Model-based methods that make assumptions on the data distribution, such as Vizier, can degrade in performance without further care to adjust this signal. We attempted to alleviate this by capping perplexity scores at 1000 but this still significantly hampered the performance of Vizier. We view robustness to these types of scenarios as an additional benefit of ASHA and Hyperband.

[1] At the time of running the experiment, it was brought to our attention by the team maintaining the Vizier service that the early-stopping code contained a bug which negatively impacted its performance. Hence, we omit the results here.

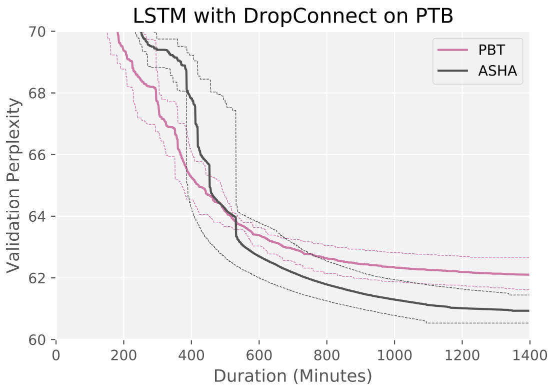


Figure 5. **Modern LSTM benchmark** with DropConnect (Merity et al., 2018) using 16 GPUs. The average across 5 trials is shown, with dashed lines indicating min/max ranges.

4.3.1 *Tuning Modern LSTM Architectures*

We next performed a follow up experiment on the Penn Treebank dataset to tune models with improved perplexity. Our starting point was the work of Merity et al. (2018), which introduced a near state-of-the-art LSTM architecture with a more effective regularization scheme called DropConenct. We constructed a search space around their configuration and ran ASHA and PBT, each with 16 GPUS on one `p2.16xlarge` instance on AWS. For ASHA, we used $\eta = 4$, $r = 1$ epoch, $R = 256$ epochs, and $s = 0$. For PBT, we use a population size to 20, a maximum resource of 256 epochs, and perform explore/exploit every 8 epochs using the same settings as the previous experiments.

Figure 5 shows that while PBT performs better initially, ASHA soon catches up and finds a better final configuration; in fact, the min/max ranges for ASHA and PBT do not overlap at the end. We then trained the best configuration found by ASHA for more epochs and reached validation and test perplexities of 60.2 and 58.1 respectively before fine-tuning and 58.7 and 56.3 after fine-tuning. For reference, Merity et al. (2018) reported validation and test perplexities respectively of 60.7 and 58.8 without fine-tuning and 60.0 and 57.3 with fine-tuning. This demonstrates the effectiveness of ASHA in the large-scale regime for modern hyperparameter optimization problems.

5 PRODUCTIONIZING ASHA

While integrating ASHA in SYSTEMX to deliver production-quality hyperparameter tuning functionality, we encountered several fundamental design decisions that impacted usability, computational performance, and reproducibility. We next discuss each of these design decisions along with proposed systems optimizations for each decision.

5.1 Usability

Ease of use is one of the most important considerations in production; if an advanced method is too cumbersome to use, its benefits may never be realized. In the context of hyperparameter optimization, classical methods like random or grid search require only two intuitive inputs: *number of configurations* (n) and *training resources per configuration* (R). In contrast, as a byproduct of adaptivity, all of the modern methods we considered in this work have many internal hyperparameters. ASHA in particular has the following internal settings: elimination rate η, early-stopping rate s, and, in the case of asynchronous Hyperband, the brackets of ASHA to run. To facilitate use and increase adoption of ASHA, we simplify its user interface to require the same inputs as random search and grid search, exposing the internal hyperparameters of ASHA only to advanced users.

Selecting ASHA default settings. Our experiments in Section 4 and the experiments conducted by Li et al. (2018) both show that aggressive early-stopping is effective across a variety of different hyperparameter tuning tasks. Hence, using both works as guidelines, we propose the following default settings for ASHA:

- Elimination rate: we set $\eta = 4$ so that the top 1/4 of configurations are promoted to the next rung.
- Maximum early-stopping rate: we set the maximum early-stopping rate for bracket s_0 to allow for a maximum of 5 rungs which indicates a minimum resource of $r = (1/4^4)R = R/256$. Then the minimum resource per configuration for a given bracket s is $r_s = r\eta^s$.
- Brackets to run: to increase robustness to misspecification of the early-stopping rate, we default to running the three most aggressively early-stopping brackets $s = 0, 1, 2$ of ASHA. We exclude the two least aggressive brackets (i.e. s_4 with $r_4 = R$ and s_3 with $r_3 = R/4$) to allow for higher speedups from early-stopping. We define this default set of brackets as the ‘standard’ set of early-stopping brackets, though we also expose the options for more conservative or more aggressive bracket sets.

Using n as ASHA’s stopping criterion. Algorithm 2 does not specify a stopping criterion; instead, it relies on the user to stop once an implicit condition is met, e.g., number of configurations evaluated, compute time, or minimum performance achieved. In a production environment, we decided to use the number of configurations n as an explicit stopping criterion both to match the user interface for random and grid search, and to provide an intuitive connection to the underlying difficulty of the search space. In contrast, setting a maximum compute time or minimum performance threshold requires prior knowledge that may not be available.

From a technical perspective, n is allocated to the different brackets while maintaining the same total training resources across brackets. We do this by first calculating the average budget per configuration for a bracket (assuming no incorrect promotions), and then allocating configurations to brackets according to the inverse of this ratio. For concreteness, let B be the set of brackets we are considering, then the average resource for a given bracket s is

$$\bar{r}_s = \frac{\text{\# of Rungs}}{\eta^{\lfloor \log_\eta R/r \rfloor - s}}.$$

For the default settings described above, this corresponds to $\bar{r}_0 = 5/256$, $\bar{r}_1 = 4/64$, and $\bar{r}_2 = 3/16$, and further translates to 70.5%, 22.1%, and 7.1% of the configurations being allocated to brackets s_0, s_1, and s_2, respectively.

Note that we still run each bracket asynchronously; the allocated number of configurations n_s for a particular bracket s simply imposes a limit on the width of the bottom rung. In particular, upon reaching the limit n_s in the bottom rung, the number of pending configurations in the bottom rung is at most equal to the number of workers, k. Therefore, since blocking occurs once a bracket can no longer add configuration to the bottom rung and must wait for promotable configurations, for large-scale problems where $n_s \gg k$, limiting the width of rungs will not block promotions until the bracket is near completion. In contrast, synchronous SHA is susceptible to blocking from stragglers throughout the entire process, which can greatly reduce both the latency and throughput of configurations promoted to the top rung (e.g. Section 4.2, Appendix A.1).

5.2 Automatic Scaling of Parallel Training

The promotion schedule for ASHA geometrically increases the resource per configuration as we move up the rungs of a bracket. Hence, the average training time for configurations in higher rungs increases drastically for computation that scales linearly or super-linearly with the training resource, presenting an opportunity to speed up training by using multiple GPUs. We explore autoscaling of parallel training to exploit this opportunity when resources are available.

We determine the maximum degree of parallelism for autoscaling a training task using an efficiency criteria motivated by the observation that speedups from parallel training do not scale linearly with the number of GPUs (Krizhevsky, 2014; Szegedy et al., 2014; You et al., 2017; You et al., 2017; Goyal et al., 2017). More specifically, we can use the Paleo framework, introduced by Qi et al. (2017), to estimate the cost of training neural networks in parallel given different specifications. Qi et al. (2017) demonstrated that the speedups from parallel training computed using Paleo are fairly accurate when compared to the actual observed speedups for a variety of models.

Figure 6 shows Paleo applied to Inception on ImageNet (et al., 2016) to estimate the training time with different numbers of GPUs under strong scaling (i.e. fixed batch size with increasing parallelism), Butterfly AllReduce communication scheme, specified hardware settings (namely Tesla K80 GPU and 20G Ethernet), and a batch size of 1024.

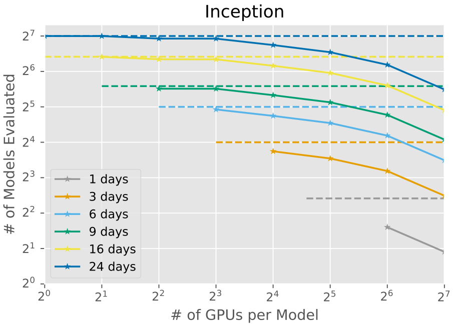


Figure 6. **Tradeoffs for parallel training of Imagenet using Inception V3.** Given that each configuration takes 24 days to train on a single Tesla K80 GPU, we chart the estimated number (according to the Paleo performance model) of configurations evaluated by 128 Tesla K80s as a function of the number of GPUs used to train each model for different time budgets. The dashed line for each color represents the number of models evaluated under perfect scaling, i.e. n GPUs train a single model n times as fast, and span the feasible range for number of GPUs per model in order to train within the allocated time budget. As expected, more GPUs per configuration are required for smaller time budgets and the total number of configurations evaluated decreases with number of GPUs per model due to decreasing marginal benefit.

The diminishing returns associated with using more GPUs to train a single model is evident in Figure 6. Additionally, there is a tradeoff between using resources to train a model faster to reduce latency versus evaluating more configurations to increase throughput. Using the predicted tradeoff curves generated using Paleo, we can automatically limit the number of GPUs per configuration to guarantee a certain degree of efficiency relative to perfect linear scaling, e.g., if the desired level of efficiency is at least 75%, then we would limit the number of GPUs per configuration for Inception to at most 16 GPUs.

5.3 Resource Allocation

Whereas research clusters often require users to specify the number of workers requested and allocate workers on a first-in-first-out (FIFO) fashion, this scheduling mechanism is poorly suited for production settings for two main reasons. First, as we discuss below in the context of ASHA, machine learning workflows can have variable resource requirements over the lifetime of a job, and forcing users to specify static resource requirements can result in suboptimal cluster utilization. Second, FIFO scheduling can result in poor sharing of cluster resources among users, as a single large job could saturate the cluster and block all other user jobs.

We address these issues with a centralized fair-share scheduler that adaptively allocates resources over the lifetime of each job. Such a scheduler must both (i) determine the appropriate amount of parallelism for each individual job, and (ii) allocate computational resources across all user jobs. In the context of an ASHA workload, the scheduler automatically determines the maximum resource requirement at any given time based on the inputs to ASHA and the parallel scaling profile determined by Paleo. Then, the scheduler allocates cluster resources by considering the resource requirements of all jobs while maintaining fair allocation across users. We describe each of these components in more detail below.

Algorithm level resource allocation. Recall that we propose to use the number of configurations, n, as a stopping criteria for ASHA in production settings. Crucially, this design decision limits the maximum degree of parallelism for an ASHA job. If n is the number of desired configurations for a given ASHA bracket and κ the maximum allowable training parallelism, e.g., as determined by Paleo, then at initialization, the maximum parallelism for the bracket is $n\kappa$. We maintain a stack of training tasks $\mathbf{S}$ that is populated initially with all configurations for the bottom rung n. The top task in $\mathbf{S}$ is popped off whenever a worker requests a task and promotable configurations are added to the top of $\mathbf{S}$ when tasks complete. As ASHA progresses, the maximum parallelism is adaptively defined as $\kappa|\mathbf{S}|$. Hence, an adaptive worker allocation schedule that relies on $\kappa|\mathbf{S}|$ would improve cluster utilization relative to a static allocation scheme, without adversely impacting performance.

Cluster level resource allocation. Given the maximum degree of parallelism for any ASHA job, the scheduler then allocates resources uniformly across all jobs while respecting these maximum parallelism limits. We allow for an optional priority weighting factor which allows certain jobs to receive a larger ratio of the total computational resources. Resource allocation is performed using a water-filling scheme where we compute the allocation per job, and then distribute any allocation above the maximum resource requirements to the remaining jobs uniformly.

For concreteness, consider a scenario in which we have a cluster of 32 GPUs shared between a group of users. When a single user is running an ASHA job with 8 configurations in $\mathbf{S_1}$ and a maximum training parallelism of $\kappa_1 = 4$, the scheduler will allocate all 32 GPUs to this ASHA job. When another user submits an ASHA job with a maximum parallelism of $\kappa_2|\mathbf{S_2}| = 64$, the central scheduler will then allocate 16 GPUs to each user. This simple scenario demonstrate how our central scheduler allows jobs to benefit from maximum parallelism when the computing resources are available, while maintaining fair allocation across jobs in the presence of resource contention.

5.4 Reproducibility in Distributed Environments

Reproducibility is critical in production settings to instill trust during the model development process; foster collaboration and knowledge transfer across teams of users; and allow for fault tolerance and iterative refinement of models. However, ASHA introduces two primary reproducibility challenges, each of which we describe below.

Pausing and restarting configurations. There are many sources of randomness when training machine learning models; some source can be made deterministic by setting the random seed, while others related to GPU floating-point computations and CPU multi-threading are harder to avoid without performance ramifications. Hence, reproducibility when resuming promoted configurations requires carefully checkpointing all stateful objects pertaining to the model. At a minimum this includes the model weights, model optimizer state, random number generator states, and data generator state. We provide a checkpointing solution that facilitates reproducibility in the presence of stateful variables and seeded random generators. The availability of deterministic GPU floating-point computations is dependent on the deep learning framework, but we allow users to control for all other sources of randomness during training.

Asynchronous promotions. To allow for full reproducibility of ASHA, we track the sequence of all promotions made within a bracket. This sequence fixes the nondeterminism from asynchrony, allowing subsequent replay of the exact promotions as the original run. Consequently, we can reconstruct the full state of a bracket at any point in time, i.e. which configurations are on which rungs and which training tasks are in the stack.

Taken together, reproducible checkpoints and full bracket states allow us to seamlessly resume hyperparameter tuning jobs when crashes happen and allow users to request to evaluate more configurations if desired. For ASHA, refining hyperparameter selection by resuming an existing bracket is highly beneficial, since a wider rung gives better empirical estimates of the top $1/\eta$ configurations, thereby resulting in fewer incorrect promotions.

6 Conclusion

In this paper, we addressed the problem of developing a production-quality system for hyperparameter tuning by introducing ASHA, a theoretically principled method for simple and robust massively parallel hyperparameter optimization. We presented empirical results demonstrating that ASHA outperforms state-of-the-art methods Fabolas, PBT, BOHB, and Vizier in a suite of hyperparameter tuning benchmarks. Finally, we provided systems level solutions to improve the effectiveness of ASHA that are applicable to existing systems that support our algorithm.

References

Agarwal, A., Duchi, J., Bartlett, P. L., and Levrard, C. Oracle inequalities for computationally budgeted model selection. In *COLT*, 2011.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Bergstra, J., Bardenet, R., Bengio, Y., and Kegl., B. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, 2015.

Dvoretzky, A., Kiefer, J., and Wolfowitz, J. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, 27:642–669, 1956.

et al., D. M. Announcing tensorflow 0.8 now with distributed computing support!, 2016. URL https://research.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pp. 1436–1445, 2018.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. Efficient and robust automated machine learning. In *NIPS*, 2015.

Ginsbourger, D., Le Riche, R., and Carraro, L. Kriging is well-suited to parallelize optimization. In *Computational Intelligence in Expensive Optimization Problems*, pp. 131–162. Springer, 2010.

Golovin, D., Sonik, B., Moitra, S., Kochanski, G., Karro, J., and D.Sculley. Google vizier: A service for black-box optimization. In *KDD*, 2017.

González, J., Zhenwen, D., Hennig, P., and Lawrence, N. Batch bayesian optimization via local penalization. In *AISTATS*, 2016.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv:1706.02677*, 2017.

György, A. and Kocsis, L. Efficient multi-start strategies for local search algorithms. *JAIR*, 41, 2011.

Hutter, F., Hoos, H., and Leyton-Brown., K. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, 2011.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. Population based training of neural networks. *arXiv:1711.09846*, 2017.

Jamieson, K. and Talwalkar, A. Non-stochastic best arm identification and hyperparameter optimization. In *AISTATS*, 2015.

Kandasamy, K., Krishnamurthy, A., Schneider, J., and Póczos, B. Parallelised bayesian optimisation via thompson sampling. In *International Conference on Artificial Intelligence and Statistics*, 2018.

Karnin, Z., Koren, T., and Somekh, O. Almost optimal exploration in multi-armed bandits. In *ICML*, 2013.

Kim, J., Kim, M., Park, H., Kusdavletov, E., Lee, D., Kim, A., Kim, J., Ha, J., and Sung, N. CHOPT : Automated hyperparameter optimization framework for cloud-based machine learning platforms. *arXiv:1810.03527*, 2018.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. Fast bayesian optimization of machine learning hyperparameters on large datasets. *AISTATS*, 2017a.

Klein, A., Faulkner, S., Springenberg, J., and Hutter, F. Learning curve prediction with bayesian neural networks. In *ICLR*, 2017b.

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

Krizhevsky, A. Learning multiple layers of features from tiny images. In *Technical report, Department of Computer Science, Univsersity of Toronto*, 2009.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997*, 2014.

Krueger, T., Panknin, D., and Braun, M. Fast cross-validation via sequential testing. In *JMLR*, 2015.

Li, A., Spyra, O., Perel, S., Dalibard, V., Jaderberg, M., Gu, C., Budden, D., Harley, T., and Gupta, P. A generalized framework for population based training. *arXiv:1902.01894*, 2019.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. *Proc. of ICLR*, 17, 2017.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL http://jmlr.org/papers/v18/16-558.html.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. Tune: A research platform for distributed model selection and training. In *ICML AutoML Workshop*, 2018.

Marcus, M., Marcinkiewicz, M., and Santorini, B. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

Merity, S., Keskar, N., and Socher, R. Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=SyyGPP0TZ.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *ICLR*, 2017.

Sabharwal, A., Samulowitz, H., and Tesauro, G. Selecting near-optimal learners via incremental data allocation. In *AAAI*, 2016.

Sermanet, P., Chintala, S., and LeCun, Y. Convolutional neural networks applied to house numbers digit classification. In *ICPR*, 2012.

Shah, A. and Ghahramani, Z. Parallel predictive entropy search for batch global optimization of expensive objective functions. In *NIPS*, 2015.

Snoek, J., Larochelle, H., and Adams, R. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Srinivas, N., Krause, A., Kakade, S., and Seeger, M. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2010.

Swersky, K., Snoek, J., and Adams, R. Multi-task bayesian optimization. In *NIPS*, 2013.

Swersky, K., Snoek, J., and Adams, R. P. Freeze-thaw bayesian optimization. *arXiv:1406.3896*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *arXiv:1409.4842*, 2014.

Wu, J. and Frazier, P. The parallel knowledge gradient method for batch bayesian optimization. In *NIPS*, 2016.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 595–610, 2018.

You, Y., Gitman, I., and Ginsburg, B. Scaling SGD batch size to 32k for imagenet training. *arXiv:1708.03888*, 2017.

You, Y., Zhang, Z., Hsieh, C.-J., Demmel, J., and Keutzer, K. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *arXiv:1709.0501*, 2017.

Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

A Appendix

As part of our supplementary material, (1) compare the impact of stragglers and dropped jobs on synchronous SHA and ASHA, (2) present the comparison to Fabolas in the sequential setting and (3) provide additional details for the empirical results shown in Section 4.

A.1 Comparison of Synchronous SHA and ASHA

We use simulated workloads to evaluate the impact of stragglers and dropped jobs on synchronous SHA and ASHA. For our simulated workloads, we run synchronous SHA with $\eta = 4$, $r = 1$, $R = 256$, and $n = 256$ and ASHA with the same values and the maximum early-stopping rate $s = 0$. Note that BOHB (Falkner et al., 2018), one of the competitors we empirically compare to in Section 4, is also susceptible to stragglers and dropped jobs since it uses synchronous SHA as its parallelization scheme but leverages Bayesian optimization to perform adaptive sampling.

For these synthetic experiments, we assume that the expected training time for each job is the same as the allocated resource. We simulate stragglers by multiplying the expected training time by $(1 + |z|)$ where z is drawn from a normal distribution with mean 0 and a specified standard deviation. We simulated dropped jobs by assuming that there is a given p probability that a job will be dropped at each time unit, hence, for a job with a runtime of 256 units, the probability that it is not dropped is $1 - (1 - p)^{256}$.

Figure 7 shows the number of configurations trained to completion (left) and time required before one configuration is trained to completion (right) when running synchronous SHA and ASHA using 25 workers. For each combination of training time standard deviation and drop probability, we simulate ASHA and synchronous SHA 25 times and report the average. As can be seen in Figure 7a, ASHA trains many more configurations to completion than synchronous SHA when the standard deviation is high; we hypothesize that this is one reason ASHA performs significantly better than synchronous SHA and BOHB for the second benchmark in Section 4.2. Figure 7b shows that ASHA returns a configuration trained for the maximum resource R much faster than synchronous SHA when there is high variability in training time (i.e., stragglers) and high risk of dropped jobs. Although ASHA is more robust than synchronous SHA to stragglers and dropped jobs on these simulated workloads, we nonetheless compare synchronous SHA in Section 4.3 and show that ASHA performs better.

A.2 Comparison with Fabolas in Sequential Setting

(Klein et al., 2017a) showed that Fabolas can be over an order of magnitude faster than existing Bayesian optimization methods. Additionally, the empirical studies presented in (Klein et al., 2017a) suggest that Fabolas is faster than Hyperband at finding a good configuration. We conducted our own experiments to compare Fabolas with Hyperband on the following tasks:

1. Tuning an SVM using the same search space as (Klein et al., 2017a).
2. Tuning a convolutional neural network (CNN) with the same search space as Li et al. (2017) on CIFAR-10 (Krizhevsky, 2009).
3. Tuning a CNN on SVHN (Netzer et al., 2011) with varying number of layers, batch size, and number of filters (see Appendix A.4 for more details).

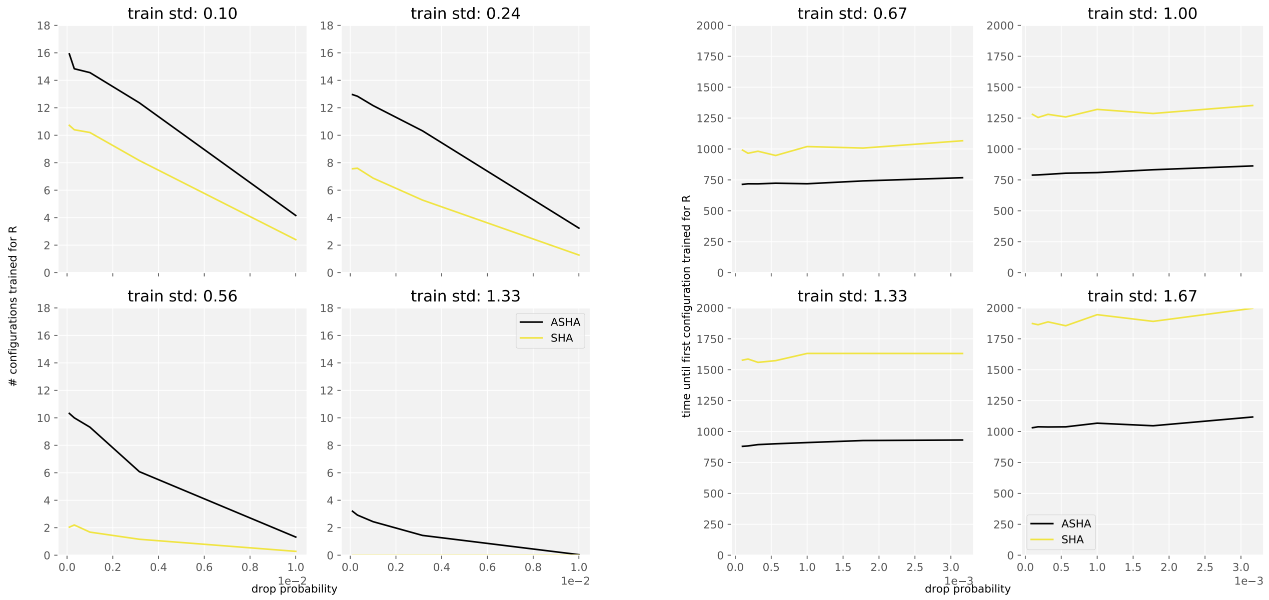
In the case of the SVM task, the allocated resource is number of training datapoints, while for the CNN tasks, the allocated resource is the number of training iterations.

We note that Fabolas was specifically designed for data points as the resource, and hence, is not directly applicable to tasks (2) and (3). However, freeze-thaw Bayesian optimization (Swersky et al., 2014), which was specifically designed for models that use iterations as the resource, is known to perform poorly on deep learning tasks (Domhan et al., 2015). Hence, we believe Fabolas to be a reasonable competitor for tasks (2) and (3) as well, despite the aforementioned shortcoming.

We use the same evaluation framework as Klein et al. (2017a), where the best configuration, also known as the *incumbent*, is recorded through time and the test error is calculated in an offline validation step. Following Klein et al. (2017a), the incumbent for Hyperband is taken to be the configuration with the lowest validation loss and the incumbent for Fabolas is the configuration with the lowest predicted validation loss on the full dataset. Moreover, for these experiments, we set $\eta = 4$ for Hyperband.

Notably, when tracking the best performing configuration for Hyperband, we consider two approaches. We first consider the approach proposed in Li et al. (2018) and used by Klein et al. (2017a) in their evaluation of Hyperband. In this variant, which we refer to as "Hyperband (by bracket)," the incumbent is recorded *after the completion of each SHA bracket*. We also consider a second approach where we record the incumbent *after the completion of each rung* of SHA to make use of intermediate validation losses, similar to what we propose for ASHA (see discussion in Section 3.3 for details). We will refer to Hyperband using this accounting scheme as "Hyperband (by rung)." Interestingly, by leveraging these intermediate losses, we observe that Hyperband actually outperforms Fabolas.

In Figure 8, we show the performance of Hyperband, Fabolas, and random search. Our results show that Hyperband



(a) Average number of configurations trained on R resource.

(b) Average time before a configuration is trained on R resource.

Figure 7. **Simulated workloads comparing impact of stragglers and dropped jobs.** The number of configurations trained for R resource (left) is higher for ASHA than synchronous SHA when the standard deviation is high. Additionally, the average time before a configuration is trained for R resource (right) is lower for ASHA than for synchronous SHA when there is high variability in training time (i.e., stragglers). Hence, ASHA is more robust to stragglers and dropped jobs than synchronous SHA since it returns a completed configuration faster and returns more configurations trained to completion.

(by rung) is competitive with Fabolas at finding a good configuration and will often find a better configuration than Fabolas with less variance. Note that Hyperband loops through the brackets of SHA, ordered by decreasing early-stopping rate; the first bracket finishes when the test error for Hyperband (by bracket) drops. Hence, most of the progress made by Hyperband comes from the bracket with the most aggressive early-stopping rate, i.e. bracket 0.

A.3 Experiments in Section 4.1 and Section 4.2

We use the usual train/validation/test splits for CIFAR-10, evaluate configurations on the validation set to inform algorithm decisions, and report test error. These experiments were conducted using `g2.2xlarge` instances on Amazon AWS.

For both benchmark tasks, we run SHA and BOHB with $n = 256$, $\eta = 4$, $s = 0$, and set $r = R/256$, where $R =$ 30000 iterations of stochastic gradient descent. Hyperband loops through 5 brackets of SHA, moving from bracket $s = 0, r = R/256$ to bracket $s = 4, r = R$. We run ASHA and asynchronous Hyperband with the same settings as the synchronous versions. We run PBT with a population size of 25, which is between the recommended 20–40 (Jaderberg et al., 2017). Furthermore, to help PBT evolve from a good set of configurations, we randomly sample configurations until at least half of the population performs above random guessing.

We implement PBT with truncation selection for the exploit phase, where the bottom 20% of configurations are replaced with a uniformly sampled configuration from the top 20% (both weights and hyperparameters are copied over). Then, the inherited hyperparameters pass through an exploration phase where $3/4$ of the time they are either perturbed by a factor of 1.2 or 0.8 (discrete hyperparameters are perturbed to two adjacent choices), and $1/4$ of the time they are randomly resampled. Configurations are considered for exploitation/exploration every 1000 iterations, for a total of 30 rounds of adaptation. For the experiments in Section 4.2, to maintain 100% worker efficiently for PBT while enforcing that all configurations are trained for within 2000 iterations of each other, we spawn new populations of 25 whenever a job is not available from existing populations.

Vanilla PBT is not compatible with hyperparameters that change the architecture of the neural network, since inherited weights are no longer valid once those hyperparameters are perturbed. To adapt PBT for the architecture tuning task, we fix hyperparameters that affect the architecture in the explore stage. Additionally, we restrict configurations to be trained within 2000 iterations of each other so a fair comparison is made to select configurations to exploit. If we do not

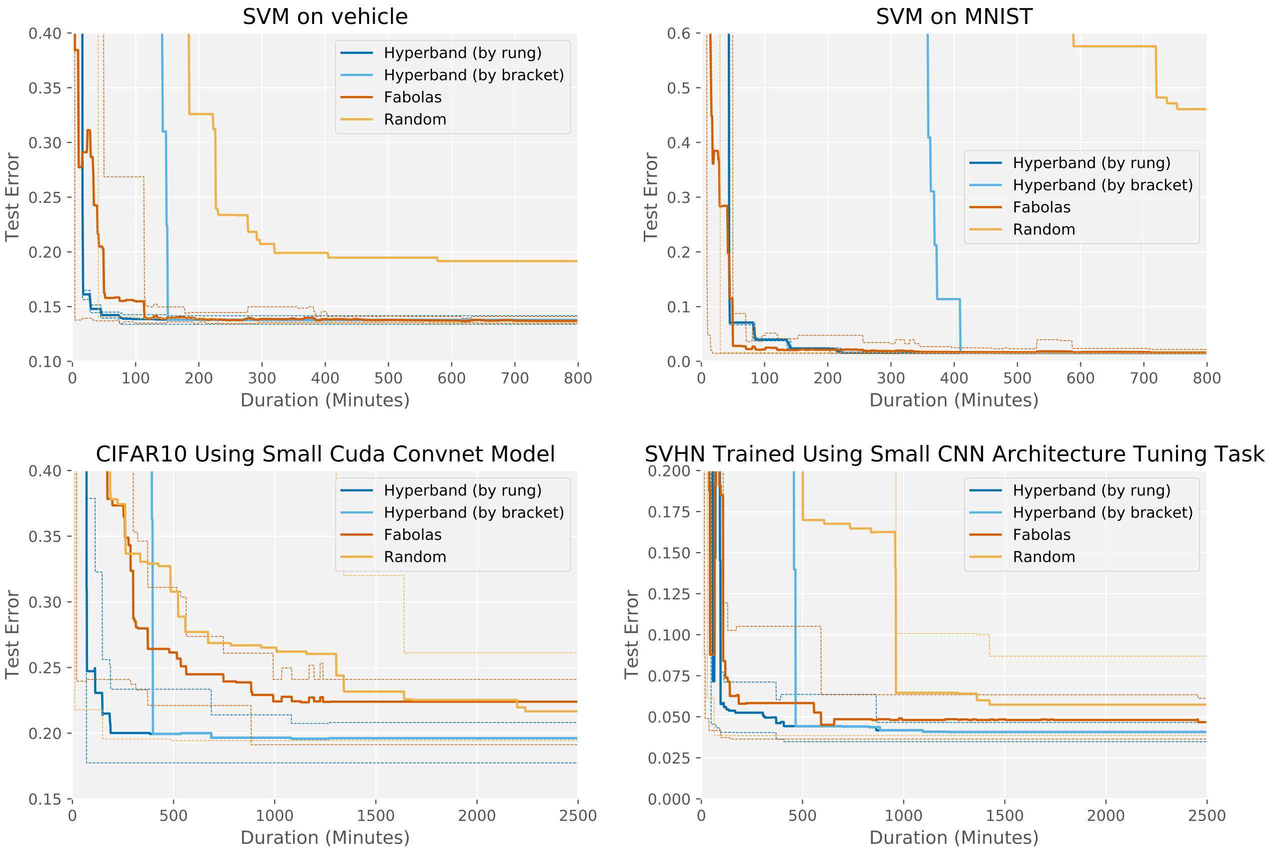


Figure 8. **Sequential Experiments** (1 worker) with Hyperband running synchronous SHA. Hyperband (by rung) records the incumbent after the completion of a SHA rung, while Hyperband (by bracket) records the incumbent after the completion of an entire SHA bracket. The average test error across 10 trials of each hyperparameter optimization method is shown in each plot. Dashed lines represent min and max ranges for each tuning method.

Hyperparameter	Type	Values
batch size	choice	$\{2^6, 2^7, 2^8, 2^9\}$
# of layers	choice	$\{2, 3, 4\}$
# of filters	choice	$\{16, 32, 48, 64\}$
weight init std 1	continuous	$\log\ [10^{-4}, 10^{-1}]$
weight init std 2	continuous	$\log\ [10^{-3}, 1]$
weight init std 3	continuous	$\log\ [10^{-3}, 1]$
l_2 penalty 1	continuous	$\log\ [10^{-5}, 1]$
l_2 penalty 2	continuous	$\log\ [10^{-5}, 1]$
l_2 penalty 3	continuous	$\log\ [10^{-3}, 10^2]$
learning rate	continuous	$\log\ [10^{-5}, 10^1]$

Table 1. Hyperparameters for small CNN architecture tuning task.

impose this restriction, PBT will be biased against configurations that take longer to train, since it will be comparing these configurations with those that have been trained for more iterations.

A.4 Experimental Setup for the Small CNN Architecture Tuning Task

This benchmark tunes a multiple layer CNN network with the hyperparameters shown in Table 1. This search space was used for the small architecture task on SVHN (Section A.2) and CIFAR-10 (Section 4.2). The # of layers hyperparameter indicate the number of convolutional layers before two fully connected layers. The # of filters indicates the # of filters in the CNN layers with the last CNN layer having 2 × # filters. Weights are initialized randomly from a Gaussian distribution with the indicated standard deviation. There are three sets of weight init and l_2 penalty hyperparameters; weight init 1 and l_2 penalty 1 apply to the convolutional layers, weight init 2 and l_2 penalty 2 to the first fully connected layer, and weight init 3 and l_2 penalty 3 to the last fully connected layer. Finally, the learning rate hyperparameter controls the initial learning rate for SGD. All models use a fixed learning rate schedule with the learning rate decreasing by a factor of 10 twice in equally spaced intervals over the training window. This benchmark is run

Hyperparameter	Type	Values
batch size	discrete	$[10, 80]$
# of time steps	discrete	$[10, 80]$
# of hidden nodes	discrete	$[200, 1500]$
learning rate	continuous	$\log\ [0.01, 100.]$
decay rate	continuous	$[0.01, 0.99]$
decay epochs	discrete	$[1, 10]$
clip gradients	continuous	$[1, 10]$
dropout probability	continuous	$[0.1, 1.]$
weight init range	continuous	$\log\ [0.001, 1]$

Table 2. Hyperparameters for PTB LSTM task.

on the SVHN dataset (Netzer et al., 2011) following Sermanet et al. (2012) to create the train, validation, and test splits.

A.5 Experimental Setup for Large-Scale Benchmarks

Hyperparameter	Type	Values
learning rate	continuous	$\log\ [10, 100]$
dropout (rnn)	continuous	$[0.15, 0.35]$
dropout (input)	continuous	$[0.3, 0.5]$
dropout (embedding)	continuous	$[0.05, 0.2]$
dropout (output)	continuous	$[0.3, 0.5]$
dropout (dropconnect)	continuous	$[0.4, 0.6]$
weight decay	continuous	$\log\ [0.5e-6, 2e-6]$
batch size	discrete	$[15, 20, 25]$
time steps	discrete	$[65, 70, 75]$

Table 3. Hyperparameters for 16 GPU near state-of-the-art LSTM task.

The hyperparameters for the LSTM tuning task comparing ASHA to Vizier on the Penn Tree Bank (PTB) dataset presented in Section 4.3 is shown in Table 2. Note that all hyperparameters are tuned on a linear scale and sampled uniform over the specified range. The inputs to the LSTM layer are embeddings of the words in a sequence. The number of hidden nodes hyperparameter refers to the number of nodes in the LSTM. The learning rate is decayed by the decay rate after each interval of decay steps. Finally, the weight initialization range indicates the upper bound of the uniform distribution used to initialize all weights. The other hyperparameters have their standard interpretations for neural networks. The default training (929k words) and test (82k words) splits for PTB are used for training and evaluation (Marcus et al., 1993). We define resources as the number of training records, which translates into the number of training iterations after accounting for certain hyperparameters.

For the task tuning a modern LSTM architecture, we use the code provided by Merity et al. (2018) and construct a search space around the hyperparameter setting that they used. The hyperparameters that we considered along with their associated ranges are shown in Table 3.