Reuse in Pipeline-Aware Hyperparameter Tuning

Liam Li[*1], **Evan Sparks**[2], **Kevin Jamieson**[3], **and Ameet Talwalkar**[1,2]
[1]CMU, [2]DeterminedAI, [3]UC Berkeley

Abstract

Hyperparameter tuning of multi-stage pipelines introduces a significant computational burden. Motivated by the observation that work can be reused across pipelines if the intermediate computations are the same, we propose a pipeline-aware approach to hyperparameter tuning. Our approach optimizes both the *design* and *execution* of pipelines to maximize reuse. We design pipelines amenable for reuse by (i) introducing a novel hybrid hyperparameter tuning method called gridded random search, and (ii) reducing the average training time in pipelines by adapting an early-stopping hyperparameter tuning approach. We then realize the potential for reuse during execution by introducing a novel caching problem for ML workloads which we pose as a mixed integer linear program (ILP), and subsequently evaluating various caching heuristics relative to the optimal ILP solution. We conduct experiments on simulated and real-world machine learning pipelines to show that a pipeline-aware approach to hyperparameter tuning can offer over an order-of-magnitude speedup over independently evaluating pipeline configurations.

1 Introduction

Modern machine learning workflows combine multiple stages of data-preprocessing, feature extraction, and supervised and unsupervised learning [Sánchez et al., 2013, Feurer et al., 2015]. The methods in each of these stages typically have configuration parameters, or *hyperparameters*, that influence their output and ultimately predictive accuracy. Although tools have been designed to speed up the development and execution of such complex pipelines [Meng et al., 2016, Pedregosa et al., 2011, Sparks et al., 2017], tuning hyperparameters at various pipeline stages remains a computationally burdensome task.

Some tuning methods are sequential in nature and recommend hyperparameter configurations one at a time for evaluation [e.g. Hutter et al., 2011, Bergstra et al., 2011, Snoek et al., 2012], while other batch mode algorithms [e.g. Li et al., 2017, Krueger et al., 2015, Sabharwal et al., 2016] evaluate multiple configurations simultaneously. In either case, holistically analyzing the functional structure of the resulting collection of evaluated configurations introduces opportunities to exploit computational reuse in the traditional black-box optimization problem.

Given a batch of candidate pipeline configurations, it is natural to model the dependencies between pipelines via a directed acyclic graph (DAG). We can eliminate redundant computation in the original DAG by collapsing shared prefixes so that pipeline configurations that depend on the same intermediate computation will share parents, resulting in a *merged DAG*. We define *pipeline-aware hyperparameter tuning* as the task of leveraging this merged DAG to evaluate hyperparameter configurations, with the goal of exploiting reuse for improved efficiency.

Consider the example of the three toy pipelines in Figure 1(a). All of them have the identical configuration for operator A, and the first two have the same configuration for operator B, leading to the merged DAG illustrated in Figure 1(b). Whereas treating each configuration independently

[*]Corresponding author email: `me@liamcli.com`.

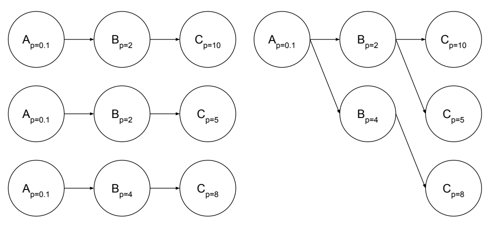


Figure 1: Three pipelines with overlapping hyperparameter configurations for operations A, B, and C. (a) Three independent pipelines. (b) Merged DAG after eliminating shared prefixes.

requires computing node $A_{p=0.1}$ three times and node $B_{p=2}$ twice, operating on the merged DAG allows us to compute each node once. Applying the same sort of common prefix elimination to real pipelines can eliminate a substantial amount of redundant work.

In this work, we tackle the problem of pipeline-aware hyperparameter tuning by optimizing both the *design* and *evaluation* of merged DAGs to maximally exploit reuse. In particular, we identify the following three core challenges, and our associated contributions addressing them.

Designing Reusable DAGs. Hyperparameter configurations are typically randomly sampled, often from continuous search spaces. Since pipelines that differ in even one node represent different operations on the data, the resulting merged DAGs consequently are extremely limited in terms of their shared prefixes, which fundamentally restricts the amount of possible reuse. To overcome this limitation, we introduce a novel configuration selection method that encourages shared prefixes. In Section 2.1 we introduce our hybrid approach, called *gridded random search*, which limits the branching factor of continuous hyperparameters, while performing more exploration than standard grid search. We show empirically that gridded random search performs comparably with random search, in spite of the known empirical limitations of grid search.

Designing Balanced DAGs. Modern learning models are computationally expensive to train. Therefore, the leaf nodes of the DAG can be more expensive to evaluate than intermediate nodes, limiting the impact of reuse. To balance backloaded computation, we turn to hyperparameter tuning strategies that exploit early-stopping and partial training. In particular, we show how the Successive Halving algorithm [Jamieson and Talwalkar, 2015], can be used to drastically reduce average training time, resulting in a more balanced DAG.

Exploiting Reuse via Caching. Exploiting reuse requires an efficient cache strategy suitable for machine learning pipelines, which can have highly variable computational profiles at different nodes. Although the general problem of caching is well studied in computer science [Sleator and Tarjan, 1985, McGeoch and Sleator, 1991], there is limited work on caching machine learning workloads, where the cost of computing an item can depend on the state of the cache. We thus introduce the *cache-dependent generalized paging problem* for this setting. We show that the optimal cache strategy is the solution to a mixed integer linear program (Section 3), and then empirically compare this optimal solution with faster heuristics. Surprisingly, our results demonstrate that the least-recently used (LRU) cache eviction heuristic used in many frameworks (e.g., `scikit-learn` and `MLlib`) is not well suited for this setting, and we thus propose using WRECIPROCAL [Gunda et al., 2010], a caching strategy that accounts for both the size and the computational cost of an operation.

Finally, in Section 4, we apply these three complementary approaches to real pipelines and show that pipeline-aware hyperparameter tuning can offer more than an *order-of-magnitude* speedup compared with the standard practice of evaluating each pipeline independently. We focus on total execution time required as opposed to accuracy in our experiments to isolate the computational savings from reuse. Note that our pipeline-aware hyperparameter optimization can be used with many hyperparameter optimization algorithms.

2 Designing 'Good' DAGs

In this section, we focus on the two design challenges discussed in Section 1: (i) designing reusable DAGs with prefix sharing across pipelines when faced with search spaces with continuous hyperparameters, and (ii) designing balanced DAGs in spite of backloaded computation due to computationally expensive model training routines.

2.1 Reusable DAGs via Gridded Random Search

Grid search is a classical approach for hyperparameter tuning that discretizes each hyperparameter dimension, rendering it naturally amenable to reuse via prefix sharing. However, sampling randomly in each dimension (random search) has been shown to be more efficient empirically than grid search [Bergstra and Bengio, 2012]. We proposed a novel hybrid method called gridded random to combine reuse from grid search and improved accuracy from random search.

Gridded random encourages prefix sharing by controlling the branching factor from each node in a given stage of the pipeline, promoting a narrower tree-like structure. The key difference from grid search is that the children of different parent nodes from the same level are not required to have the same hyperparameter settings (see Appendix: Figure 6). In effect each node performs a separate instance of gridded random search for the remaining downstream hyperparameters.

Gridded random search introduces a tradeoff between the amount of potential reuse and coverage of the search space. Generally, the branching factor should be higher for nodes that have more hyperparameters in order to guarantee sufficient coverage. We compare gridded random search to random search in Section A.2 and show that they perform comparably, despite the former's limited coverage of the search space relative to the latter.

2.2 Balancing DAGs via Successive Halving

Algorithm 1 Successive Halving Algorithm.

```
input: set of configurations $P$, maximum resource per configuration $R$, elimination rate $\eta$,
generations $G \geq 1$

assert $n \geq \eta^{G-1}$ so that at least one configuration will be trained for $R$ resource
$r = R\eta^{-(G-1)}$
for $g$ in $1, \dots G$ do
    step: Train each configuration in $P$ for $r\eta^{g-1}$ resource
    $P \leftarrow$ prune($P$): evaluate and keep top $\lfloor |P|/\eta \rfloor$ configurations
    $P \leftarrow$ populate($P$): warmstart with remaining configurations
end for
```

The benefit from reuse is limited for pipelines with long training times due to Amdahl's Law, i.e., the relative amount of redundant computation is only a small fraction of the total time. Hence, for backloaded pipelines, we can increase the potential for reuse by reducing the average training time per configuration, e.g., by employing iterative early-stopping hyperparameter tuning algorithms when evaluating a batch of configurations.

The Successive Halving algorithm [Karnin et al., 2013] is a hyperparameter optimization routine that uses early-stopping to speedup the search for a good configuration. As shown in Algorithm 1, SH with an elimination rate of η begins with a set of configurations P and proceeds as follows: allocate a small amount of resource to each configuration, keep the top $1/\eta$ configurations, and repeat with the remaining configurations, while increasing the resource per configuration in each generation by a factor of η. Notably, the specified inputs η, G (generations) and R (maximum resource per configuration) define the minimum resource per configuration is $r = R\eta^{-(G-1)}$.

As an example of how SH can be combined with pipeline-aware hyperparameter tuning, consider the DAG in Figure 2 where the time needed for the preprocessing steps in the pipeline is the same as the training time. In this example, SH use $3R$ resources for training compared to the $16R$ that would be needed without early-stopping. In general, for a maximum resource of R per configuration, a total budget of nR is needed to evaluate n configurations without early-stopping. In contrast, Successive

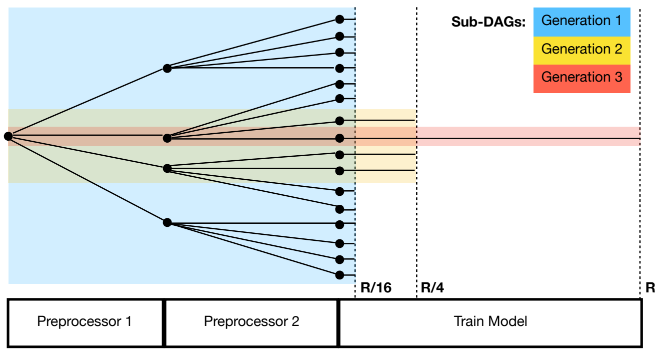


Figure 2: In this DAG, all pipelines start with the raw data in the root node and proceed to subsequent steps of the pipeline with a branching factor of 4 for a total of 16 pipelines. The lengths of the edges correspond to the time needed to compute the subsequent node. Hence, for the depicted pipelines, the time needed to train a model on R resource is equal to the time needed to compute the two preprocessing steps. Switching over to Successive Halving (SH) with elimination rate $\eta = 4$ and generations $G = 3$, the shaded blocks indicate the resulting pruned DAGs after each generation. The lengths of the edges in the 'Train Model' phase correspond to resources allocated to those configurations by SH: 16 configurations are trained for $R/16$ in the first generation, 4 configurations for $R/4$ in the second generation, and one configuration for R in the last generation. Hence, the total training time required for SH is $3R$, which is over $5\times$ smaller than the $16R$ needed without early-stopping, leading to a DAG with more frontloaded computation.

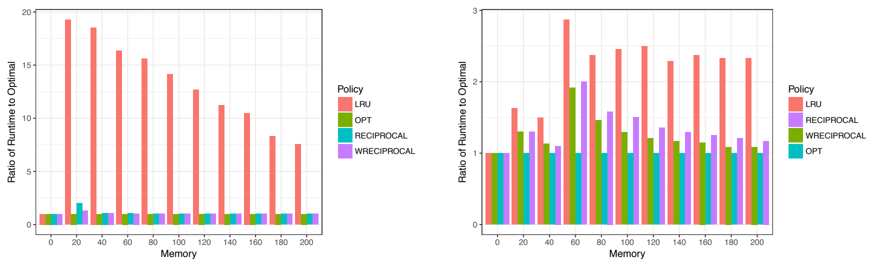
Halving requires a budget equivalent to $nR\eta^{-(G-1)}G$, where $nR\eta^{-(G-1)}$ indicates the total resource allocated in each generation. Hence, SH can be used to balance backloaded computation by reducing the total training time by a factor of η^{G-1}/G relative to no early-stopping.

3 Exploiting Reuse via Caching

Given the discussion in Section 2 on optimizing the design of DAGs, we now tackle the third challenge of realizing this potential through efficient caching of intermediate results. We consider a setting central to caching for machine learning pipelines, where the cost of computing an item depends on the state of the cache, i.e., it is cheaper to compute a node if any of the intermediate computation it depends on are in cache. We term this the *cache-dependent generalized paging problem*. We provided a discussion of related work in Appendix A.1 to distinguish the setting we consider from previously studied cache problems.

In the perfect information setting, where the caching algorithm has access to not only the DAG but also the computational costs and memory requirements of all nodes, the cache-dependent generalized paging problem can be solved to recover the optimal cache policy. Specifically, as we show in Appendix A.3, the cache-dependent generalized paging problem can be formulated as a mixed integer linear program (ILP). Solving the ILP for a particular DAG and memory constraint will yield the optimal caching strategy. However, solving ILPs is NP-hard and it is not feasible to solve for the optimal caching strategy in a reasonable time for larger DAGS (Appendix A.3.1). Additionally, the computational cost and memory requirement for all nodes within a pipeline search DAG may not be known beforehand, so an online strategy that makes cache decisions solely based on previous experience is needed.

Hence, we compare the optimal strategy (OPT) from solving the ILP to the following online strategies, which are discussed in more detail in Appendix A.1: (1) LRU: evict least recently used items first; (2) RECIPROCAL: items are evicted with probability inversely proportional to the cost [Motwani and Raghavan, 1995]; and (3) WRECIPROCAL: items are evicted with probability inversely proportional to cost and directly proportional to size [Gunda et al., 2010]. We evaluate the caching algorithms on synthetic DAGs of k-ary trees with branching factor $k = 3$ and and depth $d = 3$. In the following experiments, randomized caching strategies are simulated 100 times to reduce variance.



(a) All nodes have size 10; root node has cost 100 and all other nodes have cost 1.

(b) For each node, size is 10 or 50 and cost is 1 or 100 (both randomized).

Figure 3: A comparison of cache management policies on a 3-ary tree with depth 3.

First we consider a scenario where the memory size of each result is fixed at 10 and the root node is assigned a computational cost of 100 while all other nodes have a cost of 1. Figure 3a shows the results of this experiment. LRU performs poorly for this configuration precisely because the root node in the tree uses 100 units of computation and LRU does not account for this. The other strategies converge to the optimal strategy relatively quickly.

Next, we allow for variable sizes and costs. Each node is randomly given a size of either 10 or 50 units with equal probability and a corresponding cost of 1 or 100. Figure 3b shows the competitiveness of each strategy vs. the optimal strategy. In this setting, LRU is again the least effective policy while WRECIPROCAL is closest to optimal for most cache sizes. These results on synthetic DAGs suggest that LRU, the default caching strategy in `scikit-learn` and `MLlib`, is poorly suited for machine learning pipelines. Our experiments in the next section on real-world tasks reinforce this conclusion.

4 Empirical Studies

In this section we evaluate the speedups from reuse on real-world hyperparameter tuning tasks gained by integrating our proposed techniques. We consider tuning pipelines for the following three datasets: 20 Newsgroups,[2] Amazon Reviews [McAuley et al., 2015], and the TIMIT Huang et al. [2014] speech dataset. Our results are simulated on DAGs generated using profiles collected during actual runs of each pipeline. Details on how the DAGs are created as well as the search spaces considered for each pipeline are available in Appendix A.4.

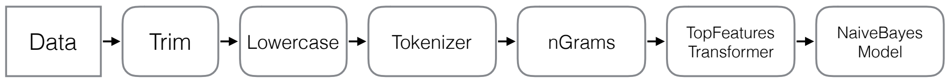
In Section 4.1, we examine the speedups from the caching component of pipeline-aware tuning on a text classification pipeline for the Newsgroup and Amazon Reviews datasets. The DAGs associated with both datasets are naturally amenable to reuse due to discrete hyperparameters and front-loaded computation. In Section 4.2, we then leverage our entire three-pronged approach to pipeline-aware hyperparameter tuning on a speech classification task that at first glance is not amenable to reuse.

4.1 Tuning Text Classification Pipelines

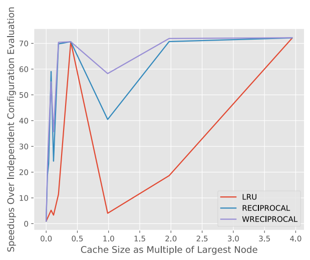
We tune the pipeline in Figure 4a for the 20 Newsgroups and the Amazon Reviews datasets. As shown in Figure 4, RECIPROCAL and WRECIPROCAL offer $70\times$ speedup over independent configuration evaluation for the 20 Newsgroups dataset. Similarly, for the Amazon Reviews dataset, all the caching strategies offer more than $40\times$ speedup over independent configuration evaluation. In these instances, caching offers significant speedups due to the front-loaded nature of the pipelines.

The frontloaded computation also explains why LRU underperforms on small memory sizes; by biasing older nodes, earlier and more expensive pipeline stages are more likely to be evicted. Another drawback of LRU is that it does not take into account the cost of computing each node and it will always try to cache the most recent node. Hence, once the size of the cache is large enough to cache the output of the largest node, LRU adds the node to cache and evicts all previous caches, while RECIPROCAL and WRECIPROCAL are less likely to cache the node. This explains the severe drop in performance for LRU once cache size exceeded 1. These results also show WRECIPROCAL

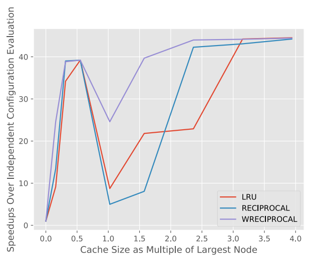
[2] `http://qwone.com/~jason/20Newsgroups/`



(a) Example pipeline for text classification.



(b) 20 Newsgroups



(c) Amazon Reviews

Figure 4: Effect of reuse on two real-world text processing hyperparameter tuning tasks. Speedups over independent configuration evaluation when evaluating a DAG with 100 pipelines are shown for each cache policy given a particular cache size.

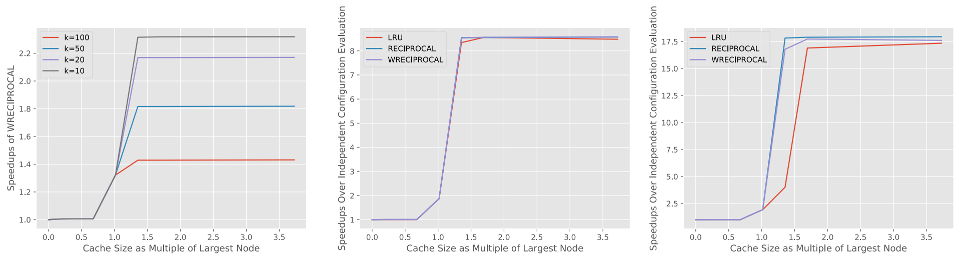


Figure 5: Speedups over independent configuration evaluation on TIMIT. Left: Speedups using WRECIPROCAL for a DAG with 100 pipelines generated using gridded random search with different branching factors in the random feature step. As expected, speedups from reuse is higher for DAGs with smaller branching factors. Middle: DAG with 100 pipelines and a maximum downsampling rate of $R/64$ for SH. Right: DAG with 256 pipelines and a maximum downsampling rate of $R/256$ for SH.

to be more robust than RECIPROCAL because it is less likely to cache large nodes with relatively inexpensive cost. Hence, we propose using WRECIPROCAL as an alternative to LRU in popular machine learning libraries.

4.2 Designing DAGs for Reuse on TIMIT

The TIMIT pipeline tuning task performs random feature kernel approximation followed by 20 iterations of LBFGS. The first stage of the pipeline is a continuous hyperparameter, so we first examine the impact of promoting prefix sharing through gridded random search. Figure 5(left) shows the speedups when using WRECIPROCAL over independent configuration evaluation for gridded random search with different branching factors. As expected, speedups increase with lower branching factor, though the speedups are muted for all branching factors due to the long training time for this task. We proceed with a branching factor of 10 for the random feature stage of the pipeline.

Next, we use Successive Halving (Section 2.2) with training set size as the resource to evaluate the impact on speedups when reducing the portion of time spent on the last stage of the pipeline. We run SH with the following parameters: $\eta = 4$ and $G = 4$ (which implies $r = R/64$). Figure 5 (middle) shows the speedups over independent configuration evaluation increases from $2\times$ for uniform allocation to over $8\times$ when using Successive Halving. Finally, we examine a more realistic setup for SH where the number of pipelines considered is higher to allow for a more aggressive downsampling rate ($n = 256$, $G = 5$, and $r = R/256$). Figure 5 (right) shows that in this case, speedups from reuse via caching reaches over $17\times$.

References

S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 31–40. Society for Industrial and Applied Mathematics, 1999.

N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. In *Foundations of Computer Science, 2007. FOCS '07. 48th Annual IEEE Symposium on*, pages 507–517, Oct 2007. doi: 10.1109/FOCS.2007.43.

N. Bansal, N. Buchbinder, and J. S. Naor. Randomized competitive algorithms for generalized caching. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 235–244. ACM, 2008.

L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

D. S. Berger, N. Beckmann, and M. Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):32:1–32:38, June 2018. ISSN 2476-1249.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 2012.

J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. In *Neural Information Processing Systems (NIPS)*, 2011.

S. Bubeck, M. B. Cohen, Y. T. Lee, J. R. Lee, and A. Mądry. K-server via multiscale entropic regularization. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 3–16, 2018.

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

A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, Dec. 1991.

P. Gijsbers, J. Vanschoren, and R. S. Olson. Layered tpot: Speeding up tree-based pipeline optimization. *arXiv preprint arXiv:1801.06007*, 2018.

P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

I. Gurobi Optimization. Gurobi optimizer reference manual, 2015. URL `http://www.gurobi.com`.

S. Hagedorn and K.-U. Sattler. Cost-based sharing and recycling of (intermediate) results in dataflow programs. In *ADBIS*, pages 185–199, 07 2018.

P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on TIMIT. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 205–209. IEEE, 2014.

F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization (LION)*, 2011.

S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 701–710, 1997. ISBN 0-89791-888-6.

M. Jaderberg and et. al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

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

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

T. Krueger, D. Panknin, and M. L. Braun. Fast cross-validation via sequential testing. *Journal of Machine Learning Research*, 16(1):1103–55, 2015.

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

J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*. ACM, 2015.

L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1):816–825, Jun 1991.

X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *JMLR*, 2016.

R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1995. ISBN 9780521474658. URL `https://books.google.com/books?id=QKVY4mDivBEC`.

C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine learning in Python. *JMLR*, 12: 2825–2830, 2011.

A. Sabharwal, H. Samulowitz, and G. Tesauro. Selecting near-optimal learners via incremental data allocation. In *AAAI*, pages 2007–2015, 2016.

J. Sánchez, F. Perronnin, T. Mensink, and J. Verbeek. Image classification with the fisher vector: Theory and practice. *International journal of computer vision*, 105(3):222–245, 2013.

D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL `http://doi.acm.org/10.1145/2786.2793`.

J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems (NIPS)*, 2012.

E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *International Conference on Data Engineering*, 2017.

A. Tyukin, S. Kramer, and J. Wicker. Scavenger – a framework for efficient evaluation of dynamic and modular algorithms. In A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, and M. Spiliopoulou, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 325–328. Springer International Publishing, 2015.

D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. G. Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *PVLDB*, 11:1958–1961, 2018.

A Appendix

We provided a detailed related work followed by supplementary material for Section 2, Section 3 and Section 4 are shown below.

A.1 Related Work

Hyperparameter tuning and caching are both areas with extensive prior work. We highlight pertinent results in both areas in the context of reuse.

Hyperparameter Tuning Methods.

Several methods, e.g., Jamieson and Talwalkar [2015], Li et al. [2017], Sabharwal et al. [2016], Jaderberg and et. al. [2017], Gijsbers et al. [2018], have exploited early-stopping to speed up hyperparameter search. These methods often reduce average training time per configuration by over an order-of-magnitude. As a result, they can benefit from significant computational speedups when deployed in a pipeline-aware context to generate balanced DAGs amenable for reuse. In our empirical results in Section 2.2, we focus on Successive Halving [Karnin et al., 2013, Jamieson and Talwalkar, 2015], a simple early-stopping approach that is widely applicable, theoretically principled, and which has been shown to achieve state-of-the-art results on a variety of hyperparameter tuning tasks.

Caching with Perfect Information. One straightforward way of reusing intermediate computation is to save results to disk. However, the associated I/O costs are large, thus motivating our exploration of efficient caching in memory as the primary method of realizing reuse in pipeline-aware hyperparameter tuning.

Finding an optimal cache management policy is known as the *paging problem* Bansal et al. [2007]. The classic paging problem assumes that all pages are the same size (i.e. memory requirement), and that all reads have the same cost (i.e. training time). Under these two assumptions, Belady's algorithm Belady [1966] is an optimal cache management policy for the perfect information setting, where the size and cost of all pages are known. If we relax the first assumption and allow for variable page sizes, solving the optimal policy is known to be NP-hard [Irani, 1997, Berger et al., 2018]. If we relax the second assumption and allow for variable read costs, we now have the *weighted paging problem*, for which the optimal policy can be obtained by solving an integer linear program [Bansal et al., 2007]. Lastly, if we relax both constraints, we have the *generalized paging problem*, which encompasses both aforementioned settings. The optimal perfect information policy for the generalized paging problem can similarly be obtained by solving an integer linear program Albers et al. [1999].

The aforementioned problem settings assume all reads have fixed cost. In contrast, we consider a setting central to caching for machine learning pipelines, where the cost of computing an item depends on the state of the cache, i.e., it is cheaper to compute a node if any of the intermediate computation it depends on are in cache. We study this *cache-dependent generalized paging problem* in Section 3.

Online Caching. In the previously described perfect information setting, the caching algorithm has access to the DAG, as well as the computational costs and memory requirements of all nodes. In contrast, in the online setting, the caching algorithm must make decisions based solely on its prior experience. Online algorithms often have significantly smaller computational overheads. In Section 3, motivated by computational concerns, we compare the optimal strategy to the cache-dependent generalized paging problem to three online strategies: (1) LRU: Least recently used items are evicted first. LRU is *k-competitive* with Belady's algorithm [Sleator and Tarjan, 1985] for the classic paging problem with uniform size and cost.[3] (2) RECIPROCAL: Items are evicted with probability inversely proportional to the cost. RECIPROCAL is k-competitive with the optimal caching strategy for the weighted paging problem [Motwani and Raghavan, 1995]. (3) WRECIPROCAL: Items are evicted with probability inversely proportional to cost and directly proportional to size. WRECIPROCAL is a weighted variant of RECIPROCAL that has been shown to work well for datacenter management [Gunda et al., 2010].

[3] An algorithm is k-competitive if its worst case performance is no more than a factor of k worse than the optimal algorithm, where k denotes the size of the cache.

For the classic paging problem, randomized online caching strategies outperform deterministic ones and are more robust [Fiat et al., 1991]. However, we include LRU in our comparison since it is widely used on account of its simplicity (including in some of the machine learning methods described below). Additionally, there is a large body of work on efficient randomized online caching policies for the weighted paging problem and generalized paging problem [Bansal et al., 2008, Bubeck et al., 2018]. Studying the empirical effectiveness of these more recent and complex caching methods for tuning machine learning pipelines is a direction for future work.

Caching in Machine Learning. There are existing methods that specifically address the issue of caching for machine learning pipelines. Hyperparameter tuning algorithms like TPOT and FLASH offer LRU caching of intermediate computation. However, as we show in Section 3, LRU is unsuitable for machine learning pipelines. Tyukin et al. [2015] created a framework for users to specify the logic of which operations to cache for reuse, but did not provide an automated scheduler. Sparks et al. [2017] addressed the issue of caching when evaluating a single pipeline, but did not account for the time-varying nature cache (i.e., did not allow for items to be evicted from cache to free up space). Lastly, Xin et al. [2018] and Hagedorn and Sattler [2018] address reuse in an iterative workflow where a user may run similar pipelines multiple times in the course of development, but the exact nature of future workflows is unknown. In contrast, we address a setting where all the pipelines to be evaluated are known and items can be stored or evicted from cache (Section 3).

A.2 Gridded Random Search Experiments

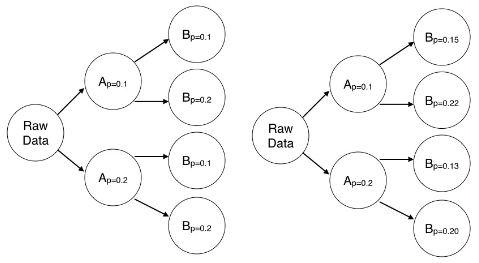


Figure 6: Comparison of grid search versus gridded random search. Grid search considers two different values for each hyperparameter dimension. In contrast, gridded random considers two hyperparameter values for operator A and four different values for operator B, while still maintaining a branching factor of two for the DAG. Note that random search would consider four different values for A and four different values for B but allow no potential for reuse.

We evaluate the performance of gridded random search compared to standard random search on 20 OpenML classification datasets. The datasets were selected from OpenML with the following criteria:

1. status is active;
2. ratio of majority class to minority class is less than 10;
3. no missing values;
4. not sparse;
5. is a classification task;
6. more than 5k and less than 100k instances;
7. more than 20 and less than 1k features.

We use $2/3$ of the data for training and $1/3$ for testing. We exclude datasets for which we can learn a perfect classifier on the test set.

The search space we consider includes the following pipeline stages: (1) Preprocessor: choice of none, standardize, normalize, or min/max scaler; (2) Featurizer: choice of PCA, select features with top percentile variance, or ICA; (3) Classifier: choice of linear classifier using stochastic gradient descent, random forests, or logistic regression. For each dataset, all four preprocessors are considered, but a single featurizer and a single classifier are randomly selected. For gridded random search, the branching factor in the first stage is 4, one for each preprocessor type; the branching factor is 5 per node for the selected featurizer; and then 5 per featurizer node for the selected classifier. This results in 100 total pipelines for gridded random search.

The search space considered for each component is shown in Table 1.

Table 1: Hyperparameters considered for the gridded random search experiments in Section 2.1.

Component	Name	Type	Values
PCA	variance to keep	continuous	$[0.5, 1.0]$
	whiten	binary	$\{True, False\}$
Select Percentile	percentile to keep	$[0.01, 1.0]$	
FastICA	# components	discrete	$[10, 2001]$
	whiten	binary	$\{True, False\}$
Random Forest	min samples to split	discrete	$[2, 21]$
	min samples per leaf	discrete	$[1, 20]$
	bootstrap	binary	$\{True, False\}$
RBF Kernel SVM	kernel scale	continuous	log $[1e-4, 10]$
l2 regularizer	continuous	log $[1e-3, 1e3]$	

Table 2: For each OpenML dataset, we randomly sample a featurizer and classifier to construct the search space. The right three columns show the difference in aggregate statistics between random search and gridded random search. Positive values indicates the performance of random is better than that of gridded random. For each task, statistics for the best hyperparameter setting found over 100 pipelines are computed over 10 independent runs.

Dataset ID	Featurizer	Classifier	Mean	20%	80%
OpenML 182	SelectPercentile	LogisticReg	0.1%	0.0%	0.0%
OpenML 300	SelectPercentile	SVC	0.6%	0.6%	0.6%
OpenML 554	PCA	LogisticReg	0.0%	0.1%	0.0%
OpenML 722	PCA	RandForest	-0.1%	-0.1%	-0.1%
OpenML 734	SelectPercentile	LogisticReg	0.0%	0.1%	0.0%
OpenML 752	SelectPercentile	LogisticReg	-0.0%	0.0%	-0.0%
OpenML 761	SelectPercentile	LogisticReg	0.6%	2.1%	0.0%
OpenML 833	SelectPercentile	RandForest	-0.0%	0.1%	-0.1%
OpenML 1116	PCA	RandForest	-0.2%	-0.2%	-0.3%
OpenML 1475	SelectPercentile	SVC	-0.8%	-0.9%	-0.9%
OpenML 1476	PCA	SVC	0.3%	-0.4%	0.0%
OpenML 1477	PCA	SVC	0.4%	-0.8%	0.1%
OpenML 1486	PCA	SVC	0.2%	0.3%	0.3%
OpenML 1496	FastICA	LogisticReg	0.0%	0.1%	-0.0%
OpenML 1497	PCA	RandForest	-0.3%	-0.1%	0.2%
OpenML 1507	SelectPercentile	SVC	0.1%	0.2%	0.0%
OpenML 4538	SelectPercentile	RandForest	-0.0%	-0.4%	0.3%
OpenML 23517	SelectPercentile	RandForest	-0.0%	0.0%	-0.1%
OpenML 40499	PCA	LogisticReg	2.9%	6.1%	1.5%
OpenML 40996	SelectPercentile	RandForest	-0.0%	-0.1%	-0.1%

For each dataset, we evaluated a set of 100 pipelines sampled using either random or gridded random and recorded the best configuration for each sampling method. To reduce the effect of randomness, we averaged results across 10 trials of 100 pipelines for each dataset. Our results in Table 2 show that on average gridded random search converges to similar objective values as standard random search on a variety of datasets. With the exception of one task, the differences in performance between random and gridded random are all below 1% accuracy. While by no means exhaustive, these results demonstrate the viability of using gridded random search to increase reuse.

A.3 Cached-Dependent Generalized Paging Problem as an ILP

We start with a few definitions. Let G denote a DAG consisting of vertices V and edges E. The *sources* of the graph are any $v_x \in V$ such that $v_x \neq v_j \; \forall (v_i, v_j) \in E$, while the *sinks* are any $v_x \in V$ such that $v_x \neq v_i \; \forall (v_i, v_j) \in E$.

An *execution plan* is a sequence of paths from sources to sinks that together include all edges in E. The execution plan can be viewed as all of the outputs the program would have to compute in the absence of a cache, with the only item in working memory being an operator's results and its direct inputs. As we progress through the execution plan, we call the path of the current node the *active path*. If we consider the graph in Figure 1, then the execution plan matching the $A_{p=0.1}B_{p=2}C_{p=10}C_{p=5}B_{p=4}C_{p=8}$ depth-first traversal is:

$$A_{p=0.1}B_{p=2}C_{p=10}, A_{p=0.1}B_{p=2}C_{p=5}, A_{p=0.1}B_{p=4}C_{p=8}$$

Let $(\boldsymbol{i})$ be a vector of length T where i_t is the tth node of the execution plan. At each time t, the cost of computing a node is given by the state of the cache. If there exists a node between i_t and the "active" sink that is cached, then the time devoted to i_t is 0, otherwise its $c_t = C(i_t)$, where C is a cost vector mapping each node to a non-negative real number. Intuitively, if a successor step of the current node is saved in cache, there is no need to perform the computation at the current node. Similarly, let $m_t = M(i_t)$ be the memory required to cache i_t, where M is a vector mapping each node to a non-negative real number.

Let Δ be a $V \times T$ matrix representing a sequence of valid actions on the cache, and define Δ as follows

$$\Delta_{j,t} = \begin{cases} 1 & \text{if } j \text{ is \textit{added} to the cache at time t} \\ -1 & \text{if } j \text{ is \textit{removed} from the cache at time t} \\ 0 & \text{otherwise} \end{cases}$$

Additionally, let $X \in \{0,1\}^{V \times T}$ represent the state of the cache through time with entries

$$X_{j,t} = \sum_{s<t} \Delta_{j,s} \; \forall j \in V$$

Then, the set of *valid* actions on the cache is severely limited; $\Delta_{j,t}$ may only be positive if $j = i_t$, and may only be negative if $X_{j,t-1} > 0$. That is, the system can only add the active item, and can only evict items that are currently in the cache.

Let $A \in \{0,1\}^{V \times T}$ indicate whether a downstream operator influences the state of the cache. That is,

$$A_{j,t} = \begin{cases} 1 & \text{if } j \text{ is on the active path between} \\ & \quad i_t \text{ and the sink} \\ 0 & \text{otherwise} \end{cases}$$

Specifically, $A_{j,t} = 1$ for the active node and all of its successors along the active path.

We now formally define our caching problem of interest:

Definition 1. (Cache-Dependent Generalized Paging) *The goal is to find the cache policy, Δ, that minimizes the following expression:*

$$
\begin{aligned}
\underset{X}{\text{minimize}} \quad & \sum_{t \in T} c_t \max(0, 1 - X_t^\top A_t) \\
\text{subject to} \quad & \Delta_{j,0} = 0 \quad \forall j \in V, \\
& \Delta_{j,t} \leq 0 \quad \forall j \neq i_t, \\
& \Delta_{i_t,t} \geq 0, \\
& -1 \leq \Delta_{j,t} \leq 1 \quad \forall j, t, \\
& \Delta_{j,t} \in \mathbb{Z} \quad \forall j, t, \\
& X_{j,t} = \sum_{k=1}^{t} \Delta_{j,k} \quad \forall j, \\
& 0 \leq X_{j,t} \leq 1 \quad \forall j, t, \\
& 0 \leq \sum_{i=1}^{n} X_{i,t} m_i \leq M, \quad \forall t
\end{aligned}
$$

That is, the goal is to minimize the total runtime of the DAG where the runtime is dependent on the structure of the graph (A_t) and the state of the cache at each point in time (X_t). The cost of evaluating a node is included in the runtime if it is not in cache and must be evaluated to reach the sink ($\max(0, 1 - X_t^\top A_t) > 0$). The first four constraints on Δ correspond to the following requirements: (1) the cache must be initially empty; (2) only the active node may be added to cache; (3) only non-active nodes may be removed from the cache; and (4) Δ is bounded above and below by 1 and -1. The fifth constraint requires Δ to take on integral values, making this a mixed-integer linear program, as opposed to a constrained linear program. Next, the equality constraint simply says that X is the cumulative sum of the *changes* to the cache up to time t. The final constraint requires that at any time t, the total size of the objects in cache must be below a positive real number, M.

The objective function is convex in X. This can be seen as follows: 0 is convex, $1 - X_t^\top A_t$ is convex, and the max of any two convex functions is convex, as is the same function scaled by a non-negative number. Finally, the sum of convex functions is also convex. Furthermore, each of the constraints on the program are either linear or equality constraints, with the exception of one integral constraint. Hence, this program is a mixed integer linear program (ILP).

Note that extending this formulation to account for a scenario where evicted items can be saved to disk and reloaded later is fairly simple. First, we should save a result to disk if reading from disk is cheaper than recomputing from the root node. If it is cheaper to read from disk, we modify the cost c_t of computing a node after the first time it is seen to be the cost of reading from disk.

It is well known that solving ILPs is NP-hard [Papadimitriou and Steiglitz, 1998]. Nonetheless, in Section A.3.1, we investigate when it is feasible to solve the ILP to recover the optimal policy. Then, equipped with this information, we use the optimal cache policy to inform the selection of an online cache strategy with smaller overhead when executed on larger DAGs.

A.3.1 Limitations on Solving the ILP

We evaluate the runtime required to solve the ILP on synthetic DAGs designed to look like common hyperparameter tuning pipelines in terms of size and computational cost. We consider DAGs of k-ary trees with varying depth d and branching factor k; this is akin to generated DAGs using gridded random search (Section 2.1) on pipelines of length d for which there are k choices of hyperparameter values at each node. We use Gurobi Gurobi Optimization [2015], called via the cvxpy package, to solve the cache-dependent generalized paging problem for each synthetic DAG.

The empirical runtimes of solving for the optimal caching strategy for different trees are shown in Table 3. While by no means exhaustive, this table provides guidance for when solving the ILP may be worthwhile. The results show that in under 5 minutes one can effectively compute an optimal cache policy for 10s of pipelines.

Table 3: Time required to solve the ILP for perfect k-ary trees of depth d. For each k and d, p counts the number of total pipelines, n the number of nodes, and T the length of the execution plan. As in (Definition 1), X represents the state of the cache. Observations with an 'x' timed out after 5 minutes.

k	d	p	n	T	Size of X	Runtime (s)
2	2	4	7	12	84	0.5
2	3	8	15	32	480	1.3
2	4	16	31	80	2480	3.8
2	5	32	63	192	12096	12.9
3	2	9	13	27	351	3.5
3	3	27	40	108	4320	70.9
3	4	81	121	405	49005	x
3	5	243	364	1458	530712	x
4	2	16	21	48	1008	27.3
4	3	64	85	256	21760	x
4	4	256	341	1280	436480	x
4	5	1024	1365	6144	8386560	x

A.4 Details for Experiments in Section 4

We constructed a simulation framework to study the effectiveness of different cache policies at achieving maximal reuse using pipeline-aware hyperparameter tuning. Our framework estimates the time needed to execute a fully merged hyperparameter pipeline DAG under a given cache policy for different memory sizes. The pipelines enter the simulator as DAGs with all of the necessary metadata required to estimate the execution time for each cache policy. Specifically, the nodes in the DAG are populated with information about the local execution time, memory footprint of each node's output, and number of iterations associated with each node using profiled statistics on sample workloads. These experiments were run using KeystoneML[4] [Sparks et al., 2017] with Scala and Apache Spark.

The search space considered for the 20 Newsgroups pipeline is shown in Figure 7; that for Amazon Reviews is shown in Table 8; and that for TIMIT is shown in Table 9. We used Apache Spark to profile the computational cost and memory requirement of different pipelines. We then created DAGs from these profiles to evaluate the different caching strategies. In order to generate the sub DAGs for Successive Halving, we assumed that training time scales linearly with the dataset size for LBFGS.

[4]`keystone-ml.org`

Figure 7: Search space for Newsgroups workload.

Name	Type	Values
nGrams	Integer	(2,4)
Top Features	Integer	(10^3, 10^5)
Naive Bayes λ	Continuous (log)	(0, 10^4)

Figure 8: Search space for Amazon Reviews workload.

Name	Type	Values
nGrams	Integer	(2,4)
Top Features	Integer	(10^4, 10^6)
λ	Continuous (log)	(10^{-5}, 10^5)

Figure 9: Search space for TIMIT workload.

Name	Type	Values
γ	Continuous (log)	$(5.5 \times 10^-4, 5.5 \times 10^4)$
Distribution	Discrete	{Cauchy, Gaussian}
λ	Continuous (log)	(0, 10^5)