Deep Learning Optimized on Jean Zay

Dataset optimization

Storage spaces and data format

Authors: Bertrand Cabot, Myriam Peyrounette

Commented slides

October 2023

Dataset optimization

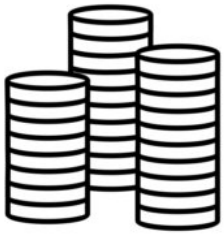
Main bottlenecks

- Data storage – various disk spaces
- Data format – at sample level
- Data format – at dataset level

2

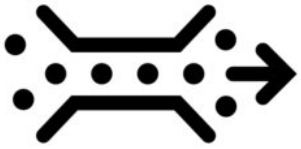
Bottlenecks upstream of DataLoader

Storage Disks



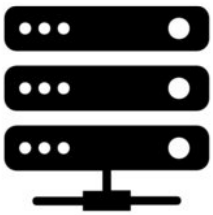
1. I/O performance

Interconnection Network
Omnipath



2. Shared Bandwidth

CPU workers



3. Decoder performance

3

In this section, we will present different optimization possibilities in regard to the input data format.

We are seeking to avoid three main obstacles: Slow I/O read speed on disks, bandwidth sharing in the OPA interconnection bus and the cost of input data decoding on CPU.

Dataset optimization

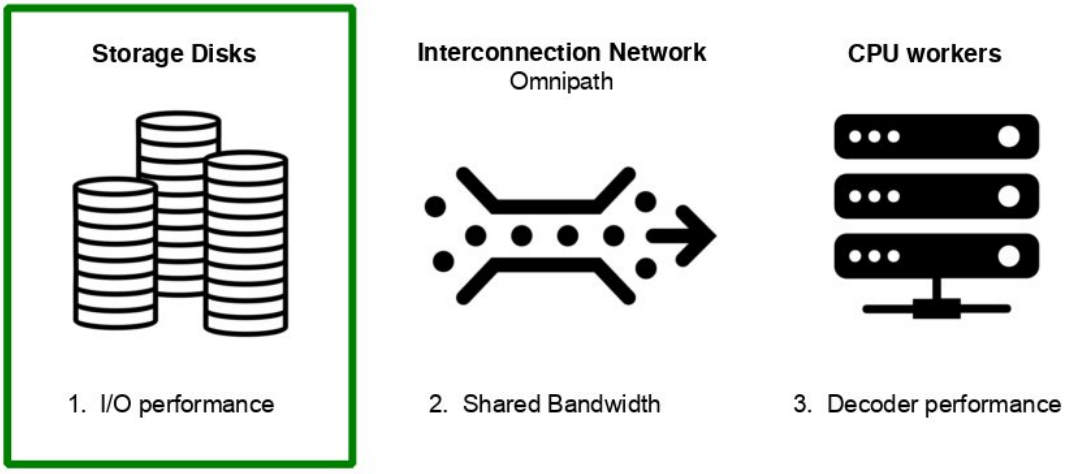
Main bottlenecks ◂

Data storage – various disk spaces ◂

Data format – at sample level ◂

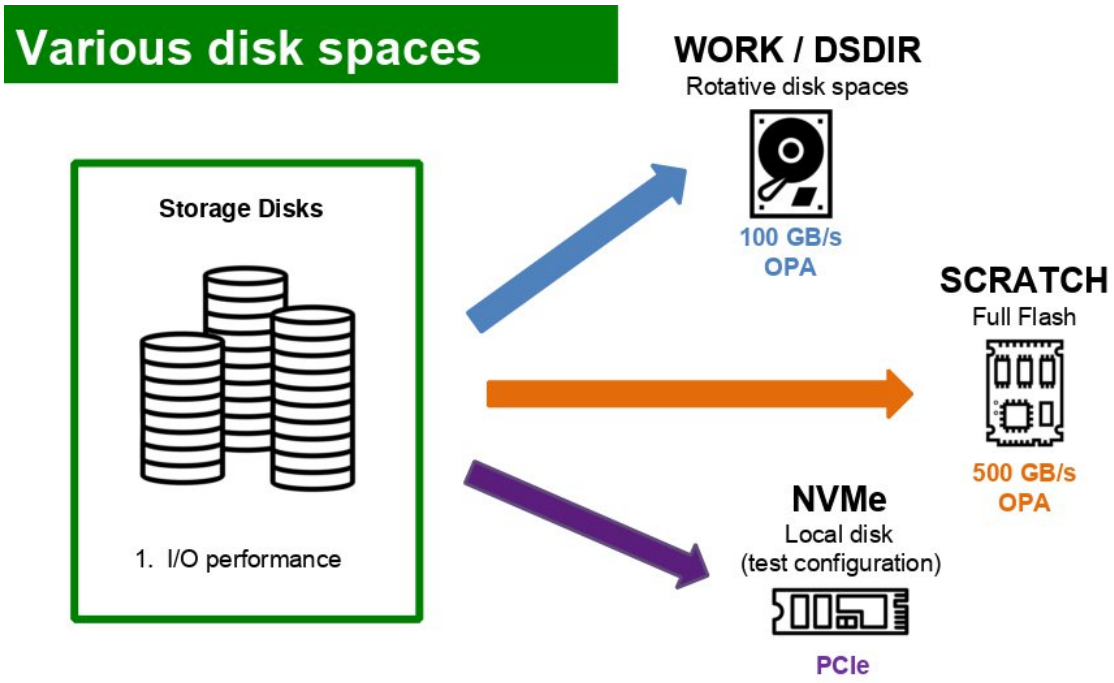
Data format – at dataset level ◂

Bottlenecks upstream of DataLoader



Where should I store my dataset?

There are different storage technologies on Jean Zay. Where is it pertinent to store your input data?

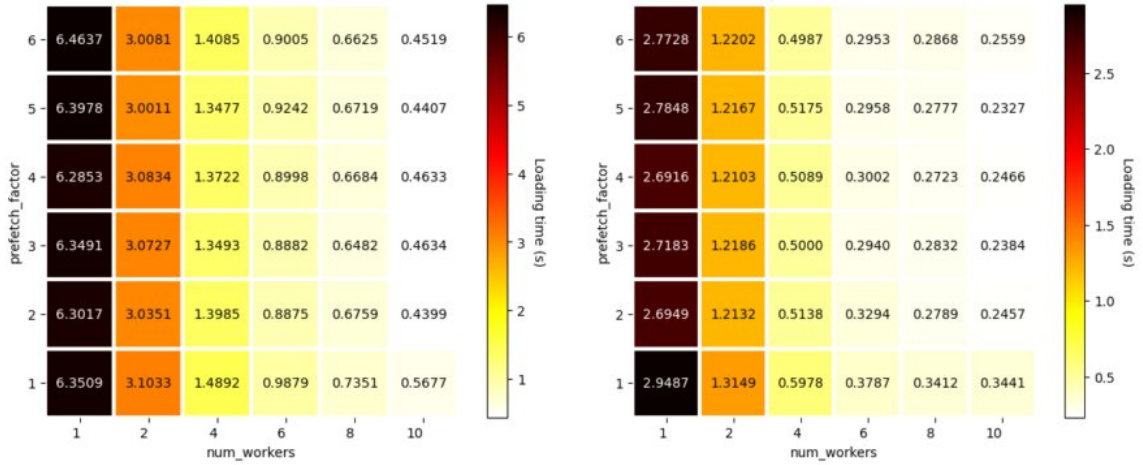


The WORK and DSDIR spaces store information on rotative disks with a read/write speed of 100 GB/s.

The SCRATCH stores information on a Full Flash (SSD) disk with a read/write speed which is 5 times faster than with rotative disks. It is possible to connect local disks (NVMe) directly to a compute node but this technology is not manageable at the level of a supercomputer and is only available internally for tests of technology watch.

Various disk spaces

dlojz.py - 50 iterations - test partition gpu_p4



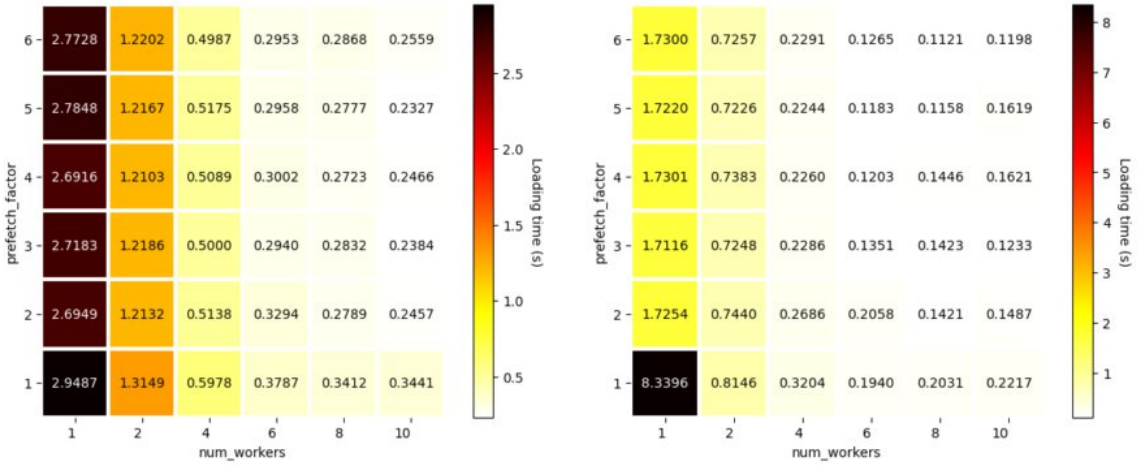
Here we ran the `dlojz.py` script on 50 iterations on a GPU of the `gpu_p4` (A100 PCIe) partition.

We see the loading time and the data transformations (`DataLoader`) in relation to the storage technology used, the number of workers and the `prefetch_factor`.

We observe that the time is divided in half when we store the database on the SCRATCH rather than on the WORK.

Various disk spaces

dlojz.py - 50 iterations - test partition gpu_p4



SCRATCH
500 GB/s - OPA

÷ 2

NVMe
PCIe

We observe that the time is divided by 2 when we store the database on NVMe rather than on the SCRATCH.

Various disk spaces

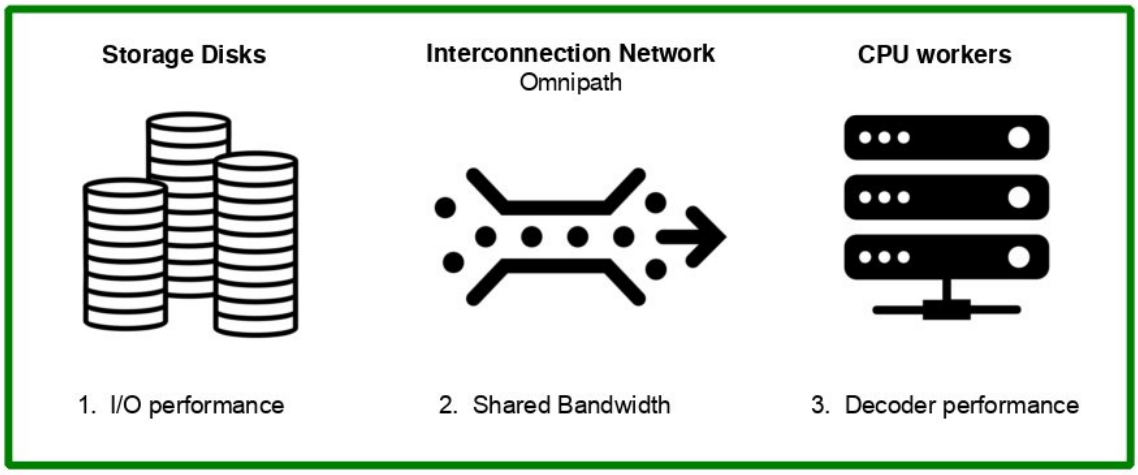
- **NVMe**
  - ✓ Best IO performance
  - – You need to copy your dataset on the local disk first, which can take a very long time
  - – This solution is not suitable at the scale of a supercomputer so it is not available to users
- **SCRATCH**
  - ✓ Second best IO performance
  - ✓ Very large quota (bytes and inodes)
  - – 30 days file lifespan
  - – Not backed up
- **WORK / DSDIR**
  - – Worst performance (but it is still acceptable)
  - – Only 5 TB and 500k inodes
  - ✓ IDRIS support team manages the dataset for you in the DSDIR (downloading, preprocessing,...)
  - ✓ Backed up

The advantages and disadvantages of the different storage spaces.

Dataset optimization

- Main bottlenecks
- Data storage – various disk spaces
- **Data format – at sample level**
- Data format – at dataset level

Bottlenecks upstream of DataLoader



Which format for my data?

The input data format will have an impact on all the steps.

At sample level - Sample decoding

Binary format: Pickle format, hdf5,...
Decoded more quickly, takes more space

- ✓ Decoder performance
- – Shared bandwidth
- – Storage volume

Compressed format: jpeg, png,...
Decoded more slowly, takes less space

- – Decoder performance
- ✓ Shared bandwidth
- ✓ Storage volume

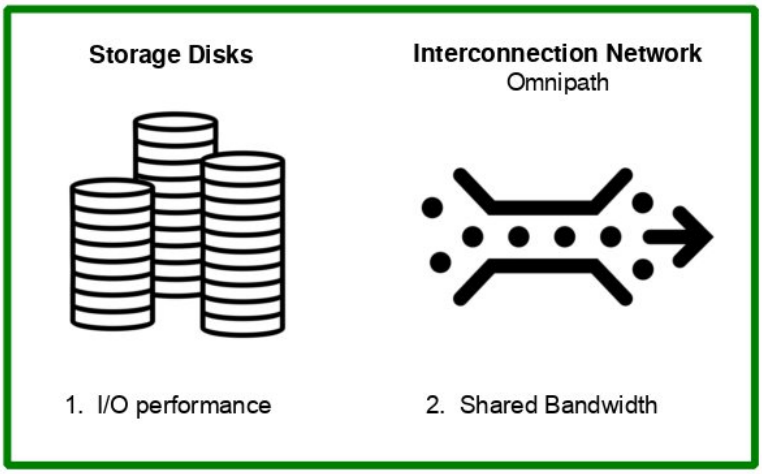
The binary formats enable a rapid decoding on CPU but use more disk space (~10x) and, therefore, more OPA bandwidth.

The compressed formats reduce the necessary storage volume and use less OPA bandwidth but the decoding time on CPU is longer.

Dataset optimisation

- Main bottlenecks
- Data storage – various disk spaces
- Data format – at sample level
- **Data format – at dataset level**

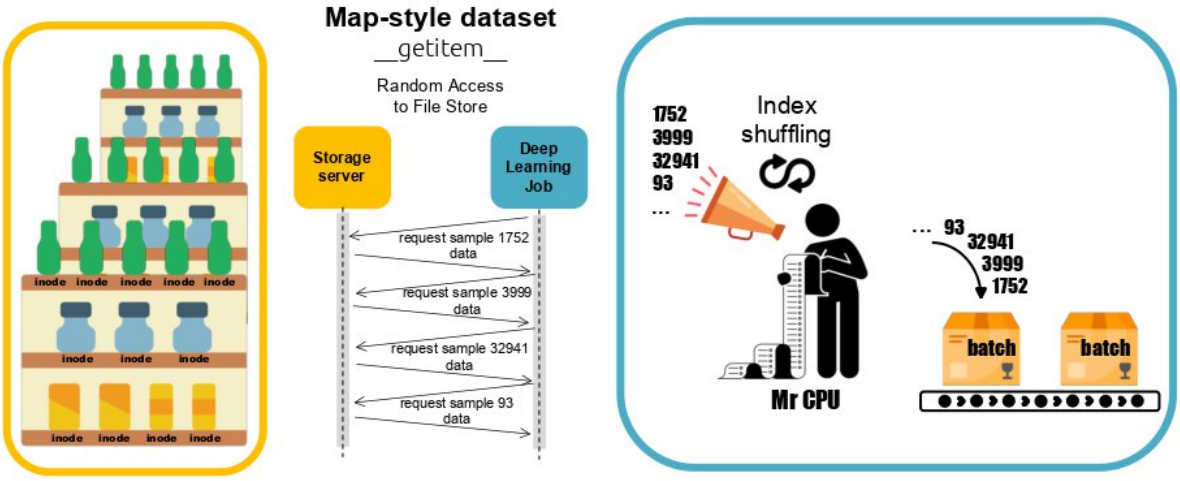
Bottlenecks upstream of DataLoader



Which format for my dataset?

Which format should I use for my database?

Intuitive way



Pros: Easy to handle, random access possible
Cons: Lots of inodes, lots of I/Os

Here: 1 image = 1 file.

A map-style dataset corresponds to a dictionary (idx, image).

We access one data at a time from its index.

We generate one I/O request per each accessed data.

The shuffling is carried out on the indexes of the whole dataset before creating the batches.

Too many inodes is an issue

Error: Disk quota exceeded

Reminder:

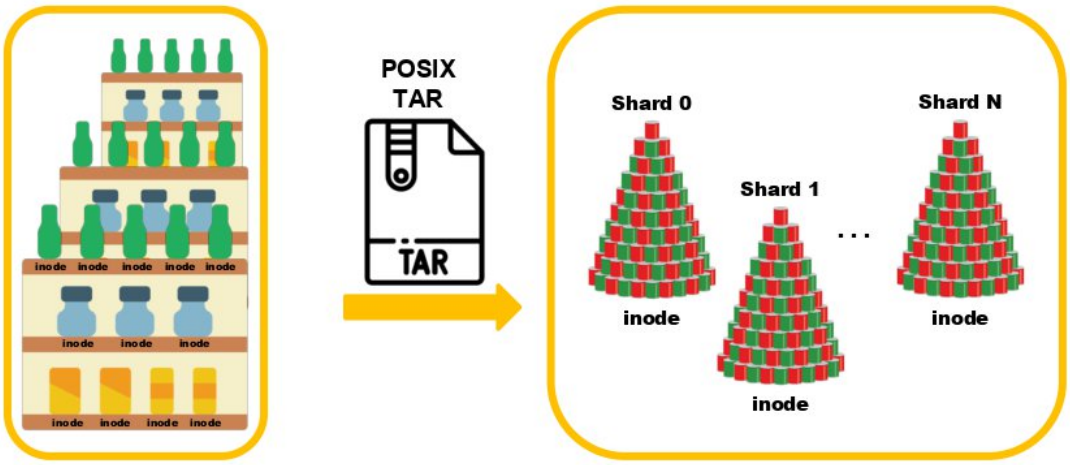
- $WORK quota per user is 5 TB / **500 kinodes**
- $SCRATCH safety quota per user is 250TB / **150 Minodes**

+ IBM Spectrum Scale file system does not like small file I/O intensive workloads

Reminder of the inode quotas on the Jean Zay disk spaces.

Originally, the IBM Spectrum Scale file system was designed to optimize parallel access to the same large file. It is adapting to new practices but is still suboptimal for I/O-intensive workloads on small-size files.

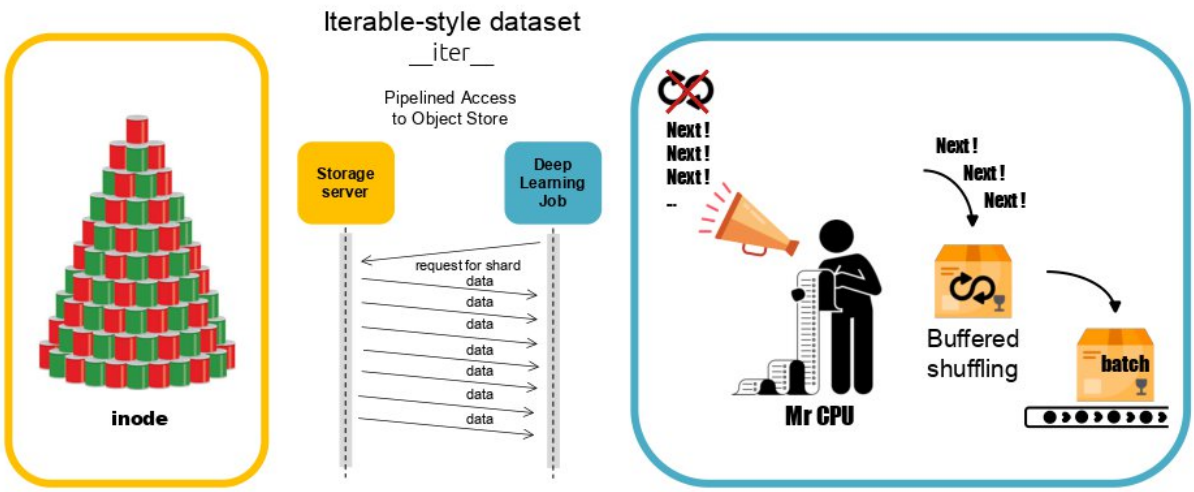
WebDataset format – Gathering inodes



We present the alternative format, WebDataset, in order to minimize the number of inodes.

The idea is to group the data in archive form.

WebDataset format – Iterable dataset



Pros: Fewer I/Os, fewer inodes
Cons: Difficult to shuffle or distribute, unknown dataset length

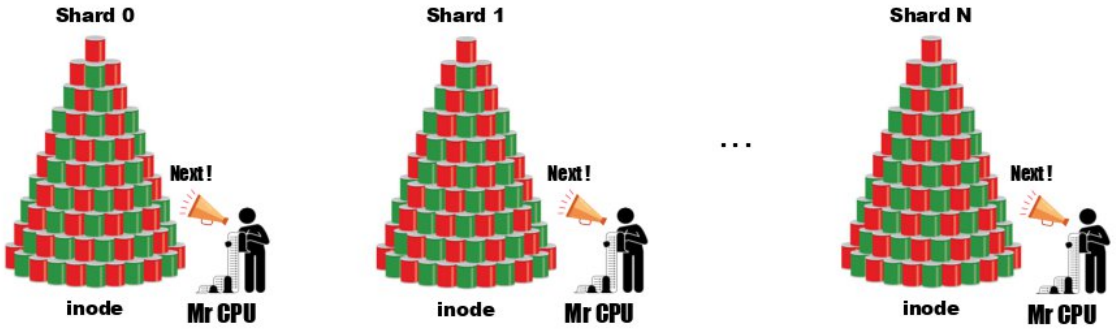
WebDataset is an iterable-style dataset. We access data contiguously. There is no mapping (idx, image).

This style of dataset is adapted to databases whose structure is not known beforehand. This is the case here because the data are « hidden » in the archives.

Shuffling is managed by the CPU at the level of a buffer. The size of this buffer is chosen by the user (proportionate to the batch size, for example).

WebDataset format - Sharding

Sharding is necessary to benefit from parallel implementation
(DataLoader multi-processing and Distributed Data Parallelism).



The number of shards should be a multiple of the number of tasks/GPUs.

19

1 shard = 1 archive.

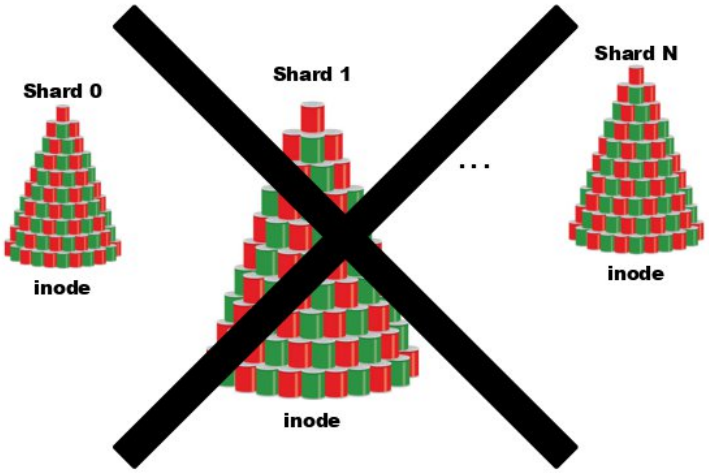
Sharding consists of dividing the original dataset on multiple archives. A minimum number of archives is necessary to generate parallelism at the I/O level.

For example, 2 DataLoader workers will access two different archives.

Likewise, 2 GPUs must be supplied by different archives.

For example: nshards = ngpus x nworkers

WebDataset format - Sharding

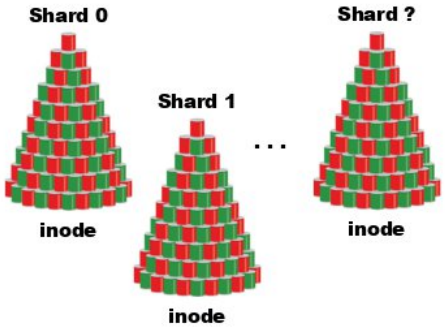


Samples must be evenly distributed among the shards to balance the workload between processes.

For the workload to be optimally distributed on the different processes, the data must be distributed as equally as possible in the archives.

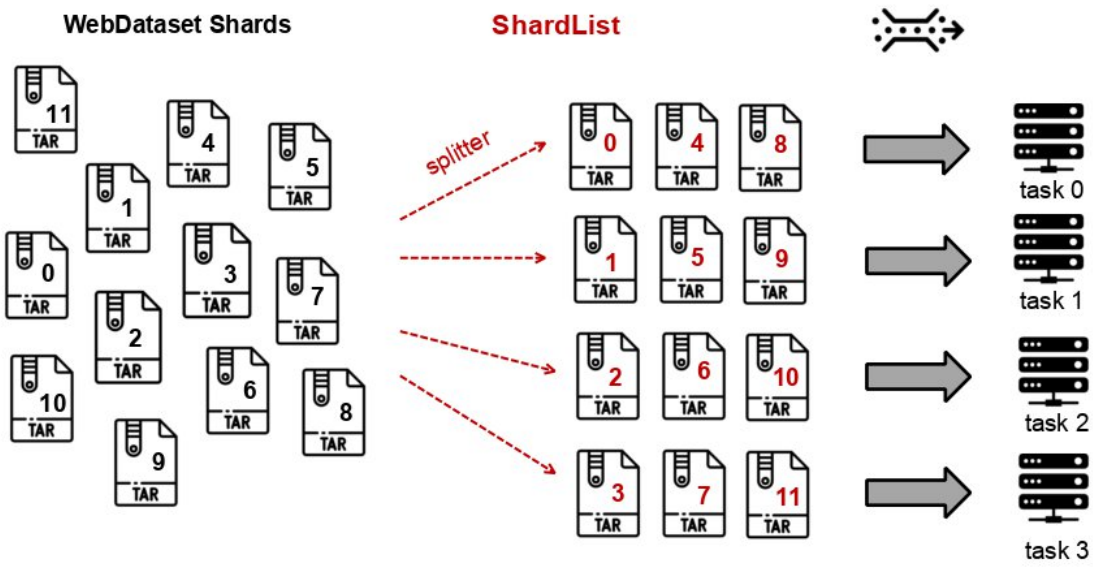
WebDataset format - More or less shards?

	More shards	Less shards
Large scale distribution	+	-
Shared bandwidth	+	-
Inodes quota	-	+
Number of I/O	-	+



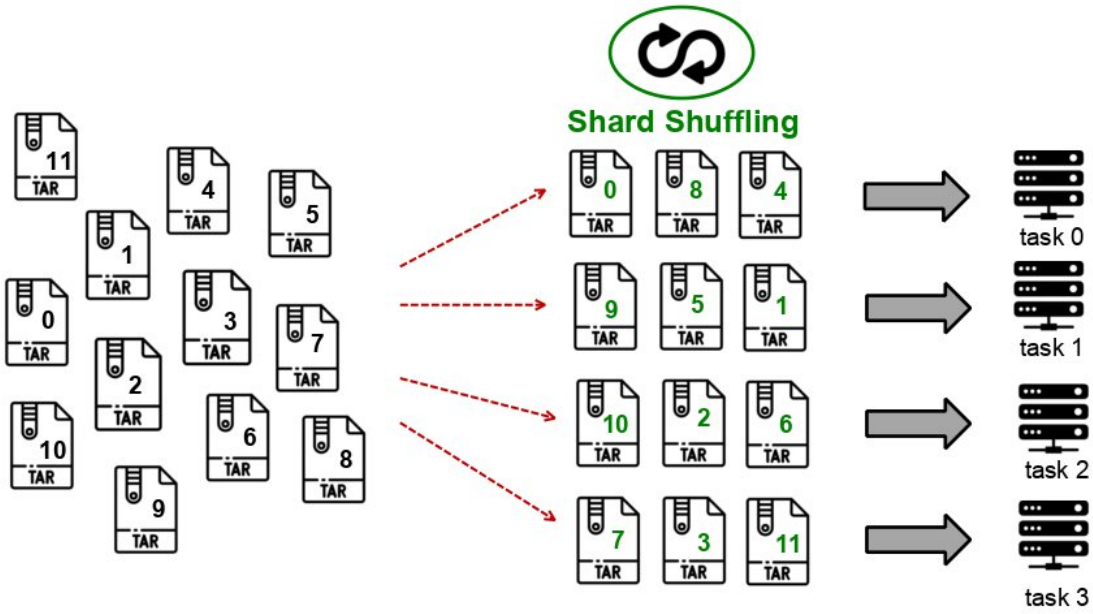
The advantages and disadvantages of having a dataset with more/less shards.

WebDataset – Multiworker sharding



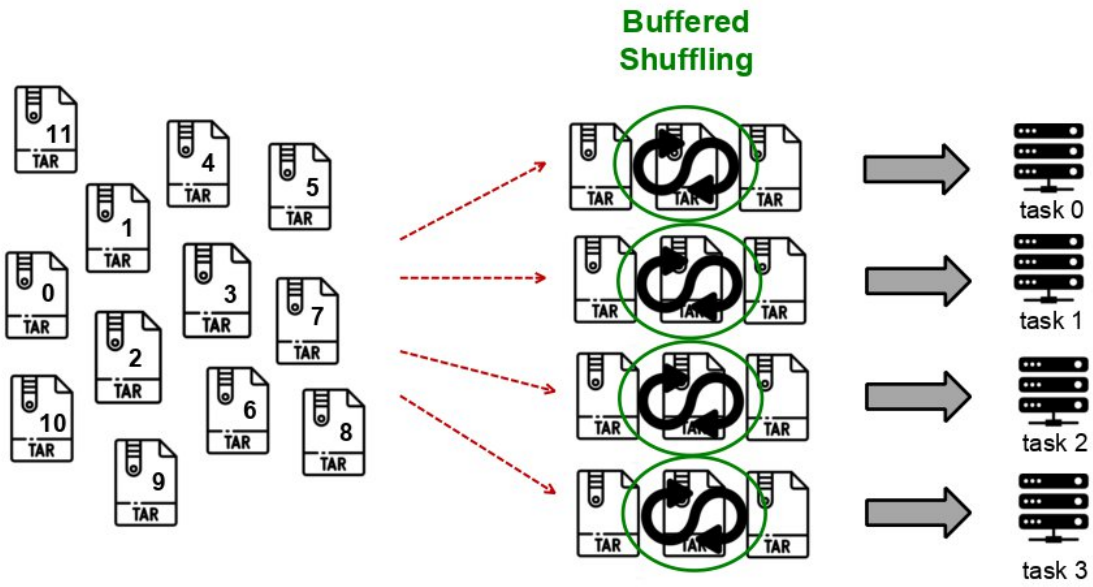
The shards are distributed on different processes according to the splitter function.

WebDataset - Shuffling



We can activate the first shuffling on the shard indexes. This shuffling is performed per process.

WebDataset - Shuffling



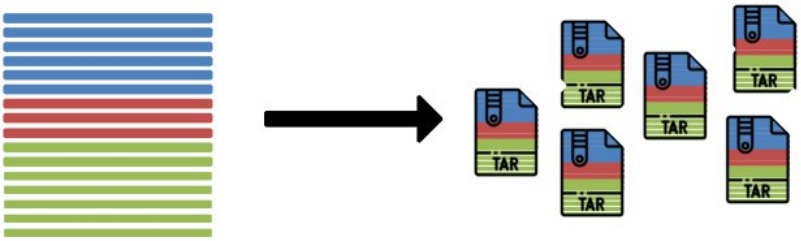
We can also activate a shuffling at the level of all the data contained on a process.

Combining the two types of shuffling (shard shuffling and buffered shuffling) is the most effective method.

WebDataset - Generation

When generating WebDataset shards, don't forget to:

- Distribute the samples as **evenly** as possible among the shards.
- Choose the number of shards **according to the number of GPUs** you will use.
- Distribute the samples so that each shard contains a **representative part of the dataset.**



+ Converting data before creating the archives to improve decoding performance?

Some advice for generating a dataset of the WebDataset type.

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```


WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

distribute shards among processes

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

shuffling shards indexes per process

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

shuffling samples per process

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

description of shard content

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

transforming and batching

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

define len(train_dataset)

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

batching handled by WebDataset class

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

usual DataLoader args

WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                               .shuffle(1000)\
                               .decode("torchrgb")\
                               .to_tuple('input.pyd', 'output.pyd')\
                               .map_tuple(transform, lambda x: x)\
                               .batched(mini_batch_size)\
                               .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

drop_last equivalent

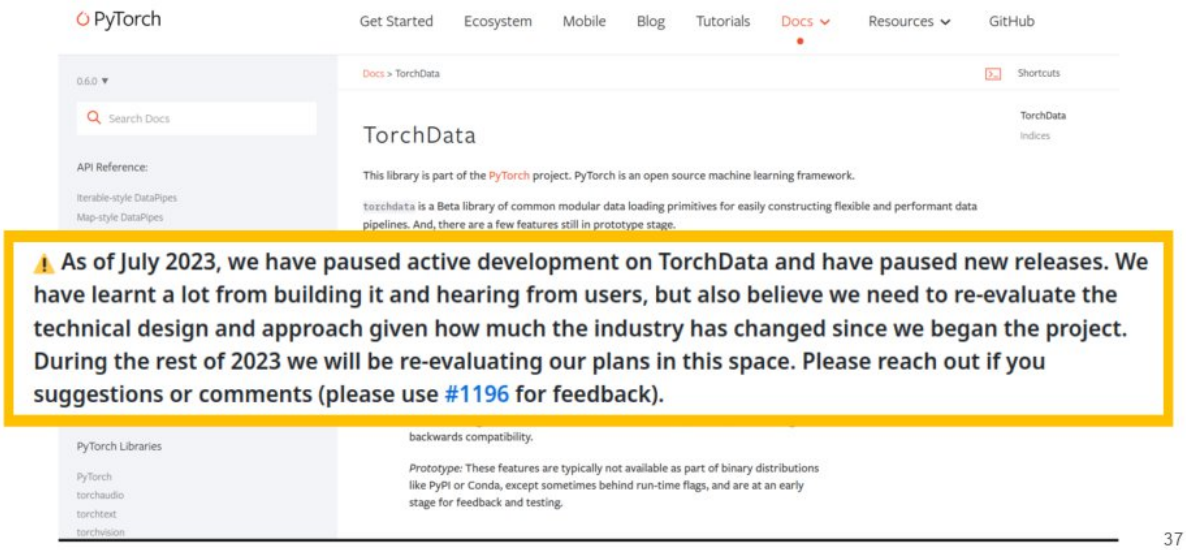
WebDataset - Implementation

```
import webdataset as wds

def my_splitter(paths):
    paths = list(paths)
    return paths[idr_torch.rank::idr_torch.world_size]

paths = os.environ['DSDIR']+'/imagenet/webdataset/imagenet_train-{000000..000127}.tar'
train_dataset_len = 1281167
train_dataset = wds.WebDataset(paths, nodesplitter=my_splitter, shardshuffle=True)\
                                .shuffle(1000)\
                                .decode("torchrgb")\
                                .to_tuple('input.pyd', 'output.pyd')\
                                .map_tuple(transform, lambda x: x)\
                                .batched(mini_batch_size)\
                                .with_length(train_dataset_len)

nbatches = train_dataset_len // global_batch_size
train_loader = wds.WebLoader(train_dataset, batch_size=None,\
                             num_workers=num_workers,\
                             persistent_workers=persistent_workers,\
                             pin_memory=pin_memory,\
                             prefetch_factor=prefetch_factor\
                             ).slice(nbatches)
train_loader.length = nbatches
```

define len(train_loader)

Soon in TorchData?

TorchData url



It is expected that the WebDataset format will be integrated into the PyTorch library via a new class in construction: TorchData.

This new class should simplify the whole data pipeline, from reading to pre-processing (DataLoader).

Unfortunately, development of this class has been paused since July 2023.

WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

```
start_time = datetime.datetime.now()

for i, (images,labels) in enumerate(loader):
    print(f'{i} / {nb_batches}', end="\r")

end_time = datetime.datetime.now()
delta_time = (end_time_it - start_time_it).total_seconds()
```

- Execution on 1 GPU

38

On a Resnet50/Imagenetcomplete training, no advantage is seen in using the WebDataset format.

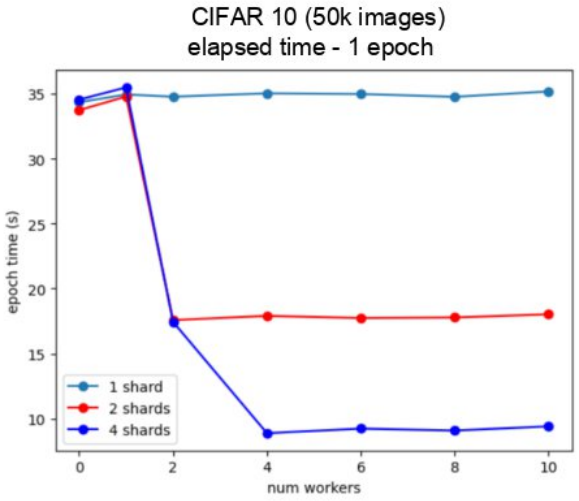
I have reduced the test to a simple dataset path to isolate the I/0 performance. This test is executed on only 1 GPU.

WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

CIFAR10 ~ 50k images

- Sharding is necessary to benefit from parallel implementation (DataLoader multi-processing).



Test on CIFAR10.

Observe that the more shards there are, the more we generate parallelism.

Here we are using a small dataset to be able to rapidly generate WebDataset versions with a higher or lower number of shards.

In reality, using WebDataset on such a small dataset deteriorates the I/O performances.

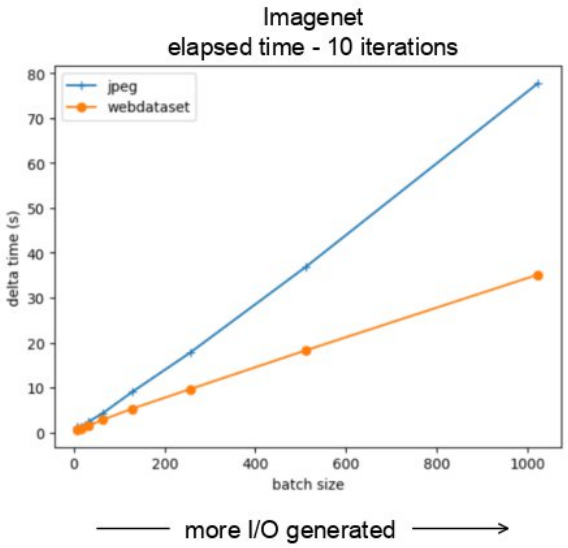
The next tests will be conducted on Imagenet.

WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

Imagenet ~ 1.3M images
128 shards ~10k images per shard (+labels)
1 shard (images + labels) ~ 6GB

- The more samples are needed per batch, the more efficient is the WebDataset format (fewer I/Os).



Test sur Imagenet.

In WebDataset format on128 shards, Imagenet weighs 826GB (versus 153GB for the jpeg version).

When increasing the batch size, we increase the number of I/O requests.

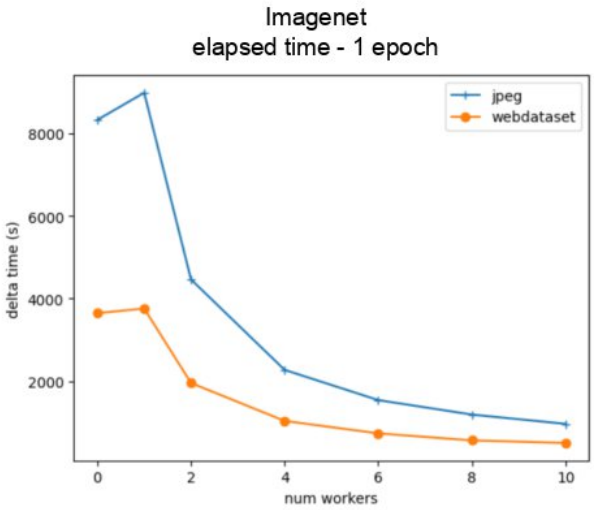
We see that with the WebDataset format, a decrease in the number of I/Os optimizes the code speed.

WebDataset - Perfomance test

I/O loop over the dataset
(calculation-free iterations)

Imagenet ~ 1.3M images
128 shards ~10k images per shard (+labels)
1 shard (images + labels) ~ 2GB

- The WebDataset format scales up.



We increase the number of workers to verify the scalability of the WebDataset version.

WebDataset - Perfomance test

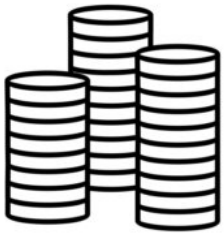
A complete training over the Imagenet dataset (dlojz.py)

	Original jpeg dataset	WebDataset format
Elapsed time (41 epochs)	30min43s	29min56s
Test accuracy	72%	72%

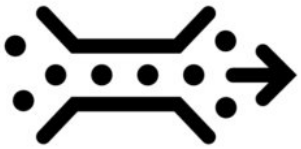
On a complete Resnet50/Imagenet training, we have verified that using the WebDataset format does not degrade the performances, either in time or in accuracy.

Conclusion

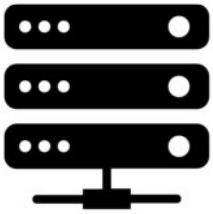
Storage Disks



Interconnection Network
Omnipath



CPU workers



1. I/O performance
2. Shared Bandwidth
3. Decoder performance

- Disk spaces: WORK / DSDIR or SCRATCH
- Data format: binary or compressed
- Dataset format: alternative format like WebDataset

Appendix A – HuggingFace Datasets

Hugging Face Hub 

`dataset = load_dataset("dataset_name")`, get any of these datasets ready to use in a dataloader for training/evaluating a ML model (Numpy/Pandas/PyTorch/TensorFlow/JAX) - **from remote access or from local copy**.

Two types of dataset objects: **Dataset** or **IterableDataset** .
- **IterableDataset** is ideal for big datasets (think hundreds of GBs!)
- **Dataset** is great for everything else.

General :
- In-memory data (dictionary, Pandas DataFrames, generator)
- CSV
- JSON
- Parquet
- Arrow
- SQL
- **WebDataset**

Audio :
- Local Files Dictionary
- AudioFolder
- AudioFolder with metadata

Text :
- Text Files list
- TextFolder

Vision :
- Local Files Dictionary
- ImageFolder
- **WebDataset**

Tabular :
- CSV files
- Pandas DataFrames
- Databases (SQLite, PostgreSQL)

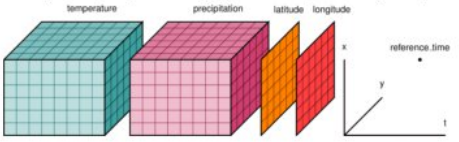
https://huggingface.co/docs/datasets/index

Appendix B – ESPRI-IA Use Case

Context : Large Training Scientific Dataset

NetCDF (network Common Data Form) is a file format for **storing multidimensional scientific data** (variables) such as temperature, humidity, pressure, wind speed, and direction.

Xarray is a library for working with domain-agnostic data-structures, labeled arrays, NetCDF, Zarr, ...



Test :

Storage format : Numpy, HDF5, WebDataset, Zarr

Zarr is a high-level storage format
Dataset-level abstraction with indexing

High-performance Compressor : BLOSC + LZ4

BLOSC is a meta-Compressor

Slide: https://espri.ipsl.fr/wp-content/uploads/2023/12/storage_formats.pdf
video: https://www.youtube.com/watch?v=w8TJcBf87zw

Appendix B – ESPRI-IA Use Case

Conclusion:

with BLOSC + LZ4, loading (I/O + com + decoding) compressed data is faster than loading uncompressed data!
Recommended for WebDataset and Zarr!

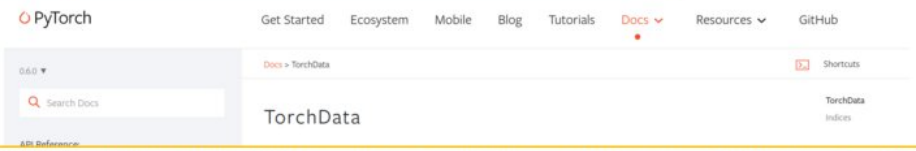
For Short Dataset:
Numpy/Pickle is the best suitable storage format !!

For Long Dataset:
- Map style: **Zarr >** HDF5
- Iterable: **WebDataset >** Zarr ≈ HDF5

Slide: https://espri.ipsl.fr/wp-content/uploads/2023/12/storage_formats.pdf
video: https://www.youtube.com/watch?v=w8TJcBf87zw

Annex – Attempt at Standardization

- TorchData?



⚠️ **As of July 2023, we have paused active development on TorchData and have paused new releases. We have learnt a lot from building it and hearing from users, but also believe we need to re-evaluate the technical design and approach given how much the industry has changed since we began the project. During the rest of 2023 we will be re-evaluating our plans in this space. Please reach out if you suggestions or comments (please use #1196 for feedback).**

- MLCommons/Croissant

Croissant 🥐 is a high-level format for machine learning datasets that combines metadata, resource file descriptions, data structure, and default ML semantics into a single file; it works with existing datasets to make them easier to find, use, and support with tools.

Croissant builds on schema.org, and its Dataset vocabulary, a widely used format to represent datasets on the Web, and make them searchable.

Croissant is currently under development by the community. **Since Jul. 2023**