An Overview of the Data-Loader Landscape: 
Comparative Performance Analysis

Iason Ofeidis *
Yale University
New Haven, CT, USA

Diego Kiedanski *
Tryolabs
Montevideo, Uruguay

Leandros Tassiulas
Yale University
New Haven, CT, USA

Abstract—The efficiency of Deep Learning (DL) training jobs is critically dependent on dataloaders, which facilitate the transfer of data from storage to DL-accelerated hardware during training. Recent advancements in data loading technology have demonstrated significant improvements, not only in reducing training times but also in introducing capabilities such as seamless integration with cloud storage. This paper examines the dataloader as a distinct component within the DL workflow, offering a detailed analysis of its structure and functionalities. We present a systematic evaluation of various dataloading libraries, investigating their performance across different configurations, including worker count, batch size, GPU scaling, data access patterns and remote loading. The evaluation highlights trade-offs in functionality, usability, and performance. Additionally, we examine the impact of dataset characteristics on data loading performance, showing that throughput decreases exponentially with image resolution. To support ongoing research and practical advancements, we introduce the first open-source benchmarking suite for DL data loading, which allows the community to replicate, extend, and build upon our experiments.

Index Terms—Machine Learning, Benchmarks, Data-Loading, Performance, Cloud Storage, Open-Source

I. INTRODUCTION

Building effective Deep Learning (DL) systems requires careful selection and design of several key aspects, typically including the dataset, the model architecture, the underlying hardware, as well as scalability efficiency. The recent prevalence of deep learning-driven applications has led to breakthroughs for them, such as increased dataset sizes [1] [2], novel model architectures [3], improved scalability [4] and custom hardware accelerators [5]. While the time needed for training and inference computations can be drastically shortened by these advances, achieving peak end-to-end training performance also requires an efficient data input pipeline. Notably, research efforts have consistently demonstrated that data loading is one of the most significant bottlenecks in DL pipelines [6], [7].

In DL frameworks, such as PyTorch [8] and TensorFlow [9], the data input pipeline is managed by the component known as dataloader. Dataloaders are responsible for retrieving data from its storage location—whether in RAM, on disk, or across a network—performing necessary transformations, and efficiently transferring the processed data to the appropriate computing device for model consumption. Given the critical role of the data input pipeline in the overall performance of DL systems, the data loading process has become a focal point of optimization efforts.

Consequently, a range of specialized libraries and research initiatives have been developed to integrate with existing DL frameworks, each dedicated to enhancing the efficiency and scalability of data loading operations. For instance, FFCV [10], a recently developed open-source library, has demonstrated the capability to train the ImageNet [11] dataset in significantly reduced time compared to the default PyTorch dataloader. Such gains can dramatically reduce the operational costs of companies and research teams that depend on Infrastructure as a Service (IaaS), such as Amazon Web Services (AWS) and Google Cloud Platform (GCP). Another promising feature offered by dataloaders is the ability to load data stored remotely; for example from cloud storage, such as AWS S3 buckets [12] or Azure blobs [13]. This capability, which has been available for some time, offers several practical advantages: it eliminates the need for local dataset setup, reduces the disk capacity required on the computing machine, and minimizes the risk of team members working with different versions of the dataset.

Despite their benefits, these advanced capabilities are not widely recognized or utilized within the community. Even with the availability of sophisticated features, determining the optimal configuration of basic dataloader parameters, such as the number of workers and batch size, remains challenging. The impact of these variables varies significantly across different libraries and datasets, leading to uncertainty about the most effective parameter combinations for maximizing efficiency. Additionally, the effect of specific dataset characteristics—such as image resolution and sample length—on library performance is not thoroughly explored. This gap in comprehensive understanding, coupled with the growing number of available libraries, highlights the urgent need for systematic benchmarking to identify best practices for configuring dataloaders across various scenarios.

Given the above, our paper aims to bridge this gap in the literature. To that end, our contributions are as follows:

- **Comprehensive Benchmarking**: We systematically evaluate the performance of data loading libraries for computer vision tasks across various configurations, including the number of workers, batch size, number of GPUs, model pass type, data access patterns, and remote data storage.
loading (e.g., from cloud storage). This analysis provides deep insights into how these factors influence data loading efficiency.

- **Impact of Dataset Characteristics**: We assess the effects of dataset size and image resolution on the performance of different data loading libraries, offering guidance for optimal configurations based on specific dataset characteristics.

- **Open-Source Benchmark Suite**: We offer the first open-source benchmarking suite for DL data loading that enables the community to replicate, extend, and update our experiments, promoting continued research and practical optimization in data loading.

II. RELATED WORK

This section describes several efforts in the community to analyze deep learning data loading, models, and frameworks.

**Data Loading Analysis.** Mohan et al. [6] offer a comprehensive analysis of the relationship between input data pipeline and training time for computer vision and audio DL Networks. They introduce both a tool to measure data stalls in DL training and CoorDL, a data loading library to mitigate them. Their major difference lies in how they use NVIDIA DALI, based on internal empirical evaluations, while in our work we systematically evaluate different data loading libraries across various configurations. Svogor et al. [14] share in a technical report how they designed a series of benchmarks that outline performance issues of certain steps in the data loading process for PyTorch for image classification tasks. There they introduce the ConcurrentDataloader that can reach improvements in GPU utilization and significantly reduce batch loading time. Yang et al. [15] study the data loading performance in large-scale distributed training, and based on their insights, they propose a locality-aware data loading method. However, in both of these projects there exists no comparison with different state-of-the-art data loading libraries besides PyTorch. Furthermore, both CoorDL and ConcurrentDataloader libraries, while being open-source, do not provide sufficient documentation for use by the community, thus could not be evaluated in our benchmarks. In Plumber [16], the authors’ analysis of over two million ML jobs in Google datacenters reveals that a significant fraction of model training jobs could benefit from faster input data pipelines. While they mention and utilize remote storage in their experiments, they do not investigate the tradeoff and the difference in performance between local and remote data source in DL training.

**Comparison of frameworks.** This section includes efforts toward benchmarking and comparing different deep learning frameworks. Deep500 [17] provides a modular framework for measuring deep learning performance but lacks hyperparameter benchmarking and ease of use for novel libraries. AIBench [18] and DAWNBench [19] offer end-to-end benchmarks but do not examine alternative loading libraries. Wu et al. [20] analyze CPU and memory usage patterns across parallel computing libraries, but this work lacks an open-source resource for new library benchmarking. Shi et al. [21] compare DL frameworks across various neural networks, while dPRO [22] and DLBench [23] focus on distributed training and performance across tools like Caffe and TensorFlow. Unlike these studies, our work uniquely benchmarks additional data loading libraries on top of PyTorch.

III. DATALOADERS

In the process of training a DL model, the dataset needs to be read from memory and pre-processed, before it can be passed as input to the model. While the performance of current Machine Learning (ML) accelerated hardware (TPUs, GPUs, etc.) is exceptional, this data pre-processing and loading becomes a potential bottleneck that leads to considerable training latency and added overhead for both CPU and memory, especially when datasets are too large to fit into memory [24]. DL libraries attempt to resolve this bottleneck by utilizing a structure called dataloader. This structure provides a way to iterate over the dataset by leveraging parallel processing, preprocessing, and other techniques to reduce data loading time and memory overhead as much as possible [8].

The main goal of a dataloader is to perform the actions responsible for transferring data samples from a storage location to the memory co-located with the processing units for training in order to form a batch of samples to be fed into the model. These actions are restrained by the storage system’s bandwidth and specifically its I/O performance. Thus, depending on the system’s hardware specifications, its filesystem serving it, and the throughput of the link with the computing units, it can have an immense influence on the total amount of time needed to complete the training.

When employing a dataloader, apart from providing the dataset for input, the user has the option to configure a number of hyperparameters, tailored to their needs and resources. A common one available in all dataloaders is the batch size (batch), which defines the number of samples that will be used before updating the internal model parameters. This parameter is intrinsically linked with the concept of mini-batch in Stochastic Gradient Descent (SGD) [25] and is one of the primary settings to be adjusted in order to tackle issues such as overfitting and long training time [26].

Dataloaders come with a component called workers (py_num_workers), whose purpose is to optimize this data-transferring process. Workers are defined as sub-processes responsible to carry out the data loading in an asynchronous fashion. When creating an instance of a dataloader, the user has the option to specify the number of workers that will be spawned and will be in control of this operation. If the number of workers is equal to zero, no sub-processes will be created, which, in turn, means that data fetching happens synchronously in the same (main) process and, thus, the

1https://github.com/smartnets/dataloader-benchmarks

2For brevity, we overview PyTorch’s torch.DataLoader and its TensorFlow counterpart, tf.data, in parenthesis.
computing units (GPU) have to wait for the data loading to be completed [27]. Reversely, there will be generated sub-processes equal to the number of workers, which will prevent the blocking of computation code with data loading. This is accomplished by pre-fetching future batches in advance to be ready when needed. However, utilizing more workers increases the memory usage, ultimately creating a serious overhead. Therefore, careful tuning of the number of workers is crucial to maximize throughput and maintain a balanced and efficient data processing workflow.

IV. Experimental Setup

A. Libraries

After extensive research, we identified a selection of both established and prominent libraries available for optimizing the data input pipeline in DL systems for computer vision tasks. While the field is rapidly evolving with new libraries emerging regularly, the following list represents a robust overview of current data loading capabilities.

Seven libraries were selected for the experiments conducted in this study. PyTorch [8], which stands as one of the most popular DL frameworks and was developed by Facebook AI Research Lab (FAIR). NVIDIA DALI [28] is a widely adopted, high-performance library for data pre-processing and augmentation in deep learning, known for its efficient GPU acceleration and integration with popular frameworks like PyTorch. Torchdata [29], a library of common modular data loading primitives for performant data pipelines, that aims to provide composable building blocks that work well out of the box with PyTorch’s default DataLoader. Deep Lake [30], developed by Activerse, is an open-source dataset format optimized for rapid streaming and querying of data while training models at scale that utilizes the Tensor Storage Format (TSF), which permits storing of unstructured data with all its metadata in deep learning-native columnar format, and thus, enables rapid data streaming. FFCV [10], as mentioned in Section I, is a drop-in data loading system that dramatically increases data throughput in model training, by making use of the .beton format and just-in-time (jit) compilation. Squirrel [31] enables ML teams to share, load, and transform data in a collaborative, flexible, and efficient way, by utilizing the MessagePack serializer, and also enables the chaining of iterables. Lastly, Webdataset [32] is a standards-based storage format and library that permits efficient access to very large datasets, as these are represented as standard POSIX tar files in which all the files that comprise a training sample are stored adjacent to each other in the tar archive.

B. Datasets, Models & Pre-processing

Regarding the datasets, we initially opted for two widely used datasets to support two different learning tasks: CIFAR-10 for image classification, and COCO for object detection. To investigate the impact of dataset characteristics on library performance, we constructed an additional image classification dataset named RANDOM_K_N, that consists of randomly generated color images of size $K \times K$ (pixels) and sample length $N$. A specific iteration of RANDOM, the RANDOM_45000_256, consists of images of size 256x256 pixels, contains 45,000 images for training and will be used for the majority of our experiments, unless otherwise stated. For the image classification task, we used the ResNet-18 [33] model, while the Faster R-CNN with ResNet-50 as the backbone [34] model was employed for the object detection task, where both models are taken from TorchVision [35].

The same transformations were applied across all libraries to ensure the benchmarks remained comparable. For image classification the transformation stack consisted of the following: Conversion to PIL format (where necessary), Random Horizontal Flip, Normalization and Transformation into Tensor. For object detection, we utilized Albumentations’ [36] implementation of transformations. The stack looked as follows: Random Sized Crop, Random Horizontal Flip, Normalization and Transformation into Tensor. These transformations apply to both, images and bounding boxes.

C. Training Environment & Metrics

We employ the following server configuration for our analysis. Server A is equipped with 4 NVIDIA RTX A5000 GPUs, 128 GB of RAM, an AMD EPYC 7352 CPU and 1TB NVMe SSD with 32 cores with Ubuntu 22.04 and CUDA 11.4.

All experiments were conducted using PyTorch 2.0.1 as the underlying deep learning framework, with the data loading component replaced by the corresponding dataloader from each benchmarked library. This choice was made because most of the data loading libraries evaluated do not yet support TensorFlow integration, ensuring a consistent and unbiased comparison across the libraries. Additionally, previous comparisons between PyTorch and TensorFlow end-to-end DL tasks already exist [37], and based on these findings, we expect the results presented here to extend to TensorFlow as well.

All experiments were conducted for two iterations of three epochs, and we report the average throughput in samples per second, calculated over the final two epochs. The first epoch, which begins with a cold cache, is excluded from the average as it serves as a warm-up phase to stabilize the system’s performance.

V. Numerical Results

A. Runtime Analysis

To gain a better understanding of how time is distributed across different stages of each dataloader, we conducted a detailed analysis of a single run (2 iterations of 3 epochs), examining the entire process from initialization to termination. We measured the time required to execute each batch and to initialize the dataloader. Figure 1 illustrates, for a specific set of parameters — RANDOM_45000_256 dataset, Server A, 1 worker, batch size 64 — the time distribution (median values). For clarity and brevity, the figure shows only the time distribution for the first 10 batches. Notably, the first batch requires significantly more time to process compared to subsequent batches. This can be
explained as follows: since mostdataloaders rely on lazy data loading at this point, future calls will benefit from pre-fetching, data already in memory, and parallelization. The size of the bands after the first batch provides the best indication of how well each library scales, as the time taken on a large dataset grows linearly with that width. While all libraries spend roughly the same amount of time on the first batch, their behavior diverges significantly beyond this point. This observation highlights that short initialization times do not necessarily reflect the overall throughput performance of the libraries. For example, although the Squirrel library has the shortest first batch time at 2.06 seconds, it ranks only second to last in overall throughput performance for this experiment. This discrepancy underscores that initial batch times alone are not sufficient to gauge a library’s efficiency and scalability. Most libraries exhibit a uniform width in terms of the remaining batches, with FFCV and NVIDIA DALI emerging as the two fastest overall.

**B. Workers & Batch Size**

As mentioned above, workers refer to the number of subprocesses used to load data in parallel, enabling faster data retrieval during model training by pre-fetching batches. We conducted an experiment where we varied the number of workers while keeping all other parameters constant, using the RANDOM_45000_256 dataset with a batch size of 64 on a single GPU. The results, presented in Figure 2, show that throughput increases steadily for all libraries as the number of workers increases up to 4. Beyond this point, however, the throughput begins to plateau, and in some cases, even slightly decreases, which can be explained by the increased communication overhead between workers, which leads to the main thread becoming overwhelmed. This indicates that selecting the optimal number of workers—not too high or too low—is essential for both utilizing efficiently the available resources, subprocesses in this case, and achieving the best possible throughput.

The relationship between batch size and GPU computational efficiency has been extensively explored in the literature [38]. Larger batch sizes more effectively leverage the underlying accelerator’s parallel processing capabilities and reduce the frequency of weight updates (and thus inter-GPU communication) per epoch, leading to accelerated training. However, a larger batch size means more data is processed simultaneously, which increases the memory needed to store intermediate activations, gradients, and model parameters during forward and backward passes. If the batch size is too large for the available accelerator memory, it can lead to out-of-memory (OOM) errors [39].

Figure 2 illustrates the impact of varying batch sizes on throughput for CIFAR-10, CoCo, and RANDOM_45000_256 datasets on Server A. For CIFAR-10, the throughput is consistently non-decreasing across all libraries, with significant differences between libraries becoming apparent as the batch size grows from 32 to 128.

For the RANDOM_45000_256 dataset, we conducted the same experiment with 1 worker on Server A and observed that, apart from FFCV, all libraries exhibited nearly identical performance across different batch sizes. Unlike the previous experiment with CIFAR-10, which involves relatively small images, this result suggests that when dealing with larger images (≥ 256 × 256 pixels) and only one worker, adjusting the batch size offers no significant throughput advantage. To further assess the libraries’ performance, we varied the number of workers. Most libraries, except FFCV, achieved their highest throughput with a batch size of 128, particularly when using 4 workers, which aligns with the findings from our worker variation experiment that a careful choice of number of workers is needed for optimal throughput. Additionally, FFCV’s shift from first to second place as the number of workers increases can be attributed to its typically constant memory usage [10], meaning it does not scale with the number of workers.

For the CoCo dataset, the libraries exhibit similar performance, with the notable exception of NVIDIA DALI, which consistently maintains high throughput across various batch sizes. This similarity in performance across libraries is expected, as in object detection tasks, the forward and backward pass computations dominate the overall runtime, whereas in image classification, data loading and pre-processing tend to play a more significant role. Additionally, varying the number of workers does not seem to yield significant gains for this computer vision task. The substantial variation in throughput is also noteworthy, ranging from as high as 50,000 samples per second to as low as 20 samples per second.

**C. Multi-GPU**

Given that the most intensive training jobs require the utilization of multiple GPUs, we conducted additional experiments with the RANDOM_45000_256 dataset on Server A, using a batch size of 64 and 1 worker (on each GPU), to assess the scaling performance in a multi-GPU environment. We implemented the experiments using PyTorch’s Distributed Data Parallel (DDP) [40], which is the recommended approach for multi-GPU training on a single node and is widely supported and documented by most libraries.

We evaluated the performance of the libraries by comparing their speedups when using 1, 2, and 4 GPUs. Figure 3 presents...
the results, with each bar showing the speedup relative to the 1-GPU baseline to emphasize scaling efficiency, while also stating the corresponding throughput on top of the bars. Most libraries demonstrate significant performance improvements with increased GPU usage, with Deep Lake approaching near-linear scaling—showing increases of 98.8% from 1 to 2 GPUs and 90.8% from 2 to 4 GPUs. Conversely, Torchdata exhibits limited scalability, with only modest throughput gains. Notably, FFCV stands out with exceptional performance, achieving a throughput of 3134.08 samples per second when using 4 GPUs. Lastly, as shown in the plot, the Squirrel library experienced issues with 4 GPUs, while it performed as expected with 1 and 2 GPUs.

D. Effect of Forward/Backward Passes on Performance

When evaluating data loading performance, a key consideration is whether to include the time taken by the forward and backward passes in the speed calculation. On the one hand, including these passes provides a more comprehensive measure of the total training time, reflecting the algorithm’s actual runtime. However, this approach can mask the differences between libraries, especially when the time spent on model operations overshadows data loading. Additionally, some libraries may leverage prefetching, optimizing data loading to overlap with computation, which could be overlooked if the focus remains solely on model training time.

To assess the impact of our decision to ultimately include the forward and backward passes in our measurement algorithm, we used the RANDOM_45000_256 dataset (64 batch size, 1 worker, single GPU) to compare the average speed with and without including the model operations. The results, shown in Figure 4, indicate that most libraries exhibit a slight increase in speed when excluding the model operations, except for FFCV, where performance drops significantly. Importantly, the relative performance ranking among libraries remains consistent, highlighting that prefetching strategies, where applicable, are indeed captured in our analysis.

E. Filtering

For certain DL projects, users may need to access only a subset of a larger dataset, whether for inspecting specific examples or focusing training on a particular subset. For those cases, having the ability to quickly filter the required data points without having to iterate over the whole dataset can drastically reduce the total training time. Some libraries allow filtering based on certain features, such as the class (for image classification tasks). Fast filtering is not necessarily trivial to implement as it requires an index-like additional structure to be maintained to avoid iterating over all samples.

We investigated the impact on speed when using the library’s filtering method, if available, versus not filtering at all. Whenever the library did not offer a filtering method, we implemented them naively, i.e., scanning the whole dataset and keeping only those elements that match the specified condition. For our filtering experiments, we selected two out of twenty classes from the RANDOM_45000_256 dataset (64 batch size, 1 worker, single GPU). The results are illustrated in Figure 5.

We observed that most libraries do not have a good filtering mechanism that avoids iterating over the whole dataset. For example, the PyTorch filtering implementation requires building a custom sampler with the indices of the desired images, which is done upfront, considerably affecting the total running time. In terms of throughput, both Deep Lake and PyTorch
achieve minimal loss in their performance, while TorchData and Webdataset exhibit dramatic speed decrease.

F. Remote

In an ideal scenario, dataset storage would be decoupled from the DL training process, allowing users to connect their database to the DL framework of choice, regardless of the location of these components. However, transferring training data over a network introduces significant latency, possibly leading to substantial reductions in speed. Given the high costs associated with utilizing cloud-based DL hardware, time efficiency becomes critical, and the trade-off between convenience and performance might seem unfavorable.

Nonetheless, three libraries evaluated in this study—WebDataset, Squirrel, and Deep Lake—offer the ability to specify datasets accessible via an internet connection, with claims of optimized performance for such scenarios. This advancement makes it feasible to assess the balance between ease of use and the impact on runtime, providing insight into whether the added convenience is justified despite potential slowdowns.

The following experiment was arranged to offer some insight into this question. We ran two iterations of two full epochs of the RANDOM_45000_256 dataset (128 batch size, 32 workers, single GPU) for these three libraries, while changing the origin of the data. Two locations were considered: a local copy in the machine running the experiments’ hard drive and a copy in an S3 bucket (in the closest region to our server). Regarding latency, the Round Trip Time from the server running the experiments to the S3-bucket in AWS servers was $11.8 \pm 0.2\text{ms}$ (min. 11.6ms, max. 14.9ms).

Figure 5 depicts the total running times for the experiment, while the percentages denote the slowdown (increase in running time) compared to the local case. We can observe that even though for Squirrel and Webdataset there is a significant increase when moving to AWS, Deep Lake managed to maintain almost the same speed with an increase of less than 50%.

Additionally, it is important to highlight that the number of workers plays a crucial role in remote data loading scenario. In this context, maximizing the number of workers is essential, as it significantly reduces the overall processing time. For instance, utilizing 16 workers led to a slowdown of 180%, 475%, and 491% for Deep Lake, Squirrel, and WebDataset, respectively—substantially worse than the results shown in

Figure 5. This difference from the findings in Section V-B suggests that, in remote data loading, workers also undertake the additional burden of downloading samples locally, which can severely impact performance.

G. Relationship with Image Resolution

It has been empirically observed in the DL community that throughput tends to decrease gracefully as image resolution increases, yet to the best of our knowledge, this relationship has not been quantified in the context of data loading. In order to investigate this relationship between image resolution and throughput, we created scatter plots for each library and dataset size (64 batch size, 1 worker, single GPU). Figure 6 shows an example plot for the RANDOM_K_100000 dataset.

We conducted an analysis of the data using an exponential decay model of the form $(y = a \cdot e^{-b \cdot x} + c)$. The goodness of fit was assessed using the $R^2$ statistic [41]. The $R^2$ values for the exponential decay fits ranged from 0.952 to 0.999 across all libraries and dataset sizes, underscoring the robustness of the model. This strong correlation indicates that throughput decreases following an exponential decay pattern as image size increases, suggesting that the impact on performance is less severe than what would be expected from a quadratic decline of the form $y = y_0/(x/x_0)$, with $x_0$ and $y_0$ the initial image resolution and the corresponding throughput at that resolution. In Figure 6, this relationship is visually depicted using the mean throughput across all libraries. This finding highlights that all libraries exhibit a similar behavior in handling increased image sizes, suggesting that they are optimized for higher loads, while also providing a reliable basis for predicting how data loading throughput will scale with image pixel resolution.

H. Relationship with Dataset Size

In this experiment, we explored the impact of varying dataset sizes on data loading throughput, using the RANDOM_512_N dataset (64 batch size, 1 worker, single GPU).

\begin{align*}
\text{Model Fit Parameters: a}=5402.76336, \text{b}=0.00995, \text{c}=63.70686
\end{align*}
The image resolution was kept constant, while the dataset size was systematically increased from 1,000 to 10,000 to 100,000 samples. For reference, the RANDOM_S12_100000 requires more than 26GB of storage for each library.

As dataset size increases, the likelihood of cache misses rises, potentially leading to significant performance degradation. The results, presented in Figure 7, demonstrate that most libraries exhibit minimal throughput reduction when moving from 1,000 to 10,000 samples. However, a substantial performance drop, approximately 50%, is observed when the dataset size increases from 10,000 to 100,000 samples. This sharp decline underscores the challenges that libraries face in maintaining performance under heavier data loads.

Among the libraries tested, FFCV and NVIDIA DALI stand out for their robustness in handling larger datasets. While most libraries suffered significant slowdowns, they exhibited only 8.05% and 33.64% decrease in throughput respectively as the dataset size expanded from 10,000 to 100,000 samples. This result highlights their efficient scaling capabilities and suggests that they are additionally suited for tasks involving large-scale datasets.

VI. DISCUSSIONS

This section highlights key insights from our analysis of deep learning data loading performance across libraries and addresses current limitations.

**Superiority of FFCV and DALI.** Across the majority of our experiments, FFCV and NVIDIA DALI consistently outperformed the other libraries in terms of throughput. This performance gap can be attributed to some key architectural characteristics. FFCV’s efficiency stems largely from its underlying .beon file format, which organizes datasets into pages, thereby minimizing random read penalties. Additionally, FFCV employs a just-in-time (jit) compiled data processing pipeline that bypasses the constraints of Python’s global interpreter lock (GIL). By executing machine code outside Python’s runtime, it can leverage threads instead of relying on subprocesses, as is common with most other libraries. For NVIDIA DALI, its main advantage lies in its ability to offload parts of the data pre-processing pipeline to the GPU using a GPU-accelerated prep mode, further enhanced by the optimized nvJPEG library. Even in CPU-only scenarios, DALI’s optimization strategies result in significantly faster pre-processing. We believe that a performant data loading library must integrate these key characteristics—optimized data formats, improved code compilation and hardware-accelerated processing.

**Ease of Use.** An important, yet more subjective, aspect to consider is the ease of use of the libraries benchmarked in this project. Many of these libraries lack comprehensive documentation, relying heavily on concrete examples rather than detailed guides or tutorials. This limitation hinders wider adoption and makes it challenging to assess their full potential compared to more established options like the default PyTorch DataLoader. Furthermore, integrating these libraries into existing deep learning workflows often requires a non-trivial investment of time and effort, particularly when adapting or modifying an existing codebase. This learning curve may deter users from exploring alternative solutions, despite their potential performance benefits.

**Varied Functionalities and Emerging Remote Capabilities.** Our analysis reveals that no single data loading library excels across all metrics—there is no “one-size-fits-all” solution. Instead, each library offers distinct features and demonstrates specific strengths. While FFCV and DALI consistently deliver high throughput, their limited native support for remote datasets and filtering features has constrained their broader application. However, some of these libraries have recently begun developing remote data loading capabilities, although much of this functionality remains largely undocumented. As more libraries implement robust support for remote data access, we will move closer to a paradigm where the learning process is fully decoupled from data management, allowing seamless data handling across distributed environments.

VII. CONCLUSION

Data loading is a critical aspect of Deep Learning (DL) training and often a major bottleneck, which can significantly impact the speed and effectiveness of end-to-end DL systems. This paper offers a systematic evaluation of DL data loading libraries, assessing their performance across various configurations, including worker count, batch size, GPU scaling, data access patterns, and remote data loading. Our comprehensive benchmarking indicates that FFCV and NVIDIA DALI outperform other libraries in most scenarios. Notably, in remote (cloud) storage contexts, Deep Lake demonstrated remarkable efficiency, with only a 49.7% increase in time versus the local scenario, highlighting the potential for effective decoupling of data storage from model training. Additionally, our analysis reveals that throughput decreases exponentially with image resolution, providing critical insights for optimizing data loading configurations. To facilitate continued research and practical improvements, we have developed the first open-source data loading benchmarking suite, which enables the community to replicate and extend our experiments.

VIII. ACKNOWLEDGMENTS

This work was supported by the Army Research Office MURI under the project number W911NF-23-1-0088, by the National Science Foundation under project number NSF-AoF: FAIN 2132573 and by the Department of Energy under project number DE-FOA-0003264.

Fig. 7: Throughput as a function of Dataset Size
