

Fast data loaders and Foundation/Frontier Models

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

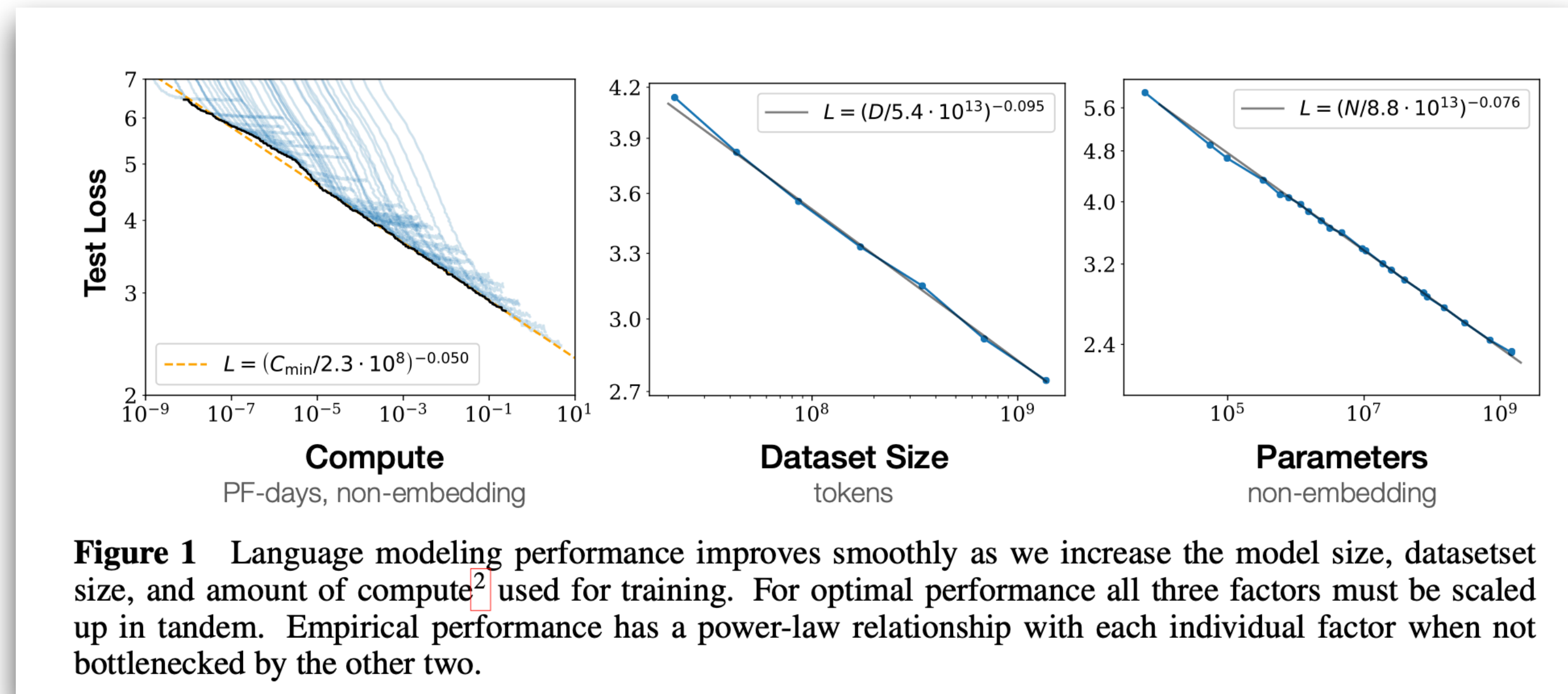
May 20, 2026

Plan

- Self-supervised Learning
- Foundation Models
- The Data Challenge
- Fast Data Loaders: Keeping the GPUs Fed

Recap on Transformers and Scaling Laws

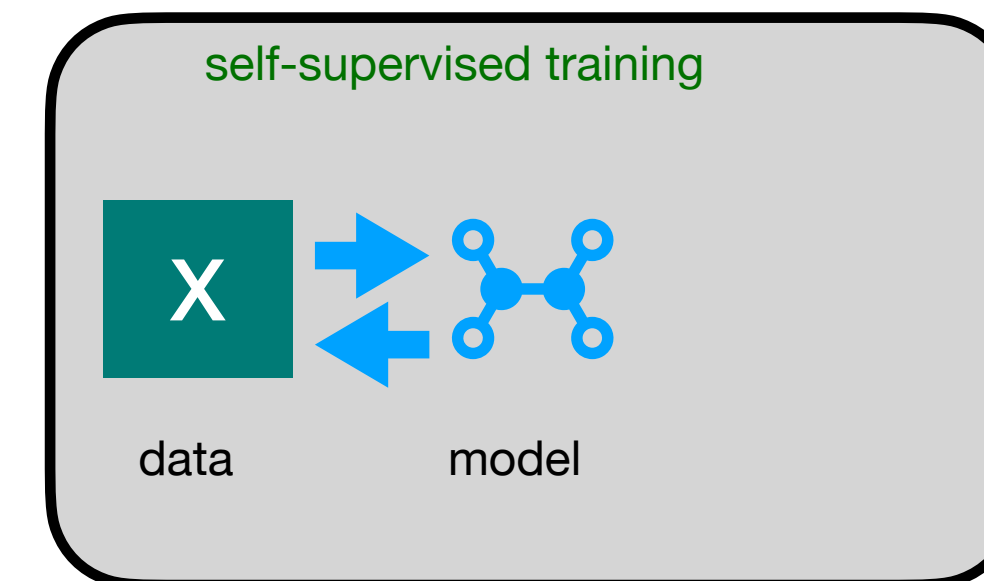
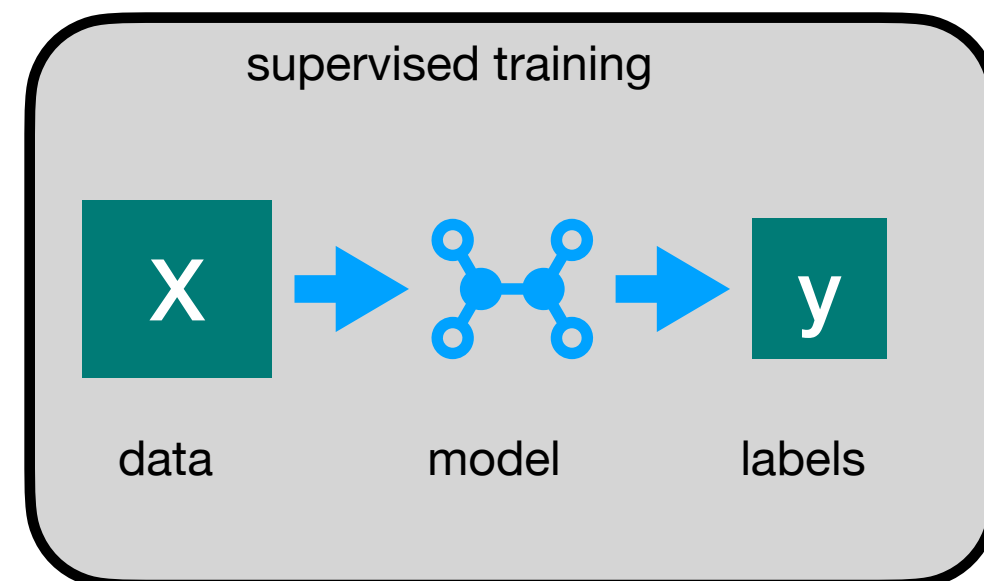
- Performance predictably improves with scale
- But where can we find the data to scale the models?
- Labeled data is scarce



<https://arxiv.org/pdf/2001.08361>
Scaling Laws for Neural Language Models

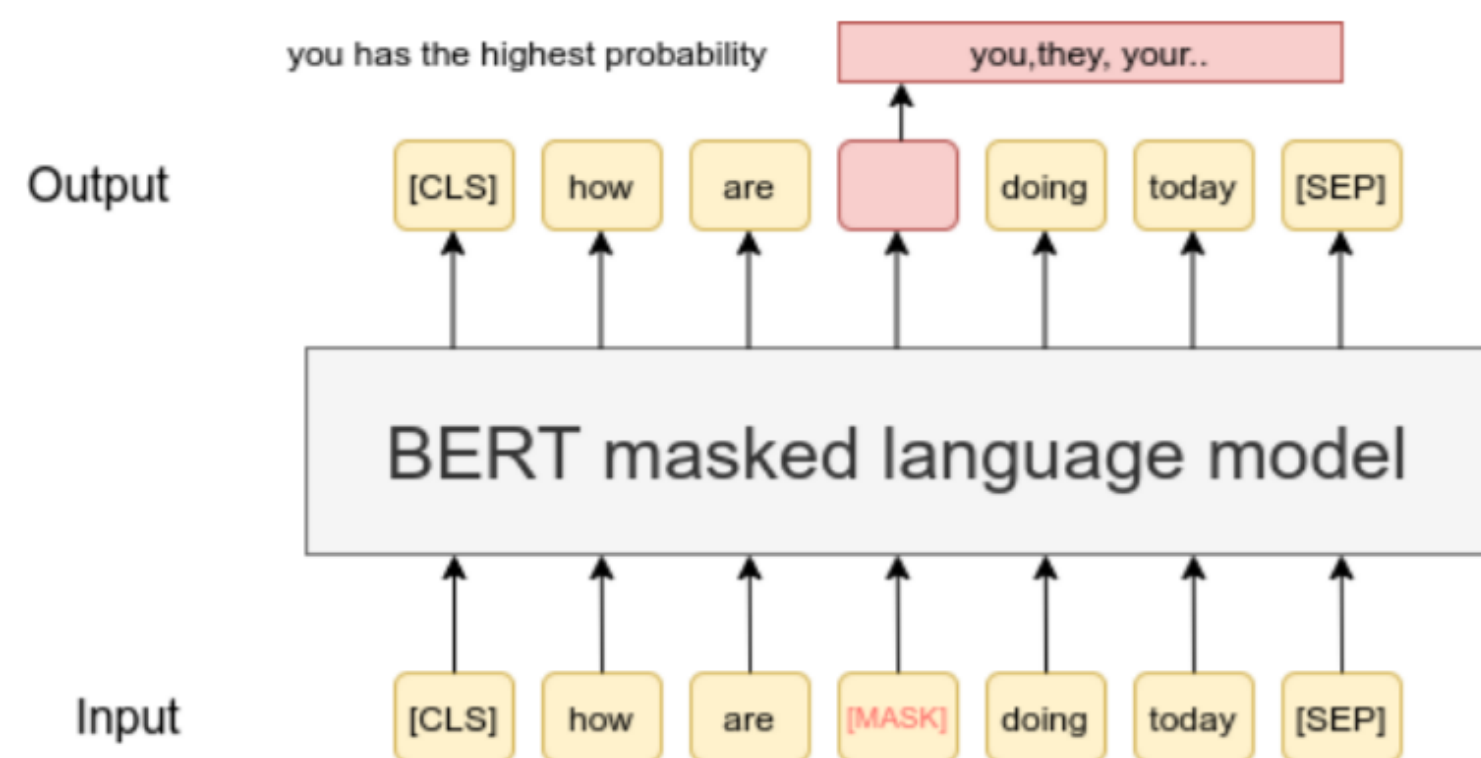
Supervised and Self-supervised Learning

- Supervised learning: we have data and labels. The model is trained to predict the labels.
- Unlabeled data is abundant. Can we do without the labels?
- Self-supervised learning



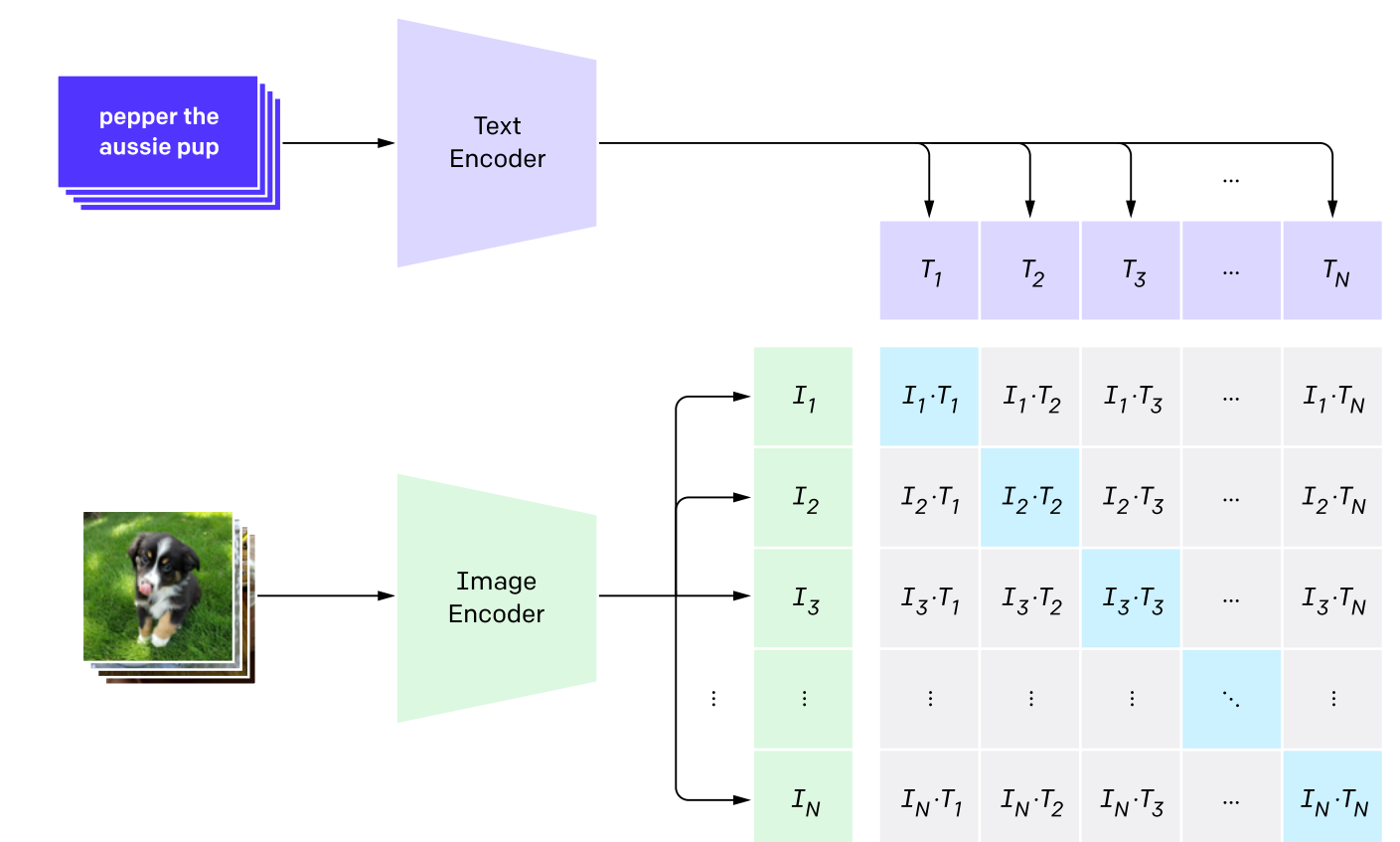
How can we train without labels?

- **Masked Prediction:** We train models to predict hidden or "masked" parts of the input data (e.g., words in a sentence or patches in an image).
- **Denoising:** The model learns to reconstruct original, clean data from a corrupted or noisy version we create.
- **Contrastive Learning:** We teach the model to pull representations of similar (e.g., augmented) data points closer and push dissimilar ones further apart.



Denoising Autoencoder

1. Contrastive pre-training



OpenAI CLIP

(Contrastive Language-Image Pre-training)

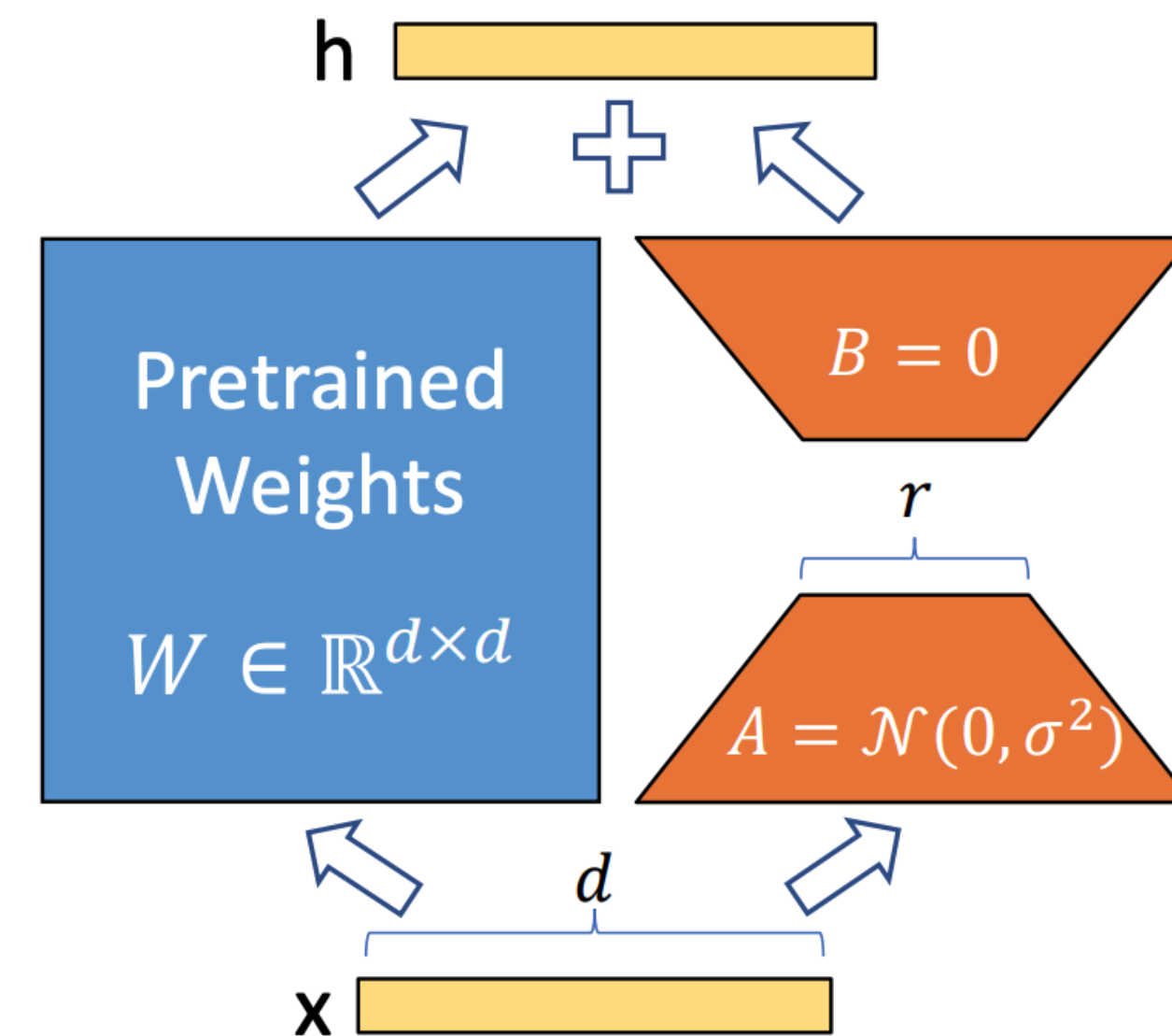
Foundation Models

- **Definition:** Models pre-trained on broad data at massive scale, designed to be adapted (e.g., fine-tuned) to a wide range of downstream tasks.
(Stanford HAI definition)
- **Scale:** Data (petabytes), Parameters (billions to trillions).
- **Generality:** Adaptable to diverse tasks, often beyond their explicit training objectives
 - **Examples:** Bert, CLIP,...
- Another term: **Frontier Models** — zero shot capabilities, no fine-tuning

Examples: GPT, Claude, Gemini, Grok, Llama, DeepSeek, Qwen

Parameter-Efficient Fine-Tuning (PEFT)

- Sometimes it is enough to tune just a subset of parameters.
- One can freeze early layers, or the full backbone.
- One can also apply techniques like **LoRA** (Low-Rank Adaptation).
- See this nice guide for LLMs: <https://docs.unsloth.ai/get-started/fine-tuning-guide>



$$h = Wx + BAx$$

BA is a low rank matrix

The Data Challenge

- Scale is crucial for foundation 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.

Fast Data Loaders: Keeping the GPUs Fed



The Deep Learning revolution was powered by Compute!

Sweat, Blood, and Data Loading

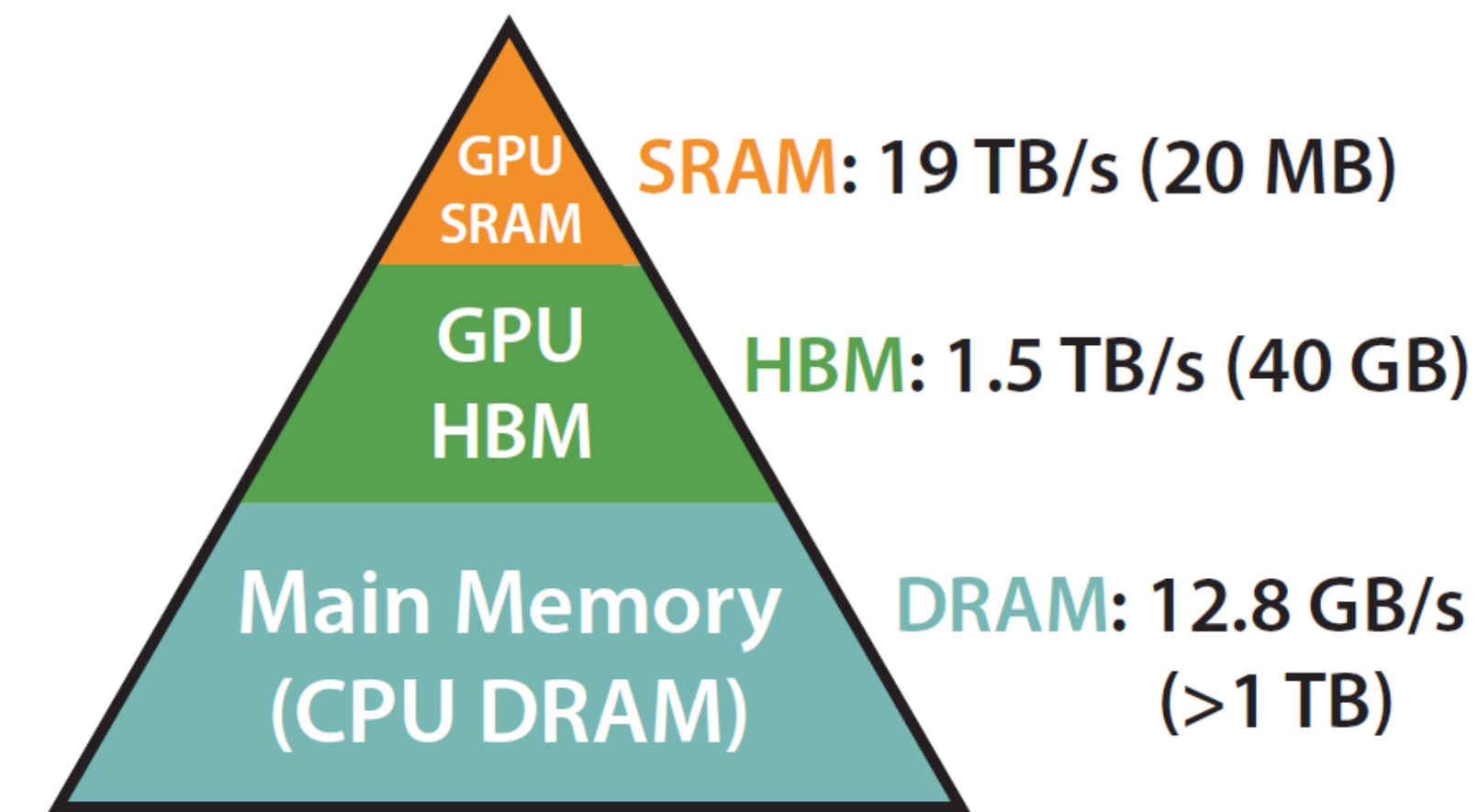
DL specific compute requirements

- NNs rely on massive matrix multiplies.

$$x' = Wx$$

multiplies is the same as number of parameters in W

- CPUs.
- CPUs can execute ~ one instruction at a time.
- GPUs perform many operations in parallel, so they give a significant speed up. Moreover, the modern GPUs have specialized tensor cores (TPUs have only those).
- GPUs have fast memory, but it is limited, so we need to transfer data to them.



Memory Hierarchy with Bandwidth & Memory Size

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. |
|                                           |                      |             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  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).

torch.utils.data.Dataset (Code on GitHub)

- 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 (Code on GitHub)

- 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

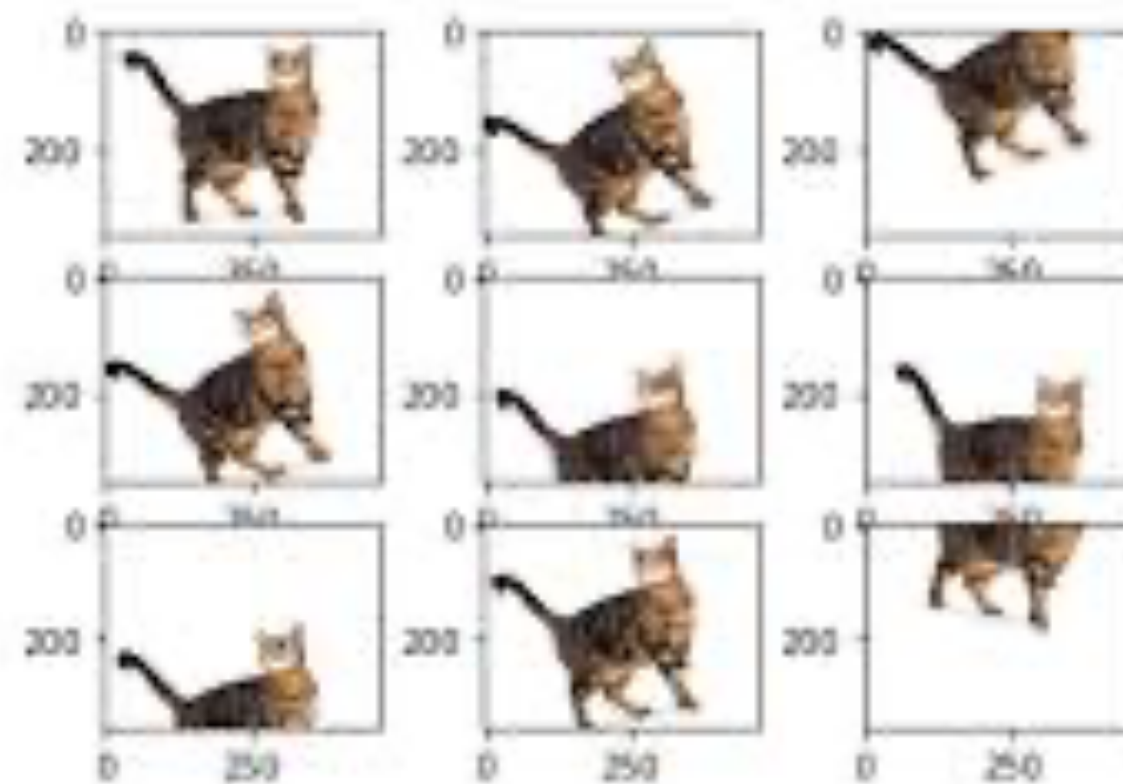
- Iterable-style Datasets (`torch.utils.data.IterableDataset`)
 - For datasets where data is read sequentially, like a data stream, rather than by random access using an index.
 - You implement `__iter__(self)` (which yields samples).
 - Note: `DataLoader` handles these differently (e.g., `num_workers` 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.
 - 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

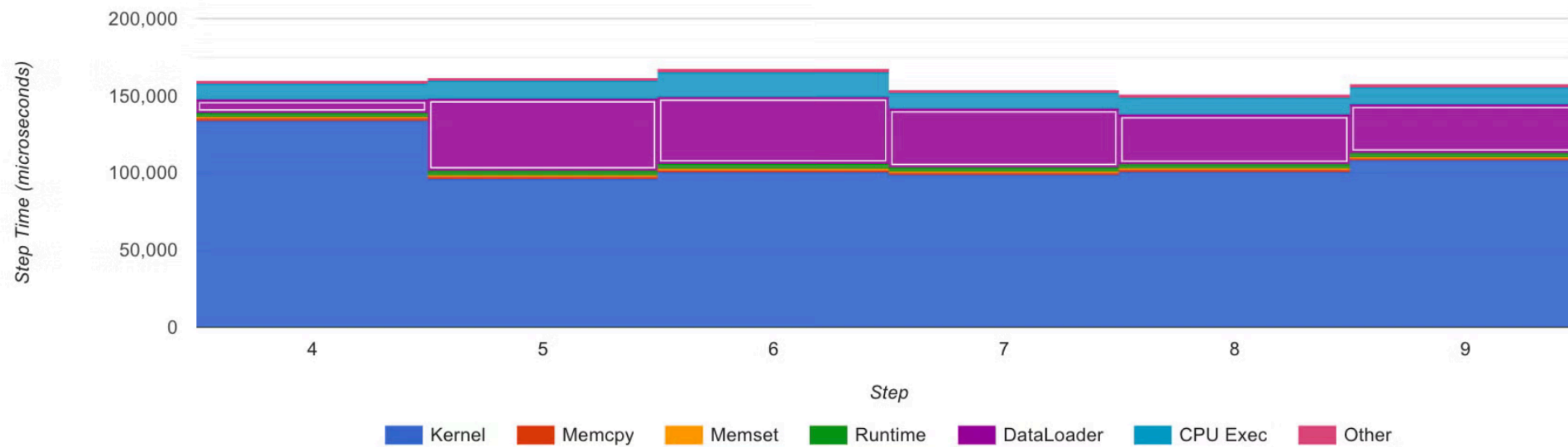
- Idea: Store frequently accessed data or pre-loaded items in faster memory (e.g., RAM, fast SSD) to avoid repeated slow reads from primary storage (HDD, network).
- Common Strategies:
 - Full Dataset in RAM: If your dataset is small enough, load it entirely into memory at the start.
 - Selective Caching: 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 their own fast storage

```
#!/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

- Use `num_workers` wisely.
- Consider `pin_memory` and prefetching.
- Choose appropriate data formats for your dataset size and access patterns.
- Profile your pipeline!
- For perspective, big labs have they own **filesystems!**
<https://github.com/deepseek-ai/3FS>