Fast Data Loaders

Inar Timiryasov (NBI)
inar.timiryasov@nbi.ku.dk

June 2, 2025

Between Models and Reality: School on Machine Learning in Physics
NBI, Copenhagen

ML vs Physics Projects

- ML projects differ from the physics ones.
- In physics, one can perform a calculation and exclude or confirm an idea.
- The model not performing well could mean anything (hyperparameters, bugs, tensor shapes, etc).
- Progress requires successive experiments.

Neural net training is a leaky abstraction

Andrej Karpathy, A Recipe for Training Neural Networks

Data is the Key

- Having your data in a suitable form is key.

- A good data loader allows for rapid experimentation.

- For perspective, big labs have they own **filesystems**!
  https://github.com/deepseek-ai/3FS

Transformer

CLASS torch.nn.Transformer(*d_model=512*, *nhead=8*, *num_encoder_layers=6*, *num_decoder_layers=6*, *dim_feedforward=2048*, *dropout=0.1*, *activation=<function relu>*, *custom_encoder=None*, *custom_decoder=None*, *layer_norm_eps=1e-05*, *batch_first=False*, *norm_first=False*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

A transformer model.

https://docs.pytorch.org

ML Progress and Scale

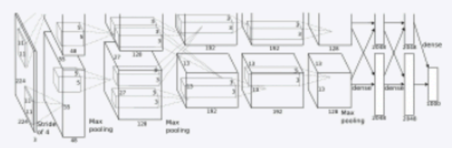
- Scale is crucial for ML models

- Large model — GPU parallelism (data, pipeline, tensor)
  - Data parallelism evenly distributes data across multiple GPUs.
  - Model parallelism distributes a model across multiple GPUs.
  - Tensor parallelism distributes large tensor computations across multiple GPUs.

- Large data — quickly loading it and transferring on GPU(s) is critical.

  If your GPU is sitting idle waiting for data, you're wasting resources and time.

AlexNet



Developer(s)	Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton
Initial release	Jun 28, 2011
Repository	code.google.com/archive/p/cuda-convnet/
Written in	CUDA, C++
Type	Convolutional neural network
License	New BSD License

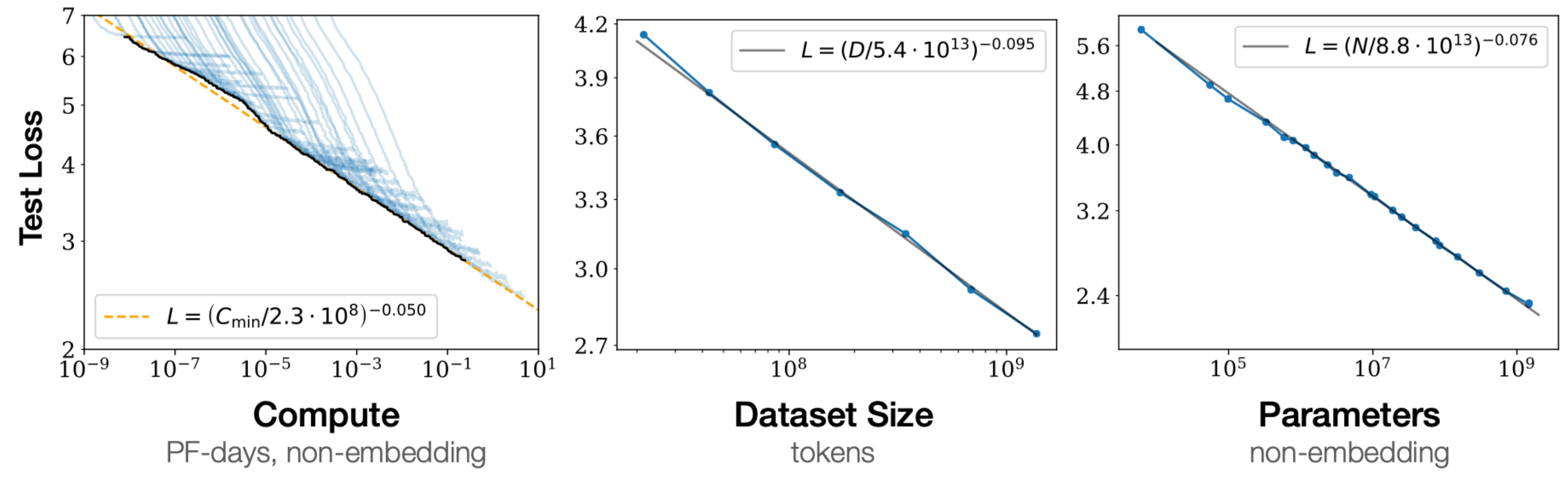


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

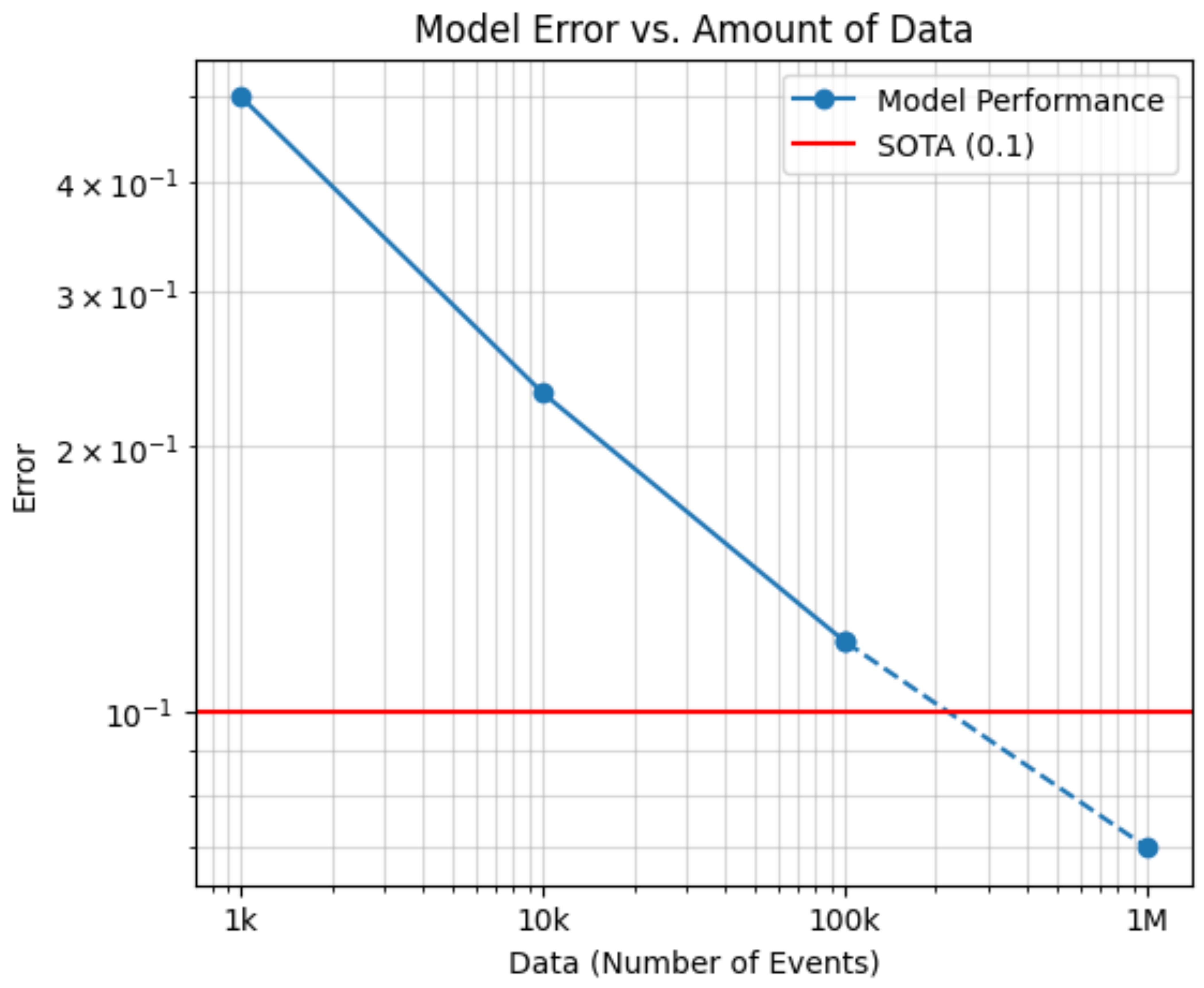
Kaplan et al

Fast Data Loaders: Keeping the GPUs Fed

Sweat, Blood, and Data Loading

A Fictional Story

- Came up with a brilliant ML for physics idea.
- Vibe coded a new model, and it works!
- It takes **one day** to train on 100k events.
- And it is almost SOTA!
- Submit an abstract to a conference. The conference is in **three months**.
- Plenty of time to train on 1M events. Should take **10 days**, right? Right?



A Fictional Story

- Start training on 1M events.
- Now it takes 100 days 😱
- Why??

```
Epoch 1:    0%|          | 9/78125 [0:16:53<100.72 days, 111.39 s/it, loss=0.223]
```

The I/O Bottleneck

- **Definition**: The **I/O (Input/Output) bottleneck** happens when your system's ability to read/write data is slower than its ability to process that data.

- **The Core Issue:** Your powerful CPU or GPU is often idle, waiting for data to be loaded from storage (like an SSD/HDD) or transferred to its memory.

- **Key Symptoms:**
  - **Low CPU/GPU utilization** during computationally intensive phases.
  - Tasks take much longer to complete than theoretically expected.
  - Data loading/preprocessing steps visibly consume a large portion of the total time.

GPU Utilization

`nvidia-smi` terminal command is your friend!
(System Management Interface)

```
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3090        Off |   00000000:21:00.0 Off |                  N/A |
| 71%   67C    P0             283W /  350W |    4733MiB /  24576MiB |     54%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

- **GPU-Util:** Percentage of time the GPU's processing cores were actively computing.
  - Aim for consistently high values (e.g., >90%) during intensive training.
  - Low Util (like 54%): Strong indicator the GPU is often idle, typically waiting for data (I/O bound) or CPU tasks.
  - **Caveat:** 100% util doesn't always mean *peak theoretical* performance. Throughput can still be limited by memory bandwidth bottlenecks or suboptimal kernel execution.

- **Memory-Usage:** Shows GPU video RAM (VRAM) currently allocated versus total available

- **Perf:** The GPU's current performance state. P0 means maximum performance. Other states (e.g., P2, P8) mean reduced performance/power.

- **Pwr:Usage/Cap:** Current GPU power consumption versus its maximum rated capacity.

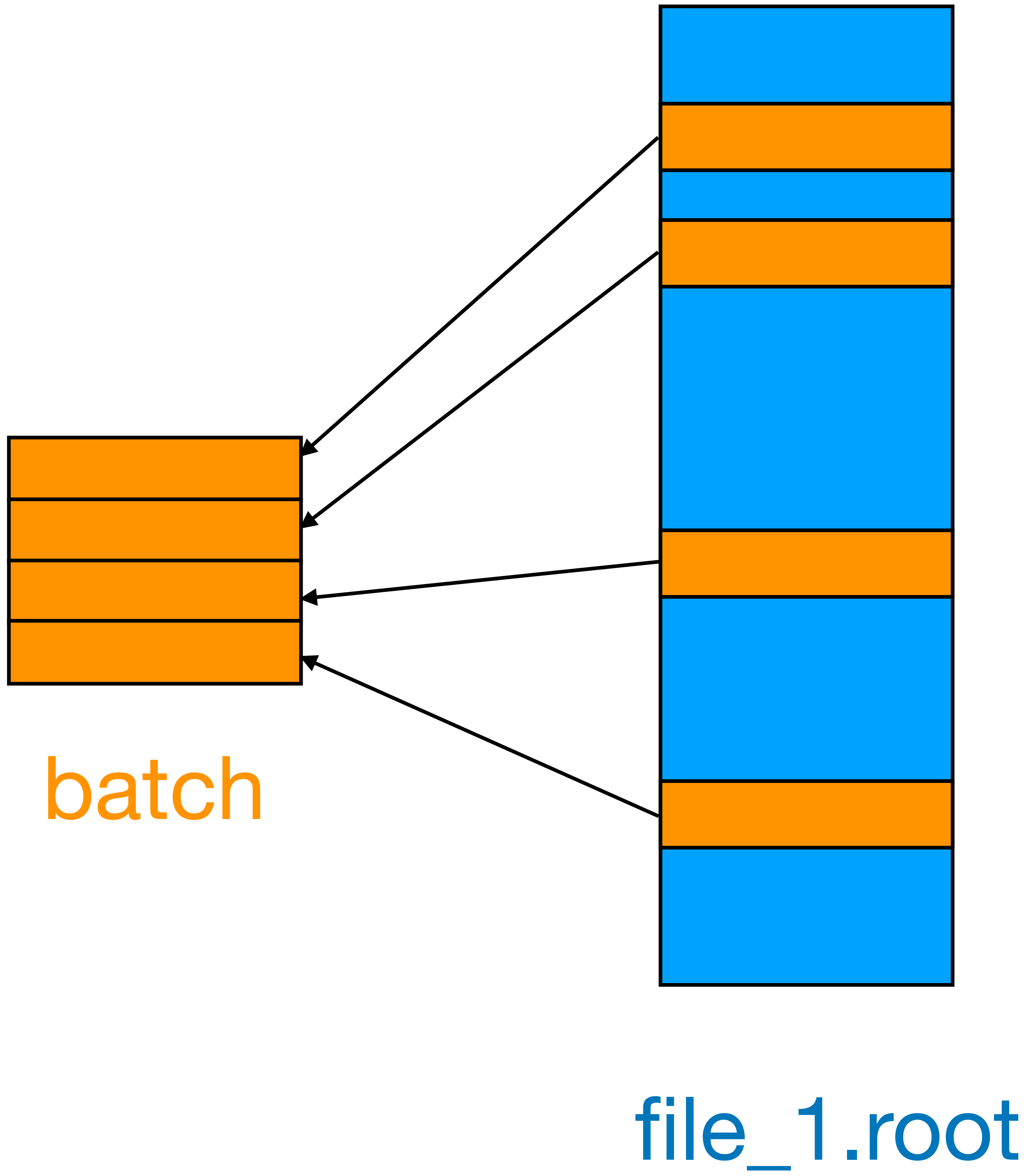
GPU Utilization

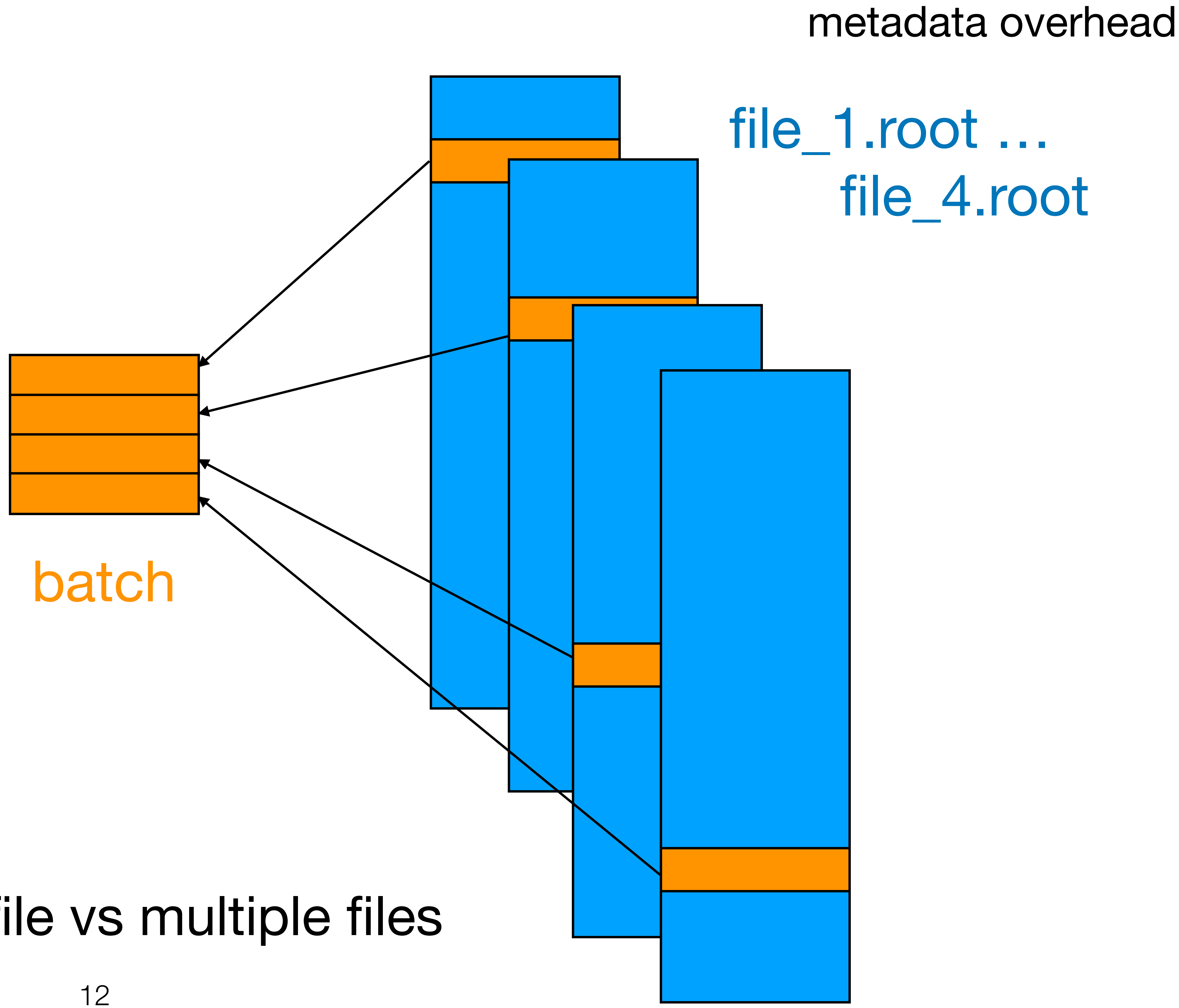
```
+-----------------------------------------------------------------------------------------+
(base) [inar@hep04 ~]$ nvidia-smi
Tue May 20 17:52:08 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 555.42.06              Driver Version: 555.42.06      CUDA Version: 12.5     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3090        Off |   00000000:21:00.0 Off |                  N/A |
| 30%   35C    P8             10W /  350W |       4MiB /  24576MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 3090        Off |   00000000:4D:00.0 Off |                  N/A |
| 47%   52C    P0            149W /  350W |    4276MiB /  24576MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
```

The I/O Bottleneck

- Slow storage media (e.g., hard disk drives vs. faster SSDs, distributed filesystems like LUSTRE could be unpredictable).
- Inefficient data access patterns (e.g., reading many small files repeatedly).
- Data formats not optimized for quick loading or random access.
- Limited bandwidth between storage, CPU memory, and GPU memory.
- CPU-bound data preprocessing or augmentation that stalls the pipeline.
- Insufficient parallelism in the data loading process (e.g., single-threaded loading).

Batch creation patterns





A batch is created from a single file vs multiple files

torch.utils.data.Dataset

- An abstract class in PyTorch representing your collection of data samples.
- Separates data loading and preprocessing logic from your model training loop.
- Seamlessly works with torch.utils.data.DataLoader for efficient batching, shuffling, and parallel data loading.
- Key Requirement: To create your own dataset, you subclass torch.utils.data.Dataset and must override two methods:
  - __len__(self): Returns the total number of samples in the dataset. Used by DataLoader to know the dataset size.
  - __getitem__(self, idx): Fetches and returns the sample (e.g., data tensor and label tensor) at the given index idx. This is where you'll typically load data from disk, apply transformations, etc.

real world example: https://github.com/timinar/BabyLlama/blob/main/babylm_dataset.py

torch.utils.data.DataLoader

- Input: A torch.utils.data.Dataset object
- Batching: Automatically groups individual samples from the Dataset into batches of a specified size.
- Shuffling: Optionally shuffles the order of data at the start of each epoch to improve model training.
- Parallel Loading: Can use multiple CPU worker processes (num_workers) to load data in the background, preventing I/O bottlenecks and keeping the GPU fed.
- Memory Management: Offers options like pin_memory for faster CPU-to-GPU data transfers.

more details: https://docs.pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Other types of Datasets

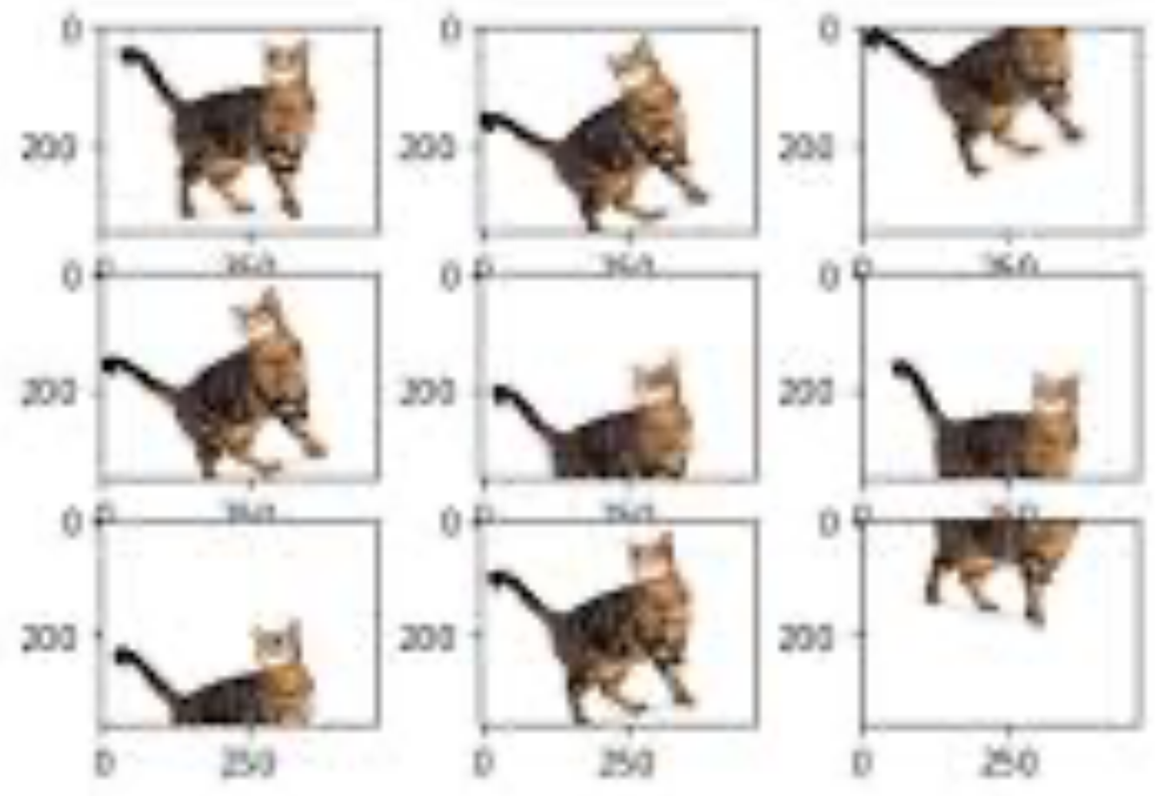
- Iterable-style Datasets (torch.utils.data.IterableDataset)
  - For datasets where data is read sequentially, like a data stream, rather than by random access using an index.
  - You implement __iter__(self) (which yields samples).
  - Note: DataLoader handles these differently (e.g., num_workers has specific considerations, shuffling is typically done within __iter__).
  - Example: https://github.com/timinar/PolarBERT/blob/main/src/polarbert/icecube_dataset.py
- Working with Image Folders (torchvision.datasets.ImageFolder)
  - You have images organized in a directory structure like: dataset_root/class_A/image1.jpg, dataset_root/class_B/image2.jpg
  - ImageFolder automatically discovers images, infers class labels from subfolder names, and can apply specified transformations.

Data Handling Technicalities & Performance Tips

- **Useful torch.utils.data Utilities:**
  - ConcatDataset: Merges multiple datasets sequentially (e.g., combining data from different sources or augmentation passes).
  - Subset: Extracts a specific portion of a dataset using a provided list of indices (useful for specific selections or k-fold cross-validation).
  - random_split: Conveniently splits a dataset into random, non-overlapping new datasets (ideal for creating train/validation/test sets).
- **Strategies for Large Datasets & Performance:**
  - Implement an Efficient __getitem__ (for map-style Dataset):
  - Lazy Loading: Crucially, load data (e.g., image from disk, specific rows from a large file) only when that specific item is requested by __getitem__.
  - Lightweight __init__: Avoid loading the entire dataset into RAM during __init__. Instead, store file paths, metadata, or pointers.
- **Specialized Data Storage Formats:**
  - WebDataset (.tar files): Excellent for streaming large image or sequence datasets, reads data sequentially from TAR archives.
  - HDF5: Hierarchical format, good for large numerical arrays; supports chunking, compression, and partial reads.
  - Apache Parquet: Columnar storage format, highly efficient for tabular data, offers good compression and predicate pushdown (filtering) when reading with libraries like pyarrow.
- Memory-Mapped Files

Data Augmentation: Where?

- Idea: Artificially create diverse training samples from your existing data (e.g., flipping images, altering text) to improve model robustness and reduce overfitting.
- Where:
    - Offline (Pre-processing): Generate and save augmented versions before training. Uses more disk space; simpler loading logic. Harder to change.
    - Online (On-the-fly): Apply augmentations dynamically during data loading for each epoch. More flexible; less disk space.
        - CPU-based: Common (e.g., in DataLoader workers using torchvision.transforms). Can be a bottleneck if transformations are heavy.
        - GPU-based: For faster, complex augmentations.



Caching & Buffering

- Idea: Store frequently accessed data or pre-loaded items in faster memory (e.g., RAM, fast SSD) to avoid repeated slow reads from primary storage (HDD, network).
- Common Strategies:
  - Full Dataset in RAM: If your dataset is small enough, load it entirely into memory at the start.
  - Selective Caching: Cache only the most frequently used samples or pre-process and cache transformed items.
  - Prefetch Buffers (e.g., in DataLoader): Automatically load upcoming batches into a memory buffer while the current batch is being processed.
  - Disk Caching: Use a fast local SSD as a cache for data originating from slower network storage or HDDs. Usually GPU nodes have they own fast storage

```bash
#!/bin/bash

if [ ! -e /dev/shm/filtered_all_big_data.db ]; then
  echo "Stage to /dev/shm/"
  time cp filtered_all_big_data.db /dev/shm/
else
  echo "File already staged to /dev/shm"
  ls -al /dev/shm/filtered_all_big_data.db
  du -skh /dev/shm/filtered_all_big_data.db
fi
```

Profiling & Tools

- Profiling — measuring time and memory consumption.
- nvidia-smi, htop / top, iotop
- PyTorch Profiler (torch.profiler)

Summary of Data Loading Best Practices

- Exploratory data analysis (EDA)!
  (Print & Plot)

- Choose appropriate data formats for your dataset size and access patterns.

- Decide about Dataset vs IterableDataset

- Use num_workers wisely.

- Consider pin_memory and prefetching.

- Profile your pipeline!

Thank you and enjoy the school!

will brown
@willccbb

being good at ML systems helps you run more experiments. being good at ML theory helps you run less experiments

4:29 AM · Jun 2, 2025 · **25K** Views

18 33 613 114