University of Münster

Department of Information Systems

Fast Forward Computer Vision: Accelerating Training by Removing Data Bottlenecks

Seminar Thesis

in the context of the seminar

Advanced Topics in Deep Learning

submitted by

Michael Vogt

Chair of Data Science:
Machine Learning and Data Engineering

Principal Supervisor	Prof. Dr. Fabian Gieseke
Supervisor	Christian Lülf, M.Sc.
	Chair for Data Science:
	Machine Learning and
	Data Engineering
Student Candidate	Michael Vogt
Matriculation Number	504831
Field of Study	Information Systems
Contact Details	mvogt1@uni-muenster.de
Submission Date	29.02.2024

Contents

List of Abbreviations

ADAM	Adaptive Moment Estimation
CephFS	Ceph File System
CPU	Central Processing Unit
FFCV	Fast Forward Computer Vision
GB	Gigabytes
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
MB	Megabytes
MiB	Mebibytes
MSE	Mean Squared Error
OS	Operating System
SSD	Solid State Drive
tf.data	Tensorflow Data
TPU	Tensor Processing Unit

1 Introduction

Training state-of-the-art deep neural networks can be time consuming and expensive. While the utilization of Graphics Processing Unit (GPU) resources has been widely studied [Moh+21], Leclerc et al. [Lec+23], Mohan et al. [Moh+21], and Murray et al. [Mur+21] found that in many machine learning projects, the data preprocessing is the limiting factor.

Before data can be used for model training, it is processed on a Central Processing Unit (CPU). In every epoch, large volumes of data are extracted from storage and transformed to a suitable format before being loaded to an accelerator [Mur+21]. If the data preparation time on the CPU exceeds the processing time on the GPU, the GPU becomes idle waiting for new data to arrive and a data bottleneck is present [Lec+23]. Data bottlenecks may account for a majority of training time [Moh+21] and can be found in many machine learning projects [Mur+21]. Especially when working with image data, data loading and preprocessing can be computationally demanding causing the CPU to slow down the process. To address data bottlenecks, researchers proposed specialized storage formats to reduce data loading time and algorithmic improvements to speed up the data preprocessing. This thesis discusses different approaches in detail, with a focus on Fast Forward Computer Vision (FFCV).

In Chapter 2, different Python libraries to tackle data bottlenecks are presented with a more detailed discussion of Tensorflow Data (tf.data), PyTorch and FFCV. Chapter 3 summarizes the findings of multiple conducted projects testing the efficiency of FFCV in different scenarios and Chapter 4 concludes this thesis.

2 Technical Background

This chapter serves as an overview of different approaches to reduce the effect of data bottlenecks in machine learning projects. First, Section 2.1 presents earlier contributions to the topic as well as drawbacks of specific solutions. Afterwards, tf.data (Section 2.2), PyTorch (Section 2.3) and FFCV (Section 2.4) are discussed in more detail.

2.1 Overview

The idea of fully utilizing available resources within a system attracted research in the field of data loading and processing. The PyTorch Dataloader resembles the baseline commonly used for machine learning projects [Lec+23] and is discussed in more detail in Section 2.3.

The MosaicML Composer[1] is a library for fast execution of distributed training workflows on computer clusters. Leclerc et al. [Lec+23] compared their framework with the MosaicML Composer which showed competitive results even before integrating FFCV (discussed in detail in Section 2.4) into this library in 2022.

Nvidia published the Data Loading Library DALI[2] which focuses on offloading data preprocessing to a GPU. It requires a `cuda` environment and offers less flexibility than other frameworks [Lec+23; Mur+21].

Apache MXNet `Data Loading API`[3] is a library written in `C++`. For custom functionality, extensions have to be written in `C++` which reduces the flexibility in a Python framework [Mur+21].

Research not only concerns data processing but also the file format in which the dataset is stored. The specialized file format MXNet RecordIO[4] was published by Apache too. There, all instances of the dataset are concatenated into a single file to allow fast processing of large chunks of data. However, RecordIO can only store datasets consisting of an image and a floating-point label which reduces the flexibility [Lec+23]. The MXNet project was retired in 2023, so there will not be further releases[5].

[1] See MosaicML.
[2] See Nvidia DALI.
[3] See MXNet Documentation.
[4] See Apache Mesos Documentation.
[5] See Apache MXNet.

When working with Tensorflow, instances of a dataset can be combined and stored in a single TFRecord file. However, TFRecord only allows sequential reads of the data and cannot offer fast access to parts of the data [Lec+23].

Aizman, Maltby, and Breuel [AMB20] present AIStore, a storage system designed for large scale input and output operations. AIStore is a distributed file system where training data can be stored. In addition, Aizman, Maltby, and Breuel [AMB20] introduce WebDataset as a file format. There, data is stored as shards, each concatenating multiple instances in a distributed fashion. WebDataset is especially usefull for datasets with a volume of multiple Petabytes.

2.2 Tensorflow Data

For deep learning tasks, Tensorflow [Aba+16] and PyTorch [Pas+19] are two of the most popular libraries widely used by researchers. Both implement unique approaches to address data bottlenecks.

With the framework tf.data, presented by Murray et al. [Mur+21], efficient input pipelines can be created and executed. tf.data was integrated into Tensorflow and became the default framework for various input pipelines.

For parallel processing, tf.data relies on threads which can address the dataset with an iterator to obtain the next instance. The benefit of threading is that the individual threads do not interfere with each other and no synchronization is necessary (as it would be when using multiprocessing). Although, a slight randomness in the ordering is produced without synchronizing the computations.

In many data transformation pipelines, identical computations are executed multiple times and slow the entire project [Mur+21]. The re-execution only concerns non-stochastic pipelines as stochastic pipelines execute identical computations on different data. tf.data implements efficient caching of the computed results[6] instead of caching the original data instances. Thus, identical transformations are not re-executed and memory usage often decreases[7].

Additionally, tf.data introduces static optimizations. Often combined transformations are fused to a single transformation step. For example, the operator `map_and_batch` leads to the same result as using the operator `map` (applying a function to data) first and `batch` (concatenation of instances) afterwards. This enables continuous com-

[6] Note that, in the case of stochastic pipelines that include deterministic transformations at the beginning, intermediate results may be cached to increase the efficiency.

[7] In many transformation steps like resizing, clipping and blurring, the instance size is reduced while it is rarely increased with a resizing transformation.

putation on the instances rather than threads needing to request the elements for multiple operations and it enhances the parallel computation.

Map vectorization reduces the computation time further. There, the ordering of operators is changed to exploit the fact that applying identical functionality to all elements of a vector is computationally faster than calling the functionality for each element individually.

Dynamic optimizations represent the core of tf.data. With the help of a background thread, the expected latency based on parameter settings as well as hardware specifications is periodically checked. The optimal configuration of the system can be computed in a few milliseconds and can lead to a large performance advantage. The auto-tuning mechanism automatically sets parameters like the amount of threads and memory usage accordingly. Depending on the calculations, the optimal resource allocation may vary and the parameters are adjusted dynamically.

In their paper, Murray et al. [Mur+21] present extreme speedups when using tf.data. It has to be noted that a sequential pipeline was used as baseline which is untypically even in the default dataloaders. Only with improved utilization of the CPU resources, part of the speedup will be achieved. Nevertheless, the parameter setting with the dynamic optimization as well as the static improvements will contribute to the speedup. Thus, the usage of tf.data leads to more efficient input pipelines in general but the reported results exaggerate the effectiveness.

2.3 PyTorch

PyTorch is a Python library tailored towards array and tensor handling [Pas+19]. For fast computation, compatibility and enhanced collaboration with accelerators is ensured.

Python code execution on the CPU is automatically overlapped with tensor operations on the GPU leading to parallel computations on different systems. PyTorch implements a work queue that allows the CPU to send a stream of data without being blocked waiting for a response. In a Python environment, the Python Global Interpreter Lock (GIL)[8] ensures thread safety at the cost of parallel computation. Restrained by the GIL, PyTorch implements multiprocessing to allow parallel computation within the same system. Compared to parallel threads, multiprocessing is less powerful as a main process needs to manage the others and communication between processes introduces additional overhead.

[8] See Python Global Interpreter Lock.

Overall, PyTorch focuses on the collaboration of multiple systems with asynchronous computations. With improvements in memory management, PyTorch reuses allocated memory in following iterations leading to fast processing times. However the Operating System (OS) cache is used for caching leading to low performances when working with large datasets which exceed the CPU memory [Mur+21].

2.4 Fast Forward Computer Vision

Similar to previous research, FFCV [Lec+23] is a state-of-the-art machine learning framework which aims at reducing the effect of data bottlenecks. The framework is designed for Computer Vision tasks (hence the name Fast Forward Computer Vision) as image augmentation can be computationally demanding. It should be flexible to allow arbitrary preprocessing pipelines while being easy to use.

FFCV adopts the idea of concatenating all training instances into a single combined file. Compared to a common dataset where each instance is stored in a separate file and can easily be inspected, the combined file is optimized for computer readability at the cost of user interpretability. When stored on a system, random read penalties from scattered files are removed, while the data collection may introduce an overhead in the first epoch if the file is stored on a distributed system (see Section 3.3).

Therefore, the file format `.beton` is introduced which is optimized for machine learning tasks. It consists of a header, data table, heap storage and allocation table (an illustration can be found in Appendix A.1). The heap storage is organized in pages with size of 8 Mebibytes (MiB) by default. On those pages, multiple instances are stored while each instance is stored on exactly one page. If the page size is not completely utilized, the leftover space remains empty resulting in an increased storage requirement of the beton file compared to the original dataset. Thus, the maximum instance size is limited by the page size leading to one instance being stored per page whereas smaller training instances are stored together per page.

For flexibility reasons, data is stored using a `Field` class defining how elements should be written to the beton file (encoder) and later read from the file (decoder). Fields like `NDArrayField`, `FloatField` and `RGBImageField` are included in FFCV and can be used for standard machine learning tasks. By implementing a custom encoder and decoder, the user is able to include arbitrary data formats (like `GeoTIFF`) in the beton file. For the conversion, the dataset can either be an indexable object including PyTorch datasets and Numpy's `ndarrays` or a Webdataset like `ImageNet`[9]. The allocation table keeps track of the storage fraction assigned to each instance and

[9] See FFCV documentation for more information.

the data table tells the system how to interpret the stored bits. For example when working with image data, the data table provides the height, width and channels of the image while the allocation table contributes a pointer to the storage.

Compared to the sharding of WebDataset, the pages are commonly much smaller than shards allowing faster random accesses and a new random order strategy.

In addition to the new file structure, FFCV implements optimized data loading and processing. To decrease the data loading time, the utilization of the system's caching is enhanced. If the dataset is small enough to be cashed entirely, the FFCV framework makes use of the OS cache. When the dataset exceeds the available memory, FFCV implements its own caching behaviour which preloads necessary instances in the next iterations earlier than the OS would do, as the OS does not have knowledge about the desired instance order after the current iteration. For expensive reads, when the data does not fit into memory or is read from another server, FFCV offers a quasi-random ordering of the instances. Instead of reading a page from memory to use one of the instances as done with random sampling, the pages themselves are randomly sampled and all instances of loaded pages are utilized. Thus, per epoch each page is loaded once whereas it might be loaded as often as there are instances per page in random sampling. With this sampling, the instance order is not completely random because instances stored together on a page are more likely to be part of the identical training iteration batch, however it leads to faster data loading (see Figure 2).

At the cost of time within the first epoch of training, the data transformation pipeline is just-in-time compiled to efficient machine code. The compilation is mainly done with the `Numba` library with custom non-compatible elements running in Python. With this compilation, the user can use customized loading and transformation functions and integrate them within the pipeline. Due to the machine code instructions, the GIL is circumvented allowing FFCV to use threads for parallel computation.

While great improvements across different machine learning tasks are presented in the paper, it should be noted that not only the dataloader was changed but also the machine learning model potentially resulting in fewer training iterations necessary to obtain the reported accuracy. In addition, in all their time measurements the first training iteration was disregarded to exclude the effect of the just-in-time compilation. However, in that time resources are used and it may impact the entire training time (as discussed in Section 3.3). Especially in their evaluation of a single model, "the dataset is cached in memory" ([Lec+23], p. 8) which means that the expensive read from storage is performed within the first epoch and crucially not included in their time measurement. Nevertheless, the presented results show that the integration of FFCV leads to faster model training across different machine learning tasks.

3 Implementation

As part of the thesis, the FFCV framework was used to obtain a deeper understanding. The most commonly used default is the PyTorch Dataloader[10], which is compared to the FFCV framework in different scenarios. To include a different environment to PyTorch, the previously mentioned tf.data framework is evaluated too. The code leading to the results as well as the generation of the figures is published on GitHub[11]. First, Section 3.1 provides the server specifications with the data described in Section 3.2. The different experiments tested the performance based on the Storage system (Section 3.3), a memory-exceeding dataset (Section 3.4), an entire machine learning project without a GPU (Section 3.5) and a project including GPU calculations (Section 3.6). Finally, Section 3.7 summarizes the findings.

3.1 Server Specifications

For this project, a JupiterLab server on the university network was provided. The server contained access to an AMD EPYC 7402P 24-Core processor with 24 physical and 48 logical cores and a NVIDIA GeForce RTX 3090 with 10,496 compute units and 24 GB memory. The access to the CPU was restricted to eight logical cores[12].

Sadly, due to the changes in the server's workload, there was a high variation in performance. For example, a stable epoch time would double at some point and remain stable at the higher time. By ensuring that there is no interference with other users and measuring the time for each project on the same day, all measured times are comparable but may not be reproducible exactly. All Jupyter notebooks, whose output is included in any graph, are included in the GitHub repository.

3.2 The Dataset

A single large satellite image was provided, which was used for tree height prediction. The image spans 57,832 pixels in height and 94,303 pixels in width for the three colour channels (RGB). An illustration of the image is shown in Appendix A.2. A major aspect of FFCV is the concatenation of a separated dataset into a single beton file. Thus, the large image was split into 2,256 smaller images with the shape `(3, 1200, 2000)` defining the channels, height and width respectively. Exemplary files are shown

[10] In this chapter the short form PyTorch always refers to the PyTorch Dataloader.
[11] GitHub repository.
[12] Multiple tests showed that parallelization with more than eight workers slowed the process likely due to context switches.

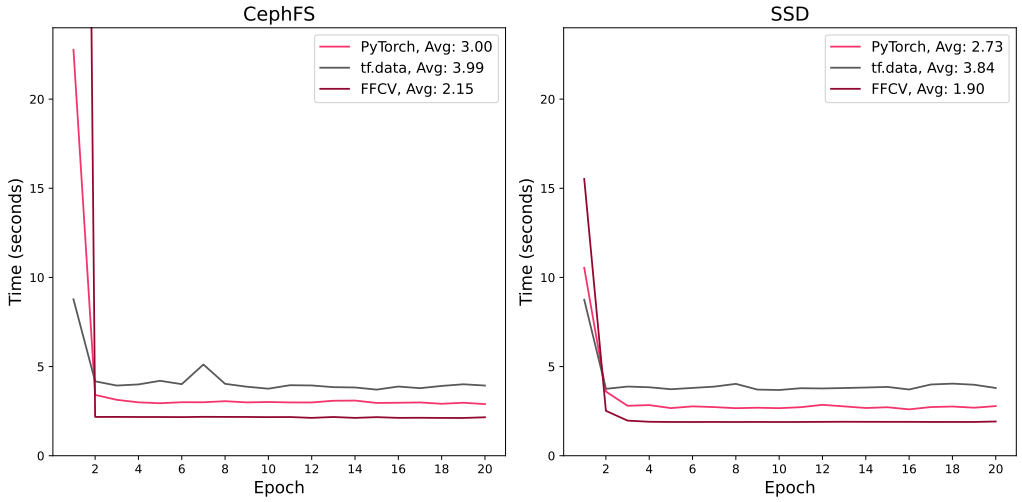

Figure 1 Individual epoch times for loading the data from a CephFS storage (left) and Solid State Drive (SSD) (right) with the PyTorch Dataloader, tf.data and FFCV where the first epoch took 131 seconds on the CephFS storage. The time average denoted in the label is excluding the first two epochs.

in Appendix A.3. If not stated otherwise, those files resemble the dataset which is used for the experiments.

The batch size for loading the dataset is chosen as 48 which is a divisor of the number of files as well as divisible by 8 parallel workers with are used in each experiment to fully utilize the available resources. To resemble a supervised learning project, a label for each image was randomly sampled and stored in a separate file. If not stated otherwise, the dataset is stored on the Ceph File System (CephFS) storage and uses 7.2 Megabytes (MB) per file totalling to 16,243 MB.

For the FFCV framework, the dataset had to be combined into a single beton file. The default page size of 8 MiB was used and a single image is stored per page. The file uses 18,933 MB of storage with an expected heap storage[13] of 18,924 MB. Thus, the header, data table and allocation table combined are responsible for 8.4 MB which is about 0.0446% of the entire beton file. Creating the beton file took roughly 15 seconds of computation, so the conversion is done very fast.

3.3 Storage System

The first experiment compares the loading times for different storage systems. The server provided access to the distributed file system CephFS as well as a SSD storage. From each of the storage systems, the dataset was loaded for 20 epochs with each of the three frameworks. PyTorch and FFCV used a random shuffling to load the data

[13] Obtained by multiplying the number of pages with the page size.

while the tf.data dataset was not shuffled (further discussed in Section 3.5). The times required by the different frameworks are shown in Figure 1. For every measurement, the first epoch takes significantly more time compared to the succeeding iterations. As the dataset completely fits into the memory of the CPU, the data is only read from storage in the first epoch and cashed for the subsequent accesses. While tf.data performed very similar regardless of the storage system, FFCV and PyTorch suffered from the distributed files and loading from the CephFS storage took significantly more time in the first epoch. The just-in-time compilation integrated in FFCV can be seen as additional time consumed within the first epoch, however the largest fraction of the consumed time is likely due to the collection of the distributed beton file[14]. For the later epochs, the data is cached with fast accessing as it can be seen from the legend of Figure 1 where the average consumed time starting from epoch three is displayed. There is a slight increase in throughput if the data was loaded from SSD, but the advantage is negligible.

3.4 Out of Memory Data Loading

A major focus of FFCV is the loading of a dataset which exceeds the available memory and has to be loaded for each individual epoch. For this experiment, the dataset had to be enlarged to require more than 200 Gigabytes (GB) of storage. Additionally, the quasi-random ordering strategy of FFCV leads to better performances for expensive reads if there are multiple instances stored per page. Therefore, each data file was split in half and each half was duplicated 13 times. This process yielded 58,656 individual files with a combined storage requirement of about 211 GB. Each page of the beton file stores two instances, so the quasi-random sampling can be used here.

The dataset was loaded for 20 epochs using the PyTorch Dataloader and FFCV with the random and quasi-random ordering strategy. The individual epoch times are displayed in Figure 2. As expected, all epochs now take the same amount of time as the data has to be read from storage. If the data would be cached as before, the time would increase by a factor of 13, however on average it increases by a factor of 188. The slight changes for each method are likely due to the order of instances in which they need to be loaded or the server inconsistency. In general, FFCV outperforms PyTorch significantly with an average of 250 seconds per epoch. The speedup can be increased by choosing the quasi-random strategy which spares additional 80 seconds per epoch at the cost of completely random shuffling. In the optimal case, this strategy divides the necessary time by the number of instances per

[14] The experiment was conducted multiple times with very similar results. Interestingly, the first epoch was faster in later experiments although the setup is identical but more computations are added.

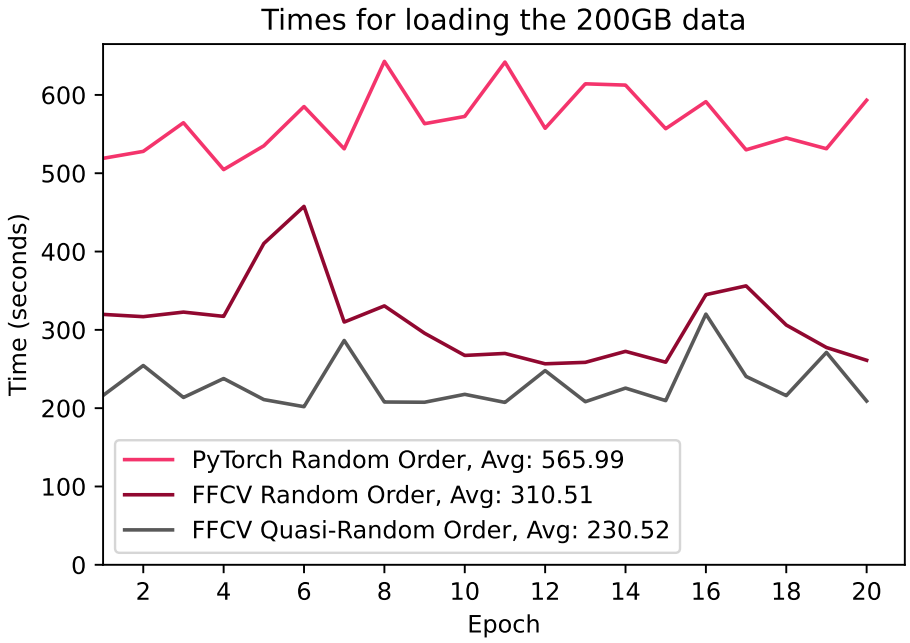

Figure 2 Epoch times for loading the large dataset with PyTorch and FFCV. For comparison, the quasi-random ordering strategy for FFCV was used too. The average contained in the label is calculated across all epochs.

page, which would be two in this case. Here, we obtain a speedup of 1.35 which is likely to increase if more instances are stored per page. Nevertheless, using the FFCV framework in the context of memory exceeding datasets increases the throughput of the data.

3.5 CPU Environment

The experiments aimed at displaying the advantages of FFCV in an end-to-end machine learning project. For this purpose, an arbitrary data transformation pipeline as well as a simple neural network were defined. For the comparison, PyTorch Dataloader, tf.data and FFCV were used as frameworks.

Transformation Pipeline

The transformation pipeline should include computationally demanding steps, which can be accelerated when using the GPU in Section 3.6. Often used functions for data augmentation were used, namely a random vertical and horizontal flip with 50% probability each. While the reflection of the axis can be computed easily, the random rotation of up to 90 degrees in both directions includes the calculation of pixel interpolation and takes significantly more resources. After the rotation some

PyTorch

```
dataset = CustomDataset(data_dir)
dataloader = DataLoader(dataset, batch_size=48,
                        shuffle=True, num_workers=8)

custom_transform = transforms.Compose([
    transforms.RandomVerticalFlip(0.5),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomRotation(90),
    transforms.RandomCrop((500, 500))
])
```

FFCV

```
loader_preprocess = Loader("data/train_data.beton",
                batch_size=48,
                num_workers=8,
                order=OrderOption.RANDOM,
                pipelines = {
                    "image": [
                        NDArrayDecoder(),
                        ToTensor(),
                        transforms.RandomVerticalFlip(0.5),
                        transforms.RandomHorizontalFlip(0.5),
                        transforms.RandomRotation(90),
                        transforms.RandomCrop((500, 500)),
                    ],
                    "label": [FloatDecoder(), ToTensor()]
                })
```

Figure 3 Implementations of the data loading and transformation with PyTorch (left) and FFCV (right).

areas consist of black pixels and the image size was reduced to 500 x 500 pixels. Examples of the transformed images are displayed in Appendix A.4.

Code Comparison

The implementation of the dataloaders for PyTorch and FFCV is displayed in Figure 3. Even without the knowledge of individual commands, one can see very similar structures between the two frameworks. First, general hyperparameters of the dataloader are defined. The attributes `dataset`, `batch_size` and `number of workers` are named identically, while `order` is a synonym for `shuffle` and allows the additional option of quasi-random sampling. In FFCV, pipelines for all `Fields` need to be defined, even if no transformations are used (see the label attribute). A sequence of transformations is normally combined with a `transforms.Compose` in PyTorch, while the transformations to the data are included in the loading pipeline in FFCV. The pipeline is compiled in a just-in-time fashion, allowing the user to include custom transformations into the dataloader. In general, most of the models using PyTorch can be transformed easily to FFCV and the model training time is reduced if a data bottleneck was present.

Machine Learning Model

To round out the experiment, a simple machine learning model was created. It consists of two convolutional layers each followed by a max pooling layer. Finally, two fully connected layers reduce the output to a single prediction. The model was trained using the Adaptive Moment Estimation (ADAM) optimizer and the Mean Squared Error (MSE) as loss function. Note that the labels for each instance were randomly sampled and thus, they do not follow any pattern. Therefore, the machine learning model cannot make use of the training process and only predicts the mean

of training instances for new data. Nevertheless, model updates are computed based on the current batch and the execution time can be measured.

Issues with Tensorflow

The major contribution of the tf.data framework is the introduction of autotuning the hyperparameters. Sadly, the `auto-tune` settings could not be used, as they detected more available resources on the CPU but were blocked when assigning them. With a suboptimal shuffle buffer, the entire workflow was heavily slowed down. Even without shuffling the data at all, tf.data performed worst across the different experiments but with a small discrepancy compared to the other frameworks[15]. Thus, for all time measurements with the tf.data framework, the data is not shuffled. With the PyTorch setup of the server, Tensorflow was not able to address the GPU at all, which results in this comparison being conducted on the CPU only. In Section 3.6, PyTorch and FFCV are compared in an environment including a GPU. While transforming the pipeline from PyTorch to FFCV requires a small effort, defining an identical data transformation pipeline with Tensorflow specific functions proofed to be challenging. Some functionality is included in the library Keras, however the transformations expect a different shape where the channels are defined in the third dimension while PyTorch uses the first dimension for the channels. Thus, multiple type conversions were conducted in the Tensorflow implementation which added additional overhead to the process.

Findings

Figure 4 displays the results of this experiment. For the time averages, data loading, data preprocessing and end-to-end training were measured individually per framework. Thus, nine notebooks which can be found in the GitHub repository are included in this graph. Compared to the other stages, the data loading (mainly from cache as discussed in Section 3.3) took a small portion of the time. Note that the average time of FFCV is the highest among the frameworks because the first epoch took 131 seconds to complete. In all following epochs FFCV was able to load the dataset faster than PyTorch and tf.data. Especially due to the random rotation transformation, data preprocessing was the most resource intensive part. With the small change in code discussed above, FFCV spares 100 seconds in data loading and data augmentation compared to PyTorch. Presumably the just-in-time compilation of the transformations leads to an optimized pipeline which is included in the loading computations and executes faster.

[15] Note that shuffling the data only caused a negligible overhead in rare cases but generally increased the time significantly.

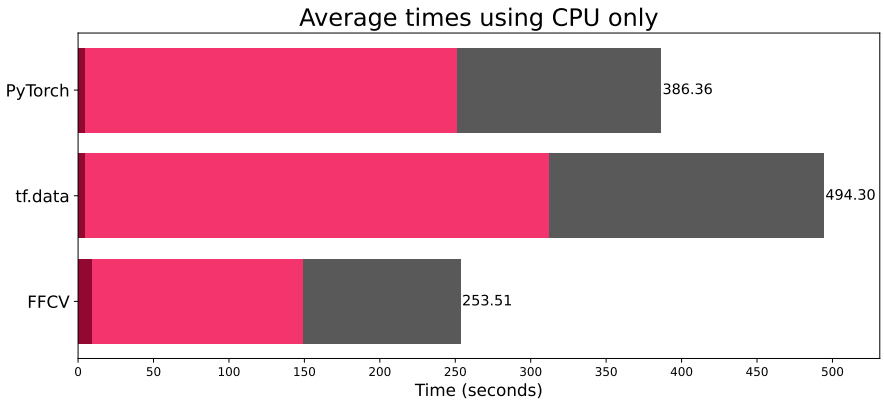

Figure 4 Average times for loading the data (dark red), applying the transformations (light red) and training a model (grey) on the CPU only. For data loading, 20 epochs were used while the other times are based on ten epochs. The numbers display the average time of an end-to-end training epoch.

Finally, the addition of the machine learning model increased the computation time further. Interestingly, PyTorch and FFCV use a completely identical model, so the additional time should be equal which is not the case as FFCV was 30 seconds faster than PyTorch in this stage alone. In total, improvements from the FFCV framework are visible within this project as the end-to-end training time was decreased compared to the PyTorch Dataloader. tf.data performed worse likely due to the overhead of implementing the same functionality in a different framework. Overall, this setup is slowed by the data processing as it consumes more time than the actual model training. FFCV was able to improve the end-to-end training time but the data preparation continues to outweigh the model training. For state-of-the-art deep learning models, a dataset size of 20 GB is very small and training with the above epoch times would be unfeasible. Therefore, GPUs are used as accelerators which is covered in the next section.

3.6 GPU Environment

In large scale machine learning projects, the use of accelerators like GPUs or Tensor Processing Units (TPUs) is essential. GPUs contain less powerful but far more compute units than CPUs and are capable of fast data augmentation especially when working with image data. With tf.data performing worst across all tasks, only PyTorch and FFCV are compared in the CUDA environment on the GPU. The results in Figure 5 indicate that the training time is significantly accelerated when the GPU is used for the data augmentation as well as the model training. Note that in many machine learning projects data augmentation is often done on the CPU to spare GPU memory or because the augmentation is not computationally expensive [Mur+21]. In

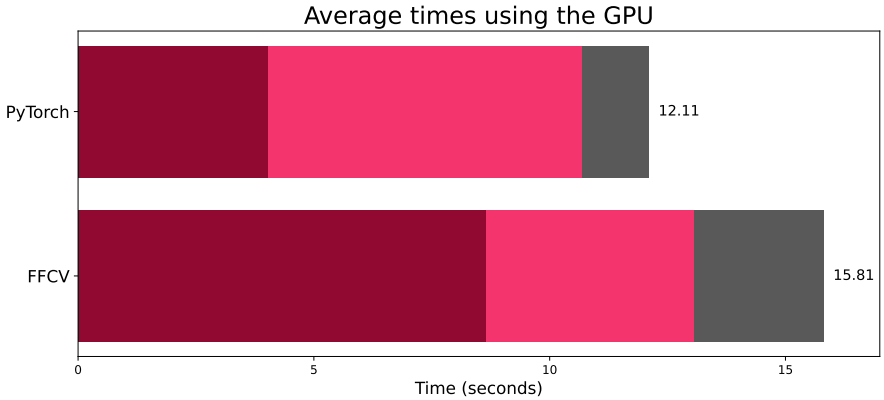

Figure 5 Average times for loading the data (dark red), applying the transformations (light red) and training a model (grey). The transformation and model training was conducted on the GPU.

this project, this would lead to the data preprocessing times presented in Figure 4 with a parallel execution of the model training on the GPU and result in a data bottleneck. Compared to the CPU environment, the end-to-end training is at least[16] 16 times faster. On the GPU, the interpolation necessary for the image rotation as well as the model weights can be computed in a highly parallel fashion leading to low epoch times.

This is the only considered experiment where PyTorch showed a better performance than FFCV with an advantage of 3.5 seconds per epoch. One might assume that the loading time in the first epoch is responsible for a large delay across all three stages, but the epoch times for the data preprocessing and end-to-end model training were consistent throughout all epochs. Somehow the first epoch time when only loading the data exceeds those with additional steps significantly, without the data being cached in previous calculations. Thus, this delay is shown within the data loading, but the data preprocessing and model training performed worse compared to PyTorch. With epoch times of 12-15 seconds, this experiment only contains few calculations, and it should be tested on more data with more representative transformations. However, even in this rather small experiment a memory leak[17] was encountered, where the GPU went out of memory after a few epochs. Based on the replies to the issues, the leak is likely caused by creating multiple iterators and non-closing threads, but none of the suggestions helped for my project[18]. Interestingly, with the same setup on the CPU no issues arised which may be due to the CPU automatically closing

[16] The best CPU computation time is compared to the worst GPU computation time.
[17] See FFCV Issue 206 and FFCV Issue 345.
[18] After a GPU restart, there was enough memory to measure ten epochs with each epoch needing a bit more time and memory than the previous one.

unnecessary threads. Nevertheless, it should be mentioned that the data transfer to the GPU can be included in the loading pipeline and thus, easily integrated in FFCV.

3.7 Summary

Multiple different experiments were conducted in which FFCV showed promising results compared to the PyTorch Dataloader. Especially in the context of loading large datasets, FFCV showed a higher throughput which can be increased when the quasi-random shuffling is used. As expected, access to a SSD storage is faster than to data that is stored on a distributed file system, where the first epoch of FFCV slows down the project significantly. The FFCV framework can be integrated into an existing PyTorch project which led to faster data loading and data preprocessing in the CPU environment. In the experiment where the GPU was used for data preprocessing and model training, these steps were greatly accelerated and PyTorch performed better than FFCV. With the obtained results, it can be assumed that FFCV outperforms the PyTorch Dataloader in most machine learning projects with a larger dataset, more complex data transformation and model training steps.

4 Conclusion

The goal of this seminar thesis was to present the FFCV framework and compare the performance to different data processing frameworks. Therefore, an overview of available python libraries and data storage formats was presented with a more detailed discussion of the benefits of tf.data, PyTorch and FFCV. Despite the benchmark tests presented in the publications not being completely representative, they indicate that the effect of data bottlenecks can be reduced without specialized hardware. By storing the data in specialized formats and better utilizing available CPU resources, the end-to-end training time of machine learning projects can be greatly accelerated.

The thesis showed that FFCV can be used as a drop-in solution for existing PyTorch models leading to an improvement across different experiments. Especially the improved caching mechanism of FFCV led to significantly faster data loading when the dataset cannot be cached entirely. Computationally demanding data augmentations benefit from the implementation in FFCV when using the CPU while the PyTorch Dataloader performed better when an accelerator was involved in the project. tf.data performed worst in all projects, showing that a completely fair comparison of entire projects programmed in two libraries is hard to achieve. The overall results indicate that projects with a large scale dataset and complex transformations profit from using FFCV over the PyTorch Dataloader.

FFCV focuses on efficient data storing and loading by a better utilization of available resources without changing any hardware component. However, reducing the runtime of data preprocessing only speeds up the project if data bottlenecks are present. When computations on the GPU slow the training, speedups can be achieved with hardware improvements by using more or faster accelerators. Other possible options include algorithmic improvements where the system learns faster and needs less epochs of training.

A Appendix

A.1 Beton format

In Figure 6, an illustration of the beton file structure is presented. Firstly, the header contains meta information about the dataset, stating the number of instances and the `Fields` stored per instance. The data table stores meta information about the individual instances per `Field`. For example, the resolution of each image is stored there which may vary across the dataset.

The data is stored in the heap storage which requires most of the space for this file. The allocation table defines the storage location within the heap storage for each instance. With the allocation table, parts of the data can be randomly accessed quickly.

A.2 Given Data

The given satellite image contains lakes, fields, forests and villages. A fraction of the image is displayed in Figure 7.

A.3 Data Parts

In many projects, the dataset is distributed with each instance being stored as a single file. Therefore, the given image is split into disjunctive parts, each with identical shape. Twelve of the 2,256 image parts are shown in Figure 8.

A.4 Transformed Images

With the pipeline displayed in Figure 3, the data images are transformed before model training. Some examples of the transformed images are shown in Figure 9. Due to the random rotation with zero fill, some corners of the images contain black areas. By cropping the images randomly, black areas are reduced but not eliminated completely. If the model should be trained on available data only, the center pixels can be cropped consistently to eliminate black areas or the transformed squares would need to be checked for many empty entries.

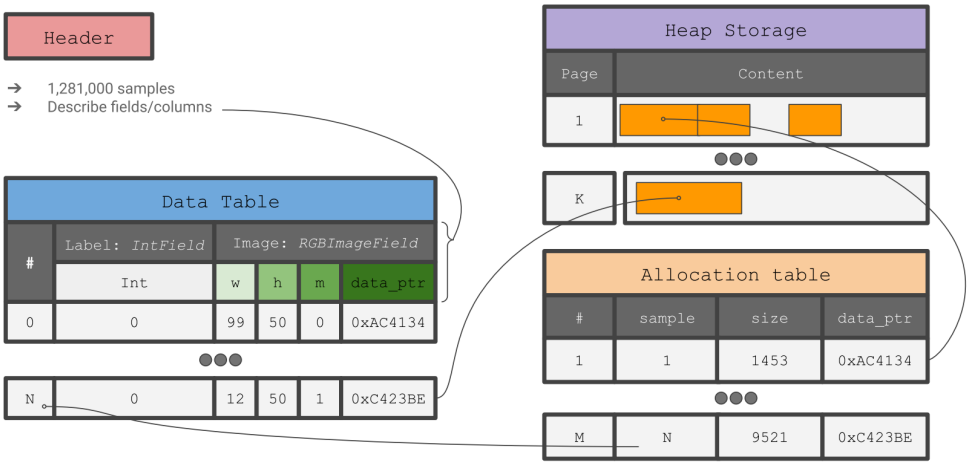

Figure 6 Figure 3 from [Lec+23], displaying the structure of the beton file format.

Figure 7 Excerpt of the given satellite image. 20,000 x 20,000 of the 57,832 x 94,303 pixels are displayed here.

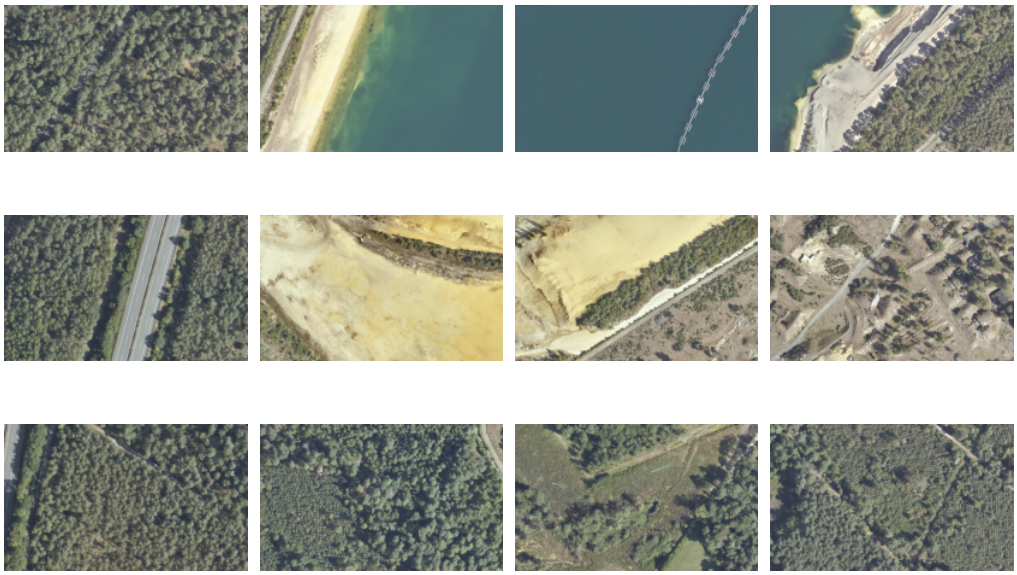

Figure 8 Twelve example image parts used within the dataset. The parts are separate areas of the large image with a shape of `(3, 1200, 2000)` each.

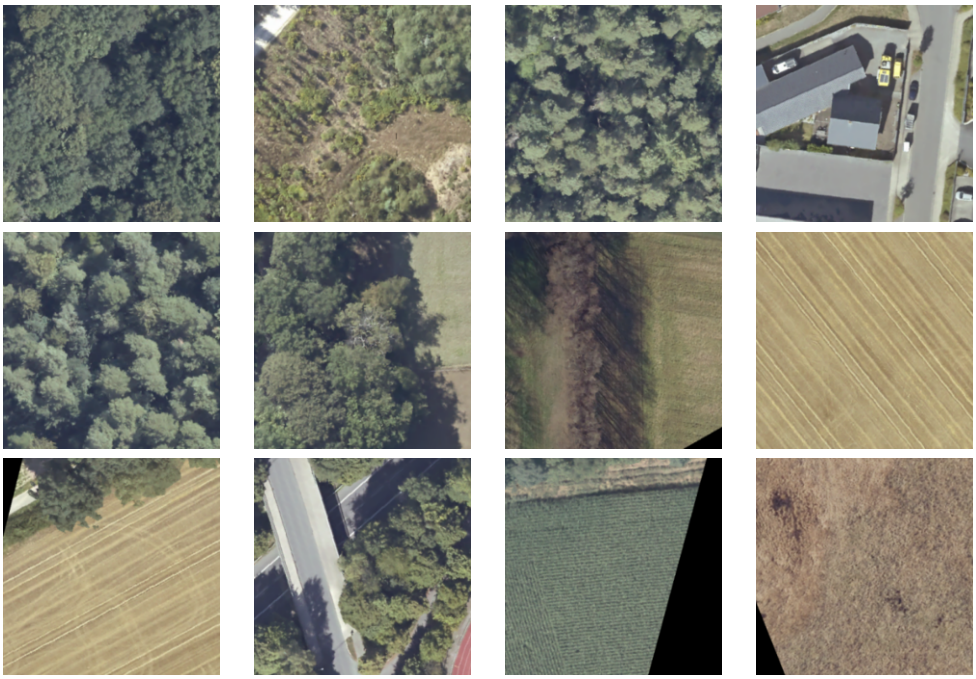

Figure 9 Twelve transformed data images which are used for model training. The black areas are caused by the random rotation transformation.

Bibliography

[Aba+16] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. arXiv:1605.08695 [cs]. May 2016. DOI: 10.48550/arXiv.1605.08695. URL: http://arxiv.org/abs/1605.08695 (visited on 02/27/2024).

[AMB20] Alex Aizman, Gavin Maltby, and Thomas Breuel. *High Performance I/O For Large Scale Deep Learning*. 2020. arXiv: 2001.01858 [cs.DC].

[Lec+23] Guillaume Leclerc et al. *FFCV: Accelerating Training by Removing Data Bottlenecks*. arXiv:2306.12517 [cs]. June 2023. DOI: 10.48550/arXiv.2306.12517. URL: http://arxiv.org/abs/2306.12517 (visited on 02/27/2024).

[Moh+21] Jayashree Mohan et al. *Analyzing and Mitigating Data Stalls in DNN Training*. 2021. arXiv: 2007.06775 [cs.DC].

[Mur+21] Derek G. Murray et al. *tf.data: A Machine Learning Data Processing Framework*. 2021. arXiv: 2101.12127 [cs.LG].

[Pas+19] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this thesis titled *Fast Forward Computer Vision: Accelerating Training by Removing Data Bottlenecks* is my own, independent work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references; this also holds for tables and graphical works.

Münster, 29.02.2024

Michael Vogt

Consent Form

Name: Michael Vogt
Title of Thesis: Fast Forward Computer Vision: Accelerating Training by Removing Data Bottlenecks

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software. The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 29.02.2024

Michael Vogt