SLURM-Managed HyperParameter Optimization

Anusha Chattopadhyay
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Hendrik Borras
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Bernhard Klein
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Holger Fröning[1]
Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University, Germany

Abstract

Hyperparameter optimization (HPO) is essential for achieving state-of-the-art performance in machine learning, yet it is computationally demanding, particularly on shared or resource-constrained clusters. We present a system that integrates the Asynchronous Successive Halving Algorithm (ASHA) with SEML, the SLURM Experiment Management Library – an experiment orchestration layer that provides declarative configuration, provenance, metric tracking, and robust SLURM job management. The resulting open-source tool enables scalable, fault-tolerant HPO on SLURM-managed infrastructure: SEML handles experiment specification, versioning, and scheduling, while ASHA performs asynchronous early stopping and resource reallocation to concentrate computation on promising configurations. Overall, the system streamlines experiment lifecycle management, enables distributed evaluations with minimal manual effort, and reduces the time required to reach high-quality configurations compared to conventional Grid and Random Search methods under similar compute budgets.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Hyperparameter optimization, Asynchronous Successive Halving Algorithm (ASHA), Experiment management, SLURM, SEML, Open Source

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.2

Supplementary Material *Software (Source Code)*: `https://github.com/TUM-DAML/seml/pull/160`

Acknowledgements The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG.

1 Introduction

Machine learning models are now widely used across multiple industries, ranging from healthcare to weather forecasting. During model development, hyperparameters play a critical role. Some hyperparameters control the structure of the model, such as the number of layers or hidden units per layer, while others govern the training process, such as the choice of optimizer, learning rate, and batch size. These hyperparameters differ from model parameters in that they are not learned from the data but are instead predefined model configurations set before training. A poor choice of hyperparameters can lead to overfitting and inefficient training. Inversely, an optimal combination can lead to well performing results.

[1] corresponding author

© Anusha Chattopadhyay, Hendrik Borras, Bernhard Klein, and Holger Fröning;
licensed under Creative Commons License CC-BY 4.0

Hyperparameter Optimization (HPO) algorithms are used to automate the process of finding such optimal combinations. Besides manual tuning, a common baseline approach is Grid Search, which exhaustively evaluates all parameter combinations. However, more efficient algorithms such as Random Search [2], Asynchronous Successive Halving [11], and Hyperband [12] have been developed to reduce computational cost. These algorithms often require multiple concurrent model evaluations, which can be computationally intensive and challenging to manage in shared or resource-constrained environments.

This raises the challenge of effectively using such algorithms on shared systems, such as high-performance computing (HPC) clusters in industry and research alike. HPC systems are usually orchestrated by workload managers. With the Simple Linux Utility for Resource Management (SLURM) [17] being a popular choice as a workload manager. It efficiently manages and schedules jobs across large-scale, shared computing resources. This allows multiple users to submit and run tasks in parallel while handling resource allocation, job prioritization, and load balancing.

This work focuses on selecting and implementing an HPO framework for experiments conducted on such a SLURM-managed compute cluster. The key objectives are efficient resource utilization and seamless SLURM integration. The framework must support parallelized optimization while maintaining predictable performance and minimal scheduling delays. Additionally, it should be well maintained and suitable for long-term use in a shared research computing environment. Keeping these factors in mind, we build on top of the SLURM Experiment Management Library (SEML) [18], due to its native compatibility with SLURM.

We further select the Asynchronous Successive Halving Algorithm (ASHA) [11] as the core HPO algorithm, due to its reliability and efficiency in distributed computing systems. As ASHA is stable and inherently asynchronous, each configuration makes promotion decisions independently based on the metrics it can access, eliminating the need for centralized coordination and scheduling delays. Its ability to cancel less promising jobs early while maintaining simple and predictable execution makes ASHA particularly well-suited for scalable hyperparameter optimization in SLURM-managed distributed environments. In detail, we make the following overarching contributions:

1. Identification of ASHA as promising HPO algorithm for SLURM-managed compute clusters, based on its efficiency, parallelizability, and robustness to resource contention.
2. Extension of SEML [18] to support ASHA, enabling scalable hyperparameter optimization on SLURM-managed compute clusters. We open-source our implementation as an extension to the official SEML GitHub repository.[2]
3. Evaluation by testing the implementation on standard models such as ResNet18, demonstrating the practical effectiveness of ASHA in realistic scenarios.

2 Background and Related Work

In machine learning, a hyperparameter is a parameter that is set before training to configure the model's training process. These can include settings such as the learning rate or the type of optimizer used during training. Others may refer to topological features such as the number of layers. Notably hyperparameters are different from parameters that are learned from data, such as weights and biases, which change during training. Most machine learning models are highly sensitive to their hyperparameters. Thus, a choice of suboptimal hyperparameters can lead to issues such as lack of convergence, inefficient training, and low quality results.

[2] https://github.com/TUM-DAML/seml/pull/160

Manual tuning is when model developers manually search for optimal hyperparameters. This process can be time-consuming, and its effectiveness relies on the developers' intuition. In contrast, *Hyperparameter Optimization (HPO) Algorithms* are algorithms designed to automate the process of hyperparameter tuning. These algorithms automatically search through often vast, non-differentiable search spaces (e.g., 5 hyperparameters with each 10 choices = 100,000 combinations). The automation of this process allows for better generalization and scalability.

Several libraries, such as Ray Tune [13] and Optuna [1], provide a wide range of hyperparameter-optimization algorithms. However, they do not integrate natively with SLURM. Ray Tune, built on the Ray [16] framework for distributed machine learning, assumes a cluster architecture with a single head node orchestrating multiple worker nodes, whereas SLURM launches independent jobs without maintaining such hierarchy. This mismatch makes Ray's structure undesirable in shared, distributed SLURM-based systems. Optuna likewise lacks direct SLURM integration and requires users to manually manage job submission when distributing trials.

2.1 Selected HPO Algorithms

Grid Search is a brute-force optimization algorithm that evaluates all possible combinations of hyperparameters within a predefined grid. While computationally expensive, it guarantees that an optimal solution will be found within the predefined discrete set of hyperparameter values. The results are also more likely to be reproducible during experimentation, as the process is generally deterministic.

Random Search [2] selects hyperparameter values randomly rather than exhaustively evaluating every possible combination. It is often more efficient than Grid Search, particularly as the dimensionality of the search space increases. However, this randomness makes it harder to reproduce results unless the random number generator is carefully seeded.

The **Successive Halving Algorithm (SHA)** [7] is an evolution of Random Search that employs early stopping to conserve computational resources. This algorithm uses a *bracket*, which consists of a sequence of evaluation *rungs*. Each *rung* within the bracket represents a stage where the algorithm evaluates whether a configuration should continue training or be terminated. The decision is based on whether the configuration ranks within the top $\lfloor n/\eta \rfloor$ configurations, where n is the number of active configurations and $\eta > 1$ is the reduction factor that determines the proportion of configurations to retain. All active configurations must reach a predefined rung before any elimination occurs, making this algorithm inherently synchronous.

Hyperband [12], similarly is an early-stopping algorithm that eliminates poorly performing trials. Hyperband is essentially a collection of multiple rounds of SHA, where each round is referred to as a stage. While this improves the inherent exploration/exploitation tradeoff, it also adds even more synchronization requirements. Making Hyperband inefficient in highly parallel and distributed settings.

The **Asynchronous Successive Halving Algorithm** (ASHA) [11] is an asynchronous version of SHA designed for parallel hyperparameter optimization. Unlike Hyperband or SHA, ASHA does not wait for all configurations in a bracket to complete before promoting the best ones at a decision *rung*. It thus evaluates configurations asynchronously, allowing for better utilization of computational resources. While this entails making decisions on incomplete information, which may lead to non-optimal choices, the issue can be compensated by algorithm configuration adjustments. This makes ASHA more efficient in distributed environments.

For its efficiency and scalability, this algorithm was chosen for hyperparameter optimization tasks. Unlike traditional methods like Random Search or Bayesian Optimization [5], it can handle a large number of configurations simultaneously due to its asynchronous nature. This allows for better utilization of computational resources, as it does not require waiting for all configurations to complete before making decisions on promotions. Additionally, ASHA's successive halving strategy effectively allocates resources to the most promising configurations, leading to faster convergence towards optimal hyperparameters. This makes it particularly well-suited for scenarios where training models can be time-consuming and resource-intensive.

2.2 SLURM and SEML

SLURM is a widely adopted, free, and open-source workload manager and job scheduler for Linux and other Unix-like based high-performance computing (HPC) clusters. It provides resource allocation, job scheduling, accounting, and monitoring to coordinate workloads across shared compute infrastructures.

Implications for multi-job algorithms. Because queued jobs start opportunistically rather than in lockstep, workflows that launch multiple jobs must handle asynchronous execution – start times, runtimes, and completion order are not guaranteed to align.

SLURM Experiment Management Library (SEML) [18] is an open-source library designed to manage SLURM jobs. It integrates with Sacred [9] for experiment management and MongoDB [15] for experiment storage. SEML generates job scripts that specify experiment details, resource requests, and commands, which are then submitted to SLURM. Once submitted, SEML tracks job statuses, logs, and results, simplifying reproducibility and debugging. The library includes implementations of Grid Search and Random Search for hyperparameter optimization. As it is specifically designed for SLURM-managed clusters, SEML is an ideal framework for extending with the selected HPO algorithm.

Sacred [9] is a tool to manage computational experiments. It aids in logging and reproduction. SEML is built on top of Sacred and thus leverages some of Sacred's capabilities in its experiment management.

SEML stores experiment information in a *MongoDB* [15] database. This meant that much of the final framework development was dependent on interacting with MongoDB.

3 HPO Algorithm Selection and System Architecture

In this work, we selected ASHA as our HPO algorithm due to its efficiency and effectiveness in hyperparameter optimization tasks, as we will elaborate on in the following.

3.1 Details of ASHA

The following section describes the ASHA [11] algorithm. In Algorithm 1, processes run in parallel, starting whenever resources are available (line 2).

Algorithm 1 Asynchronous Successive Halving Algorithm (ASHA).

1: **Input:** $r_{\max}$: maximum resource per config (e.g., epochs), η: elimination rate, $r_{\min}$: minimum resource to train a configuration for, N: number of ASHA jobs to start
2: **while** compute resources are available and $N < n$ **do** ▷ Run in parallel
3: Sample new configuration c and increment n
4: Train c for $r_{\min}$ resources
5: **for** $k = 0$ **to** $\left\lfloor \log_\eta \left(\frac{r_{\max}}{r_{\min}} \right) \right\rfloor$ **do**
6: $r \leftarrow r_{\min} \cdot \eta^k$
7: Wait until c finishes training for r resources ▷ Configuration reaches evaluation point(rung)
8: Evaluate c and store its performance ▷ Evaluation occurs here
9: **if** c ranks in top $1/\eta$ fraction at rung r **then**
10: Promote c to next rung with resource $r \cdot \eta$
11: **else**
12: Stop training c
13: **break**
14: **end if**
15: **end for**
16: **end while**

In Algorithm 1, line 5 is responsible for deciding the times to check for promotion. This formula counts how many times one can multiply the minimum resource $r_{\min}$ by the factor η before reaching the maximum resource $r_{\max}$.

After a configuration completes training at a given resource level, it is evaluated and compared to others at the same rung. If it ranks in the top η or $\lfloor n/\eta \rfloor$, it is promoted to the next rung with more resources; otherwise, training is stopped early. As this process is asynchronous, each evaluation is done only with processes that have already reached this rung.In scenarios where compute restrictions force ASHA to execute each configuration serially, each successive process can benchmark against prior ones, thereby enabling early stopping.

It is possible to further extend ASHA with a tie-breaking strategy when two configurations achieve the same accuracy. For example, the algorithm could choose which configuration to promote based on factors such as which one arrived first, or select randomly if they arrived simultaneously. However, in our implementation, we avoided tie-breaking, by simply promoting all tied configurations.

3.2 Integration with SEML

Figure 1 depicts the overall system architecture. It describes the overall program and data flow for the presented ASHA extension. When the experiment configuration reaches a stage marked as an evaluation rung, the ASHA class compares it with its neighbors by communicating with the shared MongoDB collection. The implementation is designed to seamlessly integrate ASHA within the existing SEML framework, extending SEML with efficient hyperparameter optimization.

An overview of the integration of ASHA in SEML is given in Algorithm 2. This integration leverages SEML's existing capabilities while introducing ASHA's efficient hyperparameter optimization strategy. The process begins with the configuration file `experiment.yaml`,

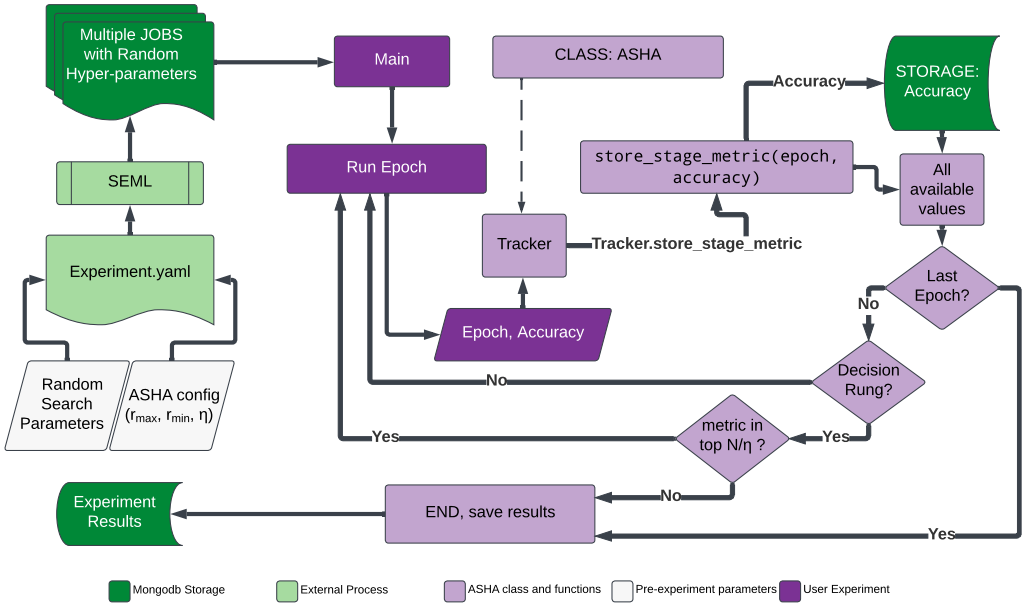


Figure 1 System Architecture of the integration of ASHA with SEML. The components are colour coded according to functionality. The overall process takes the already existing feature of Random Search in SEML to extend it to ASHA. This diagram uses accuracy as the evaluation metric and epochs as the promotion stages.

which defines the hyperparameter search space using Random Search and includes ASHA-specific parameters such as η, $r_{\min}$, and $r_{\max}$. SEML initializes multiple processes based on the defined hyperparameter configurations. Each process imports the `ASHA.py` file, which contains the ASHA class and its associated functions. During training, each process periodically calls the `tracker.store_stage_metric()` function to log the current metric (e.g., accuracy) to a shared MongoDB collection. When a process reaches a decision rung, it queries the MongoDB to retrieve metrics from peer processes that have also reached this rung. Using this data, the ASHA decision-making logic is executed to determine whether the process should be promoted to the next rung or terminated. If the process ranks within the top N/η or meets the promotion criteria, it is promoted; otherwise, a stop signal is sent to terminate the process. This integration allows ASHA to function effectively within the SEML framework, enabling efficient and scalable hyperparameter optimization.

Setting up a Hyperparameter Optimization run with ASHA revolves around 3 basic files and the SEML base libraries:

- **Configuration file (usually named `experiment.yaml`):** mandatory file defining all SLURM and SEML settings, except for MongoDB configurations, which are set separately. It also specifies additional ASHA parameters, such as minimum and maximum resources, while SEML's Random Search configurations define the hyperparameters for the experiments.
- **`main.py`:** is the main experiment script, where the model is implemented. It communicates with the ASHA class, sending experiment data and receiving stop signals. Users may act on these signals to halt the process or perform other cleanup tasks.
- **`asha.py`:** contains the ASHA class along with all functions for ASHA's stopping criteria and logging logic.

Algorithm 2 Overview of the ASHA Process Integrated with SEML.

Input: `experiment.yaml` contains Random Search hyperparameters, ASHA setup: η, $r_{\min}$, $r_{\max}$.
SEML initializes processes via Random Search defined in `experiment.yaml`.
Each process imports the `ASHA.py` file as a `tracker` object, which contains all required ASHA-related functions.
`tracker.store_stage_metric()` logs accuracy (or any other metric) to a shared MongoDB collection at each stage (e.g epoch).
if the current stage corresponds to a decision rung **then**
    Query MongoDB for peer metrics at this rung.
    Perform ASHA decision-making **only** based on available data.
    if the process ranks in the top $\lfloor N/\eta \rfloor$ or meets cutoff **then**
        Promote to the next rung.
    else
        Send stop signal to terminate the process.
    end if
end if

3.3 Storage and Access

As a centralized controller isn't ideal and sometimes non-viable in a shared cluster enviroment, we rely on a MongoDB collection in which each trial logs its progress, enabling coordination across processes.

This is achieved by using the `store_stage_metric()` function to store the current tasks accuracy (metric) in a database collection for each epoch (stage). This means that when a task reaches a decision rung, it checks the database to find the results of all tasks which have already reached and/or crossed this rung (in this case at an epoch in a predefined set calculated using $r_{\min}$ and $r_{\max}$). Following, the performance of all available tasks is compared and the corresponding ASHA promotion action is performed.

To uniquely identify each configuration and avoid conflicts, a Universally Unique Identifier (UUIDv4) is assigned to every job. Each entry stores the epoch, the metric value, and the associated job's UUID. This design prevents multiple jobs from reading and writing the same object simultaneously, ensuring concurrency safety.

With this system, once a job stores its evaluated metric, it no longer needs to modify it; as the metrics are solely used by other jobs for promotion decisions.

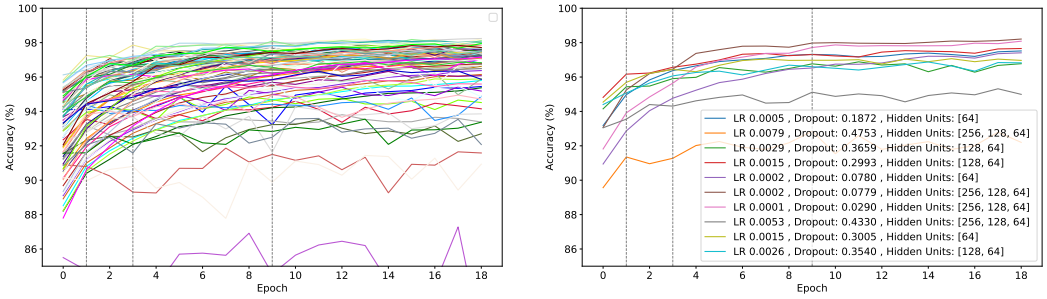
3.4 Configurations

ASHA is configured in the experiment.yaml file with the following metrics: η is the reduction factor and N/η specifies the number of jobs promoted per rung. $r_{\min}$ and $r_{\max}$ set the minimum and maximum rungs at which a job can be promoted. Metrics with "bigger is better" are specified by `metric_increases = True`, while metrics with "less is better" are specified by `metric_increases = False`.

The parameter allocations are derived from the Random Search configurations already defined in SEML's setup. For each job, the full history of the chosen metric (such as accuracy or any other performance measure) is stored in MongoDB. This allows configurations to be compared at each decision rung, enabling ASHA to make promotion or early-stopping decisions based on the recorded metrics.

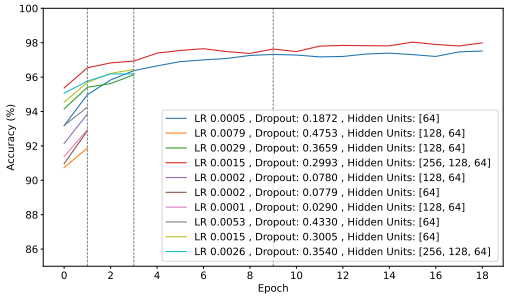
4 Results

We evaluated our ASHA implementation on two widely used image classification benchmarks, MNIST [4] and CIFAR-10 [10], using neural networks of differing complexity with no image augmentation. For MNIST, we used a configurable multilayer perceptron (*SimpleNN*) with a variable number of fully connected layers, ReLU activations, optional dropout, and a ten-class output layer. Each SimpleNN configuration was trained with ADAM [8] optimizer for 20 epochs with evaluation intervals $r \in [1, 20]$. Hyperparameters were tuned over 1 to 3 hidden layers with variable unit sizes, dropout rates and learning rates. For CIFAR-10, we used ResNet-18 [6] from Torchvision [14], training for 50 epochs with $r \in [1, 50]$. In all experiments, the reduction factor was fixed at $\eta = 3$, and the progression parameter was set to *increasing* to reflect that higher accuracy indicates better performance. Experiments were run on NVIDIA TITAN X GPUs with a memory capacity of 14 GB.



(a) Grid Search (90 trials, 20 epochs).

(b) Random Search (10 trials, 20 epochs).



(c) ASHA (10 trials, 20 epochs).

Figure 2 Comparison of search strategies on MNIST with *SimpleNN*. Random and Grid Search train all configurations to completion, whereas ASHA adaptively allocates resources by early-stopping underperforming trials and promoting only the most promising ones. In our experiments, ASHA allocated full training budgets to only two high-performing configurations, achieving accuracies comparable to the exhaustive methods while using substantially less resources. All methods used the same search space configuration.

4.1 Comparison of Hyperparameter Optimization Methods

To evaluate the behaviour of different hyperparameter optimization algorithms within the SEML framework, we conducted experiments using Random Search, Grid Search, and ASHA on the *SimpleNN* model. Random Search and ASHA were each executed for 10 trials, while Grid Search exhaustively explored a discrete configuration space of size $3 \times 6 \times 5$. The

Table 1 Comparison of total compute time, achieved accuracy, and relative speedups of different hyperparameter optimization methods on the *SimpleNN* model using MNIST. Grid Search (GS) explores $3 \times 6 \times 5 = 90$ configurations, while Random Search (RS) and ASHA each ran for $N = 10$ trials. Notably, ASHA avoided unpromising configurations through early stopping, while Grid and Random Search completed all runs regardless of performance.

Algorithm	Acc. [%]	Total Time [s]	Speedup over RS	Speedup over GS
Grid Search	98.1	20361.26	–	1.00×
Random Search	97.5	3296.61	1.00×	6.18×
ASHA	97.0	642.31	5.13×	31.70×

grid comprised three possible hidden layer configurations ranging from [64], [128, 64], and [256, 128, 64], six dropout values uniformly spaced between 0.0 and 0.5; and five learning rates sampled log-uniformly between 10^{-4} and 10^{-2}. For Random Search and ASHA, the same parameter ranges were used, but the exact hyperparameter combinations were determined stochastically rather than by enumerating a fixed set of configurations.

Figure 2 shows that Grid Search and Random Search train all configurations to completion, whereas ASHA focuses resources on promising trials by early-stopping underperforming ones. This selective allocation allows ASHA to reach comparable accuracy with far less compute resources. As summarized in Table 1, ASHA is substantially faster than both alternatives. While Grid Search achieved the highest accuracy (97.5%), both Random Search and ASHA exceeded 96% within ten trials. Overall, ASHA provides an excellent balance between accuracy and efficiency, making it well suited for scenarios with limited computational resources.

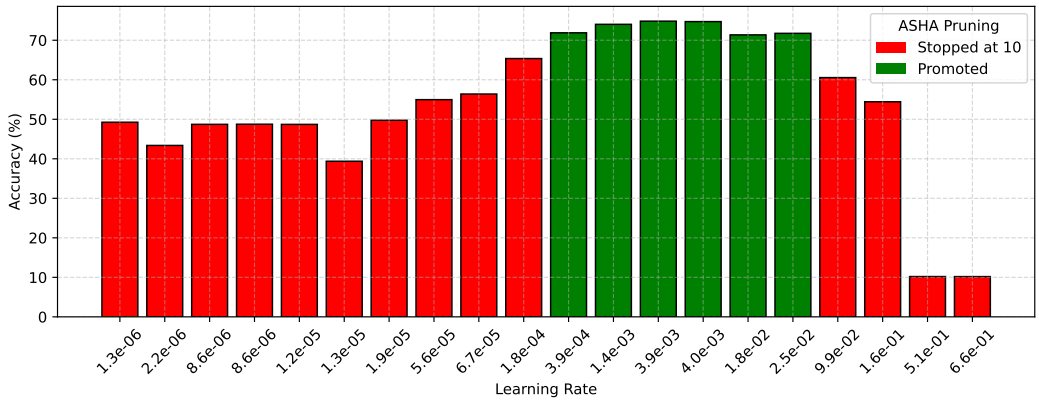


Figure 3 Summary of ResNet-18 training on CIFAR-10 over 20 sampled configurations. The learning rate was drawn log-uniformly from the range 10^{-6} to 1. Early trial termination began after $r_{\min} = 10$. It is observed that the trials with a learning rate set in the range $3.9 \cdot 10^{-4}$ to $2.5 \cdot 10^{-2}$ were promoted after the first elimination round. These results line up with intuition, in that a learning rate should be neither too small nor too large. Further decisions after the first rung are not shown, as they only reinforce this observation.

4.2 Hyperparameter Search over a Wide Learning Rate Range

To examine ASHA's behaviour under an extremely broad hyperparameter range, we conducted a search over learning rates spanning six orders of magnitude, from 10^{-6} to 1. ResNet-18 was trained on CIFAR-10 across 20 configurations sampled from this log-uniform distribution. To

avoid premature pruning given the large range, the minimum promotion threshold was set to $r_{\min} = 10$, ensuring that all configurations completed at least ten epochs before potential cancellation.

As shown in Figure 3, ASHA quickly differentiated between promising and unpromising configurations during training. Runs with learning rates that were either too high or too low were stopped after the first evaluation stage, whereas configurations with moderate learning rates advanced further through the training process. These results line up with intuition, as a learning rate that is too low leads to inefficient training, whereas an overly large learning rate will lead to overfitting. This pattern shows that ASHA can automatically identify an effective region of the learning rate space without exhaustive evaluation and while working asynchronously.

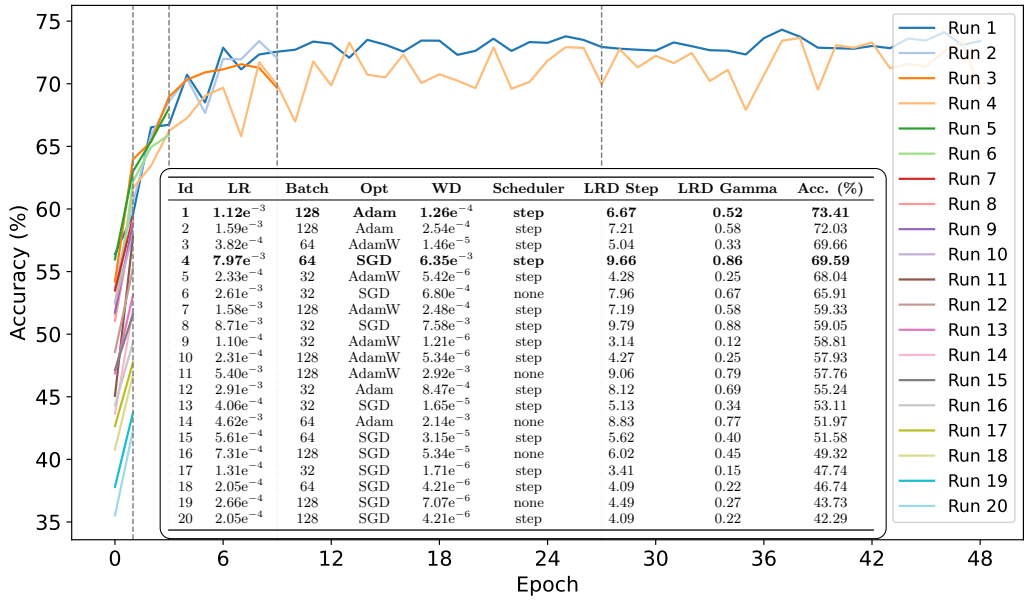
Overall, the experiment demonstrates how ASHA efficiently allocates computational resources by focusing on configurations with strong early performance while eliminating unpromising ones. The trials that progressed beyond the first rung converged consistently to higher accuracies, confirming ASHA's ability to concentrate computation on the most relevant hyperparameter ranges.

Table 2 Hyperparameter search space for ResNet-18 on CIFAR-10 (50 epochs; $r_{\min} = 1$, $r_{\max} = 50$).

Hyperparameter	Search Space	Distribution	Note / Purpose
`learning rate`	$[10^{-4}, 10^{-2}]$	log-uniform	Step size for gradient updates; controls learning stability and convergence speed.
`batch size`	{32, 64, 128}	categorical	Samples per update; larger batches improve stability but increase memory use.
`optimizer`	{SGD, Adam, AdamW}	categorical	Gradient update algorithm; AdamW includes decoupled weight decay.
`weight decay`	$[10^{-6}, 10^{-2}]$	log-uniform	L2 regularization strength to reduce overfitting.
`lr scheduler`	{none, step}	categorical	Learning rate schedule type.
`lr decay step`	[3, 10]	uniform	Interval (in epochs) for step-based LR decay.
`lr decay gamma`	[0.1, 0.9]	uniform	Multiplicative LR reduction factor per decay step.

4.3 Evaluation of ASHA on ResNet-18 with Multiple Hyperparameters

To evaluate ASHA's performance in a higher-dimensional search space, we trained ResNet-18 while jointly tuning seven hyperparameters, which corresponds to a search space dimensionality wholly unrealistic for standard Grid Search. The corresponding parameters and their search spaces are summarized in Table 2. As illustrated by Figure 4, the top-performing configurations did not share a single common hyperparameter setting, underscoring the sensitivity of the model to the joint optimization of multiple parameters. The experiment confirms that ASHA scales effectively to complex search spaces involving many interacting hyperparameters. Through its integration with the SEML framework, ASHA can now be seamlessly deployed on SLURM-managed compute clusters, enabling efficient hyperparameter optimization and helping researchers identify well-performing models with significantly reduced compute cost.



Id	LR	Batch	Opt	WD	Scheduler	LRD Step	LRD Gamma	Acc. (%)
1	**1.12e^{-3}**	**128**	**Adam**	**1.26e^{-4}**	**step**	**6.67**	**0.52**	**73.41**
2	1.59e^{-3}	128	Adam	2.54e^{-4}	step	7.21	0.58	72.03
3	3.82e^{-4}	64	AdamW	1.46e^{-5}	step	5.04	0.33	69.66
4	**7.97e^{-3}**	**64**	**SGD**	**6.35e^{-3}**	**step**	**9.66**	**0.86**	**69.59**
5	2.33e^{-4}	32	AdamW	5.42e^{-6}	step	4.28	0.25	68.04
6	2.61e^{-3}	32	SGD	6.80e^{-4}	none	7.96	0.67	65.91
7	1.58e^{-3}	128	AdamW	2.48e^{-4}	step	7.19	0.58	59.33
8	8.71e^{-3}	32	SGD	7.58e^{-3}	step	9.79	0.88	59.05
9	1.10e^{-4}	32	AdamW	1.21e^{-6}	step	3.14	0.12	58.81
10	2.31e^{-4}	128	AdamW	5.34e^{-6}	step	4.27	0.25	57.93
11	5.40e^{-3}	128	AdamW	2.92e^{-3}	none	9.06	0.79	57.76
12	2.91e^{-3}	32	Adam	8.47e^{-4}	step	8.12	0.69	55.24
13	4.06e^{-4}	32	SGD	1.65e^{-5}	step	5.13	0.34	53.11
14	4.62e^{-3}	64	Adam	2.14e^{-3}	none	8.83	0.77	51.97
15	5.61e^{-4}	64	SGD	3.15e^{-5}	step	5.62	0.40	51.58
16	7.31e^{-4}	128	SGD	5.34e^{-5}	none	6.02	0.45	49.32
17	1.31e^{-4}	32	SGD	1.71e^{-6}	step	3.41	0.15	47.74
18	2.05e^{-4}	64	SGD	4.21e^{-6}	step	4.09	0.22	46.74
19	2.66e^{-4}	128	SGD	7.07e^{-6}	none	4.49	0.27	43.73
20	2.05e^{-4}	128	SGD	4.21e^{-6}	step	4.09	0.22	42.29

Figure 4 Accuracy histories for 20 ResNet-18 trials on CIFAR-10 with an embedded table summarizing hyperparameters and final accuracies.

5 Conclusion

In this paper, we presented a framework for running hyperparameter optimization on SLURM-managed clusters by integrating the ASHA algorithm into the SEML experiment management library. The framework coordinates large numbers of concurrent trials through a shared MongoDB backend and can be deployed on existing SLURM infrastructures without modifying the underlying scheduler.

Our experiments show that ASHA substantially reduces compute time compared to simple baseline methods. On the workloads considered, it achieved up to a 5.13× speed-up over Random Search and a 31.7× speed-up over Grid Search, while reliably identifying well-performing hyperparameter configurations, including in high-dimensional search spaces that are infeasible for naive methods.

By providing an open-source implementation tailored to SLURM-based clusters, we make efficient, asynchronous hyperparameter optimization readily accessible to practitioners. Future work includes extending the framework with additional schedulers and search algorithms and exploring more advanced resource-allocation policies for heterogeneous cluster environments.

The framework is already in regular use on production SLURM clusters at our institution, where it has successfully optimized models for “Uncertainty Reasoning with Photonic Bayesian Machines” [3], which are notably sensitive to hyperparameters, further demonstrating its practical utility.

References

1 Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019. doi:10.1145/3292500.3330701.

2 James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. doi:10.5555/2503308.2188395.

3 F. Brückerhoff-Plückelmann, H. Borras, S. U. Hulyal, L. Meyer, X. Ji, J. Hu, J. Sun, B. Klein, F. Ebert, J. Dijkstra, L. McRae, P. Schmidt, T. J. Kippenberg, H. Fröning, and W. Pernice. Uncertainty reasoning with photonic bayesian machines, 2025. doi:10.48550/arXiv.2512.02217.

4 Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Process. Mag.*, 29(6):141–142, 2012. doi:10.1109/MSP.2012.2211477.

5 Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

6 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.

7 Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 09–11 May 2016. PMLR. URL: https://proceedings.mlr.press/v51/jamieson16.html.

8 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, volume abs/1412.6980, 2015. doi:10.48550/arXiv.1412.6980.

9 Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49–56, 2017. doi:10.25080/shinma-7f4c6e7-008.

10 Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

11 Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, volume 2, pages 230–246. mlsys.org, 2020. URL: https://proceedings.mlsys.org/paper_files/paper/2020/hash/a06f20b349c6cf09a6b171c71b88bbfc-Abstract.html.

12 Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(1):185:1–185:52, 2017. URL: https://jmlr.org/papers/v18/16-558.html.

13 Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training, July 2018. doi:10.48550/arXiv.1807.05118.

14 TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. https://github.com/pytorch/vision, 2016.

15 MongoDB, Inc. *MongoDB Manual*, 2025. Accessed: 2025-07-10. URL: https://www.mongodb.com/docs/.

16 Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, OSDI'18, pages 561–577, USA, 2018. USENIX Association. URL: `https://www.usenix.org/conference/osdi18/presentation/nishihara`.

17 Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. `doi:10.1007/10968987_3`.

18 Daniel Zügner, Johannes Gasteiger, Nicholas Gao, and Dominik Fuchsgruber. SEML: Slurm Experiment Management Library, 2023. URL: `https://github.com/TUM-DAML/seml`.