# Final Report: Metric Benchmarking with Limited Resources

**Markelle Kelly**
kmarke@uci.edu

## 1 Introduction

The UCI Machine Learning Repository[1] is a widely used public testbed for machine learning research with an estimated two million data downloads annually. Currently, the repository is undergoing an NSF-funded reinvention, focusing on dataset leaderboards. These leaderboards provide users with benchmarked metric values, enabling them to validate their own results. To accomplish this, I have developed a pipeline that, for a given dataset, automatically trains baseline scikit-learn models and evaluates them, computing a given metric and generating bootstrapped confidence intervals. This inclusion of uncertainty allows students and researchers to place empirical results in context, an important methodological and conceptual step beyond just tabulating algorithmic "winners" and "losers."

The simple scikit-learn models are trained as-is, without any customization or hyperparameter tuning, limiting the applicability of results. Results from well-constructed, optimized neural networks would be more useful to researchers developing new state-of-the-art machine learning techniques. However, network training and hyperparameter tuning need to happen quickly; once a new dataset is added, its benchmarks should be available soon after, without demanding too much computational power on the server side. For this project, I extend the repository's benchmarking to perform hyperparameter tuning given these constraints. To operate with limited available resources, I employ the successive halving algorithm (Jamieson and Talwalkar, 2015), which allocates more resources to promising hyperparameter settings. Based on the limitations of the repository server, I allocate a budget, which is uniformly allocated to $n$ hyperparameter configuration "arms" for a set number of iterations. Then, the worst half of the arms are thrown out, and this cycle is repeated until a single best arm remains. I also explore the HyperBand algorithm (Li et al., 2016), which effectively performs a grid search over the number of arms used. In this project, I evaluate both techniques on models for a selection of repository datasets.

[1] https://archive-beta.ics.uci.edu

## 2 Related Work

A variety of methods exist for adaptive configuration evaluation in constrained settings. For example, Bayesian optimization techniques (e.g., Hutter et al. (2011), Swersky et al. (2013), Klein et al. (2016)) are popular, but involve strong distributional assumptions regarding the performance of a model. In recent years, halving-style algorithms, which avoid these strong assumptions, have gained traction (Sparks et al., 2015; Jamieson and Talwalkar, 2015). These approaches treat the problem of hyperparameter search as a multi-armed bandit problem, reviewed in depth by Jamieson and Nowak (2014). As an extension, Li et al. (2016) introduce HyperBand, which builds on these halving-style algorithms by optimizing over possible numbers of configurations to explore.

While the problem of constrained hyperparameter search has been well explored, including within the realm of machine learning algorithms, my work applies these techniques in a novel domain. Rather than optimizing a single model on a single dataset, the benchmarking I perform must be flexible to a variety of models and datasets, with no human interference. That is, the hyperparameter search must cover reasonable values for a variety of models, for a variety of dataset sizes and contents. Further, constraints are based on a constantly-changing dataset repository. If, for example, it is determined that all benchmarks should be available within an hour of a dataset's upload, for a limited amount of server

resources, resources may need to be divided between multiple datasets if they are all uploaded in the same hour. The new design employs both successive halving and HyperBand to solve this problem.

## 3 Design

This section outlines the design and implementation of my approach. A high-level diagram can be found in Figure 1. In short, a driver program receives new datasets and controls the hyperparameter configuration space to explore. For each dataset, it creates an Evaluator, which computes an appropriate budget and creates TensorFlow models (Classification or Regression networks). These models are built according to the dataset information and hyperparameter space; the Evaluator allocates resources to each based on successive halving or HyperBand. The Evaluator determines the optimal model, benchmarks its performance, and provides this information to the driver, which can post it to the website database.

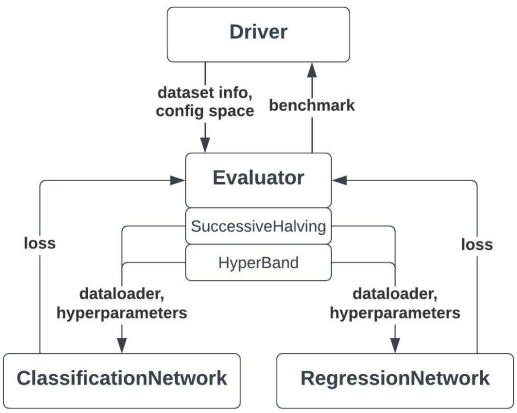


Figure 1: A diagram of the relationships between the driver, Evaluator, and individual networks.

### 3.1 Driver

The driver program, given a dataset, controls the overall model search. It specifies the hyperparameter configuration space, describing a distribution over model structures (number and sizes of hidden layers), optimization algorithms (including stochastic gradient descent, Adagrad, and AdamW optimizers), and learning rates. As new datasets are uploaded to the repository, the driver receives each dataset and its relevant metadata, reflecting the syntax with which it is stored (e.g. delimiter, locations of many separate files), the target variable and its type, and the types of other variables. The program creates an Evaluator instance for the dataset, passing in its metadata, the hyperparameter space, and which algorithm (successive halving or HyperBand) to use. The driver then waits to receive the final benchmarked performance from the Evaluator so it can be added to the dataset's webpage.

### 3.2 Evaluator

Given the dataset information, the Evaluator creates TensorFlow models to be tested. This involves determining whether the problem is classification or regression, the target variable, the number of input variables, and the number of classes (if applicable). For successive halving, this is a fixed $n$ models, one for each hyperparameter configuration specified by the driver. For HyperBand, this involves repeatedly sampling $n$ configurations from a defined configuration distribution, for each value of $s$. (The specific details on these algorithms are outlined in sections 3.3 and 3.4 below.) The Evaluator also performs generic data preprocessing and creates a custom TensorFlow DataLoader to pass to the models.

The Evaluator then computes an appropriate budget in terms of the total number of epochs. To do this, it runs a single sample epoch on the most complex model (estimated by the number of hidden layers * the maximum hidden layer size). This only needs to be an estimate, so other configuration options (i.e., the optimizer and learning rate) are chosen randomly. Based on the runtime of this single epoch, the Evaluator computes the total budget as $\frac{1800}{RT}$, that is, the number of epochs of this length that can run in half an hour. In this way, I define a budget that implicitly takes into account the worst-case model complexity and dataset size, ensuring that results are available quickly (in general, often completing in well under 30 minutes). However, for very simple or small datasets, even this allocation may be excessive. For example, on the four-column, 150-row Iris dataset, this returns a budget of over 1,000 epochs, and a runtime of about 16 minutes, although an optimal model can be trained in just seconds. Thus, we also use early stopping for individual models, quitting if the loss has not improved for $i$ iterations. Note that available resources are not taken into account in this budget; a further discussion of this will follow.

According to the computed budget and the algorithm specified by the driver, the Evaluator then

allocates resources to the various models created, eventually choosing the optimal model and computing its test performance. This test performance serves as the benchmark for the given dataset.

### 3.3 Successive Halving

Based on the given budget $B$, in each iteration, each remaining model is run for $\lfloor \frac{B}{|S_k|\lceil log_2 n \rceil} \rfloor$ epochs, where $S_k$ is the number of remaining models and $n$ is the starting number of models. After each iteration, the half of the models with the highest validation losses are dropped, and this is repeated until a single model remains. The accuracy or MSE of this model on the test set is then used as the benchmark value.

### 3.4 HyperBand

HyperBand effectively uses successive halving as its inner loop, and iterates over possible values of $n$, the number of arms explored for a given budget. To do this, it takes as input a meta-budget $R$ and a parameter $\eta$ which controls the proportion of configurations discarded per iteration. (For the rest of this paper, "budget" refers to this value $R$.) For all experiments, I use $\eta = 3$ as recommended by Li et al. (2016). Thus, $\lfloor log_3 R \rfloor + 1$ (generally around five) iterations are performed, each considering a smaller value of $n$. Starting with $s_{max} = \lfloor log_\eta R \rfloor$ and decrementing $s$ by one each iteration, we compute the value of $n$ to test as $\lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$, and the iteration budget $r = R\eta^{-s}$. Using these values, we then perform standard successive halving on a randomly selected set of $n$ models from the specified configuration space. Again, we repeatedly remove the models with the highest losses, eventually settling on a single best model (across all values of $s$), which we use to compute our benchmark performance.

### 3.5 Models

The Evaluator, depending on the task at hand, creates a custom TensorFlow model, either a ClassificationNetwork or a RegressionNetwork. These models take as input the custom DataLoader and a single hyperparameter configuration. They build a network structure according to the specified data shape, number and sizes of hidden layers, optimizer type, and learning rate. Both network types use ReLU activation functions. The ClassificationNetwork also adds dropout layers with probability 0.3 after each specified layer.

The ClassificationNetwork and RegressionNetwork use cross-entropy loss and MSE loss, respectively, and include functionality to obtain validation and test performance. If a train/test split was specified by the data donor, it is used here, otherwise the test set is a randomly chosen 25% of the full dataset. Of the training set, 10% is randomly chosen as a validation set. The benchmarked performance metric is test-set accuracy or MSE.

## 4 Results

To evaluate my solution, I generate benchmarked accuracy and MSE for the set of the top 50 most-downloaded repository datasets. These datasets vary from very small (Iris has 4 columns and 150 rows) to very large (ISOLET has 618 columns and 6,238 rows), and include binary classification tasks, multi-classification tasks, and regression tasks. A batch size of 16 is used for each model and dataset. Experiments are run on the repository server, which includes 7 GPUs with a total of 203 gigibytes of RAM.

The hyperparameter space to explore includes network architectures with one to three hidden layers with sizes ranging from 50 to 250 nodes. I test AdamW, Adagrad, and standard stochastic gradient descent optimizers, with learning rates between 0.0001 and 0.005. The TensorFlow models are run with CUDA on the GPUs.

For each dataset, the goal is to improve performance (increase accuracy or decrease MSE) over the existing benchmarks. Note that naturally, the runtimes listed are slower than training a single, default scikit-learn model, but I am not interested in improving upon that speed. Rather, the goal is to improve the performance of the benchmark models, while still training in a reasonable amount of time (in particular, in less than 30 minutes).

### 4.1 Successive Halving vs. HyperBand

To determine which algorithm to use for the full at-scale evaluation, I test both successive halving and HyperBand on the first 10 datasets. All results from this experiment can be found in Table 3 in the Appendix. Compared to successive halving, the models chosen by HyperBand are less accurate by an average of 4.9 percentage points, and the algorithm runs for an average of 7.3 more minutes. One possible reason for this discrepancy in performance is that HyperBand divides the given budget into several iterations, where different num-

bers of configuration options are considered. Thus, a smaller amount of computational resources can be dedicated to each model when using HyperBand. It makes sense that HyperBand runs for longer periods of time, since it is starting to train many more models, and thus less likely to be shortened by early stopping. Another possible reason for this difference is that HyperBand randomly chooses configurations from the hyperparameter space, while successive halving takes in a certain number of thoughtfully chosen, predefined arms. This means that the configurations investigated by HyperBand may cover a smaller, random region of the hyperparameter space. Therefore, HyperBand may "miss" an ideal model setup for a given dataset.

To better understand the impact of the budget on the performance of these two algorithms, I explore a variety of budget settings on the Glass Identification dataset. Here, the goal of the model is to predict the type of glass out of seven categories based on ten chemical attributes. In this experiment, I do not use the automatic budget allocation described earlier; I specify an exact budget and observe the resulting accuracy. The results of this experiment are plotted in Figure 2.

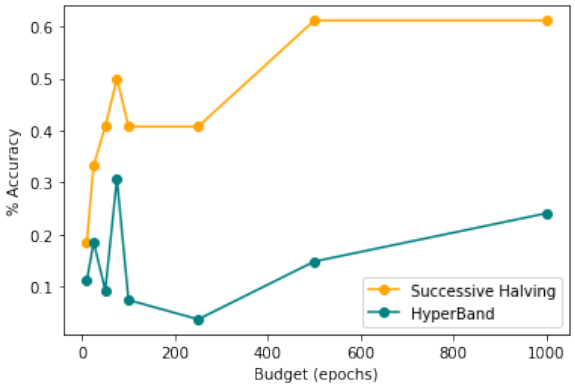


Figure 2: Computational budget vs. accuracy for the Glass Identification dataset.

For small computational budgets, HyperBand has variable success; it is dependent on which random configurations are chosen in that run. As the budget increases, HyperBand's performance begins to stabilize and steadily improve, but successive halving is able to focus the same budget on fewer models, achieving better performance faster.

Based on this analysis, I use successive halving for my approach. While HyperBand may "cover more ground," for the relatively small budgets I am using, the drawbacks of this outweigh the benefits. It is possible that, for much larger budgets, the automatic, exploratory nature of HyperBand would be a good fit. However, successive halving appears to work well with the quick-and-dirty nature of this problem.

### 4.2 Full analysis

Using the top 50 most-downloaded repository datasets, I train models with successive halving and compare the results to those from my original pipeline. Over all classification tasks, the new accuracies are an average of 3.8 percentage points higher. Across all datasets, 83% obtain better or equal benchmarks, and the average runtime is 337 seconds, or just under six minutes. This means that although each Evaluator is allotted 30 minutes of runtime, most models stop early, using only a few minutes. None of the runtimes are longer than half an hour; the longest runtime is around 18.7 minutes. The full set of benchmarks before and after, along with runtimes, can be found in Tables 4 and 5 in the Appendix.

Table 1 contains the new accuracy and the increase in accuracy (in percentage points) of the top four most improved datasets. The largest accuracy improvement is for the Glass Identification dataset, which has an scikit-learn benchmark of 22.2% accuracy, and has a new benchmark of 57.4% accuracy. One possible reason successive halving is useful for these datasets is their relative complexity. Glass Identification, Vehicle Silhouettes, and Lung Cancer are all multi-class classification problems. MONK's Problems is a difficult artificial classification task used to distinguish between learning algorithms at an international competition. Thus, these tasks might require a complex, custom model to be completed with high accuracy.

| Dataset | Accuracy | Increase |
|---|---|---|
| Glass Identification | 57.4 | 35.2 |
| MONK's Problems | 97.4 | 22.9 |
| Vehicle Silhouettes | 75.9 | 16.9 |
| Lung Cancer | 37.5 | 12.5 |

Table 1: The (new) accuracy and increase in percentage points of the most improved benchmarks.

However, not all benchmarks were improved by the successive halving approach; Table 2 shows the results for the four datasets with the worst drops in accuracy. Compared to the most-improved datasets, all four of these datasets are relatively simple. Each

| Dataset | Accuracy | Decrease |
|---|---|---|
| Hepatitis | 69.2 | 7.7 |
| Wine | 93.3 | 4.5 |
| Connectionist Bench | 78.8 | 4.2 |
| Congress. Voting | 94.5 | 1.8 |

Table 2: The (new) accuracy and decrease in percentage points of the least improved benchmarks.

one is a binary classification task, with relatively few rows and columns. It is possible that hyperparameter tuning with neural networks is simply overkill for these smaller datasets, and overfitting is contributing to a drop in performance. It is worth noting, however, that the magnitude of these drops in accuracy is much smaller than the magnitude of increases in accuracy we see elsewhere.

## 5 Discussion and Future Work

One remaining concern with this approach is the issue of many datasets being donated at once. Empirically, several Evaluators can run in parallel on the server without dominating computational power or memory. In addition, over the last two years, the maximum number of datasets donated within an hour is three, so with our average training time of under 20 minutes, benchmarks should consistently be available within an hour. However, to deploy this system on the real server, it will be necessary to better handle very extreme cases (e.g., 100 dataset uploads within an hour). One simple solution is to set a maximum number of Evaluators that can run at once, allowing some lateness. If this lateness becomes a problem, a decrease in budget, stricter early stopping, or alternative allocation strategy could be used. As an example, if too many datasets are uploaded at once, they could receive a budget of only a few minutes, and be added to a queue; the server could then re-benchmark these datasets with the standard resource allocation during idle time. Overall, considering the currently available server resources remains an important direction for future exploration.

While in general, the successive halving approach improved benchmarks, hyperparameter-tuned deep neural networks may not be appropriate for small, simple datasets. As discussed in the previous section, for several such datasets this approach significantly decreased benchmark accuracy. Simple scikit-learn models can be trained in negligible runtime, so a hybrid approach might offer high-quality benchmarks for a bigger variety of datasets — if a simple logistic regression or random forest achieves higher accuracy than the tuned successive halving model, that should be reported as well.

One important future expansion to this approach is support for non-tabular datasets (such as text or image data). In addition to the standard models explored in this paper, the Evaluator could also fine-tune a BERT model or add convolutional layers to a network.

Further, while a reasonable hyperparameter space was chosen empirically, choosing this more intelligently, perhaps based on certain dataset qualities such as dimensionality, may lead to better results. While a variety of hyperparameter settings were explored, many more, including activation functions, momentum, batch size, more complex network layers, and weight decay, were outside the scope of this project. A meta-analysis that determines the most important parameters to tune, and how best to keep these flexible for a variety of datasets, would be an interesting future project.

Finally, it is important to note that accuracy and MSE are not perfect performance metrics. A more thorough analysis might incorporate multiple metrics (e.g., precision, recall, $R^2$).

## 6 Conclusion

In this paper, I design and evaluate a system for automatically benchmarking machine learning models on a variety of datasets. To operate in a reasonable amount of time, without dominating the website server, I test successive halving and HyperBand algorithms for hyperparameter tuning. I find that, for this problem, successive halving achieves the best performance. On the top 50 repository datasets, compared to previous benchmarks, the system improves average accuracy subject to the constrained runtime.

## References

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kevin Jamieson and Robert Nowak. 2014. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *2014 48th Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6.

Kevin G. Jamieson and Ameet Talwalkar. 2015. Non-stochastic best arm identification and hyperparameter optimization. *CoRR*, abs/1502.07943.

Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2016. Fast bayesian optimization of machine learning hyperparameters on large datasets. *CoRR*, abs/1605.07079.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560.

Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 368–380, New York, NY, USA. Association for Computing Machinery.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. 2013. Multi-task bayesian optimization. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc.

# 7 Appendix

## 7.1 Successive Halving vs. HyperBand

| Dataset | Successive Halving | | HyperBand | |
|---|---|---|---|---|
| Iris | 97.4 | **14.9** | 92.1 | **71.09** |
| Adult | 80.0 | **1156.9** | 76.8 | **3559.1** |
| Heart Disease | 78.4 | **53.3** | 74.5 | **123.3** |
| Wine | 93.3 | **80.4** | 86.7 | **216.0** |
| Car Evaluation | 99.5 | **582.2** | 91.3 | **1620.0** |
| Breast Cancer WI (D) | 96.0 | **178.3** | 90.9 | **438.2** |
| Breast Cancer | 77.8 | **55.8** | 71.7 | **190.8** |
| Glass Identification | 57.4 | **72.0** | 49.1 | **218.8** |
| Mushroom | 100.0 | **853.9** | 100.0 | **1009.4** |
| Breast Cancer WI (O) | 97.7 | **173.2** | 95.4 | **164.0** |

Table 3: Successive Halving vs. HyperBand accuracy and **runtime** on top 10 datasets

## 7.2 Full Results

Note that fewer than 50 datasets (in particular, 41) appear here. Of the top 50 datasets investigated, nine were either non-tabular (e.g. image) or unsupervised problems, which were beyond the scope of this project (as well as the scope of the original benchmarking).

| Dataset | MSE: Before | After | Decrease | Runtime (s) |
|---|---|---|---|---|
| Abalone | 4.31 | 3.65 | 0.66 | 209.2 |
| Liver Disorders | 10.12 | 8.09 | 2.03 | 51.9 |
| Auto MPG | 378.7 | 329.0 | 49.7 | 25.3 |
| Automobile | 317427 | 51409 | 266018 | 1069.6 |

Table 4: Successive Halving Performance on Regression Datasets

| Dataset | Accuracy: Before | After | Increase | Runtime (s) |
|---|---|---|---|---|
| Iris | 97.4 | 97.4 | 0 | 14.9 |
| Adult | 78.4 | 80.0 | 1.6 | 1156.9 |
| Heart Disease | 78.9 | 78.4 | -0.5 | 53.3 |
| Wine | 97.8 | 93.3 | -4.5 | 80.4 |
| Car Evaluation | 98.4 | 99.5 | 1.1 | 582.2 |
| Breast Cancer WI (D) | 92.3 | 96.0 | 3.7 | 178.3 |
| Breast Cancer | 66.6 | 77.8 | 11.2 | 55.8 |
| Glass Identification | 22.2 | 57.4 | 35.2 | 72.0 |
| Mushroom | 100.0 | 100.0 | 0 | 853.9 |
| Breast Cancer WI (O) | 96.6 | 97.7 | 1.1 | 173.2 |
| Statlog German Credit | 64.4 | 75.6 | 11.2 | 234.0 |
| Optical Handwritten Digits | 96.3 | 96.9 | 0.6 | 731.8 |
| Thyroid Disease | 94.4 | 98.1 | 3.7 | 120.2 |
| Pen-based Handwritten Digits | 96.9 | 97.5 | 0.6 | 1000.2 |
| Ionosphere | 87.5 | 95.5 | 8.0 | 112.6 |
| Congressional Voting Records | 96.3 | 94.5 | -1.8 | 214.7 |
| Zoo | 92.3 | 100.0 | 7.7 | 71.5 |
| Letter Recognition | 91.9 | 92.0 | 0.1 | 1223.2 |
| Yeast | 56.9 | 58.0 | 1.1 | 106.8 |
| Lung Cancer | 25.0 | 37.5 | 12.5 | 9.0 |
| Hepatitis | 76.9 | 69.2 | -7.7 | 54.7 |
| Spambase | 91.5 | 94.0 | 2.5 | 288.4 |
| Internet Advertisements | 97.2 | 97.8 | 0.6 | 212.6 |
| Soybean (small) | 100.0 | 100.0 | 0 | 30.5 |
| Connectionist Bench | 83.0 | 78.8 | -4.2 | 59.5 |
| Tic-Tac-Toe Endgame | 99.6 | 99.2 | -0.4 | 447.9 |
| Soybean (large) | 93.9 | 94.1 | 0.2 | 234.2 |
| Nursery | 100.0 | 99.9 | -0.1 | 1102.6 |
| Statlog Landsat Satellite | 75.8 | 82.8 | 7.0 | 729.8 |
| Statlog Australian Credit | 62.4 | 65.5 | 3.1 | 513.9 |
| Statlog Shuttle | 90.6 | 91.9 | 1.3 | 82.1 |
| Waveform Database Generator | 84.0 | 87.0 | 3.0 | 336.6 |
| Statlog Vehicle Silhouettes | 59.0 | 75.9 | 16.9 | 224.5 |
| ISOLET | 96.0 | 94.8 | -1.2 | 881.0 |
| MONK's Problems | 74.5 | 97.4 | 22.9 | 60.6 |
| Credit Approval | 65.9 | 71.2 | 5.3 | 104.7 |
| Balance Scale | 96.2 | 96.2 | 0 | 56.0 |

Table 5: Successive Halving Performance on Classification Datasets