The Case for Unifying Data Loading in Machine Learning Clusters

Aarati Kakaraparthy[*†], Abhay Venkatesh[*], Amar Phanishayee[‡], Shivaram Venkataraman[*]
[*]*University of Wisconsin, Madison* [†]*Microsoft Gray Systems Lab, Madison*
[‡]*Microsoft Research, Redmond*

Abstract

Training machine learning models involves iteratively fetching and pre-processing batches of data. Conventionally, popular ML frameworks implement data loading within a job and focus on improving the performance of a single job. However, such an approach is inefficient in shared clusters where multiple training jobs are likely to be accessing the same data and duplicating operations. To illustrate this, we present a case study which reveals that for hyper-parameter tuning experiments, we can reduce up to 89% I/O and 97% pre-processing redundancy.

Based on this observation, we make the case for unifying data loading in machine learning clusters by bringing the isolated data loading systems together into a single system. Such a system architecture can remove the aforementioned redundancies that arise due to the isolation of data loading in each job. We introduce *OneAccess*, a unified data access layer and present a prototype implementation that shows a 47.3% improvement in I/O cost when sharing data across jobs. Finally we discuss open research challenges in designing and developing a unified data loading layer that can run across frameworks on shared multi-tenant clusters, including how to handle distributed data access, support diverse sampling schemes, and exploit new storage media.

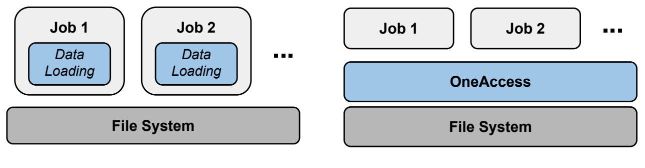
1 Introduction

With the widespread success of large scale machine learning for applications ranging from machine translation, image recognition to robotics, software frameworks like Tensorflow and Pytorch are being rapidly adopted by enterprises. Efficiently executing training jobs on shared enterprise clusters is thus an important requirement for developer productivity and resource utilization. Correspondingly, a number of previous efforts have focused on lowering the time spent in computation and communication [1, 4].

In this paper, we study the role of data loading in machine learning frameworks and characterize how the data access patterns and data pre-processing steps contribute to overall performance. In existing machine learning clusters, we find that distributed storage systems typically store raw input files, and data loading and pre-processing is performed independently by each job that runs on the cluster. While this approach is suitable for standalone clusters running individual training workloads, there are a number of opportunities for improving data access in shared, multi-tenant clusters based on the specific properties of machine learning workloads:

- **Temporally co-located jobs**: We find that a number of machine learning jobs are generated by parameter tuning experiments [11, 17]. These jobs often share the same input files and are spawned to run in parallel. Our analysis of a workload trace from Microsoft (§2.1) shows that up to 20 concurrent jobs are spawned by 40% of the experiments. On average we find that we can reduce up to 89% of I/O and 97% of pre-processing by unifying data loading.
- **Data pre-processing overheads:** Further, a number of machine learning models perform pre-processing on the input data to generate representations that are suitable for model training. Examples of this include cropping images or tokenizing text data [16]. Reusing pre-processed data could especially be effective for temporally co-located jobs.
- **Random data access patterns**: Finally, machine learning algorithms are typically iterative and at each iteration sample a random subset of the input data. Frameworks usually support a number of sampling methods including sampling at every iteration or shuffling the data at end of every epoch. This provides an opportunity to convert random access into sequential access by considering the sampling method.

Based on these opportunities, we propose developing a unified data loading layer for machine learning clusters. As shown in Figure 1, we propose, *OneAccess*, a new data loading layer that can support multiple machine learning jobs and



(a) (b)

Figure 1: **OneAccess: A unified data loading layer for machine learning. In (a), we show the design of current systems with standalone data loading; we're proposing a dedicated data loading layer in (b).**

amortize the random sampling and data pre-processing across jobs. We present the design of *OneAccess* and using a prototype implementation, we find that OneAccess can improve data access I/O time by 47.3% while running two jobs in parallel.

There are a number of open challenges in building such a unified data loading layer and we present some initial ideas on extending our system to span across multiple machines, supporting additional random sampling methods, and managing lifetime of samples generated.

2 Background

In this section, we first motivate our study by analyzing concurrent jobs that read same input data (§2.1). Following that, we describe the two main functions of data loading in machine learning frameworks (§2.2) and discuss how these functions are implemented in widely used ML frameworks (§2.3).

2.1 Concurrent Jobs in the Cloud

To help us understand the importance of unifying data loading, we attempt to quantify how many concurrent jobs access the same dataset in a shared cluster. One class of concurrent jobs comes from hyperparameter[1] tuning workloads, which involve exploring the model configuration space to find the best fit for a given problem. Hyperparameter tuning is an important step in building machine learning models, and data scientists typically use frameworks like HyperDrive [17] or Hyperband [11] to submit parameter tuning experiments.

We conduct a study of the experiments performed on HyperDrive over a period of 30 days. Each hyperparameter tuning experiment consists of multiple jobs, each of which performs training on a different configuration of the model independently. We calculate the total number of jobs launched by each experiment and also compute how many of these jobs are run concurrently on a shared cluster. We observe that on an average, each experiment has about 35 jobs (Fig. 2a). Although each experiment has multiple jobs, not all of them are launched simultaneously. We find that on an average, about 9 jobs are run concurrently (Fig. 2b).

All the jobs launched for an experiment run on the same dataset and perform the same steps of fetching and pre-processing data, thus performing repetitive computation and redundant I/O calls. This study reveals the potential for saving $(1-1/9) = 89\%$ of I/O calls if concurrent jobs share data during an epoch, and $(1-1/35) = 97\%$ of the pre-processing computation performed if pre-processed data is persisted across jobs. Given the significant benefits that can be realized from unifying data loading, we next discuss the two main steps in loading data in machine learning frameworks.

2.2 Data Loading for ML: Two Pieces

Machine learning models are typically trained in an iterative manner [10] on batches of data over multiple epochs (Fig. 3). The data loading module in a framework is responsible for generating batches, which are used for computing gradients and updating the model. On taking a closer look, there are two steps that need to be performed for generating batches.

The first step is *fetching data* from the underlying storage, which imposes a fundamental tradeoff between true randomness and the timely availability of data. As shown by previous work [13, 19], the ordering of data points has a significant impact on the training process, with random sampling across the entire dataset having superior convergence guarantees. However, this requirement inherently contradicts with the behavior of persistent storage devices, where sequential accesses are much more desirable [2, 18]. Randomly accessing persistent storage is significantly slower than computation, and can potentially become a bottleneck. In distributed cloud settings, fetching data incurs an additional (non-uniform) latency of network transfer time, as the data can be spread across multiple machines. Thus, performing random access across the entire dataset becomes extremely challenging in the cloud.

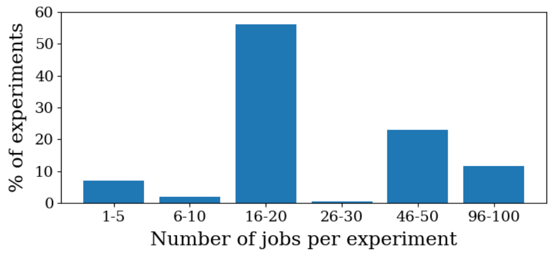
Once the data has been fetched, the next step is *pre-processing* the data. In order to obtain a standard input format and also robustly train machine learning models, data scientists often apply operations such as cropping images, removing stop words from text, etc., on the data. This computation is fairly deterministic and repetitive across epochs, unless pre-processed data is persisted in some way.

In summary, optimizing data subsystems in machine learning frameworks requires studying data access patterns to persistent storage, and the computation required in pre-processing.

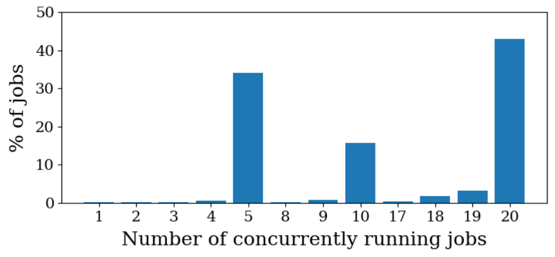
2.3 Existing ML Frameworks

We study two popular ML frameworks, with particular focus on how data loading and preprocessing is performed. Based on this we determine the features that need to be supported by a unified data loading layer.

[1]A hyperparameter defines the configuration of the model, and does not change during training. For example, the learning rate, momentum, number of hidden layers, etc. are hyperparameters found in many deep learning models.



(a) **Distribution of number of jobs launched per experiment. We find that most experiments have 20, 50, and 100 jobs. On an average, each experiment has about 35 jobs.**



(b) **Distribution of number of jobs launched concurrently. We observe that HyperDrive frequently launches 5, 10, or 20 jobs simultaneously. On an average, about 9 jobs are run concurrently for an experiment.**

Figure 2: **Experiments running on HyperDrive over 30 days. We measure the number of jobs launched by each experiment, and how many of them run concurrently.**

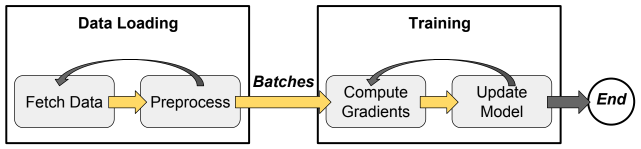


Figure 3: **Iterative training of a model. The data loading module is responsible for fetching and preprocessing data, to generate batches used in training.**

Number of Workers	**Compute(s)**	**I/O(s)**	**Total(s)**
1	416s	443s	521s
2	309s	250s, 248s	310s
4	309s	124.7s, 125.2s, 125.3s, 125.2s	309.7s

Table 1: **Breakdown of time spent by I/O threads vs. compute when varying number of workers in PyTorch running Resnet-18 on MS-COCO dataset. I/O bottleneck for a single job is typically alleviated by using multiple worker threads.**

2.3.1 PyTorch

PyTorch [15] provides a base data loading module which needs to be extended for different datasets, as each of them can potentially have different storage formats. Data points are accessed one at a time using the *__getitem__* interface, which is overwritten for each dataset. Preprocessing is performed individually for every data point before it is returned, and there is no built-in support for buffering pre-processed data.

The framework provides multiple sampling schemes (such as sequential, globally random, random within a subset), that determine the order in which points are accessed. However, the efficiency with which data is fetched is not optimized by the framework, and depends on the implementation of the data loading module for a given dataset.

A noteworthy technique incorporated into the base data loading module of PyTorch, is the utilization of multiprocessing to keep the operation of the data loading module independent from training. Multiple *worker* processes can be launched to fetch data in parallel, and repeatedly populate shared memory queues with batches that are used for training. An example of how this helps is shown in Table 1.

2.3.2 Tensorflow

Similar to PyTorch (§2.3.1), the data sub-system of Tensorflow [1] runs independently from the training process. However, Tensorflow provides a richer API for fetching and iterating over datasets. Multiple storage formats and sources of data are supported (e.g., images, text) by the framework, the most noteworthy being the *TFRecord* format which stores data linearly in a 100-200MB file for efficient access.

By default, data points in a dataset are accessed one at a time in a sequential order. However, applying a *shuffle* operation to the dataset provides some randomness by maintaining a fixed size buffer where data points are enqued sequentially and dequed in a random order. This approach leads to random sampling within a window, and is not random over all the points in the dataset. The API also provides the option to *prefetch* data points into a fixed size buffer for faster access.

Preprocessing is performed individually on each data point through the *map* operation. There are no built-in techniques for caching preprocessed data, and the map function on each point is repeated over epochs. However, preprocessed data can be optionally serialized and persisted as *TFRecord* files, in order to avoid repetitive computation.

In summary, both frameworks support a number of sampling schemes but they do not amortize data pre-processing, and do not optimize for concurrent jobs accessing the same data. While our prior discussion on concurrent jobs focused on jobs using the same framework, we also observe that we can amortize data loading for *jobs across different frameworks*.

3 Unified Data Access with OneAccess

OneAccess is an attempt towards building a unified data system for machine learning frameworks. Along with optimizing the process of fetching and preprocessing data, the framework also adds support for multiple ML training jobs running simultaneously sharing the same data. Fig. 4 shows the architecture of OneAccess on a single machine.

OneAccess optimizes the process of fetching data through reservoir sampling [20]. Reservoir sampling is a technique to generate uniformly random samples of data, while accessing the data sequentially. Thus, to realize the benefits of reservoir sampling, as well as to avoid repetitive computation, pre-processed data is serialized and stored in files on disk. There are two major sub-tasks performed independently by two types of processes.

First, sample creators running at each storage device, perform reservoir sampling on the data/samples from the layer below, and generate more compact samples to be stored in the layer above (sample creator 1 & 2 in Fig. 1). Second, the batch creator process generates batches of data from the reservoir samples in memory; batches are then consumed by multiple ML training processes. All of these processes run independently and replenish batches/samples once they are depleted. Given this design, the noteworthy features that OneAccess realizes are:

- **Support for multiple jobs**: OneAccess adds support for multiple ML training jobs running on the same data, so that steps related to fetching and preprocessing data are not duplicated. This is especially relevant in the cloud settings, where multiple concurrent programs could be replicating the same operations (§2.1).
- **Sequential storage of pre-processed data**: Sequential accesses have significantly better performance when it comes to persistent storage devices. In OneAccess, we pre-process the data and store it sequentially in files on disk, which we refer to as the *serialized intermediate data*. Persisting pre-processed data eliminates repetitive computation.
- **Uniform randomness while performing sequential accesses**: OneAccess recursively performs reservoir sampling[2] across the memory hierarchy to fetch compact and uniformly random samples into memory. Through this approach, only sequential accesses are performed on persistent storage devices while creating samples, and all random accesses during batch creation are restricted to main memory.

4 Preliminary Results

All the experiments in this section have been run on a single machine with a two-level storage hierarchy comprised of Samsung 960 EVO NVMe 500GB SSD, and a main memory of size 32GB. We benchmark and compare the time taken for generating batches by both the frameworks (§4.1), and also measure the total I/O time saved when multiple concurrent processes use the same data system (§4.2).

[2]All points have equal probability of being present in the resulting reservoir sample. Additionally, it can also be proved that all subsets of data (of reservoir sample size) have equal probability of being the final sample.

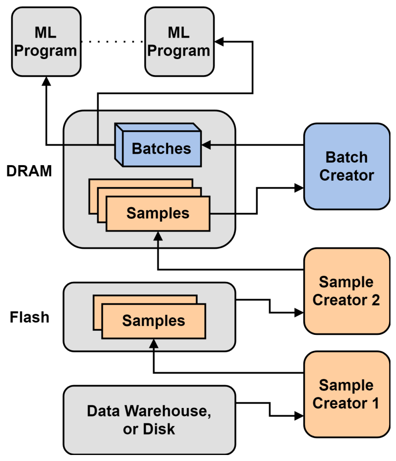


Figure 4: **The architecture of OneAccess. The Sample Creator1 process generates reservoir samples from data on disk, and stores them on SSD. These samples are consumed by Sample Creator 2, which creates smaller reservoir samples held in memory. The Batch Creator process consumes the samples in DRAM to generate batches used by multiple jobs.**

Framework	Total Time (min)
Pytorch (with 1 worker)	23
PyTorch (with 2 workers)	12
PyTorch (with 4 workers)	6.8
OneAccess (sample size 400)	6.4

Table 2: **Benchmarking batch creation time for MS-COCO. We measure the time taken to generate batches of size 32 for one epoch. OneAccess is 3.6x and 1.9x faster compared to PyTorch with 1 and 2 workers respectively.**

4.1 Benchmarking Batch Creation

We compare OneAccess against the built-in dataloader of PyTorch for the MS-COCO Detection dataset [12]. This dataset has 118K images of variable sizes, which need to be cropped to size 224×224. We measure the total time taken by both the frameworks to generate batches of 32 images during one epoch over the dataset.

Prior to creating batches, OneAccess generates a serialized intermediate data representation (§3) which results in a one-time initial overhead of 24 minutes. The serialized representation for MS-COCO consists of nine files each around 2GB in size, consisting of about 13K cropped images.

Table 2 shows the total time taken for creating batches over one epoch by both PyTorch and OneAccess. PyTorch was run with 1, 2, and 4 workers, whereas OneAccess has a single sample creator process alongside the batch creator (as we have a two-level storage hierarchy). The sample creator process repeatedly fills in an in-memory reservoir of size

Training Jobs/ Configuration	**I/O time (s)**	
	Job 1	*Job 2*
Separate Data Access	14.3	15.1
Shared Data Access	$15.4/2$	$15.4/2$

Table 3: **Measuring the reduction in I/O time when using a shared instance of OneAccess (Fig. 1b) for two independent training jobs running on the CIFAR-10 dataset. When using a shared instance, the total I/O time is amortized between the jobs. We obtain an overall reduction of 47.3% against using separate instances of OneAccess.**

12.8K images (400 batches), which are then consumed by the batch creator. Random sampling over entire dataset is used by both the frameworks in this experiment.

We find that OneAccess is 3.6X and 1.9X faster compared to PyTorch with 1 and 2 workers respectively (Table 2). This speed-up is the result of performing sequential accesses and persisting pre-processed data in a serialized manner. Although OneAccess has an initial overhead for generating the serialized intermediate representation, it is evident that this will be adequately amortized over a few epochs.

To test a distributed setup, We repeated the same experiment with two VMs on Amazon EC2 (m5d.2xlarge and p2.xlarge) running a NFS server and client. We found that OneAccess is around 1.7x faster than Pytorch with 1 worker and 1.1X slower than Pytorch with 2 workers. We found that the ratio of sequential to random SSD throughput resulted in lower benefits when running on AWS. We also plan to extend OneAccess to use multiple worker threads in the future.

4.2 Supporting Multiple Jobs

To evaluate the benefits of using a common data system, we run two independent training jobs in PyTorch, first with the conventional approach of using separate instances of OneAccess (similar to Fig. 1a), and compare the total I/O time to that of a shared setup using a common instance of OneAccess (similar to Fig. 1b). Both the jobs independently train a convolutional neural network on the CIFAR-10 dataset [9]. OneAccess was configured to run with a single in-memory reservoir size of 3.2K images (100 batches), with a serialized intermediate representation of CIFAR-10 created beforehand.

In this case we observe that the total training time remains the same but we observe a significant reduction in the total I/O time (Table 3). The training time is unaffected as training is performed by a separate CPU thread. The smaller image sizes of CIFAR-10 (32x32) and use of CPU for training mean that the storage access time is smaller than time taken for compute. However, we do see that I/O is performed by the common instance of OneAccess and the overhead is amortized between the two processes. As expected, we find that the total I/O time reduces by 47.3% (the theoretical best would be 50%), showing the potential of our system for eliminating redundancy in storage access. We discuss more about data loading vs compute bottlenecks in the topics for discussion.

5 Future Work

There are a number of research challenges in building a machine learning data system. We next outline some of these challenges and propose some initial ideas to address them.

Sampling schemes, convergence: Our prototype implementation of OneAccess has been focused on supporting random sampling with replacement over the entire dataset. However, recent theoretical research shows that other sampling schemes could result in faster convergence for algorithms like SGD [13] and coordinate descent [7]. Thus it is important for the data loading layer to be flexible and support custom sampling schemes. We plan to investigate creating a developer API that can be used to install new sampling schemes that can be used across framweorks.

Distributed Data Access Systems: Our results presented in §4.2 assume that OneAccess has a centralized view of the data being accessed by different jobs. In order to build a unified data loading system for a large cluster, we plan to split our design into a control plane that monitors which samples are created and accessed by jobs, and a data plane that performs the sampling operation and serves batches to workers. For performance efficiency we would also need to investigate how to integrate our system with underlying distributed file systems, blob stores, databases etc.

New Storage Media: The performance and sample sizes chosen by the sample creator module in OneAccess is determined by the type of media that is used to store data. With the widespread availability of NVMe based SSDs and introduction of Intel Optane drives [8], we plan to investigate how sample creation can automatically be tuned based on the device characteristics.

Sample lifecycle: While the creation of samples in the background and caching pre-processed data can improve performance, they also lead to increase in storage requirements. We plan to develop techniques to automatically manage the lifecyle of samples created by coordinating with the cluster scheduler and plan to build upon prior work in data provenance [5], and derived dataset management [6].

6 Conclusion

In conclusion, we make the case for unifiying data loading across machine learning frameworks deployed in shared clusters. We observe the ubiquity of machine learning experiments with a high number of concurrent jobs, and our analysis shows the potential for 89% of I/O operations and 97% of preprocessing computation that can be saved by unifying data loading across jobs. We propose a novel unified data loading system called *OneAccess* that serves as a first step towards achieving such savings. Our results using *OneAccess* running on CIFAR-10 that show an overall reduction of 47.3% in total I/O time of two concurrent jobs, thus showing the potential of a unified system.

Discussion Topics

This paper is likely to generate a discussion regarding the advantages and disadvantages of using a unified data loading system for machine learning jobs in the cloud. The main issues that we believe will generate discussion are

Data loading vs. compute bottlenecks Pipelining data loading with computing of gradients means that the slower of the two steps will determine the overall latency of running an iteration. Trends in how storage media is getting faster vs. availability of more compute resources will make this an interesting discussion point. Further it will be interesting to discuss other benefits of avoiding I/O requests in terms of power savings or sharing resources with other datacenter applications.

Unifying data pre-processing across frameworks. While saving I/O costs and pre-processings costs for jobs using the same framework is relatively straightforward, our proposal of sharing pre-processing across frameworks will likely generate discussion on how we can handle different languages, and if a format like Apache Arrow can be used to achieve this.

Synchronizing data access across jobs. A potential challenge for training ML systems in the wild might be synchronizing data access across jobs. If the hyperparameter variation leads to a variation in training duration, and if we are synchronously loading data then jobs could be blocked on the slowest job. Some preliminary approaches to avoid this would be to relax sampling guarantees or have an in-memory cache that can handle requests from slower jobs.

Importance of locality. OneAccess is most applicable to a cluster computing scenario as that is where we can find the most redundancy in job execution. However, there might be challenges in implementing random sampling methods due to additional latency when fetching data across a number of machines. While prior work [3, 14] has shown that locality concerns can be overcome for big data analytics and filesystems, it remains to be seen how locality will affect random sampling.

Acknowledgements

We would like to thank the reviewers for their valuable feedback. This work is supported by the National Science Foundation (CNS-1838733). Shivaram Venkataraman is also supported by a Facebook faculty research award and support for this research was also provided by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation.

References

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

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, volume 13, pages 12–12, 2011.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[5] MIchael Chow, Kaushik Veeraraghavan, Michael Cafarella, and Jason Flinn. Dqbarge: Improving data-quality tradeoffs in large-scale internet services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 771–786, Savannah, GA, 2016. USENIX Association.

[6] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[7] Mert Gurbuzbalaban, Asuman Ozdaglar, Nuri Denizcan Vanli, and Stephen J Wright. Randomness and permutations in coordinate descent methods. *arXiv preprint arXiv:1803.08200*, 2018.

[8] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[9] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[12] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[13] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 2019.

[14] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 1–15, 2012.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[16] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[17] Jeff Rasley, Yuxiong He, Olatunji Ruwase, and Feng Yan. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, December 2017.

[18] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[19] Stephen Tu, Shivaram Venkataraman, Ashia C Wilson, Alex Gittens, Michael I Jordan, and Benjamin Recht. Breaking locality accelerates block gauss-seidel. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3482–3491. JMLR. org, 2017.

[20] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.