Parallelizing Hyperparameter Tuning in Artificial Neural Networks

Roy Han[1] and Evan Jiang[1]

[1]Harvard University

June 8, 2023

Abstract

Our project aims to implement Asynchronous Successive Halving (ASH) to accelerate the search for the best hyperparameter configurations in artificial neural networks (ANNs). Hyperparameters are crucial in determining model performance (i.e. maintaining highest predictive accuracy), and they are foundational to training and/or architecture. Since the process of finding the set of hyperparameters for an ANN is lengthy, time-drawn out, and expensive, our group devised an approach that leverages parallelization techniques using openMP. Through a comparison between our parallelized implementation (ASH) to the sequential baseline of Successive Halving (SH), we show that our project yields results that indicate hyperparameter search speedup while maintaining scalability and productionizability.

1 Background and Significance

Artificial neural networks (ANNs) are a type of machine learning model that simulate how the human brain thinks. They are comprised of nodes that are connected to one another, and each connection has a weight that is learned during training. They are capable of precisely approximating functions as long as they have the proper activation functions, and put together, receive an input of data to predict an ouput. Although there are a variety of types of neural networks (e.g. feed-forward, recurring, convolutional), we will focus on feed-forward ANNs because of their popularity. Hyperparameters are crucial to ANNs in determining the model's actual training and architecture. They are settings that cannot be learned, but are preset by the user. Specific to ANNs, examples include number of layers, number of neurons per layer, activation functions, and learning rate. if the hyperparameters are not set well, then the ANN will not be as effective (e.g. overshooting or undershooting optimal weights) which would undermine the ANN to predict and generalize properly. There are a variety of techniques to set the hyperparameters of an ANN. The most foundational methods are grid search and random search as they are easy to implement and understand, but there are also newer techniques like the sequential halving algorithm (SH):

1. Grid search: discretize the hyperparameter search space into a grid, where each point on the grid represents a model's hyperparameter configuration. Evaluate the performance of all the points and select the best hyperparameter set. [2]

2. Random search: discretize the hyperparameter search space into a grid, but instead of evaluating every single point, the user selects a number of models they want to evaluate from the start and selects the best hyperparameter set from the evaluated hyperparameter sets. [2]

3. SH: discretize the hyperparameter search space into a grid, then iteratively apply a process of elimination to identify best set. The algorithm's steps are:

(a) Discretize into grid to get initial set of candidates and define computational budget.
(b) Train and evaluate all candidate hyperparameters using the full computational budget.
(c) Eliminate the worst-performing half of the candidate hyperparameters.
(d) Repeat steps 2-3 until a single best set of hyperparameters remains.

Observe that every round of SH eliminates $1/2$ of the candidates, and those leftover are trained again with double the amount of epochs since having $n/2$ models with 2x computational resources equals the full computational budget. Visually, grid search, random search, and adaptive selection (manual tuning) are represented like so in the following discretized search spaces: [1]

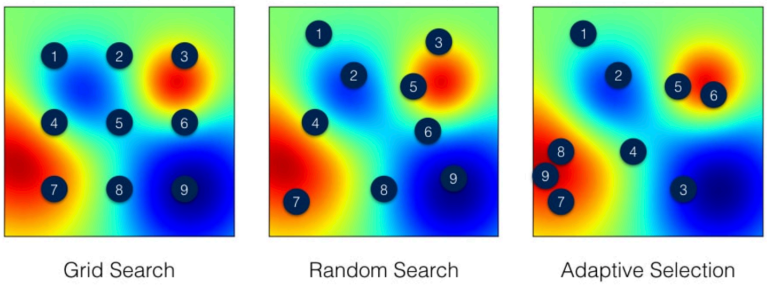

Figure 1: Grid Search, Random Search, and Adaptive Selection [1]

However, optimizing model hyperparameters has become increasingly challenging, as datasets have become more complex and high-dimensional. Therefore, traditional search methods become inapplicable when the hyperparameter search space becomes orders of magnitude larger than before. Our group's objective is to adapt and parallelize sequential halving to get asynchronous sequential halving (ASH), which could leverage new technologies and strategies to accelerate the search for the best hyperparameter configuration. The key difference between ASH and SH is that SH only uses one worker to evaluate all candidate models, whereas ASH leverages multiple workers simultaneously to expedite hyperparameter search. The ASH algorithm is explained and outlined in section 3.

2 Scientific Goals and Objectives

Our overall objective is to use more computational resources to accelerate hyperparameter tuning. As machine learning models become increasingly complex, it has become increasingly important for models to have the optimal hyperparameter configurations. Therefore, our project's completion has impacts that extend beyond CS205, where we can apply our parallelized tuning strategy to more general use cases to achieve the following scientific goals:

1. **Experimentation speed-up:** By reducing the time required for tuning via ASH, users can do more testing, iterating, and deployment with more hyperparameters than using more traditional methods for tuning. Overall, this time saved can accelerate iteration cycles, project turnaround times, and improve research productivity.

2. **Reduce costs:** Since tuning is a very expensive process, the resources needed for tuning can shoot-up in cost – and it only grows exponentially with more complex datasets and models like ANNs that require heavy computation for training. By accelerating hyperparameter tuning, we can lower computational costs since users will have insight to how they should change their workflow, projects, and direction given that they will have results much faster than before.

3. **Improve performance:** As explained earlier, tuning is critical to ensuring optimal model performance. In the context of ANNs, that means having very accurate outputs, whether that be regression or classification, but that applies more broadly to all machine learning models as well.

4. **Improve scalability:** As the machine learning industry continues to grow, so does the demand for the capacity for scaling with respect to training, computing, and deployment. Therefore, it's important to tap into distributed computing environments so that machine learning systems can scale on larger computing infrastructures.

5. **Improve decision-making:** Machine learning models have outputs that can control the direction on what decisions the user decides to act (or not act) upon. To make the best informed decisions, accuracy is crucial, which is why searching for the best hyperparameter configuration is important to the decision-making process.

Therefore, on balance our project aims to save more time and resources than expended, as our project has wider-reaching implications beyond CS205. Therefore, it makes sense to stress-test and improve our parallelization on Harvard's HPC architecture, as our group aims to get net benefits that are more resource-saving in the long-run. Specifically, since our project involves non-trivial computation, it's important for us to leverage the benefits of HPC because:

1. **Large-scale, complex data:** Since our ANN involves processing complex data, it's an intensive training process requiring computational resources not well-suited by a laptop. HPC's high-powered architecture enables us to utilize the computing power of multi-core nodes to simultaneously process data more efficiently.

2. **High computational complexity:** The computational intensity of backpropogation to train all our ANNs, and having HPC architecture to distribute the training for all these models can mitigate the training time needed.

3. **Scalability:** Scalability is crucial, as explained in item 4 above. In the context of our project, we cannot scale our paralellized code without HPC architecture so to meet the demands of our project we need HPC architecture to realize the benefits.

Hence, HPC compute hours are key to our project to get results and to observe the speed-ups we are aiming for.

3 Algorithms and Code Parallelization

3.1 ASH Algorithm

ASH is an extension of SH that's designed to use parallel computing techniques to search for the optimal hyperparameter configuration for any machine learning model, motivated by the slow computation time required for SH's sequential tuning approach. Below is the outline for the logic powering ASH with n total computing resources (in this case, epochs).

1. **Initialization:** initializing all candidate hyperparameter configurations that are to be evaluated on the lowest ladder rung (see Figure 2).

2. **Evaluation:** We take all the workers we have and distribute untrained candidate models to them, evaluating each model with 1 epoch and keeping track of their predictive accuracies (scores).

3. **Advancement:** The best-performing half of the candidates on lowest rung are advanced to the next rung, where they are then re-evaluated with double the epochs. This process repeats itself until we have one candidate model left, which is selected as the optimal hyperparameter configuration.

4. **Asynchronous execution:** Because we have multiple workers, we can train multiple candidate models concurrently across our workers (using `openMP` we give each workload, or untrained candidate model, to individual threads), which accelerates hyperparameter searching. We also have a `maxHeap` for every rung that maintains that rung's evaluated candidate models and their respective scores. For a ladder with a lowest-rung containing k candidate models total, that rung's `maxHeap` will keep track of its evaluated candidate models and their scores, and once $k/2 + 1$ models are evaluated and stored in the `maxHeap`, we know that the highest scoring model in the `maxHeap` is definitely in the top-half of performers of that rung. Hence, we can advance it to the next rung for further evaluation by popping it from `maxHeap` in $O(1)$, which would bring our `maxHeap`'s size back down to $k/2$, in which the same process repeats itself until all the models on that rung have been evaluated. This occurs for every rung's `maxHeap`, and we always prioritize giving workers jobs that are highest up on the ladder, as those strictly dominate the jobs on lower rungs in terms of computation intensity. The `maxHeaps` on every rung is how we maintain dynamic promotions and maximize our efficiency.

Every ladder rung represents a "round" of evaluating candidate models, where the lowest rung has models that are evaluated using $1/n$ epochs, the second lowest rung with models evaluated using $2/n$ epochs, etc. until the top-most rung where the only candidate model remaining is evaluated using n epochs. This way, every ladder rung will maintain the same computational resources used, as every rung only promotes half of the candidates from the rung directly below. This is visualized in Figure 2:

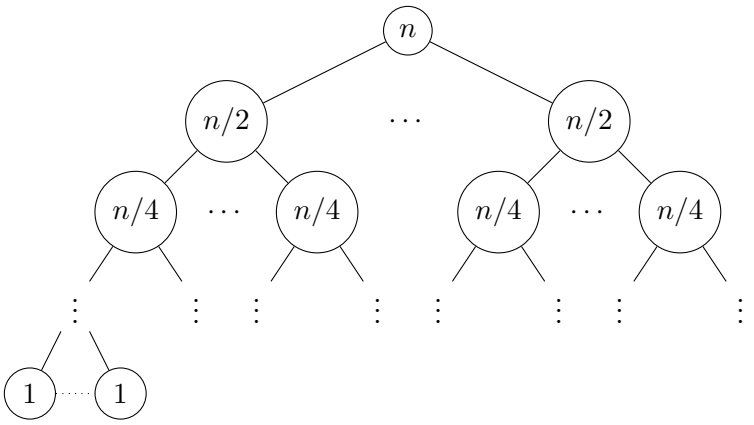

Figure 2: Asynchronous Successive Halving

Above is a ASH tree diagram shows the asynchronous successive halving process, where each level of the tree is a rung and all the nodes in the rung are the candidate models that belong to that rung. Observe how at each rung, half of the remaining configurations are eliminated, until we reach the top where only one configuration is left. The numbers inside of the nodes represent the allocated computational resources given to that candidate, which doubles every rung you move up on the tree (ladder).

3.2 Code Deliverables

We developed the following program files from scratch for our project:

1. `m5_ASH.cpp`: The ASH logic is described in Section 3.1 and we implement it accordingly. This code file implements ASH using openMP. After defining and discretizng the solution space into the vector `candidates`, it defines a `ladder`, which is a vector of max-heaps and serves as the basis of candidate rung climbing. It then spawns a parallel region, in which each thread identifies the highest (on the ladder) model available, trains it, then pushes it to the next rung. The region is terminated once the top rung is reached. Any operations on `candidates` or `ladder` are protected from race conditions with `#pragma omp critical`. As we tested up to 32 threads, we only required one node from the cluster, and used shared memory to enable result management between threads. For our experiments, we evaluate 128, 256, and 512 candidates, each with 4, 8, 16, and 32 threads.

2. `m5_SH.cpp`: The SH logic is described in Section 1, but we implement it with a `maxHeap` to emulate the overhead of managing a max priority queue when collecting our runtimes. We discretize the learning rate alpha to be the hyperparameter we are searching for, and we evaluate 128 and 256 candidates total to determine the optimal α value.

3. `Makefile`: This automates the build for our programs, and compiles and links everything together into executables.

4. `sh_check.sh` & `ash_check.sh`: These are scripts that handle program compilation and jobs with the cluster.

We adapted `neural_network.cpp` (an open-source cpp neural network file [5]) to suit our needs for the ANN model infrastructure, but this is flexible to be replaced by any other model as long as the inputs and outputs are processed in line with our ASH code. We are using open-sourced `test_features.txt`, `test_outputs.txt`, `train_classes.txt`, `train_features.txt`, `train_output.txt`, and `predictions.txt` for the data that is used during training, and we are also using a few standard cpp libraries including `random` and `cstdlib`, `omp.h` for threading.

3.3 Validation and Verification

Validating our ASH implementation is simple – use any SH implementation and compare the results to ASH. Since ASH uses the logic from SH in terms of the structure of the ladder, they differ because ASH is meant for a worksharing environment and uses the `maxHeap` and prioritization of jobs (from Section 3.1). Therefore, their ultimate results should be analogous, which can be confirmed by [3]. Moreover, our results make intuitive sense; we can observe each iteration of training to see the scores of our candidates, and they generally are increasing as they get more resources allocated to them to optimize their parameters (generally because stochastic gradient descent, the optimization method for these models, inherently has an element of randomness), which means that we have confidence that numerically, the logic of our implementation checks out.

4 Performance Benchmarks and Scaling Analysis

Table 1 below displays key metrics of each job type. We kept the memory per core constant to standardize memory access of each thread, and allocated 2 GB per core since each core is responsible

for training with the neural net, which is memory intensive as explored later in Figure 5. Only one node was needed for each job as we used up to 32 threads, allowing us to focus on using shared memory for our parallelization mechanics.

	SH	ASH (4T)	ASH (8T)	ASH (16T)	ASH (32T)
Typical job size (cores)	1	4	8	16	32
Memory per core (GB)	2	2	2	2	2
Memory per node (GB)	2	8	16	32	64

Table 1: Job Sizes for SH and ASH with N threads (T)

Models		SH	ASH (4T)	ASH (8T)	ASH (16T)	ASH (32T)
128	Walltime (min:sec)	4:25.73	3:40.23	2:46.56	1:37.06	1:27.91
	Observed Speedup	1x	1.21x	1.60x	2.74x	3.02x
256	Walltime (min:sec)	9:30.85	7:30.43	5:09.47	2:58.89	2:39.95
	Observed Speedup	1x	1.27x	1.84x	3.19x	3.57x
512	Walltime (min:sec)	-	16:41.96	11:16.47	8:07.31	5:46.35

Table 2: Runtimes and Speedups for SH and ASH with N threads (T), by number of models

After conversing with CS205 teaching staff, our group agreed that roofline analysis was not applicable to our project, as it would not contribute any insight to our analysis. The biggest bottleneck inevitably is the training for all candidate models, which given ANNs complex architecture, is even more augmented. Since we are not parallelizing the training but instead the question of when and where each model gets trained, we decided against including roofline analysis here. Moreover, our project did not include I/O related implementation, so for analysis we opted to focus what is relevant to our project: (1) strong scaling analysis for scaling and (2) resource utilization metrics for performance on top of the required statistical measurements.

We collected data on SH for 128 and 256 candidate models and ASH for 128, 256, and 512 candidate models in min:sec format in Table 5 through Table 8 (see Appendix).

We plot our data to observe the relationship between the number of threads used and walltime below.

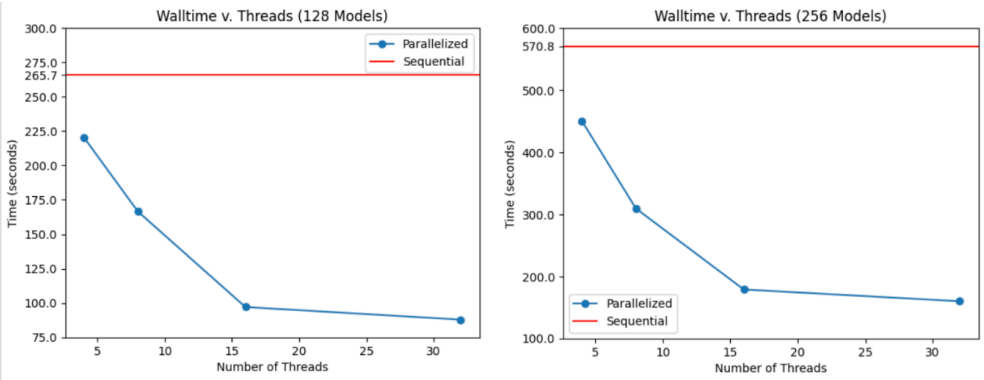

Figure 3: Walltime vs Threads for 128, 256, and 512 Models

We then performed strong scaling analyses based on the number of models using our data, plotting ideal speedup ($y = x$) as well as the actual speedup, where the actual speedup for N threads is calculated as such: Speed-up(N) = Walltime(1) / Walltime(N)

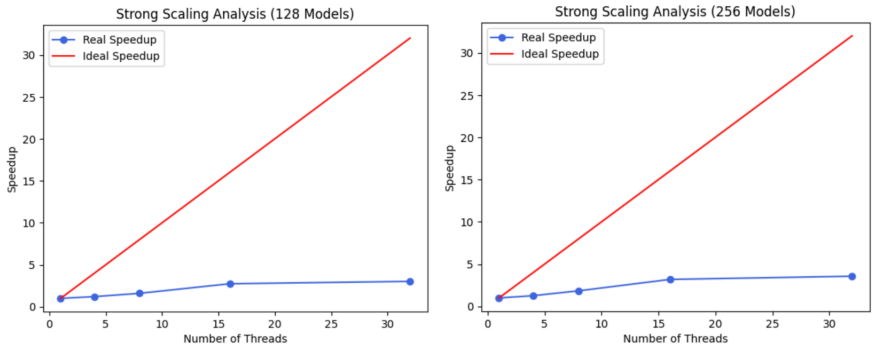

Figure 4: Strong Scaling Visualizations for 128 and 256 Models

Although the strong scaling analysis suggests, on first glance, very poor parallelization performance, it is not actually indicative of the quality of the ASH implementation due to two main factors:

1. Worksharing with OpenMP creates significant overhead associated with the parallel region. Specifically, we observed that the walltime of training a model for some number of epochs is noticeably higher for ASH than SH. This trend can be clearly seen by comparing a row in Table 5 for 128 or 256 models to the corresponding row in Table 6 or Table 7 of the Appendix. For example, when the solution space is 128 models, training a model with one epoch takes 0.35s for SH, which is much less than the 1.08s, 1.31s, 1.35, and 1.64s it takes for 4, 8, 16, and 32 threads respectively for ASH. This trend can be attributed to the additional stress placed upon compute resources by our thread pool and associated shared memory management.

2. The ideal speedup suggested in Figure 4 is not true even in theory, and is a blatant oversimplification of ASH. Recall that in ASH, we saturate our workers by assigning the most intensive workload to the next available worker. Under these circumstances in which the workers are completely saturated, the ideal speedup is representative, since the vast majority of the sequential workload is being parallelized (machine learning model training). Due to factor 1 outlined above, we would already fall short of the ideal speedup. However, as we near the top of the ladder, there are eventually less jobs available than workers. Additionally, the jobs at the top of the ladder require many epochs, since we double epochs as we move up rungs, which means that there is significant idle time by threads once ASH approaches the top of the ladder. We can see how the walltime increases with the number of epochs by traversing down any of the tables from Table 6 through Table 8. The effect of this on parallelization speedup can be seen by the way in which the walltime tapers off in Figure 3 for 128 and 256 models compared to 512 models, since the former two have less rungs and therefore start closer to the top. It is very difficult to mathematically model the true ideal speedup of ASH, but we note that it would take some form of Amdahl's law similar (but much more complex) to that of Problem 2c in Homework 4. Near the top rungs of the ladder, the workload will be bounded in terms of the number of threads that can be used to

parallelize it. We note that Li [1] was only able to achieve 10x speedup with 25 workers on over a thousand models (that were more intensive than ours), confirming our analysis.

We also analyzed memory usage and bottlenecks in memory using Valgrind with small trials for SH. The plot below, which was generated from the data in Table 4, examines the relationship between bytes and total epochs. Note that the memory allocation comes from the training of the neural nets, not the innate algorithm, and so the following table is representative of memory allocation in ASH as well.

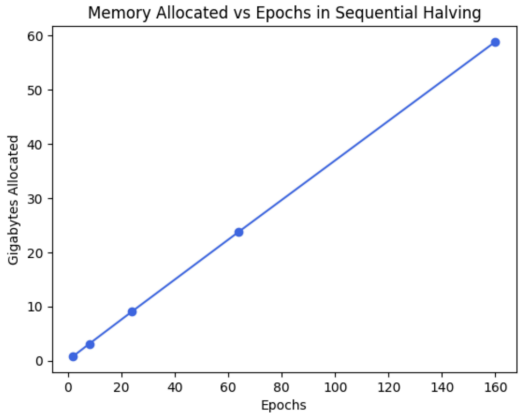

Figure 5: SH Bytes/Epochs Visualization

We see based on the visualization that there is a linear relationship between bytes allocated and number of epochs, and so the number of memory allocated per epoch stays constant. Running SH or ASH with a larger number of models, such as 1024 or 2048, would result in more epochs, requiring larger amounts of memory resources.

5 Resource Justification

We justify our project resource utilization by examining the aggregated core hours for our simulations, which were all run on a 32 core single compute node on the cluster (Harvard's Broadwell Architecture) with varying threadcounts depending on the simulation. Note that we did not use all 32 cores for every job; instead, as briefed in Table 1, we ran on the number of cores equal to the number of threads.

In this sense, given that all of our threads are running on 1 node and we use only OpenMP, it is more suitable to report the number of core hours rather than the number of node hours. Since the number of core hours consumed by a simulation is the number of cores used multiplied by the wall time (in hours) for a typical production run, we tally core hour runtimes across all trials using the formula:

$$\text{Core hours} = \text{ number of cores} \times \frac{\text{time in s}}{3600 \frac{s}{\text{hour}}}$$

	SH (128)	SH (256)	ASH (128)	ASH (256)	ASH (512)
Core hours (sequential)	0.07381	0.1586	-	-	-
Core hours (4 threads)	-	-	0.2447	0.5005	1.1133
Core hours (8 threads)	-	-	0.3701	0.6877	1.5032
Core hours (16 threads)	-	-	0.4313	1.1678	2.1657
Core hours (32 threads)	-	-	0.7815	1.4219	3.0785

Table 3: Core Hour Data for SH and ASH Trials

Pulling from our project data (see Appendix) and using the formula above, we can calculate the core hours for each of our thread implementations. Aggregating, our group requests a total of $\boxed{13.6985}$ core hours to cover all simulations, which is the total resource request of our project. This is a very reasonable amount given the multithreaded focus of our project and the need to test on multiple values of models to demonstrate the behavior of ASH.

References

[1] Liam Li. *Massively Parallel Hyperparameter Optimization*, 12 Dec. 2018. https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/

[2] Sayak Paul. *Hyperparameter Optimization in Machine Learning Models*, Aug. 2018. https://www.datacamp.com/tutorial/parameter-optimization-machine-learning-models

[3] Robin Schmucker. *Multi-objective Asynchronous Successive Halving* https://arxiv.org/pdf/2106.12639.pdf

[4] Leslie Lamport. *LaTeX: a document preparation system*. Addison-Wesley, Reading, Massachusetts, 1993.

[5] Raj Daksh. *Configurable Feed-Forward Backpropagation Neural Network written in C++*, 4 Nov. 2020. https://github.com/raj1003daksh/cpp-neural-network

6 Appendix

Models	Training	Epochs	Bytes allocated	Bytes/Epoch
2	2	2	792,790,098	396395049
4	6	8	3,078,079,270	384759908.8
8	14	24	9,048,497,806	377020741.9
16	30	64	23,789,004,701	371703198.5
32	62	160	58,869,389,053	367933681.6

Table 4: Memory Analysis for SH

SH: 128 Models		
Epochs	Models	Avg Local Runtime (m:s)
1	128	0:0.350994867187
2	64	0:0.620960125
4	32	0:1.1973700625
8	16	0:2.2616663125
16	8	0:4.453271375
32	4	0:8.91331325
64	2	0:17.6253085
Total		**4:25.727091**

SH: 256 Models		
Epochs	Models	Avg Local Runtime (m:s)
1	256	0:0.3498893125
2	128	0:0.601203703125
4	64	0:1.141992953125
8	32	0:2.19912890625
16	16	0:4.247624125
32	8	0:8.045298125
64	4	0:16.02331675
128	2	0:32.199592
Total		**9:30.847081**

Table 5: SH Runtime Data for 128 and 256 Candidate Models

Avg Local Runtimes (m:s) of ASH for 128 Models and 4, 8, 16, and 32 Threads					
Epochs	Models	4 Threads	8 Threads	16 Threads	32 Threads
1	128	0:1.081571367	0:1.3070152265625	0:1.350048875	0:1.63740825
2	64	0:2.073625359	0:2.50861965625	0:2.599252063	0:3.312751641
4	32	0:4.077447563	0:4.930083875	0:5.072138688	0:5.846190969
8	16	0:8.581533375	0:11.0901794375	0:5.700810375	0:8.133771063
16	8	0:16.39209025	0:13.80062	0:13.98892325	0:12.91279775
32	4	0:18.34304325	0:23.75596875	0:15.82415575	0:13.68504575
64	2	0:22.037281	0:26.4307675	0:21.7634555	0:24.308323
Total Runtime		**3:40.257241**	**2:46.56195**	**1:37.056908**	**1:27.908432**

Table 6: ASH Runtime Data for 128 Candidate Models

Avg Local Runtimes (m:s) of ASH for 256 Models and 4, 8, 16, and 32 Threads					
Epochs	Models	4 Threads	8 Threads	16 Threads	32 Threads
1	256	0:0.8553349492	0:1.027869531	0:01.324147973	0:1.572833293
2	128	0:1.60707893	0:1.93852718	0:02.452543219	0:2.947107859
4	64	0:3.13666675	0:3.790191781	0:04.662824672	0:6.158493813
8	32	0:6.179006313	0:7.501214844	0:10.03072978	0:8.074182688
16	16	0:12.61627381	0:14.90109706	0:10.6543485	0:13.67862275
32	8	0:25.35025163	0:29.5360245	0:10.97461313	0:14.41168088
64	4	0:42.62104525	0:47.43314725	0:21.8536185	0:24.32628825
128	2	1:7.769429	0:52.436606	0:43.805928	0:48.697185
Total Runtime		**7:30.429362**	**5:9.472301**	**2:58.89496**	**2:39.953534**

Table 7: ASH Runtime Data for 256 Candidate Models

Avg Local Runtimes (m:s) of ASH for 512 Models and 4, 8, 16, and 32 Threads					
Epochs	Models	4 Threads	8 Threads	16 Threads	32 Threads
1	512	0:0.8569402754	0:0.9527414688	0:1.198049604	0:1.453168949
2	256	0:1.614162367	0:1.800946539	0:2.250358324	0:2.738476664
4	128	0:3.180323609	0:3.5158835	0:4.408548781	0:5.57474207
8	64	0:6.255857125	0:6.947583953	0:9.146660578	0:11.93978072
16	32	0:12.29536313	0:13.81107231	0:18.95382256	0:16.11568722
32	16	0:25.27993525	0:27.55315	0:19.57586644	0:34.76844644
64	8	0:49.90705213	0:54.87334488	1:1.69183138	0:26.4449895
128	4	1:27.4670788	1:30.428955	1:17.30019625	0:48.81741325
256	2	2:7.83678399999999	2:17.261799	1:27.71943	1:37.9050175
Total Runtime		**16:41.961042**	**11:16.469526**	**8:7.310522**	**5:46.348529**

Table 8: ASH Runtime Data for 512 Candidate Models