FFCV: FAST TRAINING VIA FAST DATA LOADING

Anonymous authors
Paper under double-blind review

ABSTRACT

We present FFCV, a library for easy and fast machine learning model training. FFCV speeds up model training by eliminating (often subtle) data bottlenecks from the training process. In particular, we combine techniques such as an efficient file storage format, caching, data pre-loading, asynchronous data transfer, and just-in-time compilation to (a) make data loading and transfer significantly more efficient, ensuring that GPUs can reach full utilization; and (b) offload as much data processing as possible to the CPU asynchronously, freeing GPU cycles for training. Using FFCV, we train ResNet-18 and ResNet-50 on the ImageNet dataset with a state-of-the-art tradeoff between accuracy and training time. For example, across the range of ResNet-50 models we test, we obtain the same accuracy as the best baselines in half the time. We demonstrate FFCV's performance, ease-of-use, extensibility, and ability to adapt to resource constraints through several case studies.

1 INTRODUCTION

What's the limiting factor in faster model training? Hint: it isn't always the GPUs. When training a machine learning model, the life cycle of an individual example spans three stages: reading the example into memory, processing the example in memory, and finally updating model parameters with the example on GPU (e.g. by calculating and then following the gradient). The stage with the lowest throughput determines the overall learning system's throughput.

Our investigations (and others' (Mohan et al., 2021)) show that in practice the limiting factor is often not computing model updates but rather the data reading and data processing stages. Indeed, in standard training setups, *the GPUs can spend a majority of cycles just waiting for inputs to process!*

To better saturate GPUs and thereby increase training throughput, we present FFCV, a system designed to reduce data loading and processing bottlenecks while remaining simple to use. FFCV operates in two successive stages: *preprocessing* and *train-time loading*.

In the first stage, FFCV preprocesses the dataset into a format more amenable to high throughput loading. Then, in the *train-time loading* stage, FFCV's data loader replaces the original learning system's data loader without requiring *any* other implementation changes.

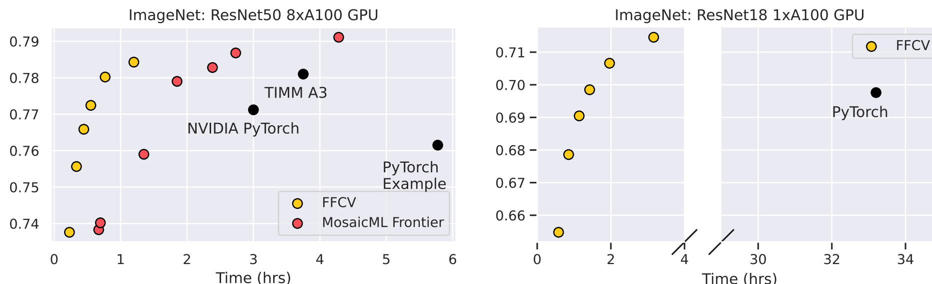


Figure 1: Accuracy vs. training time when training a ResNet-50 on 8 A100s. The FFCV accuracy/training time tradeoff outperforms all baselines. As an example, we can train ImageNet to 75% accuracy in less than 20 minutes on a single machine.

Together, FFCV data preprocessing and the FFCV data loader can drastically increase training speeds without any learning algorithm modifications. To demonstrate, we train machine learning models for a number of tasks much faster than previous general purpose data loaders can support, including single-node ResNet-50 (He et al., 2015) training (2x faster than the previous state-of-the-art to reach the same accuracies) and parallel ResNet-18 training (we can train 14 models per minute on an 8 GPU machine). While FFCV improves performance on most GPUs, its effect is most pronounced on faster GPUs, which require higher throughput data loading to saturate available compute capacity. We expect FFCV will only increase in utility as new GPUs become faster.

Contributions. We introduce FFCV, a drop-in, general purpose training system for high throughput data loading. Using FFCV requires *no* algorithmic changes, and involves a nearly identical API to standard data loading systems (e.g., the default PyTorch (Paszke et al., 2019) data loader). FFCV automatically handles the necessary data transfer, memory management, and data conversion work that users usually manually optimize (or leave to suboptimal defaults). FFCV also replaces the default data preprocessing and augmentation pipeline with one that is more efficient due to (a) just-in-time compilation to machine code and (b) highly optimized memory management. Comparing with strong baselines, we find that FFCV drastically speeds up a number of standard applications:

- **Faster ImageNet Training.** We greatly improve ImageNet single node training throughput, achieving state-of-the-art speed-accuracy tradeoffs. We reach the same accuracy as the best public baselines in less than half the time.
- **Faster bootstrapping and grid search.** We enable faster large-scale grid search by supporting same-machine, different-GPU training without any throughput penalty.
- **Faster network filesystem-based training.** Especially in cloud computing environments where network file systems are commonplace, data reading can greatly bottleneck learning systems. FFCV enables faster data loading in a realistic read-constrained environment.
- **Accelerating tasks beyond computer vision.** We demonstrate FFCV's ability to speed up almost any data loading task by using it as a drop-in replacement to the default PyTorch data loader in a GPU-enabled sparse regression solver.

2 Identifying Bottlenecks in Training

What makes a machine learning training system "slow" or "fast"? The answer varies by task, algorithm, implementation, and computing equipment available at train time. Model training is best thought of as a *pipeline* of discrete steps: data reading, data processing, and GPU computing that executes the learning algorithm.

To understand which of these steps bottlenecks training in practice, we study a standard task commonly used to benchmark training speeds (Coleman et al., 2017; Mattson et al., 2020), namely ImageNet (Deng et al., 2009) training. As a specific setup, we investigate the PyTorch ImageNet training example with the standard PyTorch ImageNet data loader, running on a standard AWS instance for GPU-based learning: p4d.24xlarge machines, which have 8 A100 GPUs, 96 vCPUs, and enough RAM to fit the ImageNet training set into memory. We benchmark each part of the system's throughput; Figure 2 shows our results. Overall, we find that data loading bottlenecks this standard training setup, and, furthermore, by fixing data loading we could achieve 30 times faster model training. Below we explore this data loading bottleneck in further detail.

Data reading throughput. We begin by only benchmarking data read throughput, measuring how long the data loader takes to read the entire dataset without performing any processing. As the machine we test on can cache the entire ImageNet dataset into memory, the data reading step is not a bottleneck (cf. Figure 2): it takes only 75 seconds.

Data processing throughput. To check whether data processing is a bottleneck, we measure how long the data loader takes to read the entire dataset while *also* performing processing: JPEG decoding, random cropping/resizing, random flipping, and normalization. We find that processing *is* a major bottleneck: adding processing to reading greatly increases loading time to 1200 seconds from the 70 seconds that loading alone took (see Figure 2).

Full training throughput. Finally, we measure the entire system's throughput, including the learning stage. We find that adding the learning stage does not greatly change the time taken to iterate

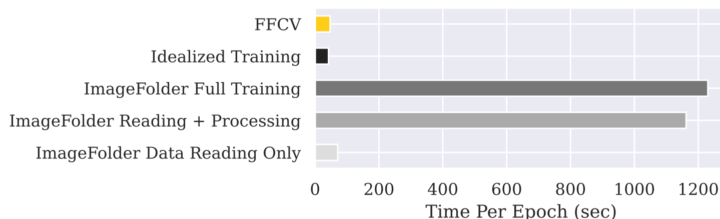


Figure 2: Time taken per set of stages in ImageNet training (median over three runs). ImageFolder refers to the default PyTorch data loader used to load ImageNet. We find that data loading, in particular data processing, is the major bottleneck of standard training. The idealized training time, or the training time we would obtain with perfect data loading, is almost 30 times smaller than the time required to just process all the training images. In the top column, `FFCV` reduces the overall training time to almost the idealized time by removing the data reading and processing bottlenecks.

through the whole dataset (cf. Figure 2). The fact that our throughput does not decrease despite adding learning on the GPUs indicates that the data loading/processing subsystem cannot supply data fast enough to saturate the GPUs. Indeed, as further corroboration, we simulate how fast model throughput could be by benchmarking our training process on a fixed data vector (here, we require no data loading). Our throughput in this idealized setting, shown in the same figure above, is much higher, and shows that we could obtain around 30x faster training with optimal data loading.

3 Eliminating Data Bottlenecks

Now, our focus turns to: how can we design a better data loading system? To maximize performance, `FFCV` manages the entire data management pipeline, from the file format used to store the training data all the way to data augmentations used at training time. Focusing on one step of the data loading pipeline at a time, we show how `FFCV`'s implementation circumvents issues in existing solutions to efficiently load data.

3.1 Challenge #1: Storing a Machine Learning Dataset

To eliminate data bottlenecks in the machine learning pipeline we start with the data format. There are already multiple existing file formats designed to store machine learning datasets: the most common of these formats (and indeed, the default one in PyTorch) is the *file-based format*, where one stores each example as an individual file. In the context of image recognition, for example, one saves each example as its own (typically JPEG-compressed) image file, and uses the enclosing folder to encode the label. The file-based approach has some advantages—most notably, users can intuitively interact with the examples on an individual level (e.g., they can open any training image in a standard image viewer). However, this format is not at all optimized for performance, and comes with several fundamental drawbacks.

`FFCV` introduces its own new file format: the `.beton` file. In the following, we discuss the different considerations involved in designing this file format, and show that `FFCV`'s new file format circumvents issues both with file-based formats as well as existing specialized solutions (namely, WebDataset, TFRecord, and MXNet RecordIO).

Reducing filesystem strain. To reduce filesystem strain, existing specialized file formats either group data examples in shards (WebDataset) or concatenate all the data into a single file (TFRecord, RecordIO). `FFCV` adopts the latter option, but goes even further—by organizing the dataset into pages (at the cost of some wasted space), it eliminates random read penalties by making it easy to read data in large chunks. `FFCV` (along with TFRecord and RecordIO) datasets are also easier to share than sharded data formats (e.g., WebDataset), since one only needs to transport a single file.

Flexibility. A data format should to be flexible enough to accommodate a wide variety of data formats and modalities. Many existing specialized solutions are hyper-specialized and support only specific modalities (e.g., RecordIO datasets can only store images with associated floating-point labels), while others are slightly more flexible (e.g., TFRecord and WebDataset). In `FFCV`, we opt for *maximal* flexibility, and use an abstract "Field" class that enables users to store arbitrary data

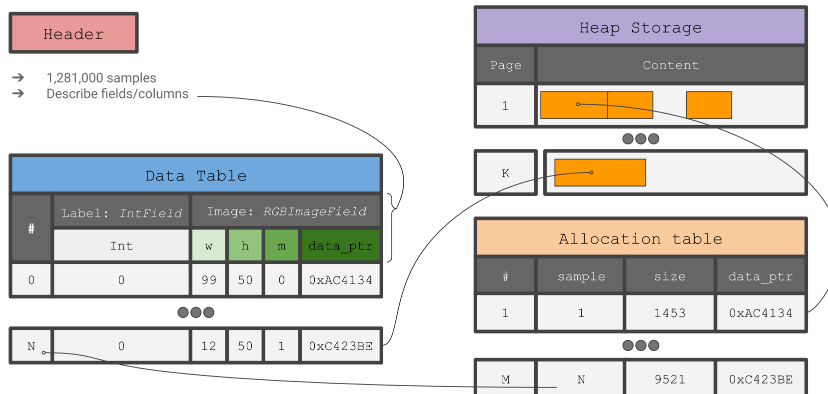


Figure 3: Structure of a `.beton` file used to store a simple image classification dataset.

modalities (with built in support for vision, text, tabular, and more), and even easily extend `FFCV`'s capabilities by writing data-specific customized encoders and decoders.

Searchability/Indexability. A good data format should also natively support *fast* access to only a particular subset of the dataset, whether for the purpose of inspecting a given example, or training on a particular subset of the training set. Specialized data formats that only support sequential reads, however (e.g., TFRecord, WebDataset) are inherently unable to support such a feature. `FFCV` datasets contain a data table that hold metadata (including indices) as well as pointers to any given sample, allowing one to easily filter and retrieve samples based on any predicate.

File structure. `FFCV` datasets are optimized for machine learning training and offer great performance regardless of the underlying storage method (RAM, HDDs, SSDs or network). Each file consists of four sections. The *Header* contains general information about the dataset like the number of samples and the fields. The *Data Table* is a small `DataFrame`-like data structure containing metadata (small fixed-width information) about a given sample, such as e.g. the image resolution in the image domain, or audio sample duration in the audio domain. The *Heap Storage* section contains pages (of default size 8MB) that store either variable size information or data that is too large for the *Data Table*, such as binary representations of images/audio examples. Finally, the *Allocation Table* at the end of the file contains book-keeping data about allocated regions in *Heap Storage*.

We show what a `.beton` file would look like for a basic image classification task in Figure 3.

3.2 Challenge #2: Efficient Data Reading

We now describe how `FFCV` achieves high read performance across a variety of compute environments with the `FFCV` file format, ranging from those featuring local SSDs (with high IOPS and low latency) to large spinning disks (which suffer under random, nonsequential accesses, such as when reading many discrete image files) to networked filesystems. `FFCV` offers built-in read strategies optimized for high throughput across all these different scenarios.

Operating system caching. For systems that can fit the dataset in random-access memory (RAM), `FFCV` can take advantage of OS-level caching. This ensures that every data read after the first one will be from RAM rather than disk, resulting in high throughput. Beyond just simplicity, OS-level caching also allows for multiple models training in parallel on the same dataset (i.e., when hyperparameter searching) to share the *same* cache without any additional memory overhead.

Process cache. On the other hand, if the dataset is larger than the main memory, an effective caching scheme will have to optimally cache, discard, and reload data at each epoch. In OS-level caching—which other data loading schemes use by default—the random access patterns induced by SGD in machine learning cause suboptimal caching behavior. `FFCV` circumvents this issue through optimized, process-level caching. By leveraging our knowledge of the sample order in data loading (since we can generate this order at the beginning of the epoch), `FFCV` can preload data much earlier than the OS can.

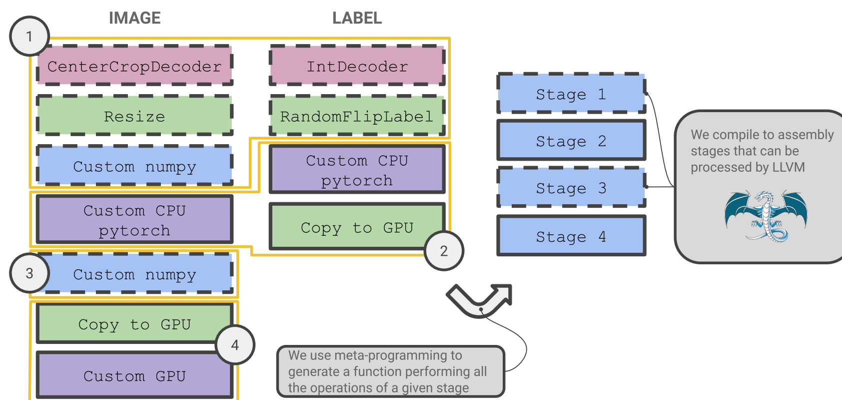


Figure 4: Illustration of the procedure followed by FFCV to generate the code of a complex image processing pipeline. Transforms are categorized together based on whether they can be JIT-ed (dashed, FFCV native or numpy based user-defined augmentations) or not (solid, Pytorch ones and others). Groups (stages) are formed based on these categories (1-4) by gluing each operation using meta-programming. Finally, the stages are compiled to machine code using Numba/LLVM.

Quasi-random sampling. For cases where disk reads are particularly expensive (e.g., when reading from a network drive and having insufficient RAM to cache the dataset), FFCV offers a *quasi-random loading strategy* that can combine with the process cache strategy above to minimize the stress on the underlying storage. Rather than reading examples in a uniformly random order, the quasi-random strategy (a) allocates a buffer large enough to fit batch_size pages of the dataset; (b) samples a permutation of all the pages of the dataset; then (c) generates a batch only from samples in the buffer.

WebDataset's shuffling procedure is similar to FFCV's quasi-random loading strategy with two crucial differences: (1) pages in FFCV are much smaller than WebDataset shards, leading to significantly better randomness [1]; (2) quasi-random loading in FFCV has a constant memory footprint, while WebDataset's footprint scales linearly in the number of workers. For further comparison, see Appendix A.

3.3 Challenge #3: Fast Data Processing

So far, we've outlined how FFCV improves both data storage and reading—we now turn to the *data processing* stage of the ML pipeline. In ML research, the data augmentation/processing pipeline requires both efficiency (in order to avoid bottlenecking the entire training process) and the flexibility to accommodate, e.g., researchers devising their own augmentations of pre-processing techniques.

FFCV tries to strike a balance between these two objectives via *just-in-time (JIT) compiled* data processing pipeline. Specifically, for a small cost paid at the start of training, FFCV analyzes the user-provided (Python) data processing pipeline, and automatically compiles it to optimized *machine code* via the following steps:

Categorization. Our main tool for compiling Python to machine code is the Numba library (Lam et al., 2015), which is by default able to compile a large (but not complete) subset of the Python language into machine code[2]. We thus first *categorize* each element of the data pipeline based on whether it can be automatically compiled by Numba. (Note that all the pipeline elements that ship with FFCV *are* Numba-compilable, so this step is primarily to enable users to write their own FFCV compatible non-compilable transformations, as in cases where it is too difficult to write a compilable verfsion of a transformation).

Grouping. After categorizing each transformation in the data pipeline, we group together all consecutive transforms of each category into groups called "stages" (see Figure 4). This will allow us to compile several separate pipeline elements into a single executable block of machine code.

[1]While one can manually make WebDataset shards smaller, this incurs a significant filesystem load.
[2]We motivate this choice and compare with other compilation systems in Appendix C.

Code generation. Finally, using meta-programming, we generate the code necessary to *fuse* each stage into a single function. Some of the stages will be then passed to Numba to be converted to machine code, the others remain unmodified and run natively in Python (albeit at much lower speed than their compiled counterparts).

Memory pre-allocation. A core tenet of FFCV is to avoid unnecessary memory allocation. Thus, every operation in the pipeline declares memory requirements in advance, and memory allocation is performed *once* at the start of an epoch. To let workers prepare the data while training happens, and to absorb potential slow downs in data preparation, FFCV relies on a circular buffer (illustrated in Figure 7 in Appendix D).

3.4 Challenge #4: Circumventing Data Transfer Costs

Since compiled machine code isn't under the supervision of the Python interpreter, FFCV can escape the constraints of Python's global interpreter lock (GIL) and can rely on threads instead of sub-processes like most libraries (the GIL typically only allows a single thread to use the Python interpreter at once, generally making multi-threading infeasible).

Threads yields two important advantages for FFCV. First, threads can collaborate directly by reading/writing memory instead of using (expensive) communication primitives. FFCV workers can therefore work together on same batch instead of having to work on their own, improving on latency and saving large quantities of memory (i.e., FFCV's memory usage is typically constant instead of scaling with the number of workers). Second, since they share the same CUDA context, all data preparation operations running on GPUs (e.g., data copying, augmentations) can be run asynchronously and—critically—in parallel with respect to the training loop, reducing the length of the critical path. See Appendix B for more details.

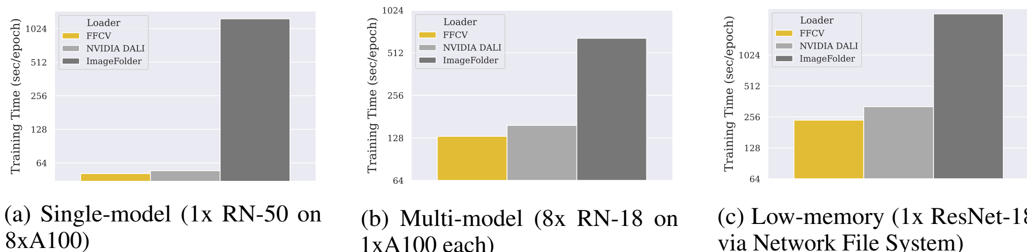
4 Case Studies

In this section, we showcase FFCV's versatility by illustrating how it can dramatically accelerate model training in three common practical settings. While one can use FFCV with any task or modality, we center our first three use cases around the ImageNet ILSVRC-2012 image classification task, which comprises 1.3 million labeled training images corresponding to 1,000 different classes. ImageNet is a standard dataset in both image classification (Russakovsky et al., 2015; He et al., 2015; Krizhevsky et al., 2012) and model training speed benchmarks (Mattson et al., 2020; Coleman et al., 2017; 2019); indeed, searching "ImageNet PyTorch" on GitHub returns hundreds of thousands of repositories.

We show, through the following use cases, that FFCV enables dramatic speedups over typical setups and more modest speedups over specialized performant training setups requiring specialized hardware (namely, NVIDIA DALI (NVIDIA, 2018)):

- **Single-model training**: We first use FFCV to train a ResNet-50 on ImageNet to 75% accuracy in 20 minutes on a single node, (Pareto) dominating all previous recorded benchmarks we are aware of (in fact, beating the next-best baseline by a factor of two). To evaluate FFCV directly with other data loaders in this scenario, we also measure the effect of using different data loading baselines in a fixed a model training setup;
- **Multi-model training**: We then consider the setting where a researcher wants to train *several* small models in parallel (e.g., to obtain confidence intervals or perform hyperparameter search). We show that FFCV can train 8 ResNet-18s at the same time (one per GPU) without incurring any additional overhead over single-GPU training. This demonstrates that FFCV enables both (a) efficient training of high throughput models and (b) low-overhead concurrent training.
- **Low-memory training**: Finally, we consider the (practically common) setting in which the dataset does not fit into machine memory (RAM). Via process-level caching and a quasi-random sampling scheme (exposed to users via just two lines of code), FFCV accelerates training even when reading data from slow disks (and even from networked file systems) with minimal performance overhead.

We then demonstrate FFCV's drop-in applicability beyond computer vision tasks:



(a) Single-model (1x RN-50 on 8xA100)

(b) Multi-model (8x RN-18 on 1xA100 each)

(c) Low-memory (1x ResNet-18 via Network File System)

Figure 5: A comparison between the time to train one epoch on ImageNet using `FFCV`, NVIDIA DALI, and PyTorch's ImageFolder.

- **GPU-enabled sparse regression:** In just a few lines of code, `FFCV` can considerably speed up an iterative SAGA (Defazio et al., 2014) solver (similar to that of Wong et al. (2021)) by simply replacing a default PyTorch data loader (loading from a memory-mapped file) with an `FFCV` loader.

4.1 Training a single model

We first study the simplest use case of `FFCV`: training a single model on ImageNet as fast as possible. By combining the data loading speed of `FFCV` with known ImageNet training optimizations, we are able to establish a new state-of-the-art speed/accuracy tradeoff for the benchmark task of training a ResNet-50 on ImageNet (Figure 1).

Fast training. We begin with an overview of the training algorithm itself—a long line of work has explored various modifications to standard training that have been shown to improve speed and/or accuracy, of which we use Blurpool (Zhang, 2019); NoWD-BN (Jia et al., 2018); linear learning rate annealing (Li et al., 2020); test-time augmentation and resizing (Touvron et al., 2019); and progressive resizing (i.e., we start training at 160px resolution and then increase to 192px 75% of the way through training).

Fast data loading. With `FFCV` we JPEG-compress 50% of the dataset, compromising between compute (i.e. faster image processing, as 50% of the images come pre-decoded) and available memory. Doing so allows our system to:

(a) reap the full benefits of progressive resizing. Even at smaller resolutions like 160px in which the GPU has much higher throughput, we can still fully saturate the GPU as we are neither data reading nor processing bottlenecked;

(b) outsource augmentations to the CPU. Since the dataset is cached (largely decoded) in memory, we can use CPU cycles for augmentations that would otherwise go to decoding.

Evaluation. We compare our `FFCV`-enabled optimized ImageNet example to these baselines:

- **PyTorch ImageNet example:** As a naive baseline, we take the PyTorch ImageNet example code, which is both unoptimized and slow to load data—we use the reported accuracy from torchvision and multiply the per-epoch time (from Figure 5a by 90 to obtain an optimistic estimate of total training time. We modify the code to use half precision. This loader is far and away the most popular loader seen in open source/research implementations, and is used by the most popular open-source ImageNet training libraries (Wightman, 2019; Falcon et al., 2019; Developers, 2016).
- **NVIDIA ImageNet example:** As a second baseline, we use the NVIDIA DALI-based ImageNet example This work uses a fixed 90 epoch schedule.
- **MosaicML:** Finally, we consider the MosaicML explorer baseline. The MosaicML Explorer plots the tradeoff between accuracy and training speed for models trained with MosaicML's training system; MosaicML explores a far larger set of algorithmic training changes than our work, including MixUp (Zhang et al., 2017), specialized optimizers outside of SGD (Foret et al., 2021), squeeze-excitation blocks (Hu et al., 2018), and more.

We run all implementations on an AWS EC2 p4d.24xlarge machine. We report our results above in Figure 1, finding that our system obtains the best accuracy vs. speed tradeoff across all baselines. In particular, to obtain 75% accuracy we require only 20 minutes, much faster than any of the tested baselines and, to our knowledge, the fastest single-node system to obtain this accuracy.

(Pessimistic) comparison with DALI loading. Swapping out FFCV for DALI and the PyTorch ImageNet train loader, we measure the training times obtained for each data loader when the implementation is held constant in Figure 5a, and find that FFCV still dominates.

MLPerf baselines. We do not compare with MLPerf baselines for two reasons: the scenarios are unfair to the MLPerf baselines (which do not allow data preprocessing), and the best MLPerf baselines have very strict hardware requirements (for example, the best NVIDIA submission requires a DGX POD[3] and is not reproducible on cloud-based machines).

4.2 Training multiple models

Another common paradigm in machine learning is training multiple models simultaneously. For example, we may want to perform a grid search for optimal parameters, or rerun a model with the same training parameters to obtain confidence intervals on results. In what follows, we show that FFCV allows for much faster parallel training than existing methods: FFCV has automatic support for OS-level caching and is high throughput enough to support even 8 ResNet-18s training at once (ResNet-18 models have nearly three times the throughput of ResNet-50 models since they are smaller).

Evaluation. Using the same AWS EC2 p4d.24xlarge machine as above, we run eight concurrent training routines for the following baselines (each originally described in Section 4.1): **(a) PyTorch ImageNet example:** ResNet-18 with code from the PyTorch ImageNet example; **(b) Scaled DALI:** ResNet-18 with code from the FFCV ImageNet example, code, swapping out FFCV for a DALI loader.

Each training routine has access to one eighth of the available vCPUs (12) and one A100. For DALI and FFCV we use image datasets that have been originally scaled to 350px. FFCV performs no JPEG compression, storing only image pixel values. Our throughput results can be found in Figure 5b; we find that FFCV has greater throughput than either DALI or ImageFolder-based methods, despite not requiring any specialized hardware for decoding.

4.3 Low-memory training

In the previous two examples, we operated in a setting where the machine being used for training has sufficient memory (RAM) to cache the entire ImageNet dataset (in particular, our 50% compressed version of ImageNet is 339GB). In many scenarios, however, we do not have sufficient RAM to cache even a fully JPEG-compressed dataset, and are thus forced to load images directly from the filesystem. Normally, this process incurs additional significant training cost, especially in settings where the filesystem is mounted on a networked drive (or any other slow disk).

Here, we show that with minimal changes to existing code, FFCV enables fast training even in such resource-constrained setups. By changing only two lines of code, we can enable *process-level caching* and *quasi-random loading*. These optimizations together ensure that read-constrained, memory-constrained systems operate at as high a throughput as possible; see Section 3 for details.

Evaluation. Just as in the last section, we compare to NVIDIA DALI and the default PyTorch Image Dataset. The results, shown in Figure 5c, illustrate that FFCV indeed enables faster training in memory-limited settings.

4.4 Beyond computer vision

Finally, we show the applicability of FFCV beyond just computer vision workloads. Specifically, we consider a large-scale sparse linear regression problem with $n = 100,000$ training points and dimensionality $d = 50,000$. We use an optimized iterative SAGA-based optimizer (Wong et al., 2021) for solving sparse linear regression problems. In Figure 6, we compare the unmodified code

[3]See: https://www.nvidia.com/en-sg/data-center/dgx-basepod/

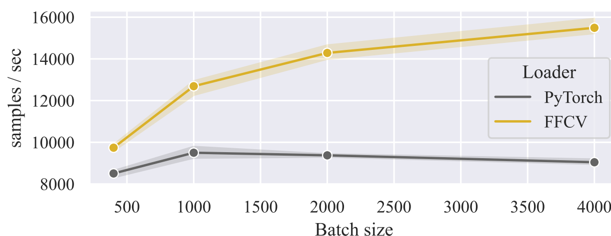


Figure 6: `FFCV` provides significant speedups over the default PyTorch data loader for non-vision modalities: here, e.g., a drop-in replacement of `FFCV` makes a sparse regression solver 1.6x faster.

(which loads data using the standard PyTorch data loader, reading from a memory-mapped file) with a drop-in `FFCV` replacement. The results indicate that even outside typical computer vision setups, `FFCV` is an effective drop-in replacement for default data loaders.

5 Related Work

Data pipelines in ML. Mohan et al. (2021) find that DNN training time is dominated by data loading in various settings. DALI (NVIDIA, 2018) uses custom input data pre-processing pipelines, with the option of off-loading some work to the GPU. (Aizman et al., 2019) introduce AIStore, a storage system, and WebDataset, a storage format based on POSIX tar, to enable high performance I/O for large scale deep learning. Murray et al. (2021) analyze millions of jobs on Google cloud and find that they spend a significant fraction of time in the input pipeline; they find that optimizing input pipeline performance is critical to end-to-end training time. Their framework `tf.data` allows users to build and execute efficient input pipelines, assisting with parallelism, caching, and static optimizations. Kakaraparthy et al. (2019) find that ML experiments with concurrent jobs (such as grid search) benefit from unifying data loading across jobs. Alternatively, `kornia` (Riba et al., 2020) implements standard image processing functions for GPUs.

Speedups from other sources. While our work and those cited above focus on removing the data bottlenecks in current ML workloads, large speed improvements also come from other sources, including hardware and algorithmic improvements, which allow models to achieve similar accuracies with less training. These include better architectures (e.g., ResNet (He et al., 2015)), optimizations (e.g., batch normalization (Ioffe & Szegedy, 2015), cyclic LR (Smith, 2017)), data augmentation (e.g., MixUp (Zhang et al., 2017)), among others.

Fast training on ImageNet. Our evaluation focuses on fast training on ImageNet, a standard in model training speed benchmarks (Mattson et al., 2020; Coleman et al., 2017; 2019). Many prior works (Goyal et al., 2017; Jia et al., 2018; You et al., 2018; Sun et al., 2019; You et al., 2017; Akiba et al., 2017) use distributed training with extremely large batch sizes to reduce training time. Beyond the increase in engineering complexity arising from distributed training, training models with large batch sizes comes with its own challenges, for instance, proper tuning of the learning rate Dettmers & Zettlemoyer (2019); You et al. (2017). Moreover, extreme resource usage, including large communication overheads necessitated by distributed training Coleman et al. (2019), reduce their usability.

6 Conclusion

In this work, we present `FFCV`, an optimized framework for eliminating data bottlenecks in machine learning model training routines. We use `FFCV` to establish a state-of-the-art speed/accuracy tradeoff for the ImageNet dataset, and demonstrate (through a series of case studies) the potential for `FFCV` to speed up almost any ML workload. The main limitation of our work is that it does not address non-data related bottlenecks in training, and might thus yield less significant (but still non-zero) improvements in settings where data is fast to load and process (e.g., NLP) or where models are very large and dominate training time (e.g., NLP).

References

Alex Aizman, Gavin Maltby, and Thomas Breuel. High performance i/o for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*, pp. 5965–5967. IEEE, 2019.

Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. In *NeurIPS ML Systems Workshop*, 2017.

Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *Operating Systems Review*, 2019.

Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in neural information processing systems (NeurIPS)*, 2014.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR)*, 2009.

Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

PyTorch Developers. Pytorch imagenet example. https://github.com/pytorch/examples/tree/master/imagenet, 2016.

William Falcon et al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. In *International Conference on Learning Representations (ICLR)*, 2021.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint 1706.02677*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

Mengtian Li, Ersin Yumer, and Deva Ramanan. Budgeted training: Rethinking deep neural network training under resource constraints. In *International Conference on Learning Representations*, 2020.

Peter Mattson, Christine Cheng, Cody A. Coleman, Gregory Frederick Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David M. Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim M. Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei A. Zaharia. Mlperf training benchmark. In *MLSys*, 2020.

Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021. ISSN 2150-8097. doi: 10.14778/3446095.3446100. URL https://doi.org/10.14778/3446095.3446100.

Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

NVIDIA. Dali: Nvidia data loading library, 2018. URL https://developer.nvidia.com/dali.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Edgar Riba, Dmytro Mishkin, Daniel Ponsa, Ethan Rublee, and Gary Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 3674–3683, 2020.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision (IJCV)*, 2015.

L. N. Smith. Cyclical learning rates for training neural networks. In *Winter Conference on Applications of Computer Vision*, 2017.

Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. *arXiv preprint arXiv:1906.06423*, 2019.

Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

Eric Wong, Shibani Santurkar, and Aleksander Madry. Leveraging sparse linear layers for debuggable deep networks. In *International Conference on Machine Learning (ICML)*, 2021.

Yang You, Zhao Zhang, C Hsieh, James Demmel, and Kurt Keutzer. 100-epoch imagenet training with alexnet in 24 minutes. *arXiv preprint arXiv:1709.05011*, 2017.

Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10, 2018.

Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

Richard Zhang. Making convolutional networks shift-invariant again. In *International conference on machine learning*, pp. 7324–7334. PMLR, 2019.

A QUASI-RANDOMNESS VS. WEBDATASET SHARDING

In Section 3.2, we mentioned that, while FFCV's quasi-random loading strategy is similar to WebDataset's sharding, the two strategies differ in key aspects. Here we discuss in more detail why this is the case.

In systems like Webdataset, sharding is done to reduce the number of files and relieve some strain on the file system. However, each file system has a block size after which random reads become sequential reads. In almost all system, this lies below 2MB. So, if Webdataset was to create shards of 2MB, it would create a dataset with way too many files that it overwhelms the file system.

On the other hand, FFCV's .beton format (by design) decouples this phenomenon by *sharding internally a single large file*. This means that our dataset is a single file the is structured to imitate sharding, as shown in Figure 3. This enables us to increase the quality of randomness as our shards are typically orders of magnitude smaller than WebDataset's. At the same time, we avoid overwhelming the filesystem as long as we pick the minimum shard size that will satisfy the file system while retaining a single file.

B WHY DOES FFCV USE MULTITHREADING INSTEAD OF MULTIPROCESSING?

In this section, we discuss in more detail (than Section 3.4) why FFCV relies to threads instead of sub-processes.

While many libraries, such as PyTorch, use multiprocessing during data loading to avoid the GIL, this comes with a large performance drop. In such strategies, all workers have to report back to the main process (running on a single thread). This often leads to lower performance with very large worker pools than with smaller as the main thread becomes overwhelmed. To avoid this, we resorted in FFCV to multi-threading instead, thus enabling each thread to write an individual sample directly in place in RAM without any inter-process communication. This also enables multiple threads to work simultaneously on the same batch, and avoids situations like when PyTorch's main thread could be sitting idle waiting for any of its workers to have their batches ready, although if these workers could accumlate their work (which is what we do in FFCV), the main thread would have not have been idle.

C COMPARISON OF JIT COMPILERS ALTERNATIVES

We motivate our decision to use Numba to compile FFCV's pipelines by elaborating on the pros and cons of potential alternatives:

`torch.script`: This JIT compiler, included as part of `Pytorch` was designed to optimize inference speed of deep neural networks and ease the transition between research code and deployment environments. In this context, it makes sense for it to only support the most basic python features and the Pytorch library itself. Relying on this solution for FFCV would have mean giving up on data augmentations written in `numpy` and interoperability with other deep learning frameworks. While `torch.scrip` can generate very fast code for some ML oriented operations (e.g matrix multiplications), it lacks some optimizations (`for` loops) and doesn't release the `GIL` which makes would have made it particularly unsuited for FFCV.

It is still important to note that it still possible to leverage `torch.script` within FFCV: users are allowed to introduce in their pipeline functions compiled with it as long as they operate on the GPU. Indeed, FFCV since doesn't compile those, it does not interacting with Numba and could even bring additional performance improvements in some scenarios.

TVM: TVM[4] shares many properties with Numba. They both are capable to generate LLVM IR, and can leverage the same optimization passes. Their performance should in theory be very similar. TVM has other advantages like being able to target other environments, this would not be useful

[4] https://tvm.apache.org/

for FFCV. The main differentiating factor remaining is the API. The familiar numpy augmented python seemed to be the most approachable for potential users. On the other end, TVM which was designed to represent and optimize complex neural network architectures would have been much more cumbersome to work with in this scenario.

D OMITTED FIGURES

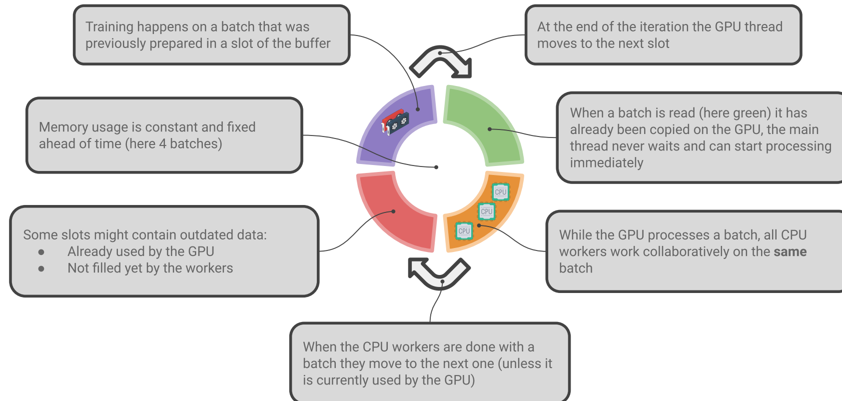


Figure 7: Illustration of `FFCV`'s circular buffer. The producer (data processing pipeline) works on an entry of the buffer while the consumer (user's code) uses a previously filled one. When done, they moving clockwise and pause when they encounter each other.