PARIO COURSE – PYTHON ML/DL (1/2)

2024-11-05 | Jan Ebert | `ja.ebert@fz-juelich.de` | Helmholtz AI Consultant | Jülich Supercomputing Centre

Part I: Outline

Outline

Topics and target audience (1/2)

First session:

1. random access patterns, memory-mapped files (`mmap`)
2. very large data
3. virtual file systems

Topics and target audience (1/2)

First session:

1. random access patterns, memory-mapped files (`mmap`)
2. very large data
3. virtual file systems

Second session:

1. general data loading optimizations, host-to-device transfers
2. efficiently saving program state
3. metric logging

Topics and target audience (2/2)

We will be using PyTorch, a scientific computing library. "NumPy on the GPU, with automatic differentiation." (NumPy is Python's de facto multi-dimensional array standard library.) TensorFlow and JAX APIs will also be addressed, concepts transfer.

Topics and target audience (2/2)

We will be using PyTorch, a scientific computing library. “NumPy on the GPU, with automatic differentiation.” (NumPy is Python’s de facto multi-dimensional array standard library.) TensorFlow and JAX APIs will also be addressed, concepts transfer.

Parallel writing of datasets is not part of these sessions!

Part II: Data formats

Parallel I/O in Machine Learning

Many algorithms used in machine learning (ML)… We will focus on gradient descent:

1. Shuffle dataset.
2. Model trains iteratively on distinct subsets of the full dataset (“mini-batches”).
3. When dataset is exhausted, go to step 1.

Gradient descent really likes to have differently shuffled iid. data each training epoch.

Data in Machine Learning

Data used in ML/deep learning (DL) comes in all kinds of shapes and forms. Images, text documents, spectra, graphs, ...

Data points in a single dataset may be:

- structured and same length. Example: same-format images
- structured and different lengths. Example: genomic data
- (unstructured and same length)
- unstructured and different lengths. Example: composition of differently structured datasets

Desiderata

What do we need from a good, (general-purpose) ML data format?

- Fast random access
- Ease of use
- Parallel writing
- (Good performance with and support for variable-length data)
- Portability

Data formats

FL = fixed-length

VL = variable-length

	Directory	HDF5	Arrow	Parquet	Custom binary
Fast FL random access	?	Y	Y	Y	Y
Fast VL random access	?	N	Y	Y	Y
FL usability	Y	Y	Y−	Y−	?
VL usability	Y	Y−	Y	Y	?
FL parallel writing	Y	Y	Y−	Y−	Y
VL parallel writing	Y	N	Y−	Y−	?
Single file	N	Y	?	?	Y
Has specification	N	Y	Y	Y	N

On JUST, files don't scale well, but have become much more lenient than in the past.

Arrow and Parquet don't support parallel writing to a single file.

Interlude: Memory-mapped files

- Usually: each read/write requires a `read` / `write` system call.
- `mmap` : one system call to associate RAM section with disk memory.
- Mapped sections always align to page size. They page disk memory blocks in/out as required.
  $\rightarrow$ Small files will still take a lot of space due to alignment.
  $\rightarrow$ Reduce system calls, instead just change out memory.
- Single-node: safely share memory between different processes.
  Multi-node: care required to prevent race conditions if writing.
- Different from memory-mapped I/O.

Part III: Data loaders under the hood

Libraries and data loaders

- **PyTorch**: `torch.utils.data.DataLoader`
- **TensorFlow/JAX**: `tf.data`

Worth mentioning: HuggingFace `datasets` as cross-framework abstraction.

How they work

Data loaders are lazy. Things only happen when they need to, in the last moment possible. How they work also depends on the type of dataset used.

In PyTorch:

- `torch.utils.data.Dataset`: “map-style”, indexable (`__getitem__`), has a length (`__len__`)
- `torch.utils.data.IterableDataset`: “iterable-style”, iterable (`__iter__`), length optional (could be unknown or ∞!)

In TensorFlow: always “iterable-style” with optional length.

Per-element processing only occurs when we actually index/iterate to a sample.

Multiprocessing and Pickle

In PyTorch, if `num_workers > 0`: upon iterating the dataloader, subprocesses are started.

Multiprocessing and Pickle

In PyTorch, if `num_workers > 0`: upon iterating the dataloader, subprocesses are started.

Can cause issues with dataset's attributes and…

- Pickle protocol. E.g., in old versions of NumPy, arrays read from memory-mapped files are `pickle`d as in-memory arrays.
  However: situation has gotten much better than in the past.
- Python objects and reference counting.

Multiprocessing and Pickle

In PyTorch, if `num_workers > 0`: upon iterating the dataloader, subprocesses are started.

Can cause issues with dataset's attributes and…

- Pickle protocol. E.g., in old versions of NumPy, arrays read from memory-mapped files are `pickle`d as in-memory arrays.
  However: situation has gotten much better than in the past.
- Python objects and reference counting.

Recommendation: do not keep files open after `__init__`, only do it in `__getitem__`/ `__iter__` because by then the subprocesses have been started.

Multiprocessing and Pickle

In PyTorch, if `num_workers > 0`: upon iterating the dataloader, subprocesses are started.

Can cause issues with dataset's attributes and...

- Pickle protocol. E.g., in old versions of NumPy, arrays read from memory-mapped files are `pickle`d as in-memory arrays.
  However: situation has gotten much better than in the past.
- Python objects and reference counting.

Recommendation: do not keep files open after `__init__`, only do it in `__getitem__`/ `__iter__` because by then the subprocesses have been started.

Always be aware of this! Memory will be replicated if not shared properly.

Data sharding APIs

Map-style: each process gets its shard's indices.

Iterable-style:

- PyTorch: manually implemented by user; simplest solution: iterate full dataset, but discard superfluous data (i.e., other shards).
- TensorFlow: each process iterates full dataset, but discards superfluous data (i.e., other shards).

Part IV: One-pot solutions

One-pot solutions

Let’s say

- we have an existing dataset with tons of files,
- an existing code base,
- we are on a file system with very slow metadata queries, and
- we are really lazy.

One-pot solutions

Let’s say

- we have an existing dataset with tons of files,
- an existing code base,
- we are on a file system with very slow metadata queries, and
- we are really lazy.

All of the following, including templates, also documented at:
`https://sdlaml.pages.jsc.fz-juelich.de/ai/guides/handling_large_datasets/`

Similar guides:
`https://go.fzj.de/ai-jsc`

One-pot solution for in-memory data

We can avoid I/O concerns and be really fast if our data fits into RAM.
One-pot solution:

1. `tar` the dataset.
2. `untar` the dataset into `/dev/shm`.
3. Read data from `/dev/shm`.

`/dev/shm` is an interface to shared memory in Linux. Optional kernel feature, so availability depends.

One-pot solution for on-disk data

Data doesn’t fit in RAM? Set up a filesystem in userspace (FUSE)!
Cheat solution:

1. Archive data into a format with FUSE support (e.g., SquashFS).
2. Mount the data into a virtual filesystem with FUSE.
3. Set up an exit handler that unmounts the FUSE.
4. Read data from mount point.

Lots of variance in library performance, especially concerning parallel access.

On our systems: SquashFS usually already installed (currently missing due to a recent update), so usually requires no additional setup; okay performance. Mount has to be in `/dev/shm` or `/tmp`.

Part V: Exercises

Setting up

1. Log into the supercomputer JUSUF (`ssh <user>@jusuf.fz-juelich.de`).
2. `cd /p/project1/training2403/ParIO_course_material/exercises/Python_ML_DL`
3. `nice bash set_up.sh` and wait until done. In the meantime, feel free to look into exercise 1.1 in the same directory!
4. Every time you (re-)connect to the machine and want to do the Python ML/DL exercises, execute the following to activate the software environment:

   ```
   cd /p/project1/training2403/ParIO_course_material/exercises/Python_ML_DL
   source activate.sh
   ```

5. Exercises should be executed like `sbatch <file>.sbatch [args...]`.

Exercise 1.1

We introduced a bunch of data formats. Now it's on you to benchmark a toy example! Important arguments: `--data-root`, `--format`, `--num-samples`.

1. How does memory mapping affect performance? (`--use-mmap`)
2. How does assuming fixed-length data affect performance? (`--fixed-length`)
3. How does shuffling affect performance? (`--shuffle`)
4. How does parallel reading affect performance? (`--num-workers`)
5. Explore other parameters if you feel like it!

The dataset contains fake image data and labels. All data formats use the same serialized NumPy array storage.

Exercise 1.2

You're given a dataset containing numbers like "01", "02", ...

1. Use the PyTorch API to shard a map-style dataset.
2. Manually modify the implementation to shard an iterable-style dataset.

Data should be sharded so that each process gets one element in turn, starting with process 0. By default, tests will be executed for sharding with 4 processes. Feel free to change this; you can also let the code run multi-node.

Hints:

1. `torch.utils.data.DistributedSampler` (by default, `shuffle=True`)
2. `itertools.islice`

Exercise 1.3 (bonus)

Let's use the `/dev/shm` one-pot solution to iterate our data quickly in-memory and without much extra work.

A basic Python code skeleton based on the "directory" format from exercise 1.1 exists. By default the code reads from the global file system. Your task is to add the taring and untaring so we read from `/dev/shm` instead, requiring no code changes other than returning a different path..

Hints:

1. `tarfile` is a helpful package for working with tar files in Python's standard library.
2. `shutil` has some file operation utilities (you may not need this).
3. Once the basics are done, you may want to make your (very sensitive) data in `/dev/shm` private, so that future users won't be able to read it.