Maggy: Scalable Asynchronous Parallel Hyperparameter Search

Moritz Meister
moritz@logicalclocks.com
Logical Clocks AB
Stockholm, Sweden

Sina Sheikholeslami
sinash@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Amir H. Payberah
payberah@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Vladimir Vlassov
vladv@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Jim Dowling
jdowling@kth.se
KTH Royal Institute of Technology
Logical Clocks AB
Stockholm, Sweden

ABSTRACT

Running extensive experiments is essential for building Machine Learning (ML) models. Such experiments usually require iterative execution of many trials with varying run times. In recent years, Apache Spark has become the de-facto standard for parallel data processing in the industry, in which iterative processes are implemented within the bulk-synchronous parallel (BSP) execution model. The BSP approach is also being used to parallelize ML trials in Spark. However, the BSP task synchronization barriers prevent asynchronous execution of trials, which leads to a reduced number of trials that can be run on a given computational budget. In this paper, we introduce Maggy, an open-source framework based on Spark, to execute ML trials asynchronously in parallel, with the ability to early stop poorly performing trials. In the experiments, we compare Maggy with the BSP execution of parallel trials in Spark and show that on random hyperparameter search on a convolutional neural network for the Fashion-MNIST dataset Maggy reduces the required time to execute a fixed number of trials by 33% to 58%, without any loss in the final model accuracy.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**; **Search methodologies**; **Parallel computing methodologies**.

KEYWORDS

Machine Learning, Scalable Hyperparameter Search, Asynchronous Hyperparameter Optimization

ACM Reference Format:
Moritz Meister, Sina Sheikholeslami, Amir H. Payberah, Vladimir Vlassov, and Jim Dowling. 2020. Maggy: Scalable Asynchronous Parallel Hyperparameter Search. In *1st Workshop on Distributed Machine Learning (DistributedML'20), December 1, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3426745.3431338

DistributedML'20, December 1, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-8182-6/20/12…$15.00
https://doi.org/10.1145/3426745.3431338

1 INTRODUCTION

Traditionally, building Machine Learning (ML) models used to be an expensive and time-consuming process. However recently, Automated ML (AutoML) approaches have enabled data scientists to automate many aspects of this process at the cost of increased computational resources. Nevertheless, many parts of building ML models behave as black-boxes without gradient information of the loss available, thus AutoML has to fall back on less efficient search algorithms to optimize them. These search algorithms are executed in *experiments*, where a model is trained with different *configurations* (such as different learning rates or convolution filter sizes) to produce a performance metric (such as any loss or accuracy metric), which are then used by the search algorithm to propose new, potentially better configurations. Training such a model configuration is referred to as a *trial*.

In Deep Learning (DL), models are ever-growing in architecture size and complexity to beat the previous state-of-the-art. However, training large models with massive amounts of data not only increases the training time, but also causes a state explosion in the search space, as the performance of these models becomes more sensitive to a growing number of *hyperparameters*. Hyperparameters are parameters of an ML model (such as learning rate or choices about the model's architecture, regularization, and optimization) that cannot be optimized by the learning algorithm itself.

These characteristics render search extremely costly, as exploding search spaces require the evaluation of exponentially more trials. Moreover, to make DL models more robust and explainable, a new best practice, called *ablation studies* [18, 21], has evolved that is in nature similar to hyperparameter search experiments. Ablation studies require many trials to evaluate the relative contribution of different architectural and regularization components to models' performance. Therefore, they also suffer from the same curse of dimensionality with increasing model size.

Current state-of-the-art solutions for *hyperparameter optimization (HPO)* mainly schedule trials and update the search model asynchronously [8, 15]. Given that Apache Spark [25, 26] has become a popular data-parallel processing framework, the industry is increasingly building tools to accommodate the advanced algorithms for HPO on Spark [4, 5]. Spark implements iterative processes, such as HPO, within the *bulk-synchronous parallel (BSP)* execution model. However, the task synchronization barriers in BSP prevent asynchronous execution of trials, which leads to a reduced number of

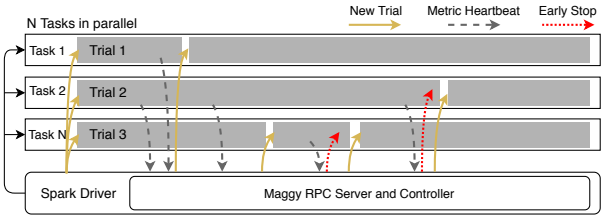

Figure 1: Maggy enables driver-to-executor communication that allows for globally managed asynchronous trials within the bulk-synchronous model.

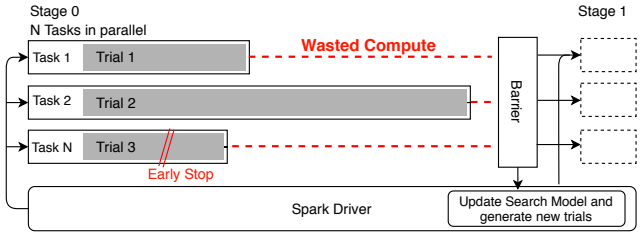

Figure 2: The bulk synchronous execution of iterations in Apache Spark only allows new trials to be executed as tasks at the beginning of a stage. This synchronization barrier results in wasted computation (when a trial is stopped early, or due to straggling trials) and delayed updates to the search model.

trials that can be run on a given computational budget. On the other hand, actor-based systems with their inherent asynchrony, like Ray [19], have shown to be a good fit for parallelized ML experiments. Nevertheless, history has shown that general-purpose programming frameworks (such as Spark), when equipped with specialized functionalities, tend to dominate specialized frameworks, in the long run.

In this paper, we introduce Maggy, a framework for parallel ML experiments that extends Spark with support for asynchronous trials, early stopping, and global trial optimization. Maggy introduces both a programming model for these experiments and a new driver-to-executor communication protocol that allows for globally managed asynchronous trials within the bulk-synchronous execution model (Figure 1). This protocol uses driver-worker heartbeats to add early-stopping and asynchronous scheduling functionality within Spark tasks, as shown in Figure 1. Maggy tackles the following challenges in modern ML model development:

(1) programming support for defining, optimizing, and running parallel ML experiments;
(2) efficient use of parallel compute resources through asynchronous trials;
(3) support for *global* directed search in high-dimensional hyperparameter search spaces. By global optimization, we mean that the optimizer has complete and up-to-date knowledge of all trials' learning curves and can make decisions on early stopping of poorly performing trials.

The experimental evaluation of Maggy shows that it can reduce the run time of experiments with a fixed number of trials, requiring between 33% and 57% of the time that of a BSP Spark implementation. This reduction in time is achieved despite the added overhead of asynchronous communication, scheduling and performance sampling. The strength of Maggy becomes apparent with early stopping, which introduces additional variation in trial run times and therefore more asynchrony.

2 PRELIMINARY AND RELATED WORK

Although Apache Spark [25, 26] was initially developed for data-parallel processing, nowadays it provides a unified analytics engine, including ML applications. With its high-level libraries for SQL queries on semi-structured data, streaming data, ML, and graph processing, it became a general-purpose framework. The fundamental data structure in Spark is resilient distributed dataset (RDD), which is a distributed collection of items [25]. The RDD provides the core abstraction in Spark, enabling data-parallel processing and fault tolerance. However, the success and ease of use of Spark come from the high-level APIs building transparently around this abstraction.

A Spark *job* is expressed as a directed acyclic graph (DAG), capturing the interdependencies between stages of independent *tasks*. Within this computational model, we can parallelize ML experiments by mapping trials to tasks. However, this approach has some limitations due to the synchronization barrier at the end of a stage that results in the inability to early stop a poorly performing trial during a stage and reuse the executor of the task's available computation resources for other trials during the rest of the stage. The early stopping can be added to Spark by enabling the driver to collect statistics on the performance of trials at executors periodically, and send messages to workers to stop poorly performing trials. However, such implementation of early stopping in Spark still wastes compute resources by not enabling new trials to be run until the end of the stage (Figure 2).

In contrary to the data-parallelism of Spark, Ray [19] is based on an actor concurrency model. It provides a flexible and asynchronous computational model expressed in stateful actors and stateless tasks, which are executed dynamically, allowing for one task or actor to spawn new actors/tasks. The Ray asynchronous computation model makes it more suitable for iterative workloads. Ray ships with a Python library for scalable hyperparameter tuning, called Tune [16]. It integrates with many ML frameworks (such as Keras, PyTorch, and XGBoost), and comes with its own implementations of popular optimization algorithms and provides support for a variety of third-party optimization libraries and services like HyperOpt [4, 5], Bayesian Optimization (BO) [23], and Google Vizier [10]. Due to its asynchrony, Tune can support early stopping, as well as multi-fidelity methods, such as HyperBand [14], BOHB [8], and ASHA [15].

HyperOpt [4, 5] is a Python library for distributed asynchronous HPO that has similar goals to Maggy and was recently extended by a backend supporting distribution via Apache Spark. To overcome the inefficiencies of synchronous stages in Spark, Hyperopt maps ML trials to jobs with only one task. These jobs can then be executed asynchronously. However, this approach requires maintaining a thread for each scheduled job in the Spark driver, even if the job

is not running yet, to retrieve the job's results. The Spark driver typically runs on few computational resources and can therefore become a bottleneck. Moreover, this design adds the overhead of starting a new job for each trial. This architecture also does not support global early stopping decisions. The optimizer is unaware of the current performance of the trials being trained. Therefore, it cannot make decisions on early stopping taking into account the knowledge about all learning curves.

Keras is a popular high-level API for TensorFlow [1] and comes with a Python library to tune models, KerasTuner [20]. KerasTuner integrates seamlessly with the Keras APIs and enables distributed experiments by starting the experiment script on different machines or processes. In the vision of Keras, KerasTuner integrates with the Google Cloud APIs to automate the process of starting worker nodes in the Google Cloud account of a user [6]. KerasTuner provides implementations of a variation of HyperBand [14] and Bayesian Optimization [23], but no explicit support for early stopping. A unique feature of KerasTuner is the possibility of intra-trial distribution to scale the training of single trials.

3 HYPERPARAMETER OPTIMIZATION (HPO)

In this section, we briefly recall some basic concepts from *hyperparameter optimization (HPO)*. While AutoML aims to automate all aspects of the ML development process, a basic subproblem to solve is finding hyperparameters to maximize the performance of a model. Hutter et al. [11] provide a rich survey of AutoML methods, systems, and challenges, and classify HPO methods along two dimensions: *black-box* HPO and *multi-fidelity optimization*. However, we believe that considering the underlying execution systems, a third dimension should be added, which is the *execution strategy*. The characteristics of methods in both the previous classes might be altered when executed in parallel or asynchronously, and the execution strategy dimension introduces more opportunities for new methods. This section serves as an overview for state-of-the-art HPO extended by considerations for the execution strategy and argues for the need for an asynchronous system to support these.

3.1 Black-box Hyperparameter Optimization

Blackbox optimization methods are split into two subsets, *model-free* (undirected search) and *model-based* (directed search) optimization. The former method, such as grid or random search, can be run in parallel without further coordination, as trials can be generated ahead of time. In particular, random search is a popular baseline, since it can find configurations with performance arbitrarily close to the optimum if it has enough computational resources [11].

On the other hand, model-based methods, like BO [23], are inherently sequential and require coordination to collect metrics and update the optimization model. BO samples the next trial to be evaluated based on previous iterations' results by using Bayesian posterior updates to a surrogate model, and encoding the prior belief over the objective function. The surrogate model's predictive distribution enables acquisition functions to determine the utility of different candidate points at low cost, trading off exploration and exploitation of the search space.

In the parallel setting, several points should be sampled based on the same information. However, if we apply deterministic strategies, each worker would evaluate the same configuration. A straightforward approach to deploy BO in an asynchronous parallel execution strategy is to impute the result of pending trials [9] with a constant (constant liar approach) or a Gaussian Process (GP) [9] predictive mean (Kriging Believer).

Other approaches, such as Thompson Sampling (TS) [13] or Tree Parzen Estimators (TPE) [5], use penalization around the locations of pending trials to encourage diversity (PLAyBOOK algorithm) [2] or sampling through a stochastic process, purposefully not to optimize the acquisition function fully to incorporate diversity. These asynchrony-enabling methods have shown to outperform their synchronous counterparts [2, 5, 13].

3.2 Multi-fidelity Optimization and Early Stopping

Multi-fidelity optimization methods rely on evaluating many trials on small computational budgets (low fidelities) and allocating more budget to a few promising trials. Here, for example, the budget can be the number of epochs for training a neural network and the amount of data used for training. Successive Halving (SHA) [12] and its successors HyperBand [14] and Asynchronous Successive Halving (ASHA) [15] are three examples of multi-fidelity optimization. Both SHA and HyperBand or ASHA rely on random sampling to generate new hyperparameter configurations. In contrast, Falkner et al. [8] introduce BOHB that uses TPE [5] together with HyperBand [14] and achieve the performance above state-of-the-art results on several ML benchmark problems. While fidelity optimization makes the budget allocation decision before starting a trial, other approaches make early stopping decisions at runtime. Such methods are performance curve prediction [3] or simple heuristics like median stopping rules, as used by Google Vizier [10]. Again, these methods benefit from a central source of truth with knowledge of all trials' learning curves to make optimal early stopping decisions.

4 MAGGY

In this section, we introduce Maggy, a system for asynchronous parallel HPO. Below, we first describe Maggy's programming model and then explain its implementation details. Maggy is open-source and available at the following link[1].

4.1 Programming Models

Parallel computing support for model training and HPO offers many benefits, such as the ability to reduce training time and hyperparameter experiments by adding more compute resources . However, parallel execution introduces additional obtrusive code artifacts and modifications, depending on the frameworks used, which leads to infrastructure code mixed with model training code. The programming model of Maggy can help avoid the problem of mixing infrastructure and training logic by enabling write-once and transparently distributed training functions. The same code, then, can be reused in Python program on a laptop or a cluster-scale PySpark program. The programs, written in Maggy framework, are *oblivious*

[1] https://github.com/logicalclocks/maggy

training functions [17] as we factor out distribution-related dependencies using best-practice programming idioms (such as functions to generate models and data batches).

In Maggy, users define the training logic in a (higher-order) function that returns the models performance metric (e.g., any loss or accuracy metric), which is to be optimized. The function is parametrized with hyperparameters and generator functions for the model and data (Listing 1). This function, then, is launched with a user-specified search space and optimizer through the *lagom*[2] API (Listing 2).

```
def train_fn(hparam1, hparam2, ..., model_fn, dataset_fn):
    model = model_fn(hparam1, hparam2)
    model.compile(hparam3)
    train, test, val = dataset_fn()
    model.fit(train, ...)
    metric = model.evaluate(test, ...)
    return metric["metric_to_be_optimized"]
```

Listing 1: Example of an oblivious training function.

```
from maggy import experiment, Searchspace
searchspace = Searchspace(hparam1=('DOUBLE', [0,1]), ...)
experiment.lagom(train_fn, controller="BOHB", searchspace)
```

Listing 2: Example of launching an experiment with lagom.

This way, Maggy instantiates the training function with different sets of parameters and launches them as trials on Spark executors, without requiring users to write code managing the distribution and execution of the training logic on the workers. In return, the users will get the metrics to be optimized from the training function, or a collection of items to be tracked along with the experiment and specify which returned metric is to be optimized. Note that the produced code is still pure Python code, and it can be run on a cluster of machines as on a single host environment by fixing the parameters and inputs.

Maggy currently ships with implementations of random search and BO (TPE [5] and GP [9]) as optimizers and HyperBand, ASHA, and a median stopping rule for early stopping. However, Maggy provides base classes for both these entities as part of a developer API to make it extensible. Users can implement their own optimizers or early stopping rules.

4.2 Design and Implementation

Maggy is built on top of Spark and provides an easy to use and scalable system for ML experiments, with support for GPUs from version 3.0. In principle, Maggy uses Spark as a resource manager with enhanced fault tolerance support. Maggy executes experiments as launching Spark applications, where the requested number of executors (degree of parallelism) are each blocked with one long-running task, executing trials in a loop until the experiment finishes.

Maggy provides the aforementioned functionality through a non-blocking RPC framework built within the Spark *driver* and *executors* (Figure 3). On the driver-side, Maggy runs a controller

[2]Lagom is a Swedish word meaning "just the right amount".

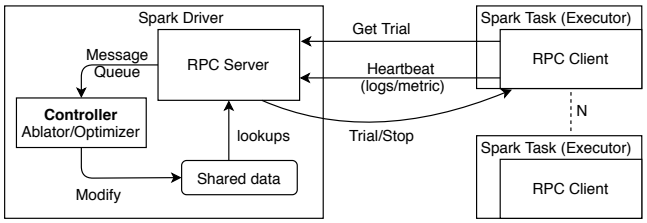

Figure 3: Maggy is setup as a RPC framework within the Spark Driver and Executors. The figure shows the entities and the flow of information for the communication protocol and runtime behaviour.

thread responsible for the experiment's global control, such as trial generation and early stopping. It communicates with a RPC server thread by modifying controls in a shared data layer and a message queue. The RPC server then responds to the clients' requests by performing lookups on the shared data or forwarding the message to the controller. The shared data layer is required for the server not to block until the controller executed the remote procedure, such as sampling a new trial. To avoid the driver becoming a single point of failure, a distributed file system or cloud storage can be leveraged to persist controller state.

On the other side, each executor runs a RPC client that requests and starts new trials, sends heartbeats with the current training metric during training, and can early stop a trial when it receives a stop signal in response to a heartbeat. A client polls for new trials and receives early stopping decisions as a response to the heartbeats sent with the current training metric. The client is stateless, hence, in case of failure, Spark can easily restart the task and start a new client polling for trials. In scenarios of experiments with runs for long periods of time, this results in the loss of single trials, which are transparently rescheduled by the controller. A worker that repeatably fails to execute trials is blacklisted from receiving future trials.

A crucial point for collecting the current training metric for early stopping is the connection between the user code and the RPC client. In order to hook into the user code, users have two options (Listing 3), either (i) make use of a reporter API to broadcast the metric with a heartbeat at the end of an iteration manually, or (ii) if a high-level framework like Keras is used, Maggy provides callbacks to be added to the training logic, doing the same thing automatically. Approach (i) is especially useful for cases when the iteration loop is programmed by the user itself, as it is the case in PyTorch, for example.

5 EXPERIMENTS

We evaluated Maggy by comparing its performance with synchronous parallel trials on Spark (equivalent to existing parallel hyperparameter tuning frameworks on Spark, such as Databricks' HyperOpt [7]). We trained a three-layer convolutional neural network with a fully connected layer on the Fashion-MNIST [24] dataset. Compared to MNIST, Fashion-MNIST requires more time to train and is more difficult to get high accuracy on, enabling us to measure the effect of early stopping. We apply the median early-stopping

```
(i)
from maggy import reporter
for current_epoch in range(epochs):
    ...
    reporter.broadcast(metric=accuracy, step=current_epoch)

(ii)
from maggy.callbacks import KerasEpochEnd
callback = KerasEpochEnd(reporter, 'val_acc')
...
keras.fit(..., callbacks=[callback], ...)
```

Listing 3: The reporter API is used to broadcast a specified metric in the heartbeats to the controller, or via the Keras Callback interface.

rule [22] in Maggy to stop trials performing worse than the median after the first four trials have completed at the same point in time during training (in terms of stochastic gradient descent optimization steps). In experiments on Maggy and Spark (synchronous parallel trials), we run a fixed number of trials (N=100) with random search for hyperparameters. We vary the number of workers from 4, to 8, to 16, to 32. The space of hyperparameters explored using random search in both Maggy and Spark is as shown in Listing 4.

```
sp = Searchspace(kernel=('INTEGER', [2, 8]),
    pool=('INTEGER', [2, 8]), dropout=('DOUBLE', [0.01, 0.99]),
    learning_rate=('DOUBLE', [0.000001, 0.99]))
```

Listing 4: Hyperparameter space for Fashion-MNIST.

The performance of hyperparameter tuning experiments is closely linked to the sensitivity of the model being tuned to small changes in hyperparameters and the relative number of points in hyperparameter space that contains poorly performing hyperparameter

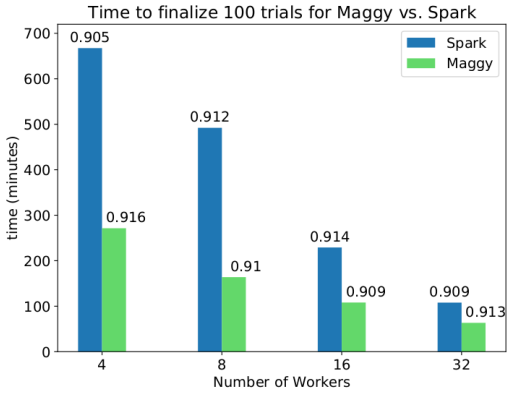

Figure 4: Asynchronous trials and the median stopping rule in Maggy enables N=100 trials to be executed in lower wallclock time compared to Spark without any loss in accuracy (denoted on top of the bars). Adding more workers linearly reduces the total time required to execute all hyperparameter trials, both for Maggy and Spark. Maggy's reduced execution time holds for varying number of workers (W=4, 8, 16, 32).

Table 1: Relative speedup of Maggy over the general Spark implementation, total experiment runtime in seconds and number of early stopped trials by Maggy.

Workers	Maggy/Spark	Maggy (s)	Spark (s)	Early-Stop
4	0.41	16284	40051	54
8	0.33	9828	29511	52
16	0.47	6486	13745	47
32	0.58	3804	6474	44

Table 2: Final accuracy after 100 trials.

Workers	Maggy Accuracy	Spark Accuracy
4	0.915	0.905
8	0.909	0.912
16	0.909	0.913
32	0.913	0.909

combinations. The Fashion-MNIST hyperparameter space used in these experiments is relatively homogeneous, and we can see that all experiments converged to very similar accuracy. Other networks, such as Generative Adversarial Networks are notoriously difficult to produce reproducible experiments.

As we can see in Figure 4 and Figure 5, Maggy reduces the wallclock time for random search hyperparameter trials by roughly half when using the median early-stopping rule, without any loss in accuracy. In Table 1, we can see that the median stopping rule stops, on average, half of the trials, reducing total execution time by approximately half. In Table 2, we can see that both Maggy and Spark converge to similar accuracy, even though half of Maggy's under-performing trials were stopped early.

6 CONCLUSION

Spark is now a popular general purpose programming framework that is used at all stages in machine learning pipelines, from feature engineering to parallel hyperparameter tuning to distributed model training. However, actor-based frameworks have shown better performance for asynchronous ML trials, leading many developers to switch part of their pipelines to such frameworks. In this paper, we introduced Maggy as an extension to Spark's synchronous processing model to allow it to run asynchronous ML trials, enabling end-to-end state-of-the-art ML pipelines to be run fully on Spark. Maggy provides programming support for defining, optimizing, and running parallel ML trials. Users can define their own global optimizer for directed search in a high-dimensional hyperparameter search space, and the Maggy runtime will manage the performance monitoring, scheduling, and early-stopping of asynchronous trials within Spark's synchronous execution model.

ACKNOWLEDGMENTS

This work is supported by the ExtremeEarth[3] project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

[3]ExtremeEarth project website: http://earthanalytics.eu

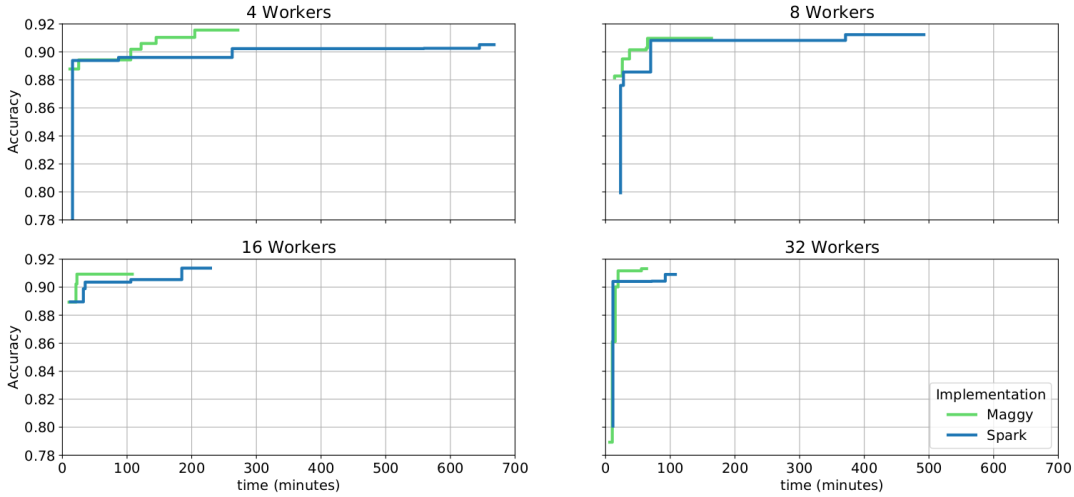

Figure 5: Maggy finds better configurations faster due to asynchronous trials and the median stopping rule in Maggy. Due to shorter trials, Maggy concludes experiments with the same number of trials in shorter wallclock time. In Spark, trials are executed to completion (no early stopping), yielding similar accuracy as expected, but resulting in higher wallclock time to execute N=100 trials compared to Maggy.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Ahsan S Alvi, Binxin Ru, Jan Calliess, Stephen J Roberts, and Michael A Osborne. 2019. Asynchronous Batch Bayesian Optimisation with Improved Local Penalisation. *arXiv preprint arXiv:1901.10452* (2019).

[3] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Practical Neural Network Performance Prediction for Early Stopping. *arXiv preprint arXiv:1705.10823* 2, 3 (2017), 6.

[4] James Bergstra, Daniel Yamins, and David Cox. 2012. *Hyperopt: Distributed Asynchronous Hyper-parameter Optimization.* Retrieved May 21, 2020 from http://hyperopt.github.io/hyperopt

[5] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning*. 115–123.

[6] Francois Chollet. 2020. Keras: The Next Five Years. Retrieved May 21, 2020 from https://www.youtube.com/watch?v=HBqCpWldPII

[7] Databricks. 2019. Scaling Hyperopt to Tune Machine Learning Models in Python. Retrieved Sep 18, 2020 from https://databricks.com/blog/2019/10/29/scaling-hyperopt-to-tune-machine-learning-models-in-python.html

[8] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. *arXiv preprint arXiv:1807.01774* (2018).

[9] David Ginsbourger, Janis Janusevskis, and Rodolphe Le Riche. 2011. Dealing with Asynchronicity in Parallel Gaussian Process based Global Optimization.

[10] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1487–1495.

[11] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated Machine Learning: Methods, Systems, Challenges.* Springer Nature.

[12] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Artificial Intelligence and Statistics*. 240–248.

[13] Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. 2018. Parallelised Bayesian Optimisation via Thompson Sampling. In *International Conference on Artificial Intelligence and Statistics*. 133–142.

[14] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[15] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2018. Massively Parallel Hyperparameter Tuning. *arXiv preprint arXiv:1810.05934* (2018).

[16] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* (2018).

[17] Moritz Meister, Sina Sheikholeslami, Robin Andersson, Alexandru A Ormenisan, and Jim Dowling. 2020. Towards Distribution Transparency for Supervised ML With Oblivious Training Functions. In *Workshop on MLOps Systems*.

[18] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. 2019. Ablation Studies in Artificial Neural Networks. *arXiv preprint arXiv:1901.08644* (2019).

[19] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[20] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. 2019. *Keras Tuner.* Retrieved May 21, 2020 from https://github.com/keras-team/keras-tuner

[21] Mohammad Pezeshki, Linxi Fan, Philemon Brakel, Aaron Courville, and Yoshua Bengio. 2016. Deconstructing the Ladder Network Architecture. In *International Conference on Machine Learning*. 2368–2376.

[22] Lutz Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.

[23] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[24] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.