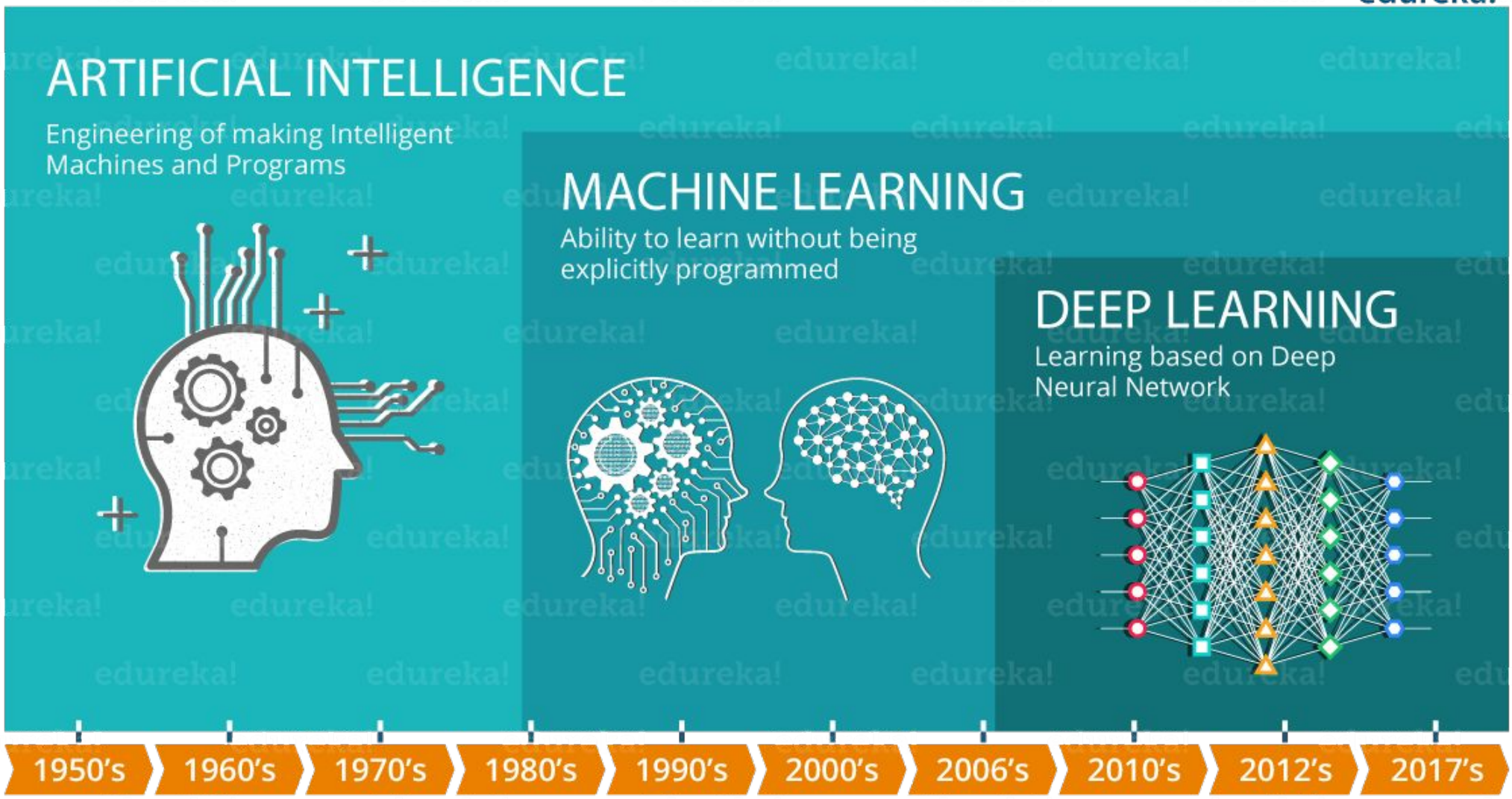
Parallel/Distributed Systems: A Driver for Artificial Intelligence

Bogdan Nicolae
Argonne National Laboratory, USA

Argonne NATIONAL LABORATORY

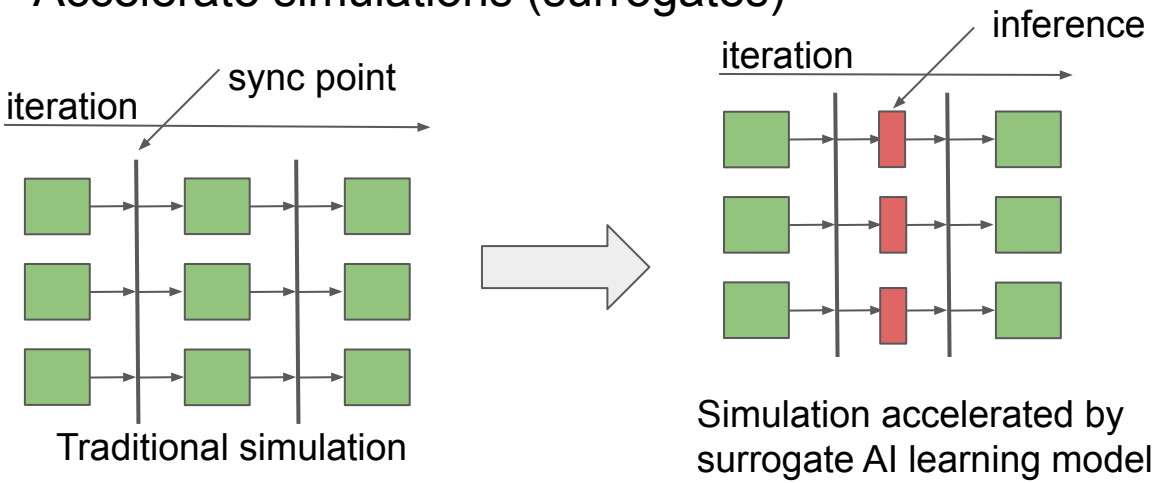
Evolution Towards Deep Learning



Source: edureka

Deep Learning in HPC/Scientific Computing

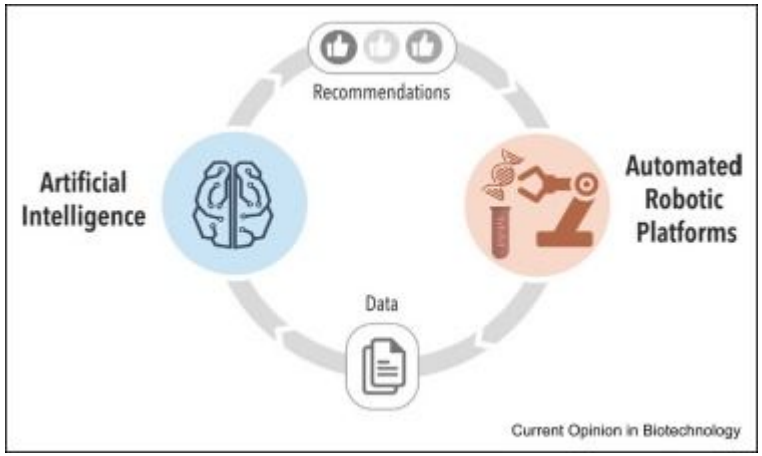
Accelerate simulations (surrogates)



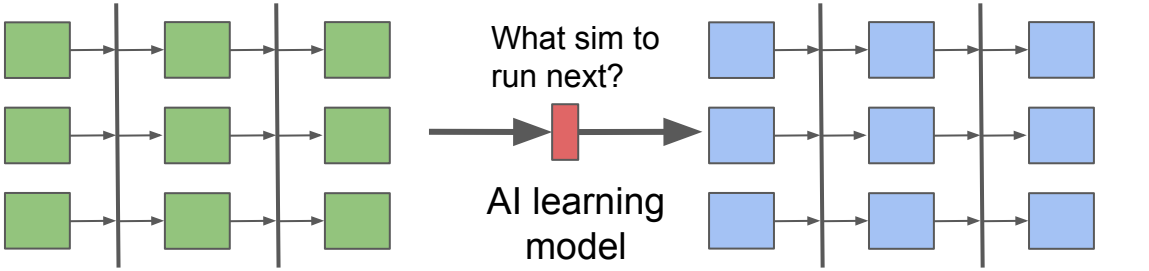
Traditional simulation

Simulation accelerated by surrogate AI learning model (trained offline or online)

Self-driving laboratories



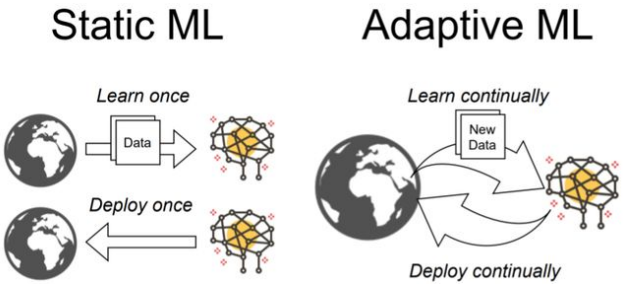
Planning and execution of ensembles



Simulation (any flavor)

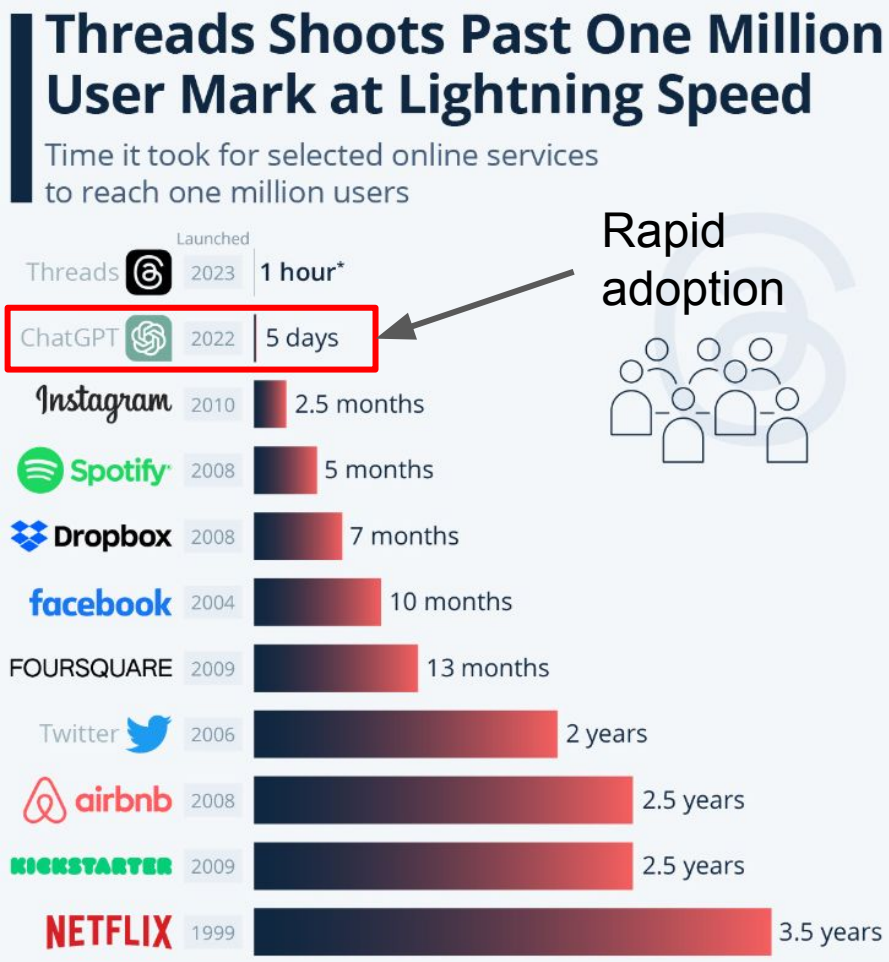
Next simulation (any flavor)

Real-time steering and acquisition



Edge Instruments + Supercomputers

AuroraGPT: 1T Params Foundational Model



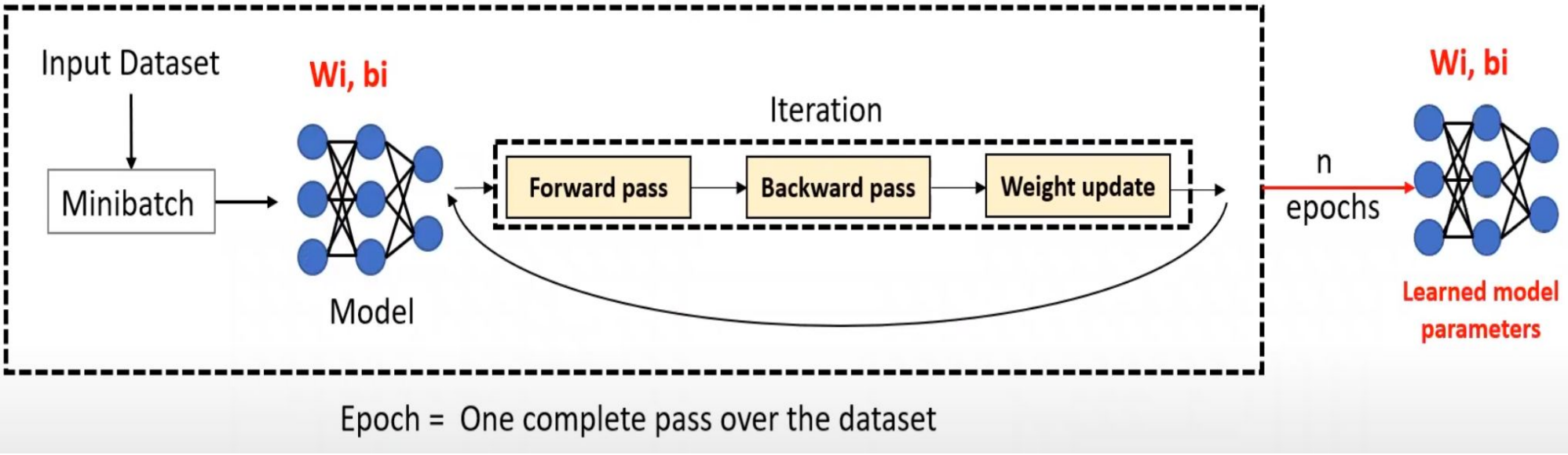
Rapid adoption

AI assistants based on foundational models (e.g. LLMs) will be the norm in the future

arXiv

Argonne to run GPT-like models Aurora Supercomputer

Deep Learning In a Nutshell

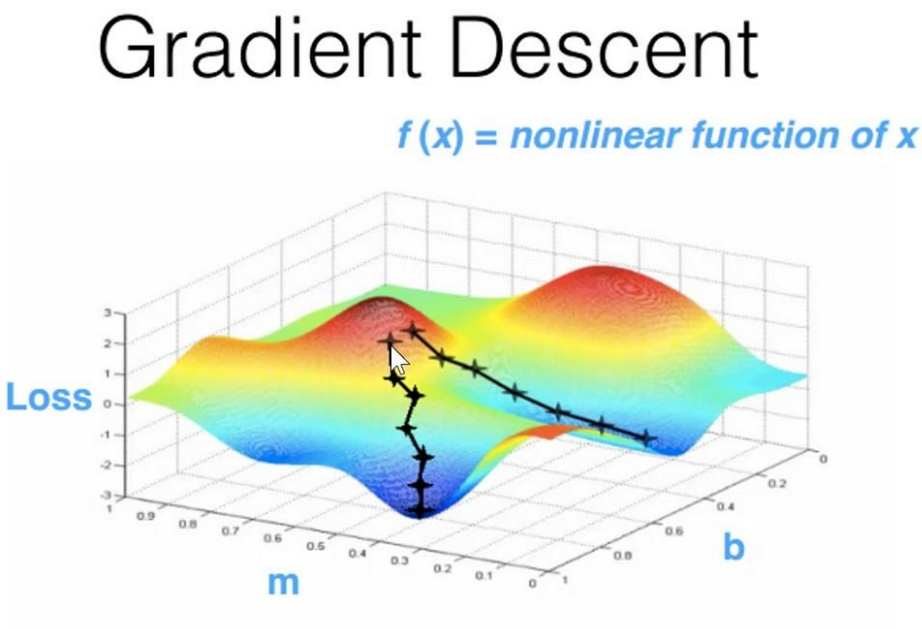


Forward Pass:

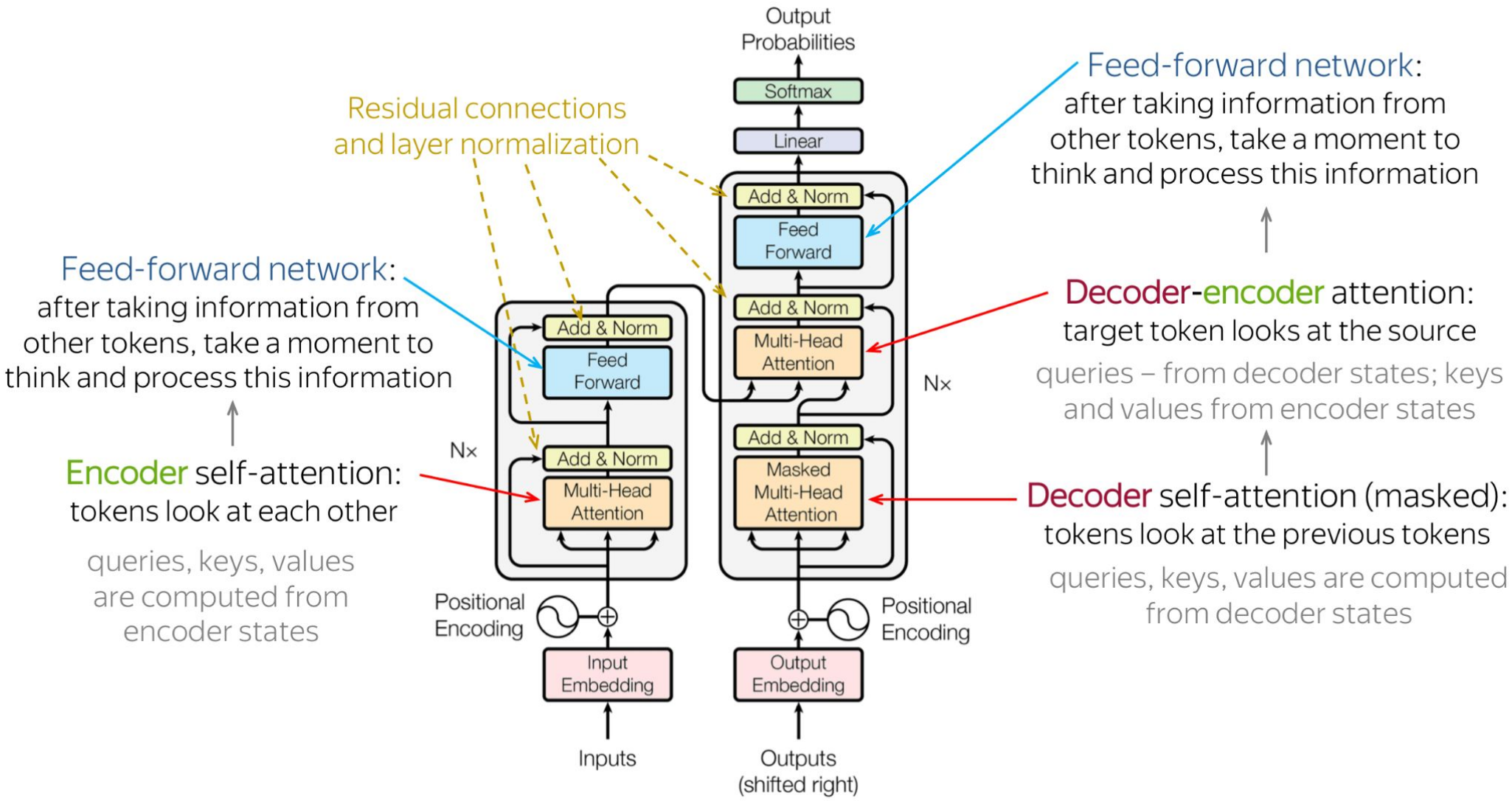
- Predict the output for the input mini-batch
- f(x) is a composition of functions (layers)

Backward Pass:

- Compute gradients layer-by-layer (in reverse)
- Update model weights to minimize loss

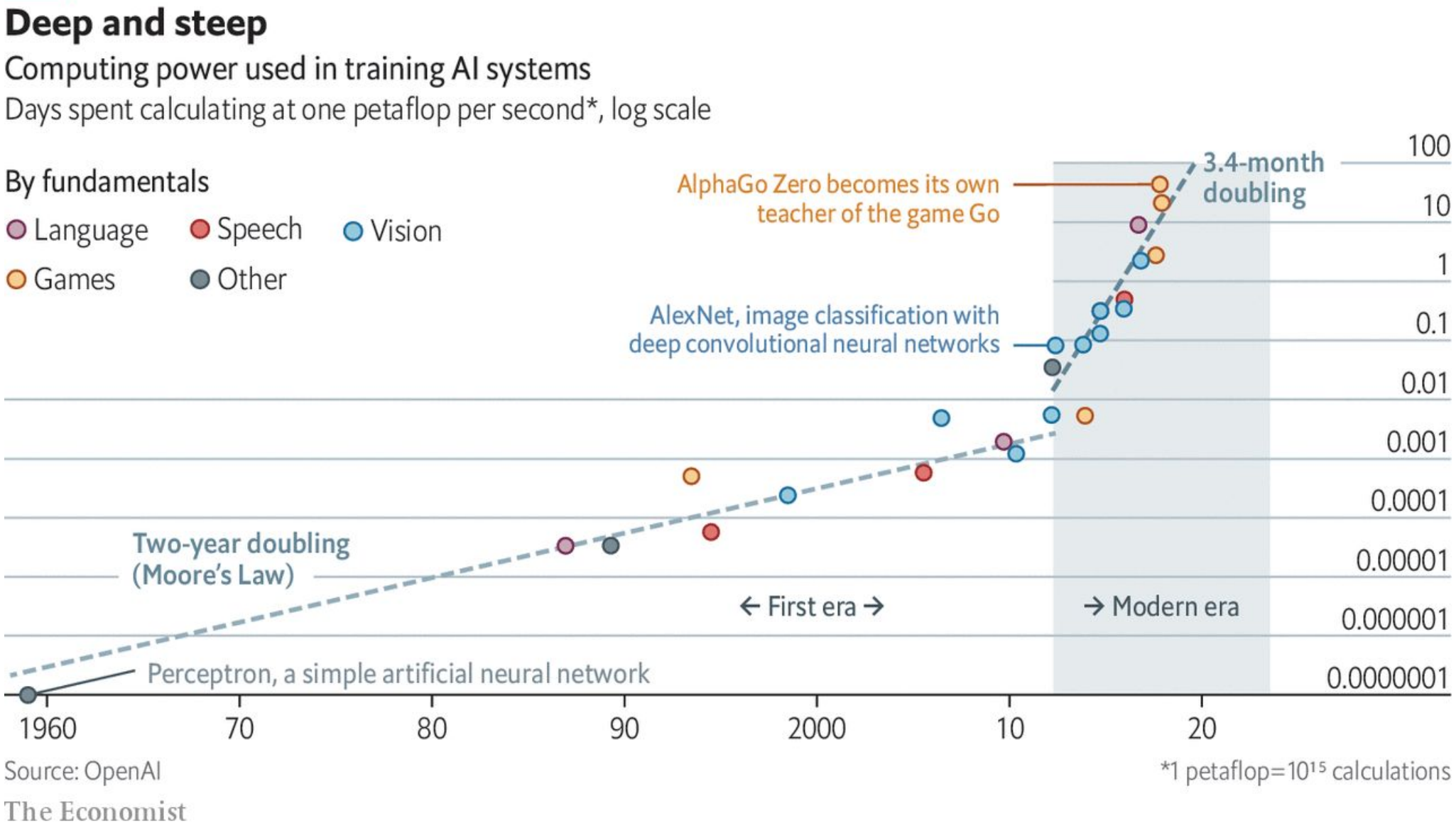


An Example: Transformers, LLMs

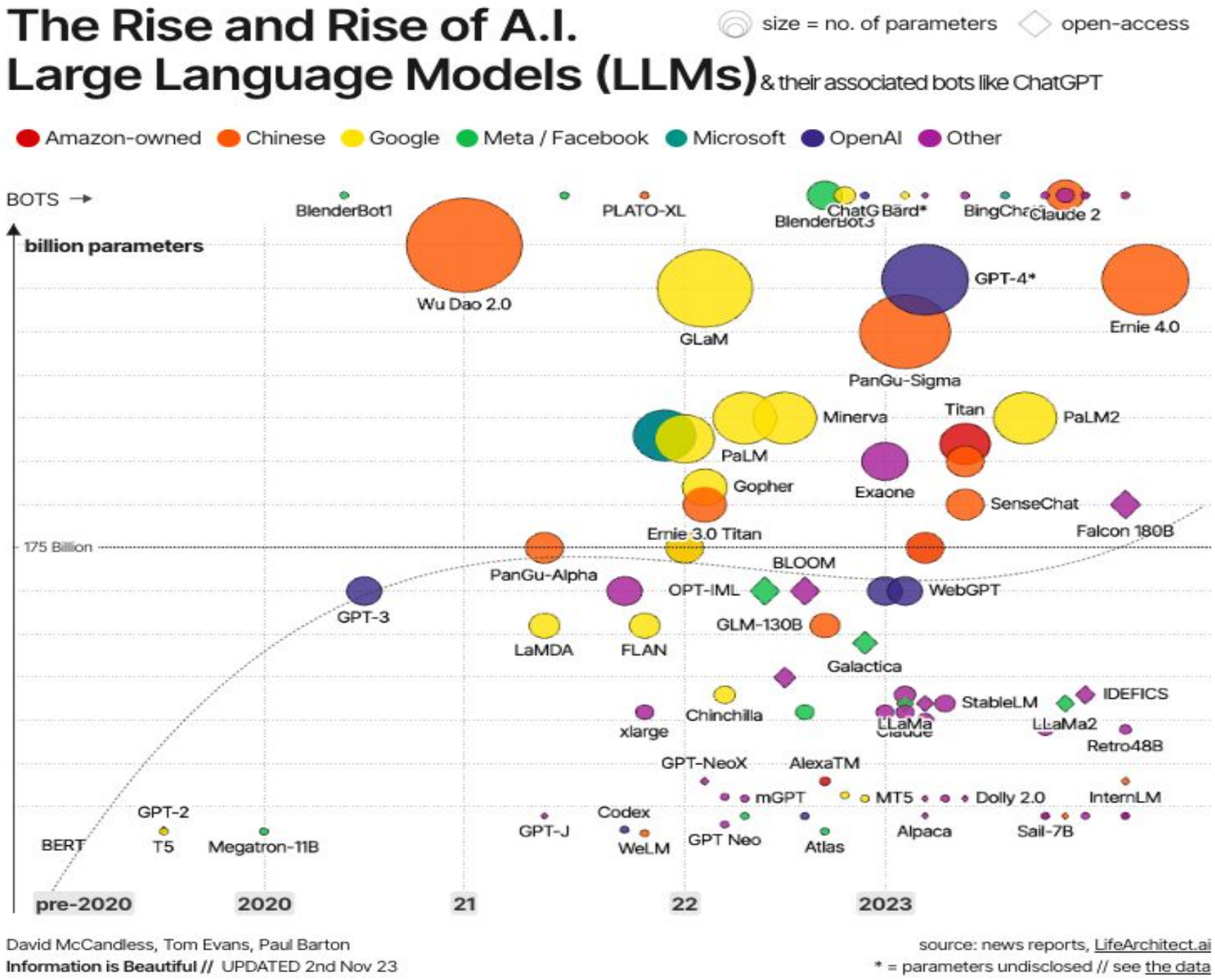


- Depth (and large number of parameters) comes from a large number of repeated blocks
- Training follows roughly the same gradient descent principle (output is ground truth)
- Inference starts with empty output and grows it iteratively

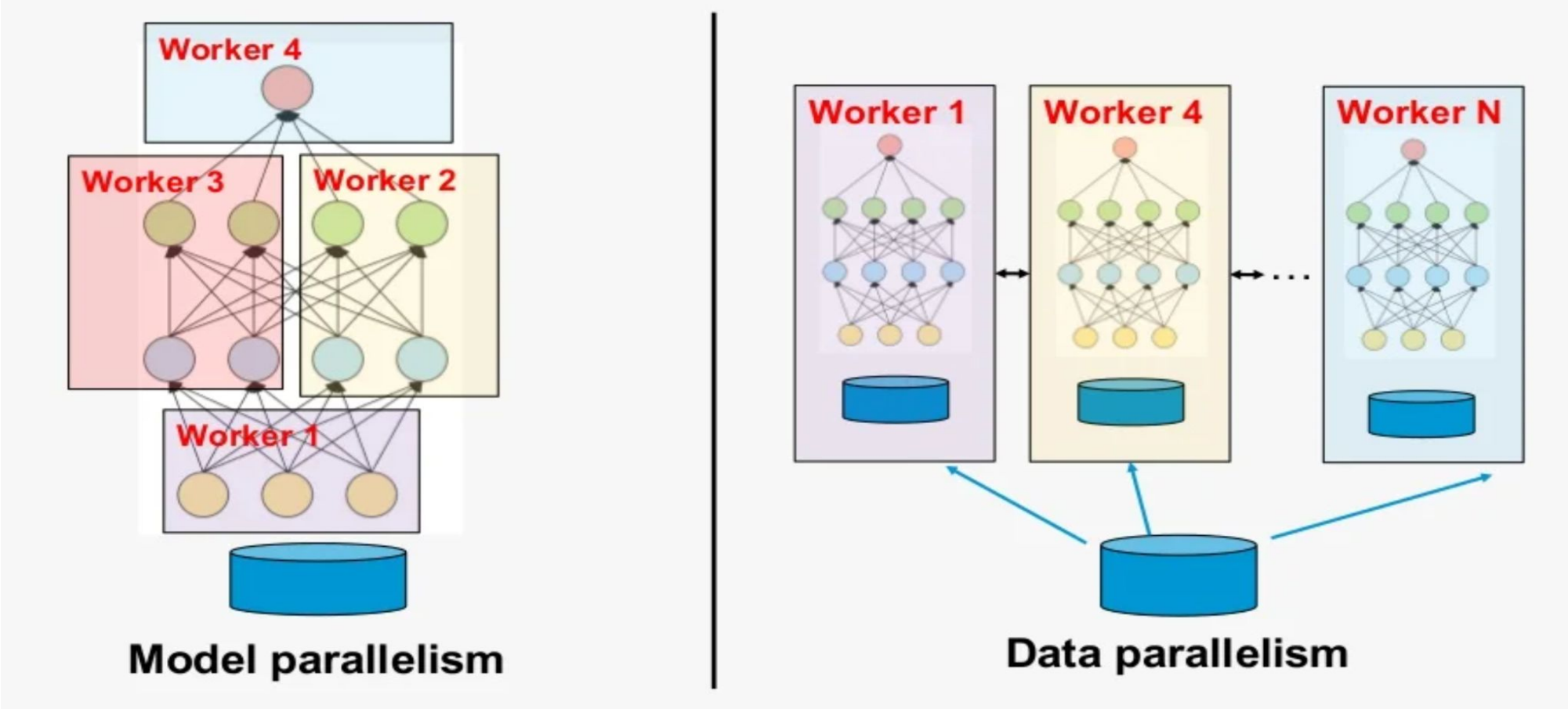
Deep Learning Training Is Expensive!



LLMs are Getting Bigger and Bigger



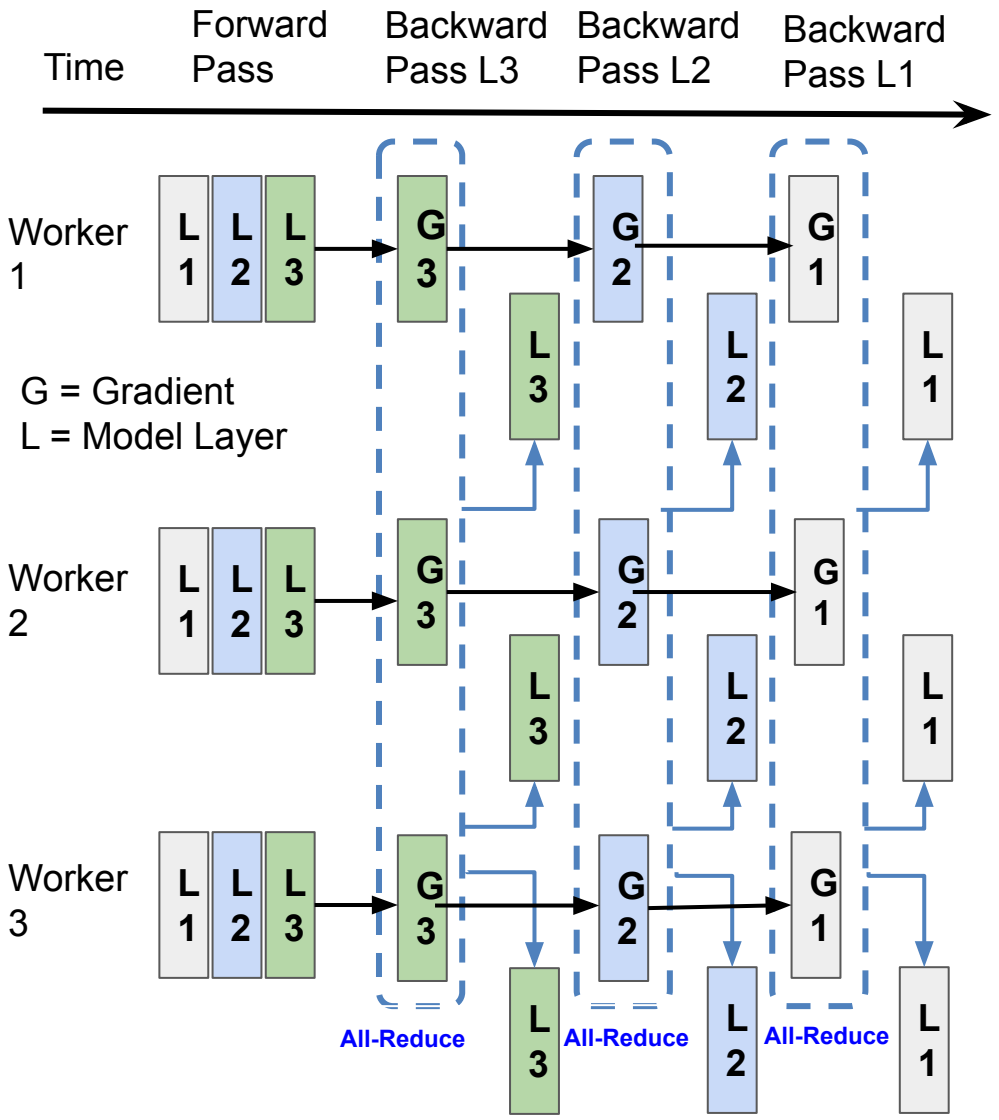
Solution: Distributed Deep Learning



- Tensor parallelism: split layers across GPUs or compute nodes (tight comm)
- Pipeline parallelism: assign different layers to different GPUs/nodes (sparse comm)

- Create model replicas on each GPU or node
- Simple and efficient solution, easy to implement
- Only works if whole DL model fits in GPU memory

Overlapped Gradient Compute and Updates



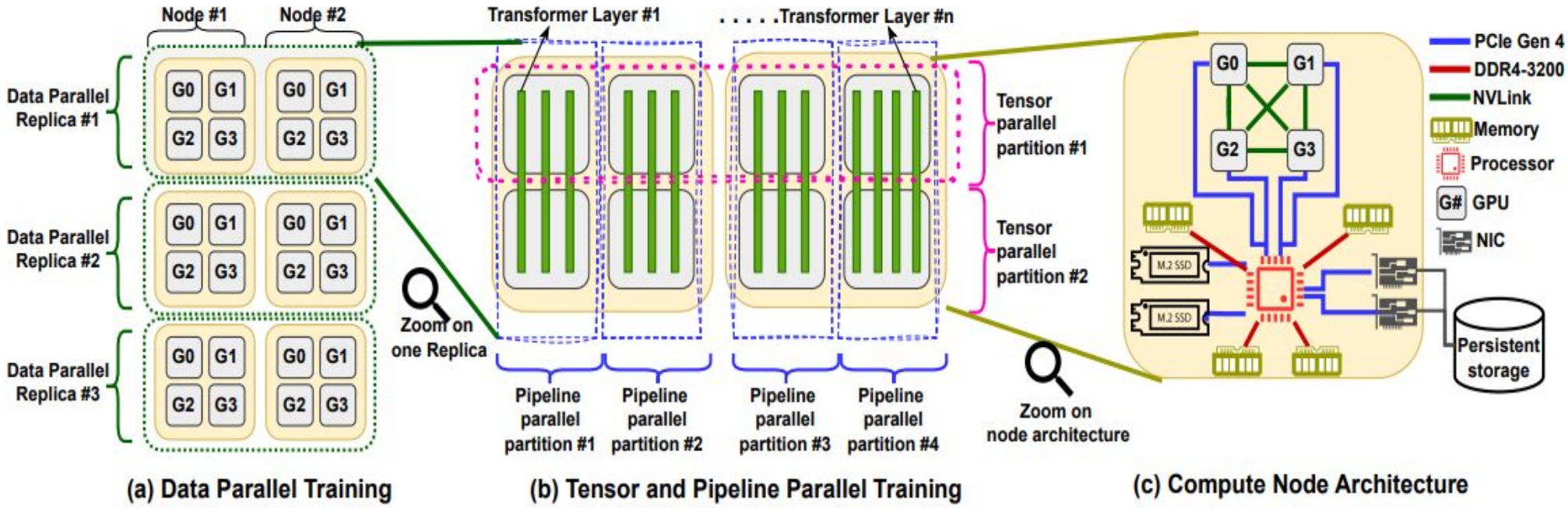
Forward Pass:

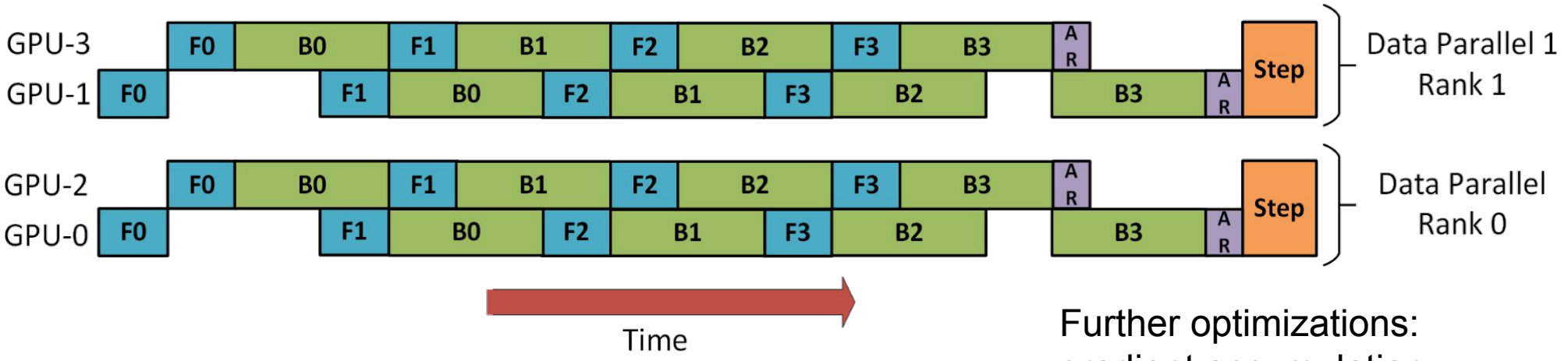
- All workers predict the output for their respective mini-batches

Backward Pass:

- Compute local gradients of layer N
- In parallel:
  - Optionally average the gradients of layer N if running in data-parallel mode
  - Update the parameters of layer N
  - Compute local gradients of layer N-1 (based on local gradients of layer N-1
- Overlap gradient computations with communication/parameter updates
- Data-parallel synchronizes gradients => model replicas remain identical

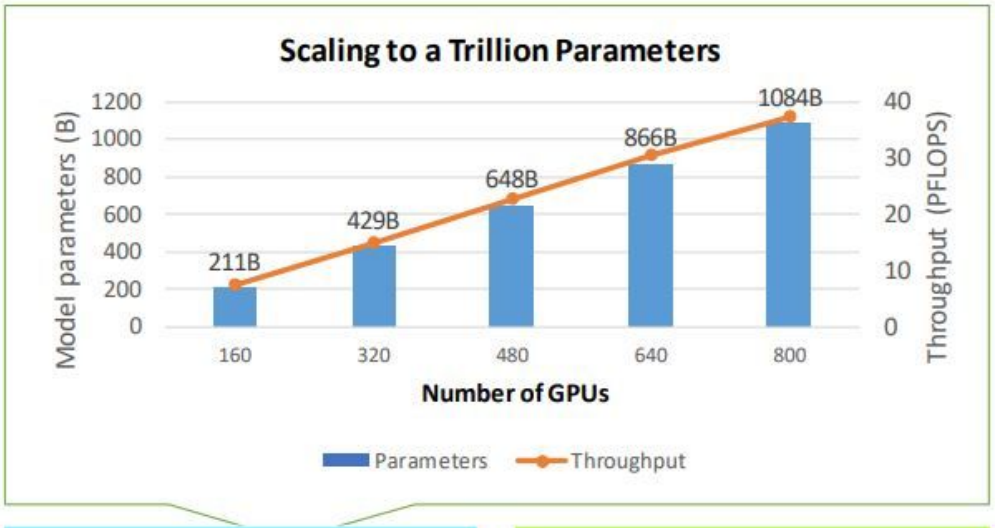
Towards a Hybrid Solution: 3D Parallelism

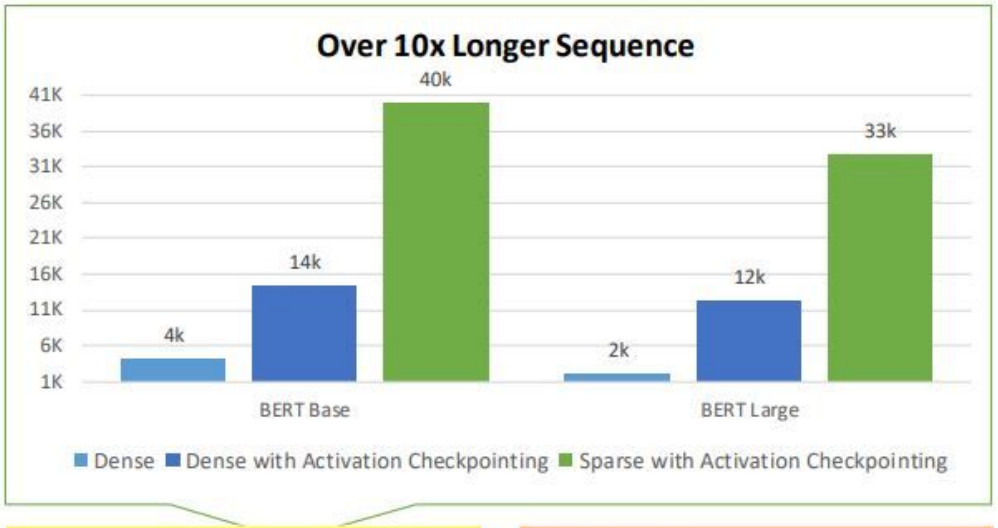




Further optimizations:
gradient accumulation

DeepSpeed: A Popular DL runtime





3D Parallelism

- 1 trillion parameter model training

ZeRO-Offload

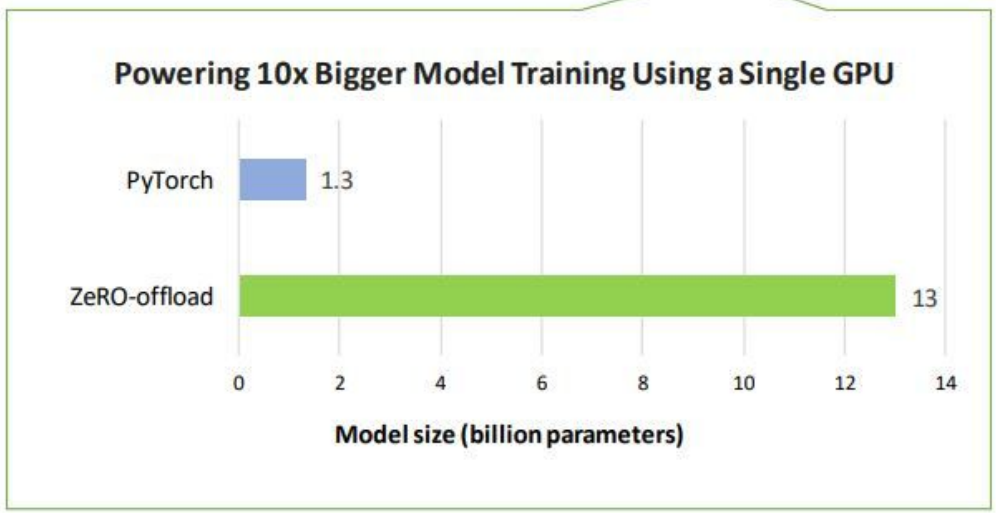
- 13B model on single GPU, 10x bigger

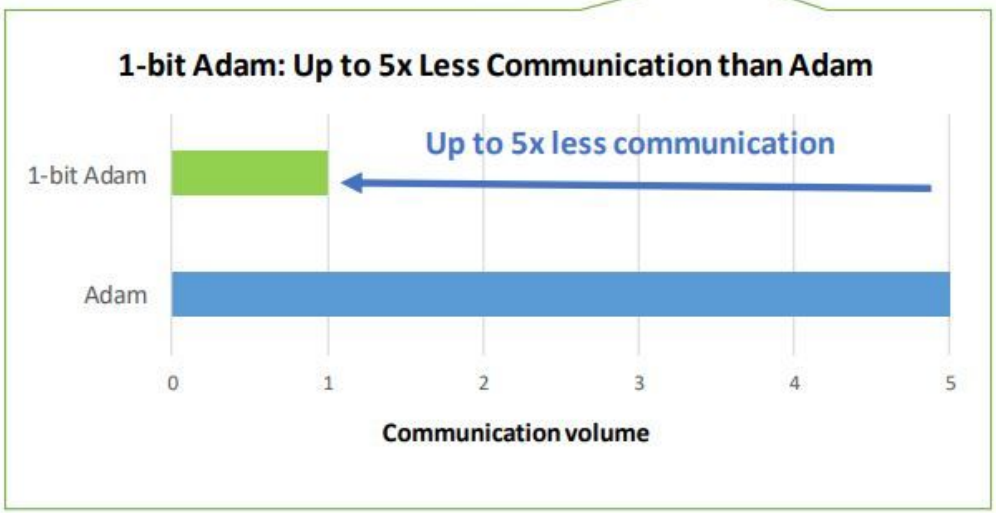
Sparse Attention

- 10x longer sequence, up to 6x faster

1-bit Adam

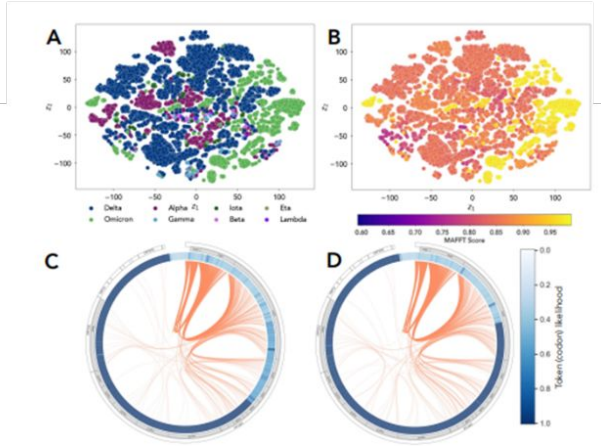
- 5x less communication



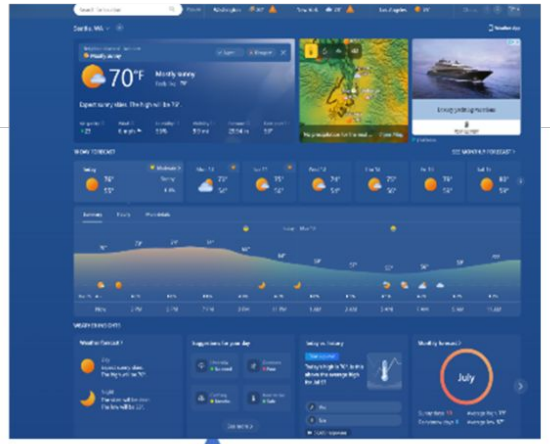


DeepSpeed4Science: Generalized Transformers

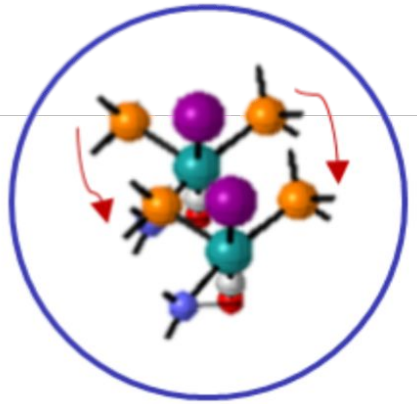
Molecular and Protein Structures, Drug Discovery, Genomes Study



Climate Model and Weather Forecasting



Quantum Chemistry



DeepSpeed4Science (4th Pillar): DeepSpeed4Science Toolkit

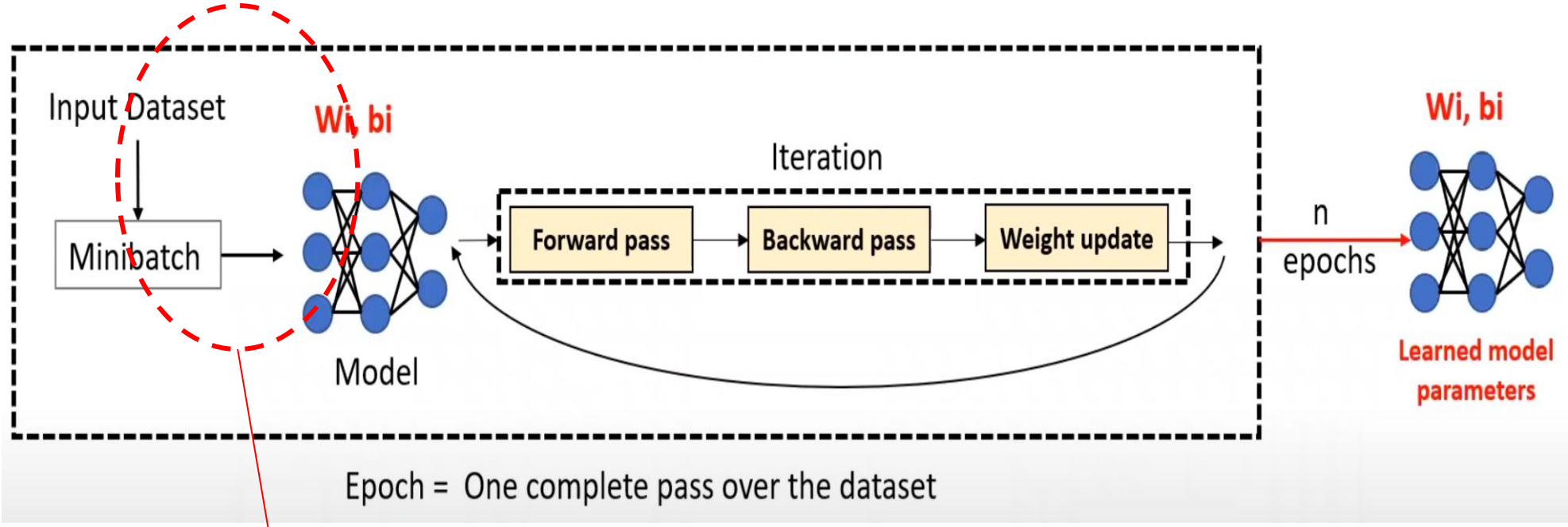
Specialized science-centric attention kernel design for performance and memory efficiency	Algorithm-system co-design for supporting various long-sequence scenarios	Scalable training with ultra-large model and data	Highly-efficient sparse inference	Multi-modality training/ inference	… …

DeepSpeed's Current Pillars As Base Technology Enabler

Training	Inference	Compression

Efficient Ingestion of the Training Data

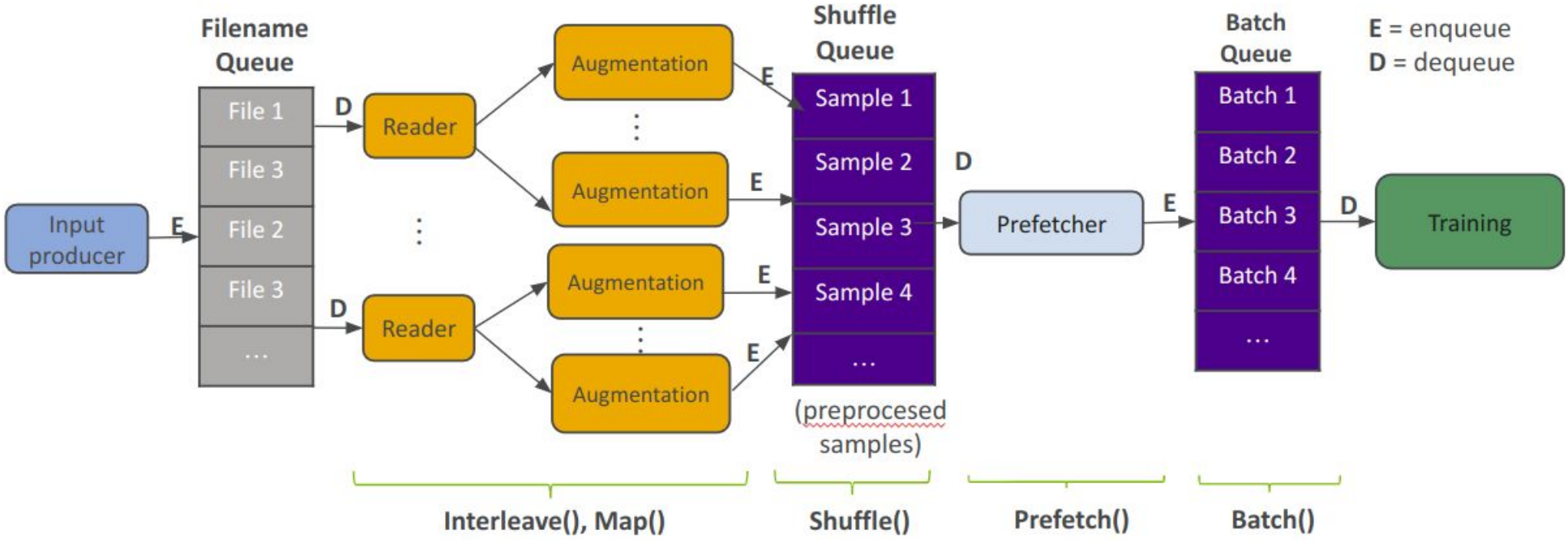
DNN Training: Data is Becoming a Bottleneck



- Ingesting the training data has a high overhead
- Need to assemble mini-batches, which is is complex:
  - Read from data repository (e.g., parallel file system)
  - Transformations (e.g., decode, augment)
  - Shuffling and batching (to avoid repeated fine-grain sampling)

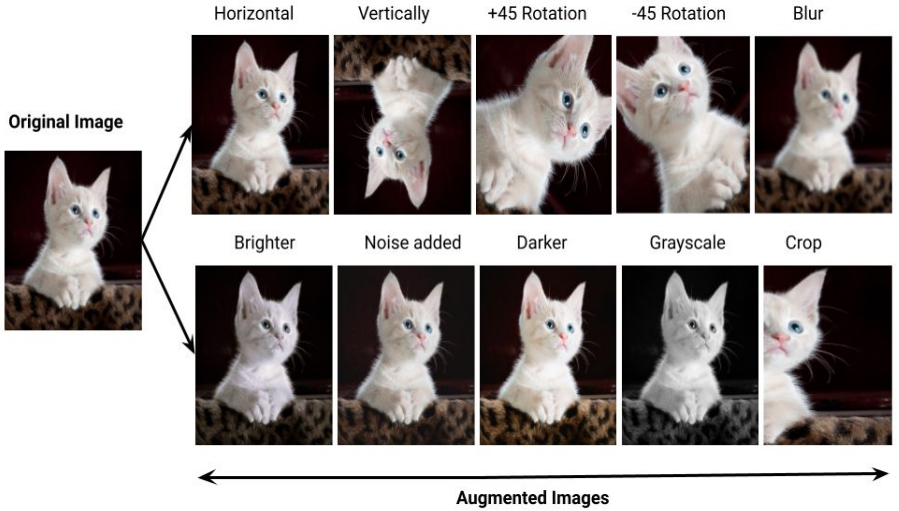
Abstraction: Training Data Pipeline

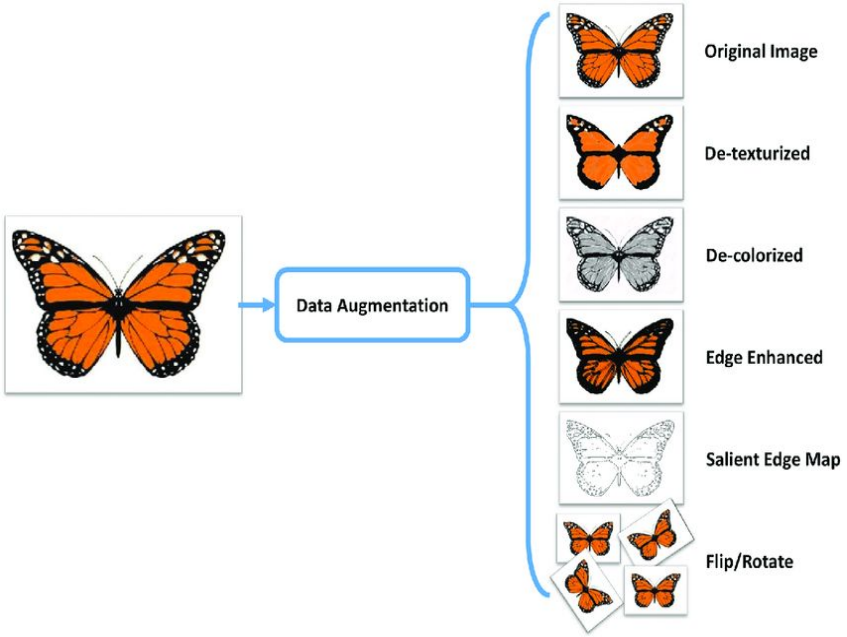
- Multi-threaded stages operating concurrently on producer-consumer queues:
  - **Enqueue (E)** blocks if queue is full
  - **Dequeue (D)** blocks if queue is empty



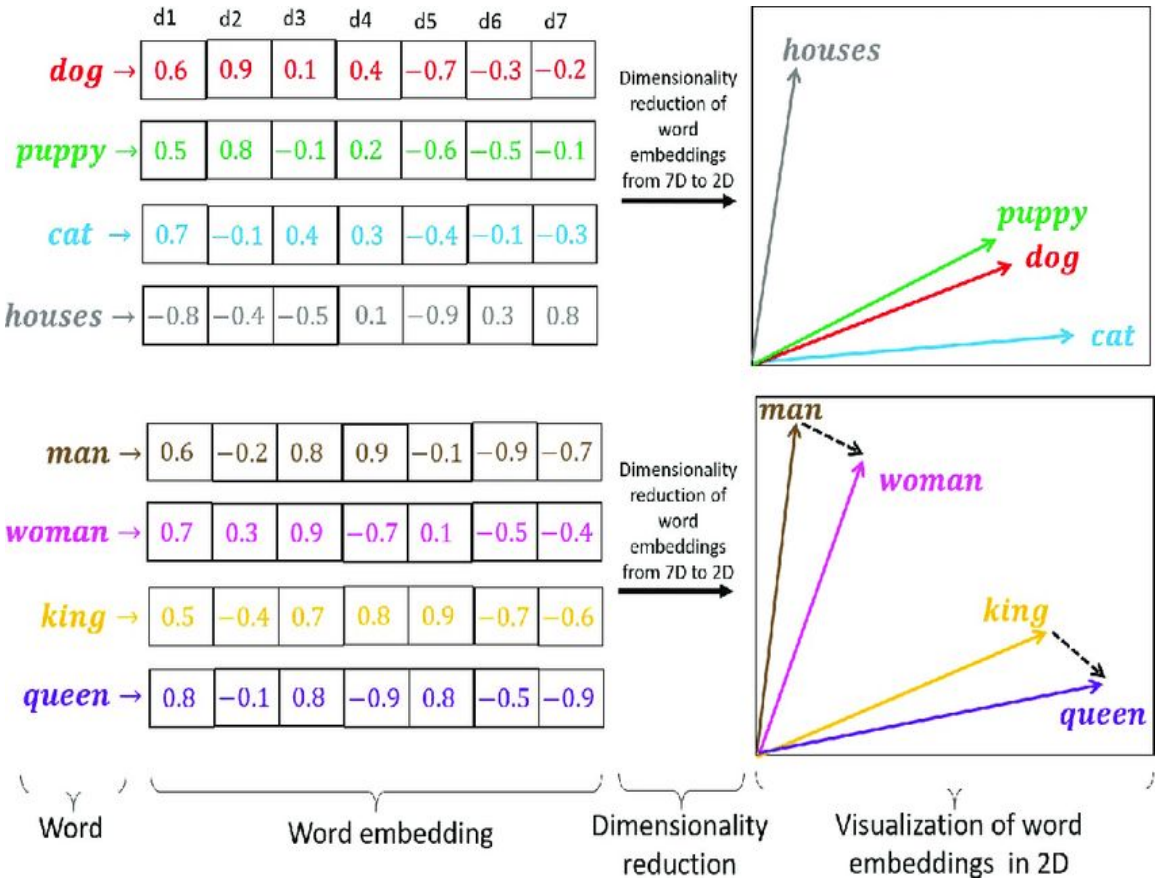
How Does an Augmentation Look Like?

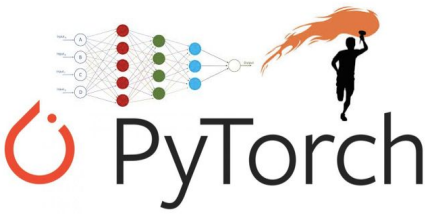
Example #1: Image Augmentation





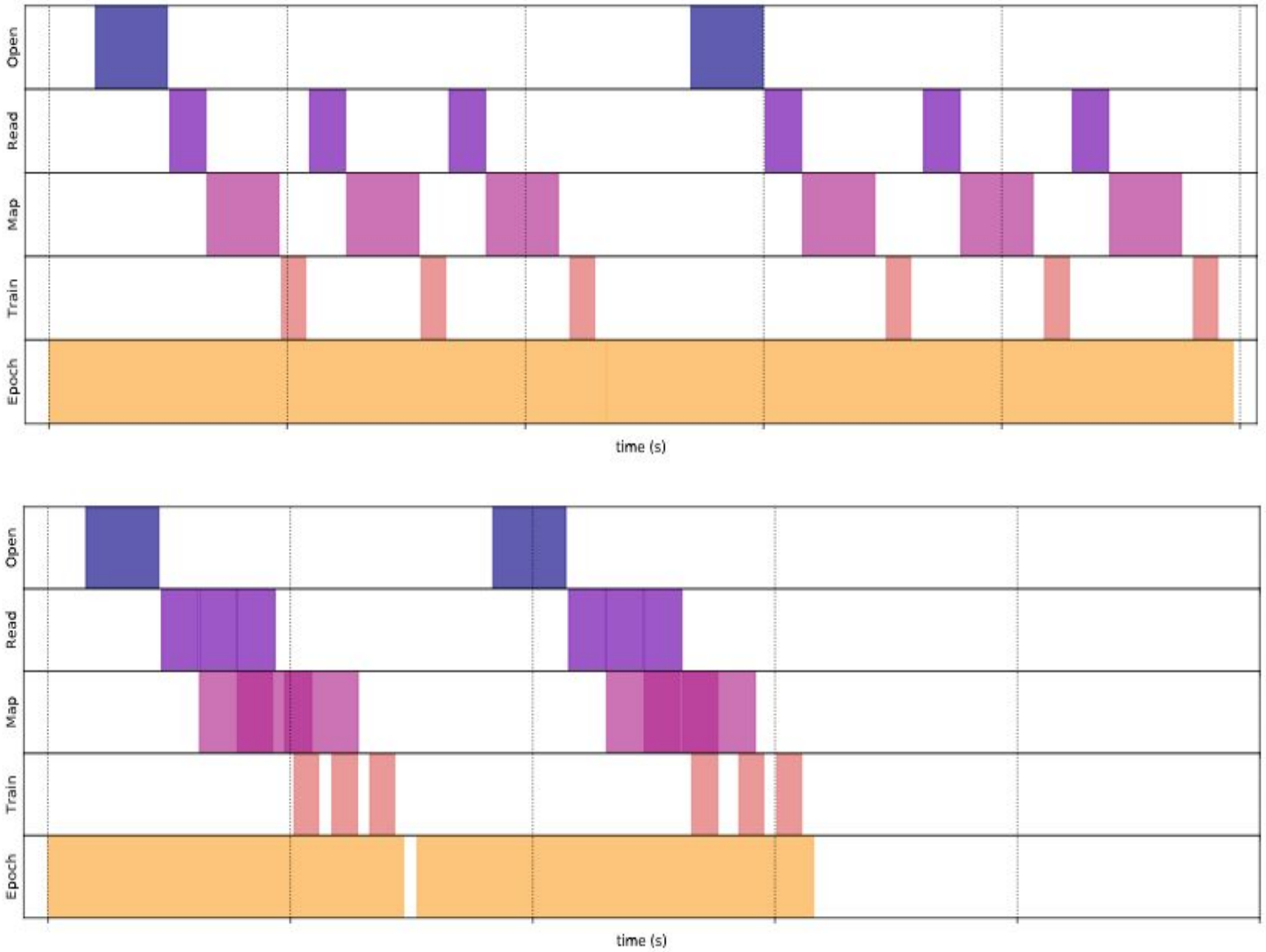
Example #2: LLM Tokenization + Embeddings



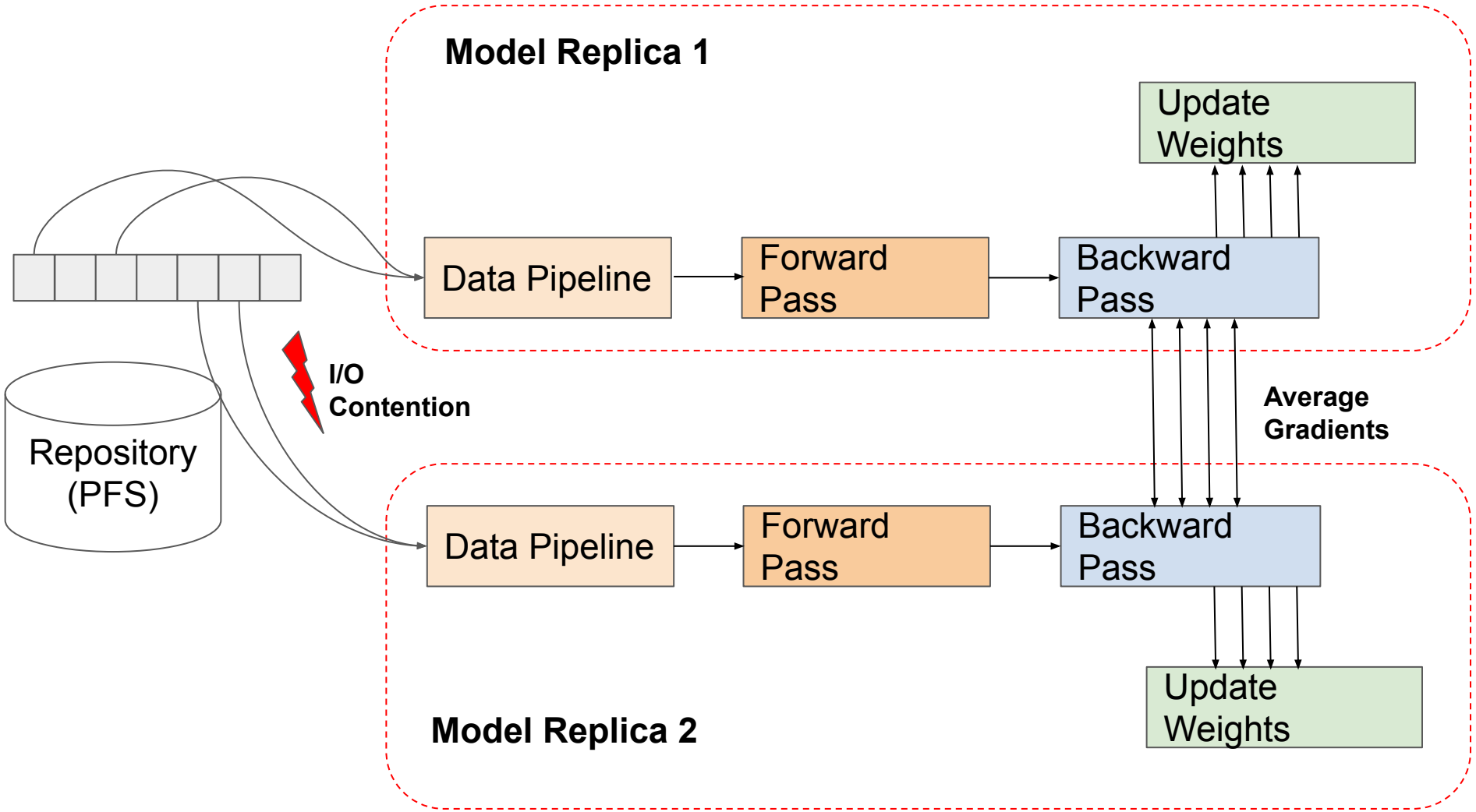


Asynchronous Overlapping with Training

- Data pipeline is running on CPUs/GPUs asynchronously with the training



Async Data Pipelines + Data Parallel Training: I/O Contention



Key Experimental Observations

- Every training sample is reused at each epoch
- Reuse distance may vary significantly due to pseudo-random sampling (Fig 4)
- Competition for I/O bandwidth has huge overheads on end-to-end runtime (Fig 5)
- Other stages in the pipeline (augmentation) scale with increasing number of threads only up to a point (Fig 6)

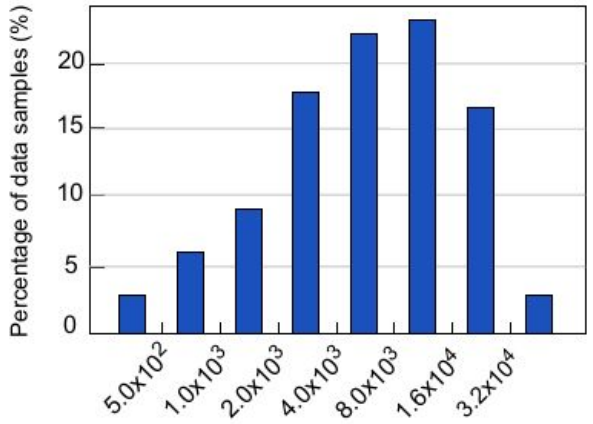


Figure 4: Histogram of the reuse distance of the training samples, measured in terms of numbers of iterations (X-axis)

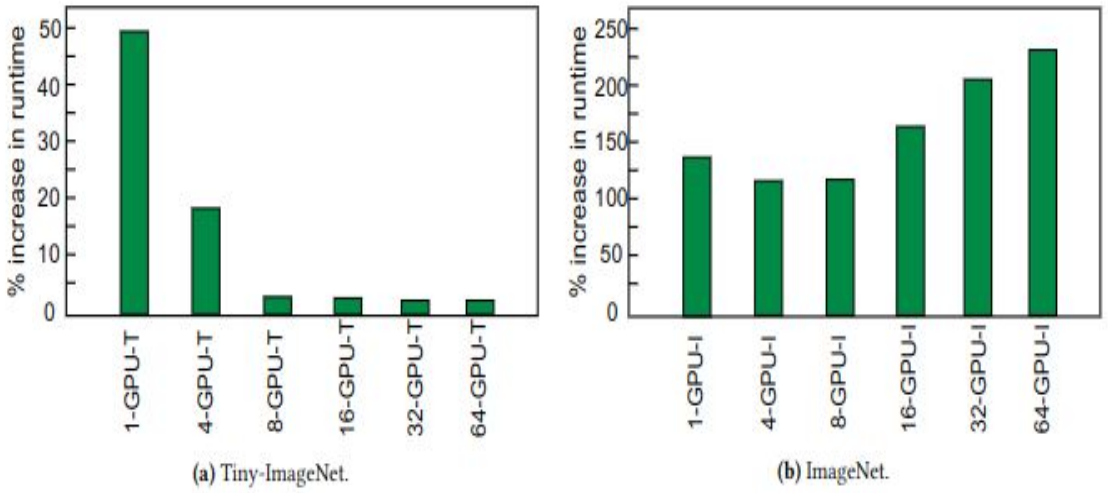


Figure 5. PFS impact measured as percent increase of end-to-end runtime over in-memory caching. The X-axis labels refer to the number of GPUs used for training with the tiny (T) or full (I) ImageNet dataset. Lower is better.

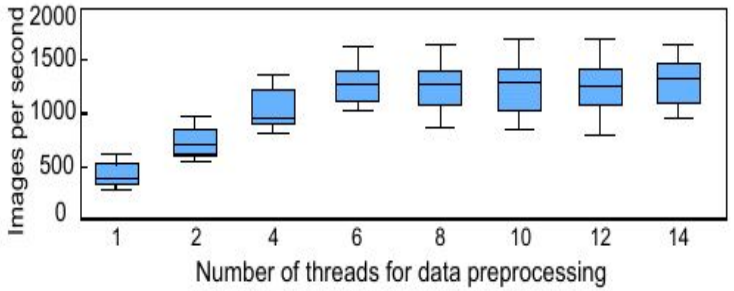


Figure 6: The impact of the number of preprocessing threads (X-axis) on the data preprocessing throughput (Y-axis)

J. Liu, B. Nicolae, D. Li: Lobster: Load Balance-Aware I/O for Distributed DNN Training. In ICPP'22
J. Liu, B. Nicolae, D. Li, J. Wozniak, T. Bicer, Z. Liu, I. Foster. Large Scale Caching and Streaming of Training Data for Online Deep Learning. In FlexScience@HPDC'22

Async Data Pipeline Stages Overheads

- Instrument data pipelines to measure data loading, decoding, pre-processing
- Zoom on the data pipelines of 3 out of 64 GPUs for various iterations
- Main observation: I/O operations become unpredictable under concurrency, leading to stragglers
- Consequences: Stragglers delay the whole iteration (due to gradient synchronization)

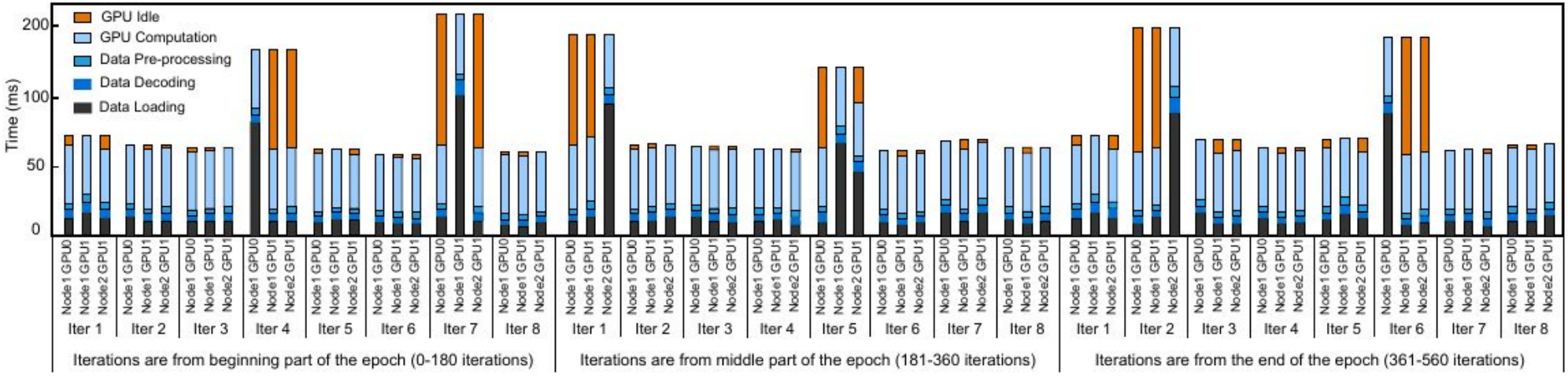


Figure 3: Execution time breakdown of the training pipeline for three GPUs.

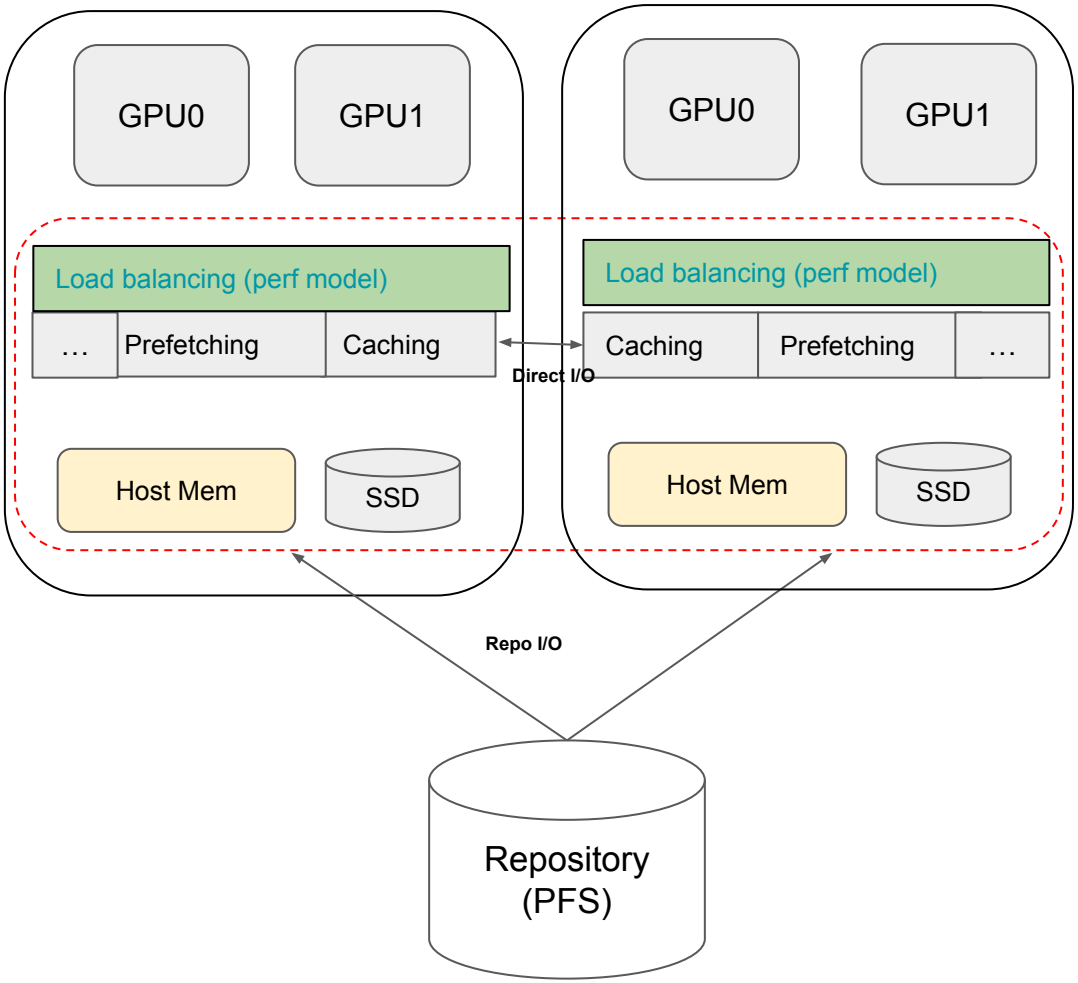
Solution: Async Collaborative Data Pipelines

Contribution:

- New runtime: Lobster: Optimized Data Pipeline for Data-Parallel Training

Key Ideas:

- Collaborative caching (get from remote cache instead of PFS to avoid I/O bottlenecks)
- Optimal prefetching (perfect knowledge about future I/O due to PRNG sampling)
- Performance model to predict stragglers
- Eviction policy based on reuse distance (coordinated with prefetching)
- Load balancing strategy to optimize the allocation of the threads to the data pipeline stages



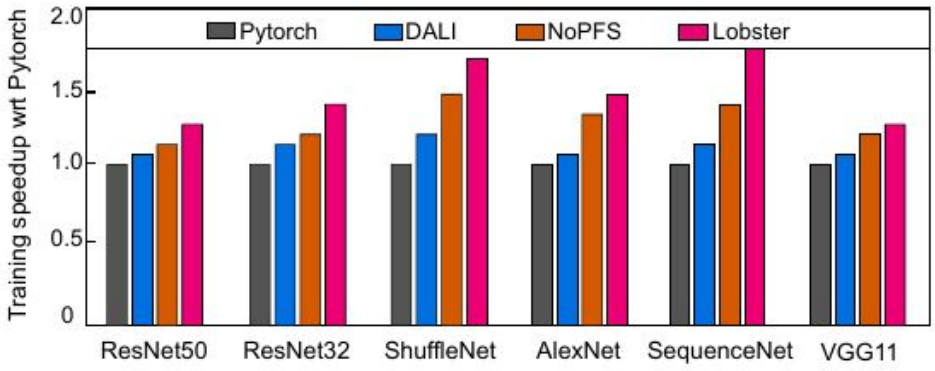
Implementation:

- Integrated with Tensorflow/PyTorch

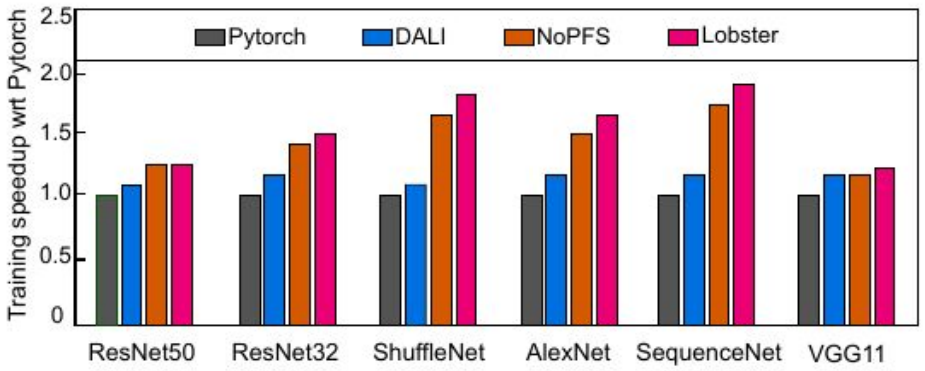
J. Liu, B. Nicolae, D. Li: Lobster: Load Balance-Aware I/O for Distributed DNN Training. In ICPP'22

Lobster Accelerates Training vs State-of-Art

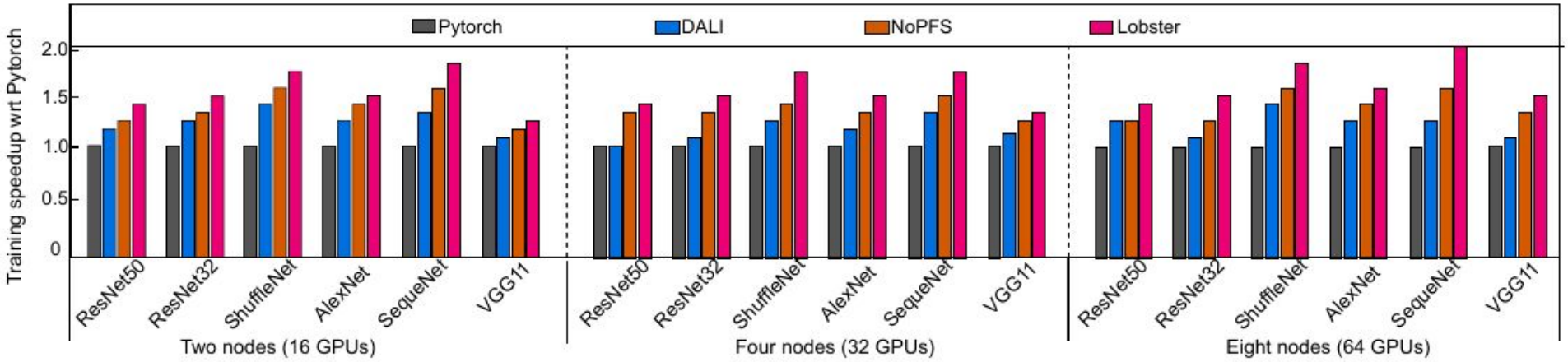
Compared approaches:
(1) PyTorch Baseline; (2) DALI (NVIDIA); (3) NoPFS (Collaborative Caching); (4) Lobster



(a) Multi-GPU training in a single node with ImageNet-1K



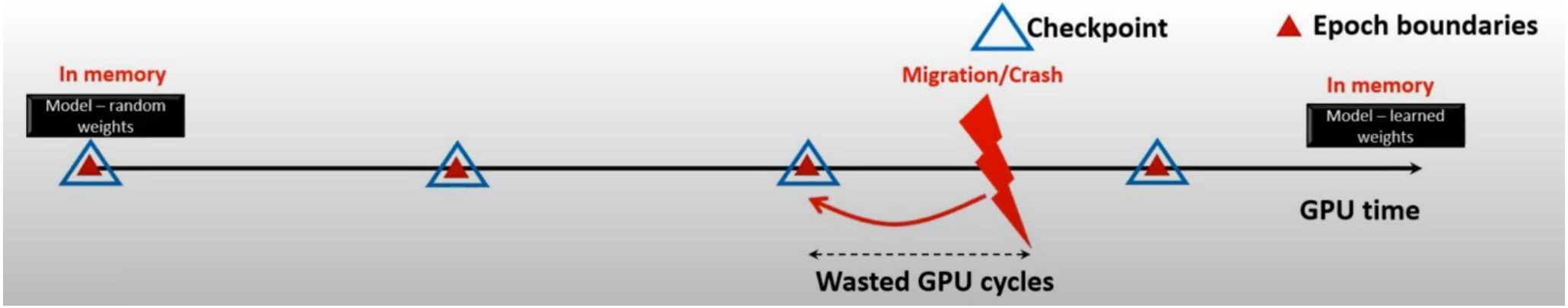
(b) Multi-GPU training in a single node with ImageNet-22K



(c) Distributed training across nodes with ImageNet-22K

Checkpoint-Restart of DL Models

Preliminaries of Checkpoint-Restart



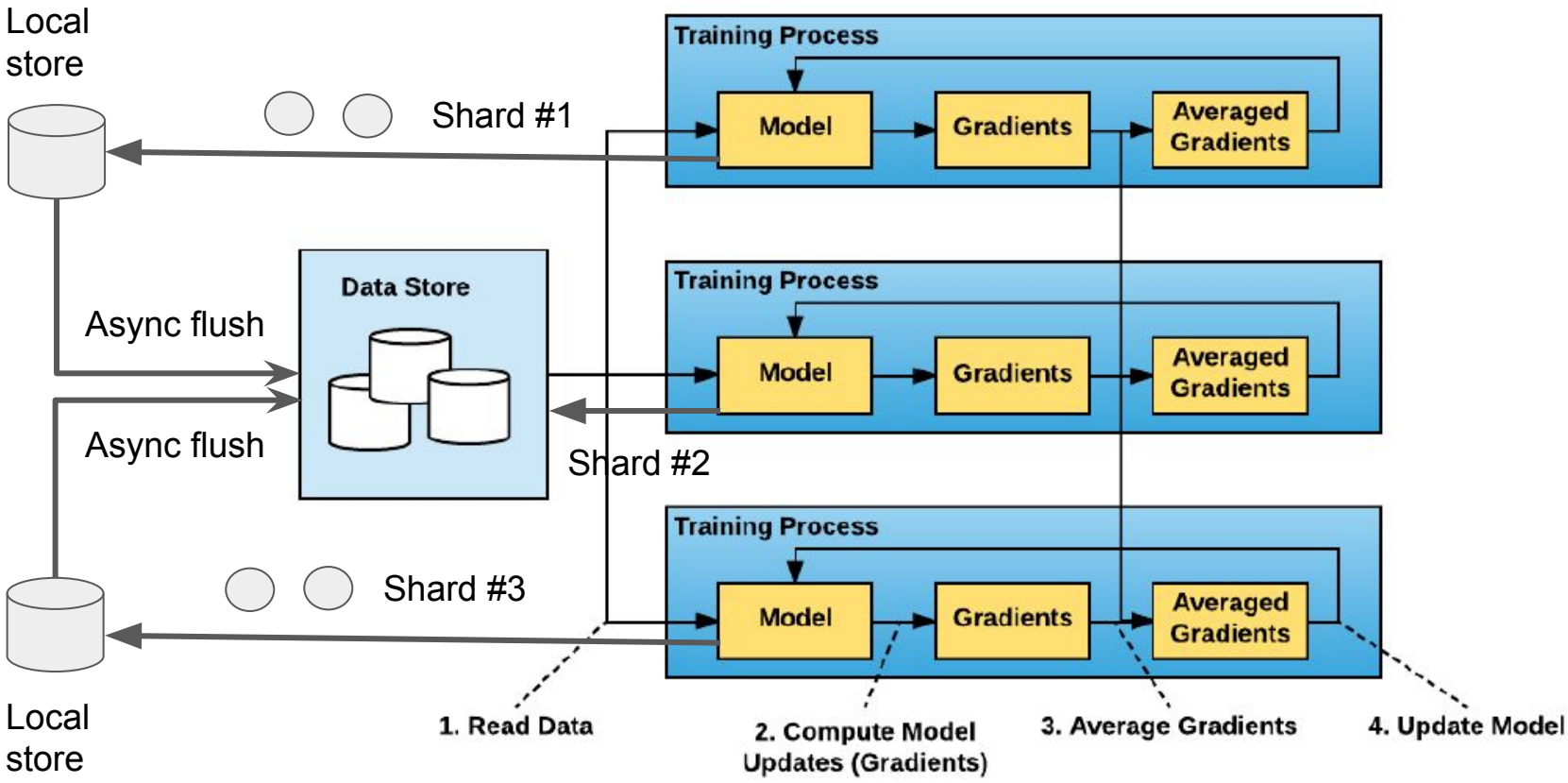
Why do we need Checkpoint-Restart?

- Fault tolerance: at large scale some GPUs always fail and we don't want to lose progress
- Reverting biases and inaccuracies: "unlearn patterns" by rolling back and restart the training using an alternative solution (e.g., exclude some mini-batches or change some layers to higher precision)

Checkpoint-Restart Techniques:

- DeepFreeze [Nicolae et al., CCGrid'20]
  - Augment execution graph (after weight updates) to capture a different layer shard on each worker from GPU to host memory (asynchronously)
  - Flush layer shards from host memory to storage (asynchronously)
- CheckFreq [Mohan et al., FAST'21]
  - Asynchronous serialization and snapshotting on the GPU (no data-parallel awareness)

DeepFreeze: Efficient Data-Parallel Checkpointing



- Naive solution: One process checkpoints, the others wait
- Better solution:
  - Each rank checkpoints a shard of the checkpoint
  - Multi-level checkpointing library flushes asynchronously to data store

DeepFreeze: How to Capture Shards Efficiently?

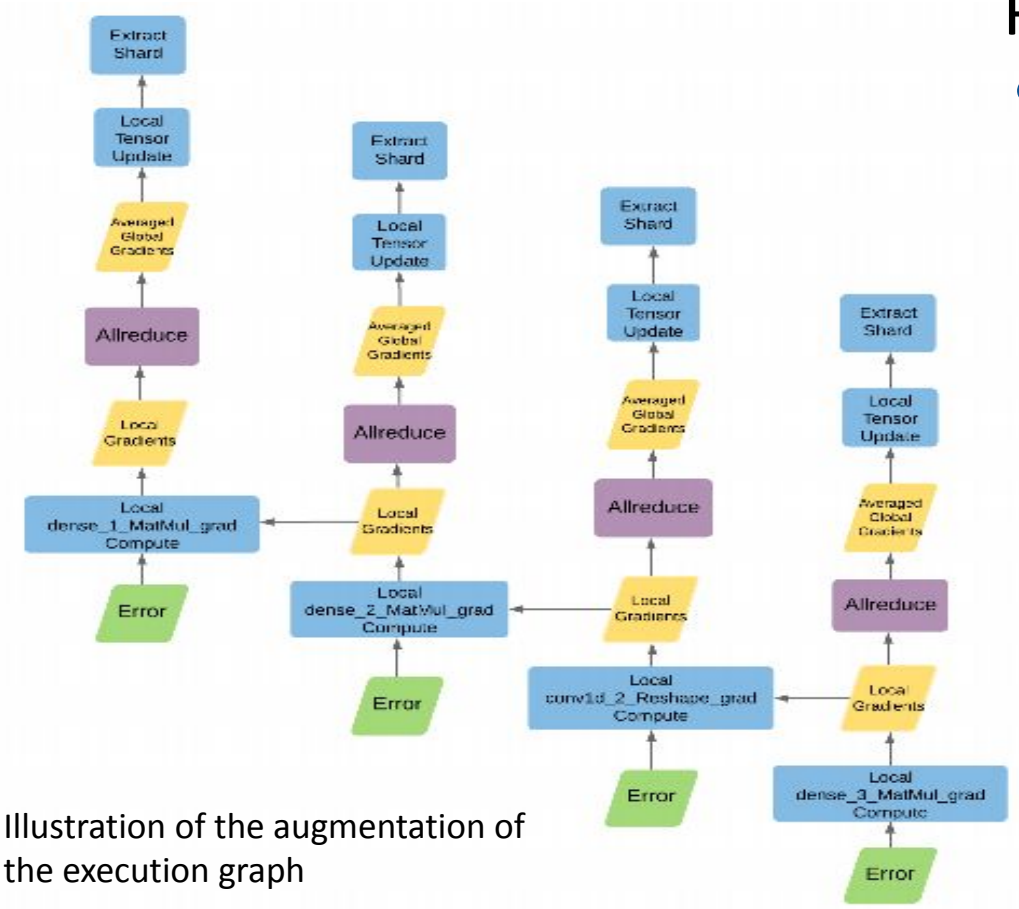


Illustration of the augmentation of the execution graph

Implementation for Tensorflow:

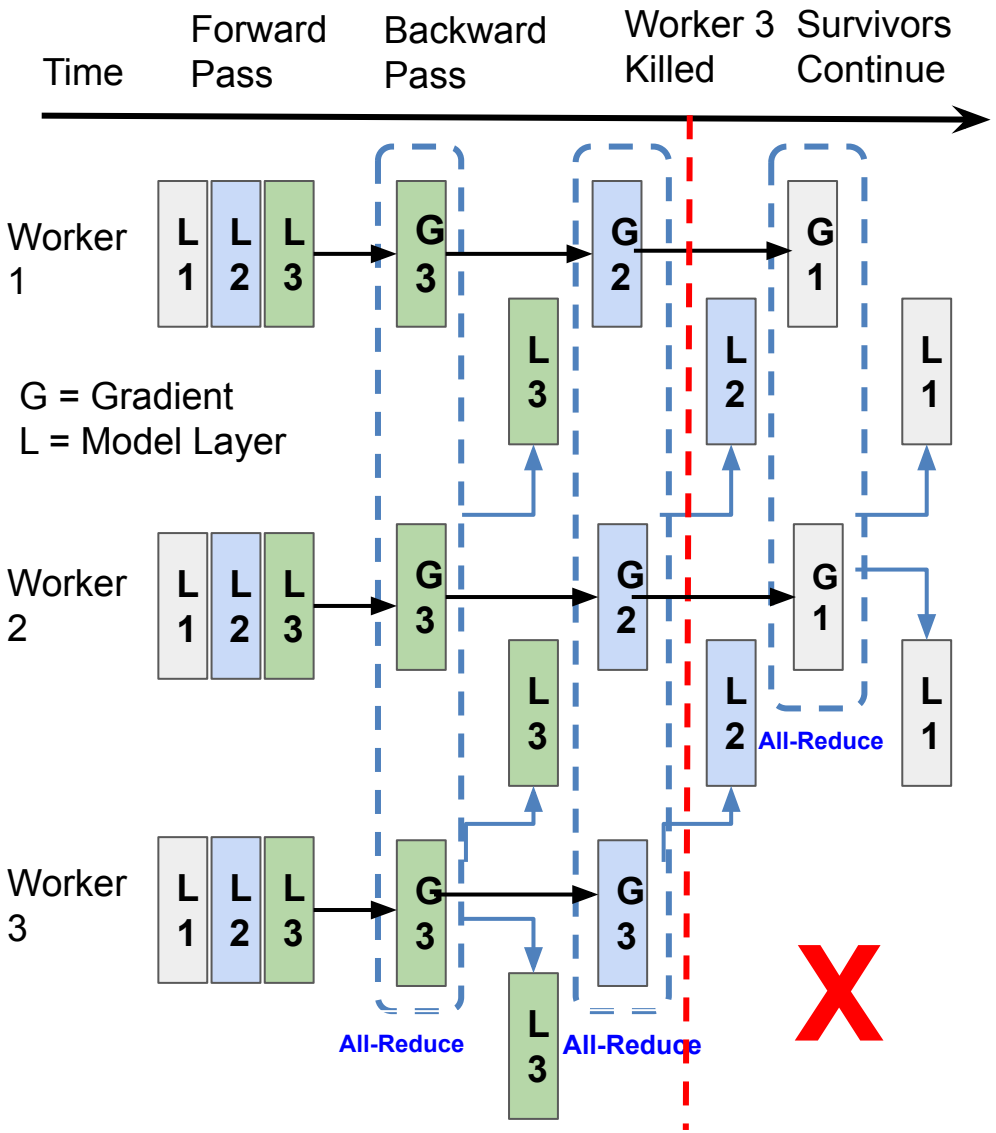
- Wrapper for optimizer that injects tensor operators after weight updates

Principles:

- **Augmentation of execution graph:**
  - No need to wait for back-prop to finish
  - Apply slice and serialize tensors immediately after weight updates (maximizes overlapping with back-prop)
- **Reduced blocking and straggler effect:**
  - Local checkpoint is ready after back-prop
  - Simply need to call checkpointing library
- **Leverage multi-level checkpointing**
  - State-of-art libraries are optimized for efficient serialization and async I/O
  - Based on VELOC (HPC checkpointing library)

Nicolae, B., Li, J., Wozniak, J., Bosilca, G., Dorier, M. and Cappello, F. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. CGrid'20

Alternative to C/R: Forward Recovery



Steps

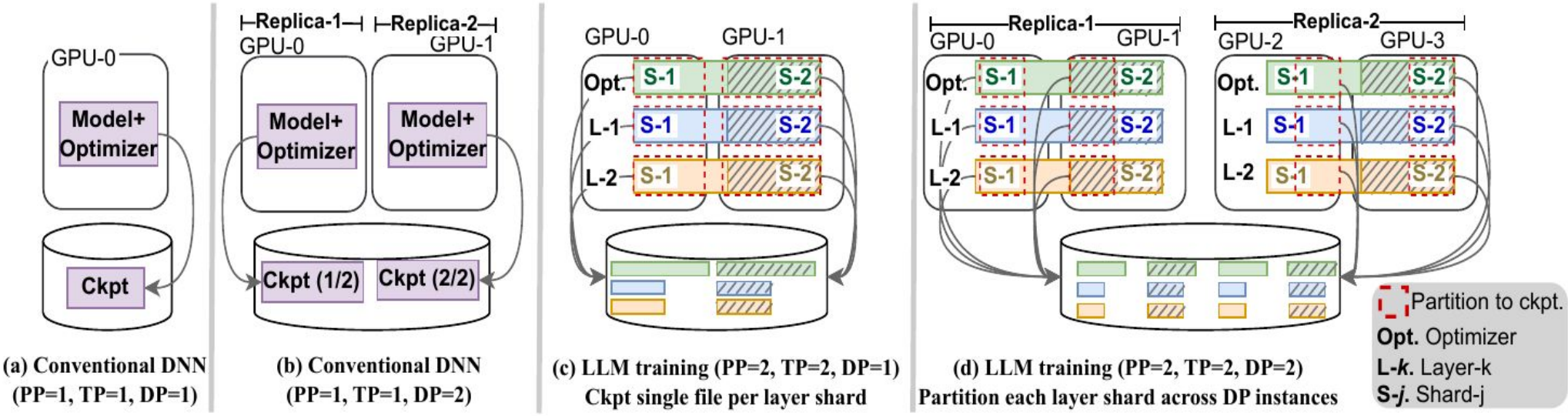
- Continue the backward pass with the survivors
- Average only the local gradients of the survivors

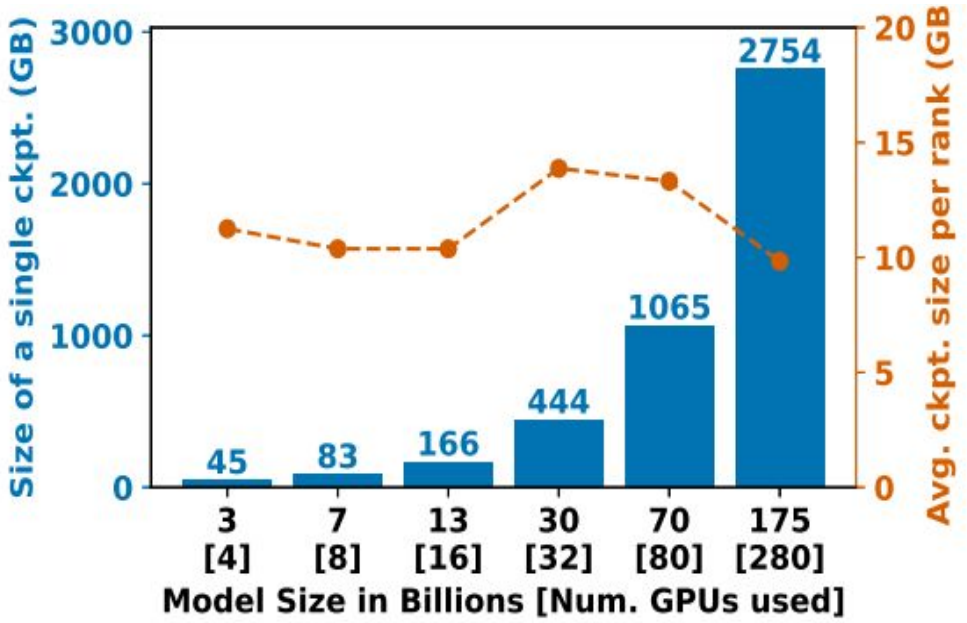
Observations

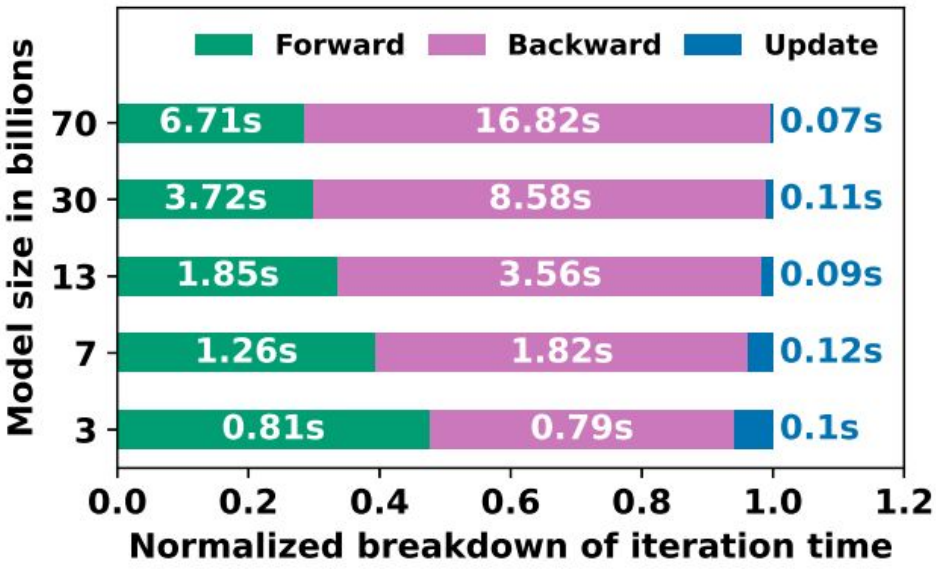
- The influence of some training samples will be lost
- No rollback overhead
- Spares can join the group later when it is convenient in asynchronous fashion

Nicolae, B., Hobson, T., Yildiz, O., Peterka, T. and Morozov, D. 2022. Towards Low-Overhead Resilience for Data Parallel Deep Learning. *CCGrid'22: The 22th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing* (Messina, Italy, 2022),

LLM Checkpointing: Analysis of Overheads



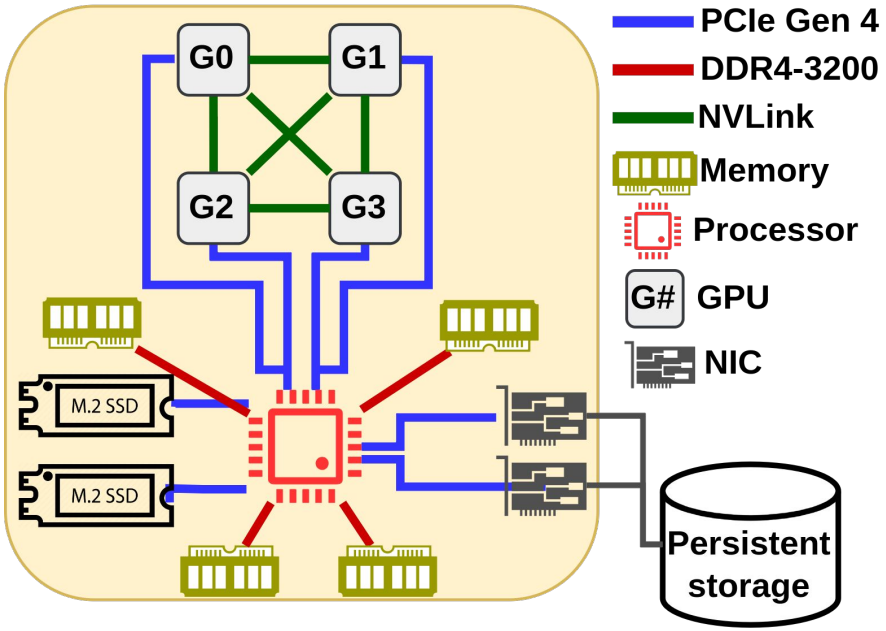




- Testbed: ALCF Polaris (A100 GPUs, maximized memory utilization)
- Studied Two LLMs: LLAMA (7B, 13B, 70B, 175B) Bloom (3B, 30B)

Contribution: DataStates-LLM Checkpointing

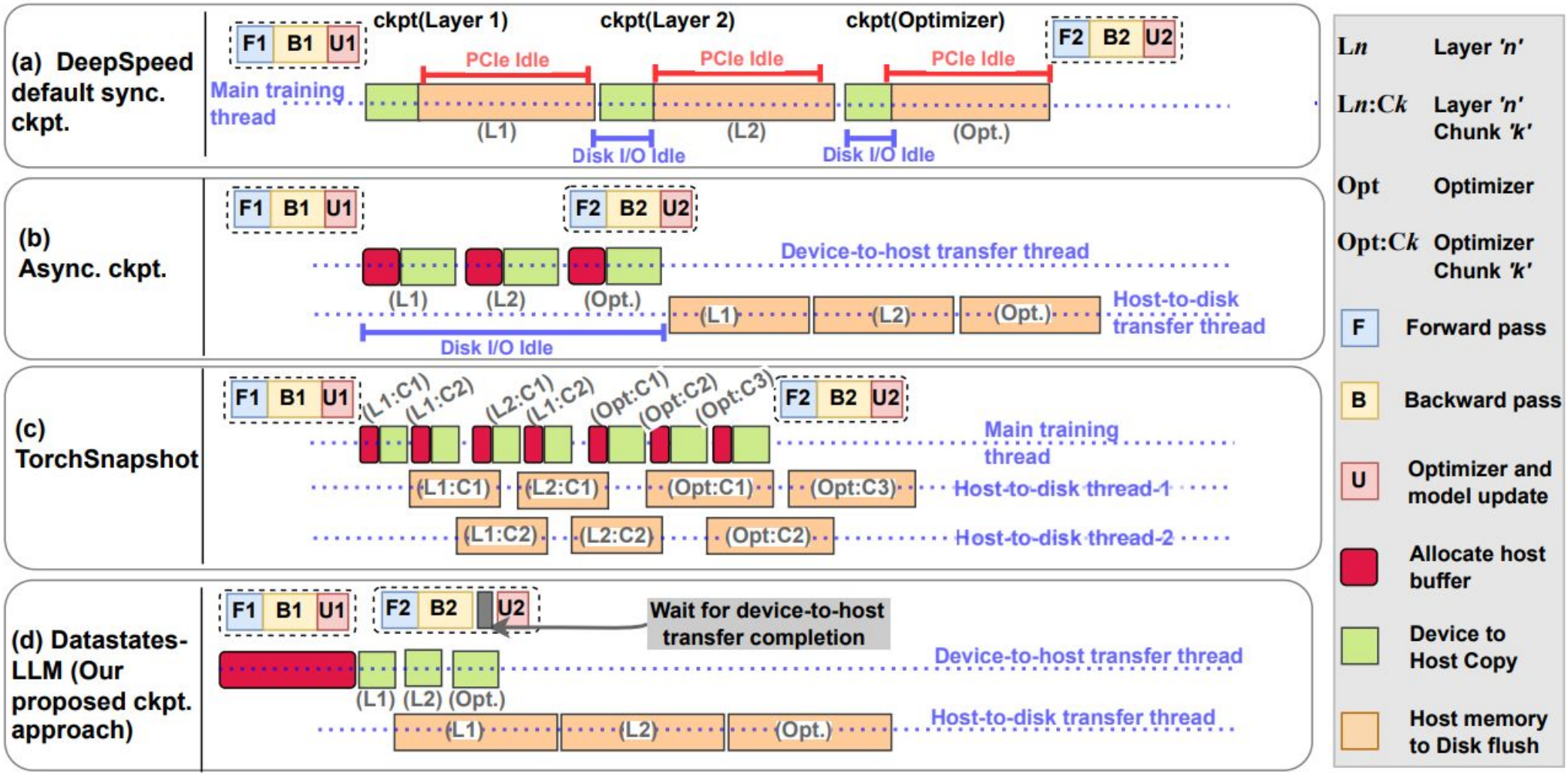
- Design objectives
  - Capture **globally consistent** model and optimizer states
  - **Maximize checkpointing throughput** to minimize blocking time during training
  - **Minimize resource competition** (memory, interconnect, network) while checkpointing



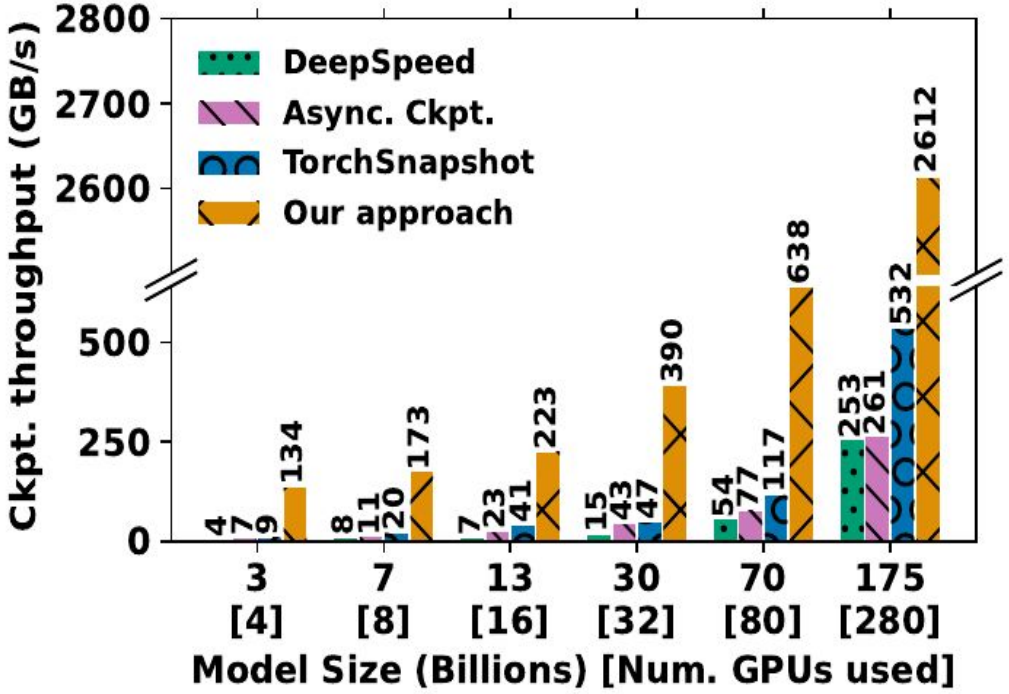
Multi-GPU node on ALCF Polaris

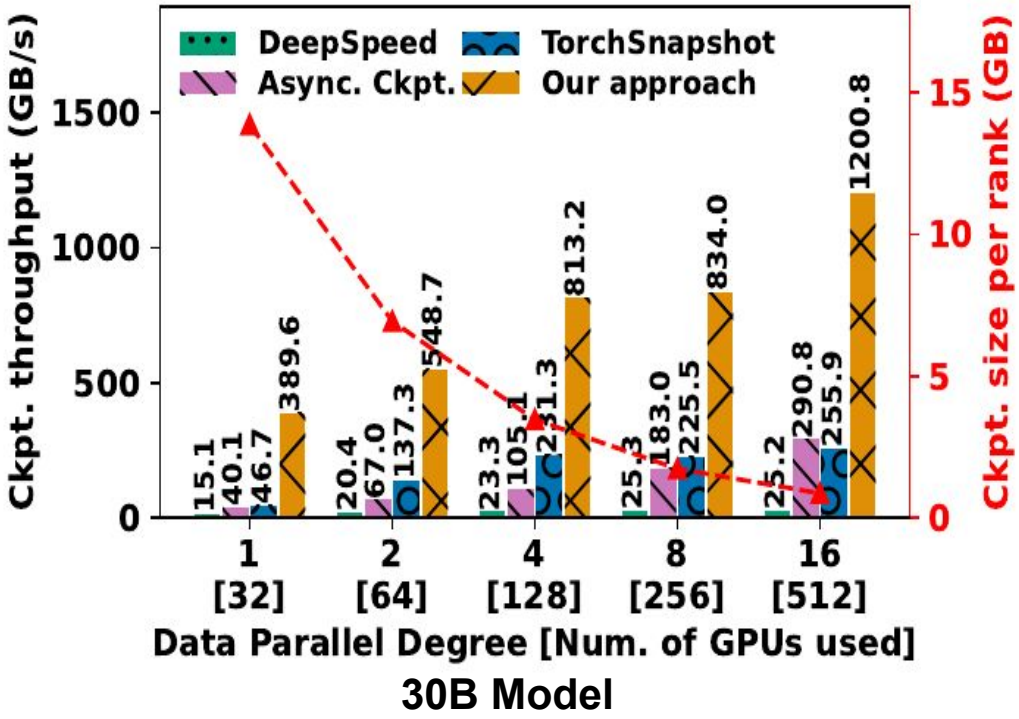
- Key ideas
  - Overlap device-to-host transfers with immutable stages of training (forward pass, backward pass)
  - Async flushes from host memory to persistent storage (PFS)
  - Deferred consolidation

DataStates-LLM Checkpointing vs State-of-Art



Highlight of Results

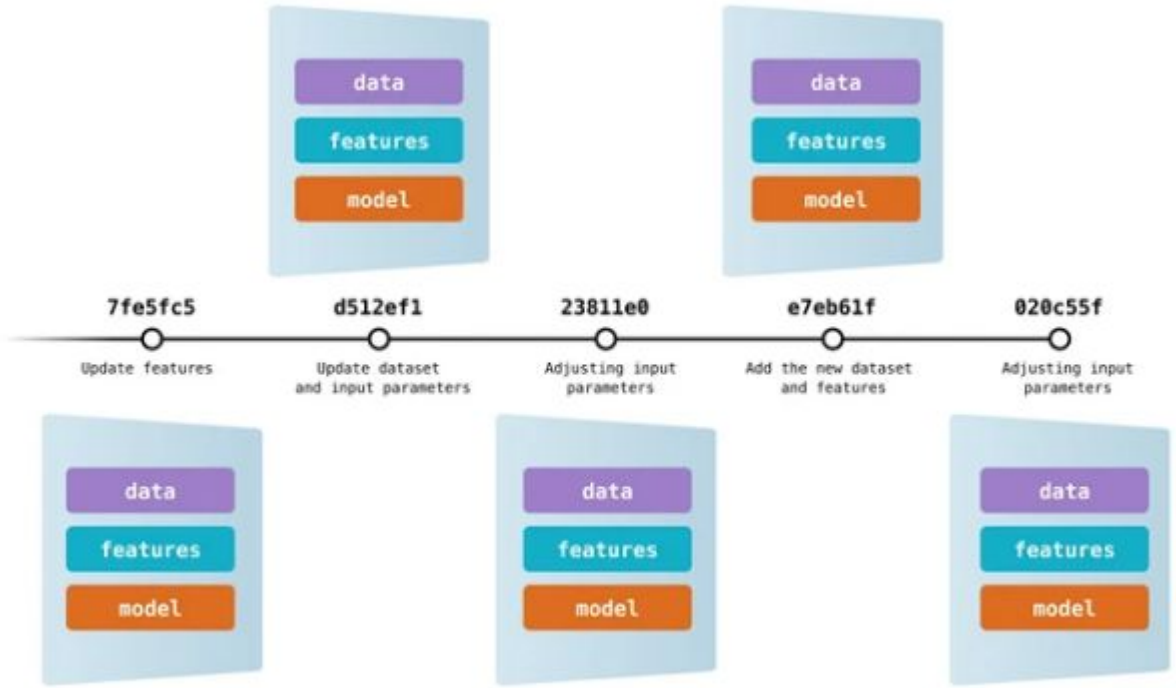




- Testbed: ALCF Polaris (A100 GPUs)
- Compared Approaches:
  - DeepSpeed (aynchronous)
  - Async Ckpt (simple async implementation)
  - TorchSnapshot (asynch multi-threaded)
  - Our approach (overlap device-to-host with fwd pass)
- Conclusions: **4x faster** for single replica, **5x faster** for data-parallel (multiple replicas)

DL Model Versioning and Lineage for Derived Models Based on Transfer Learning and Fine-Tuning

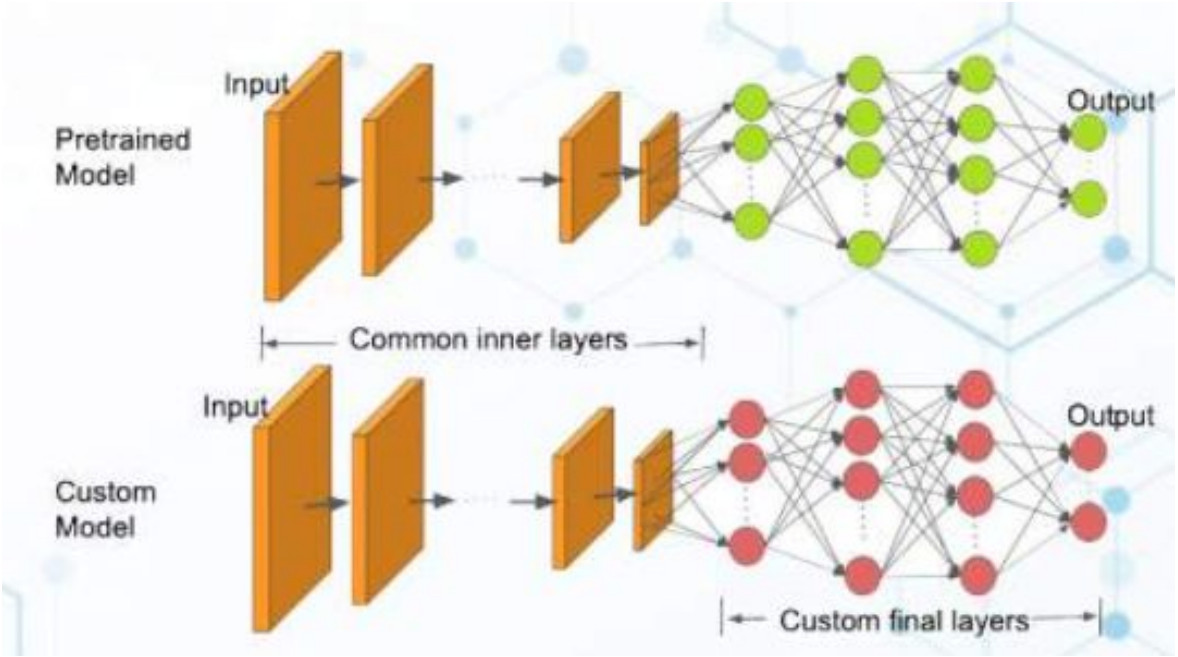
DL Model Versioning and Lineage is Fundamental



Many scenarios

- Provenance
- Explainability
- Sharing between workflow tasks, different teams, etc. (MLDevOps)
- Better AI techniques

Transfer Learning and Fine-Tuning



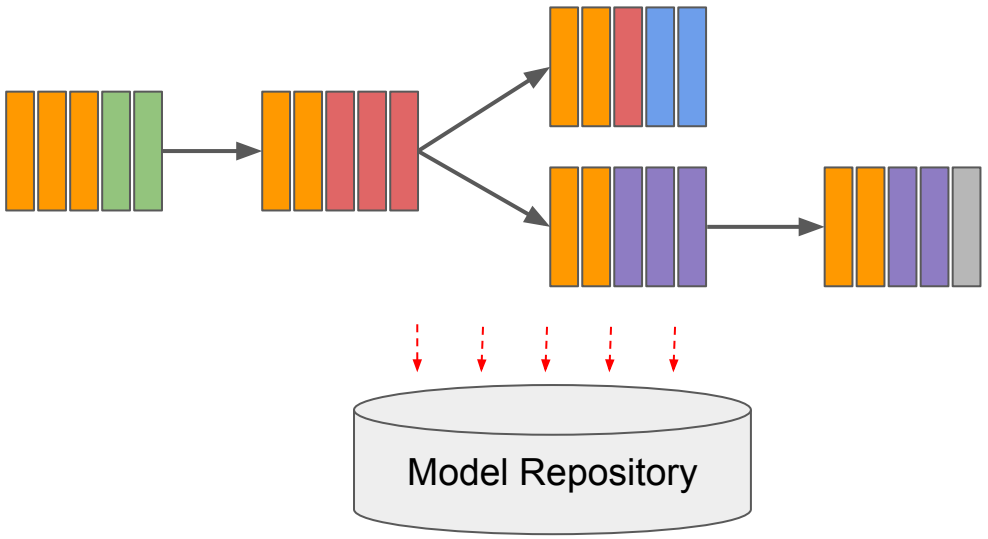
Transfer Learning

- Common layers inherit learned features
- Final layers learn specific features
- Training the custom model:
  - Copy weights of common layers
  - Initialize weights of final layers with random values
  - Freeze common layers and update the weights of final layers iteratively as usual
- Why does this accelerate the training?
  - Faster convergence
  - Faster individual iterations

Fine-Tuning

- Same architecture
- Typically updates all layers
- Specific instance of transfer learning

Data Problem: Long Chain of Derived DNN Models



Challenges

- How do we store the lineage efficiently (without duplication of layers and metadata)?
- How do we find a relevant DNN model in the lineage?
- How do we study the ancestry to understand why transfer learning works?

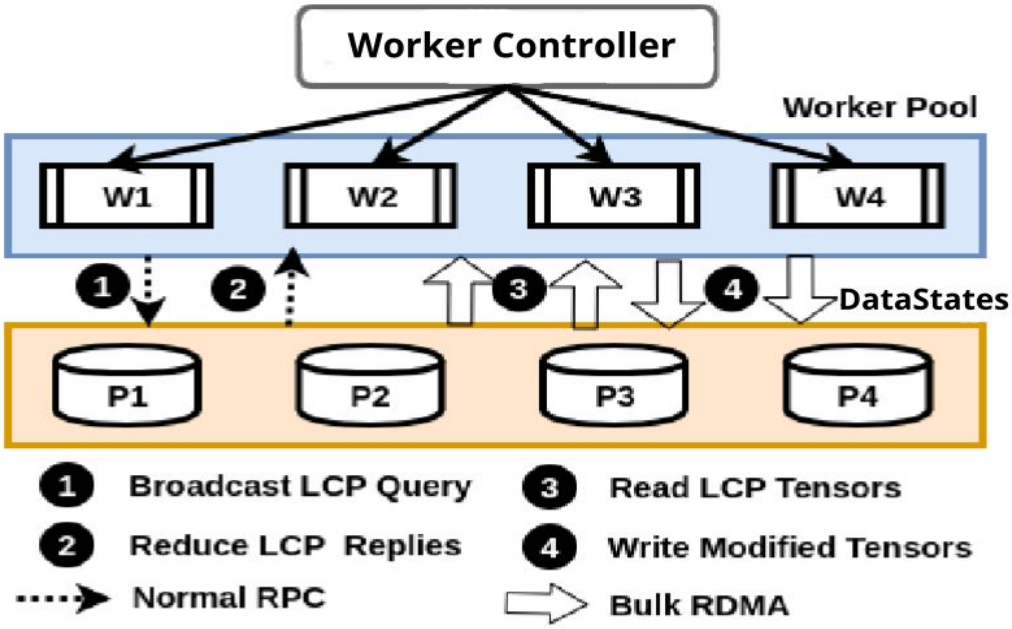
EvoStore: Lineage-Driven AI Model Repository

Challenges

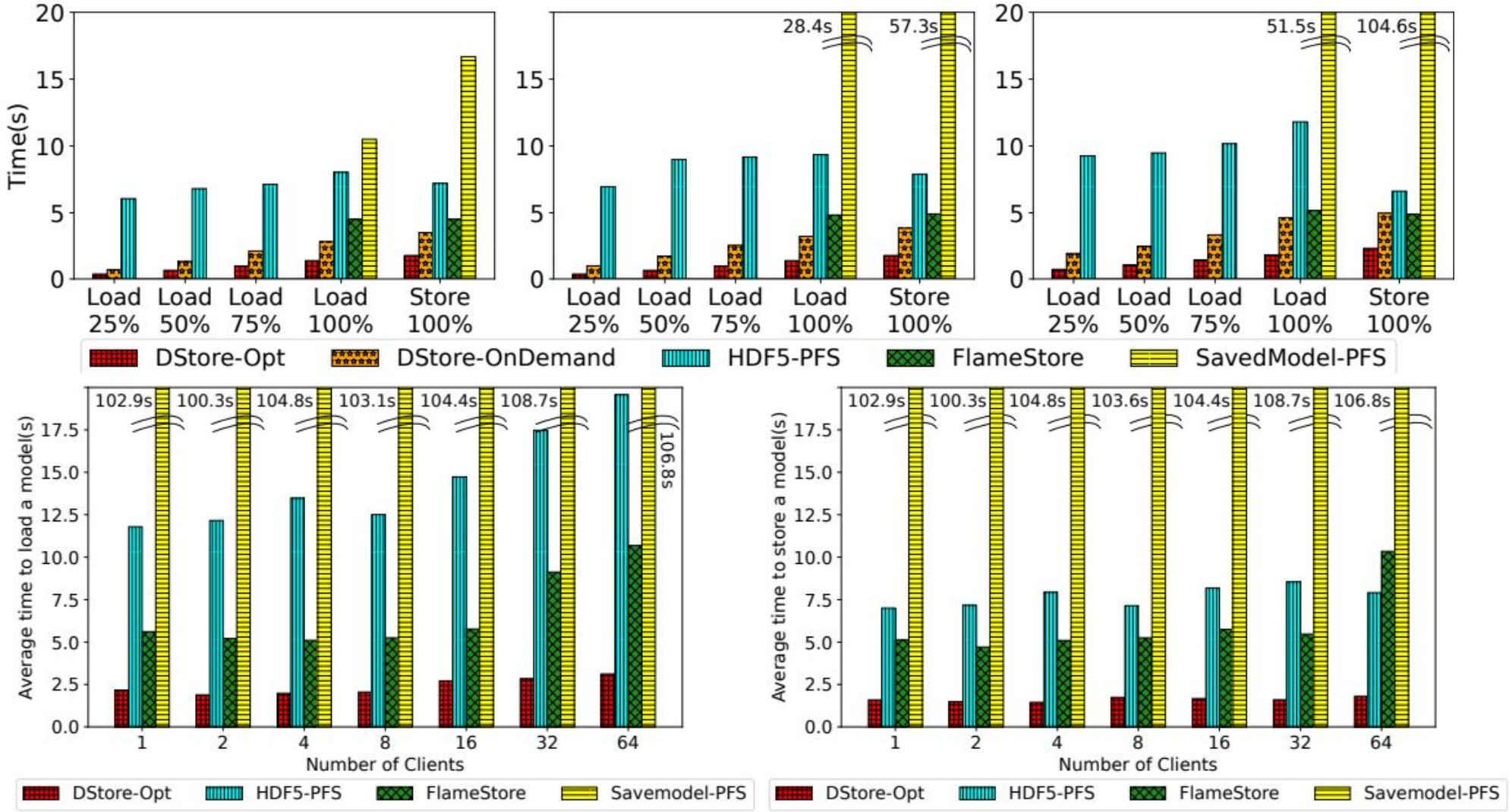
- **Search queries:** What model in the repository is the best fit for specified features?
- **Ancestry queries:** How did the models derive from each other? What is the most recent ancestor of a frozen layer?
- **Fine-grain tensor-level access:** Efficient I/O of individual/groups of tensors
- **Space efficiency:** How do we apply incremental storage techniques at tensor-level for layers that were frozen during transfer learning?
- **Scalability:** Large number of workers evaluate model candidates in parallel, they need efficient access the repository under concurrency both in terms of metadata (queries) and data (tensors)

Our approach

- Each DNN model candidate is a new data state
- Lineage captures the evolution of transfer learning and addresses search queries and ancestry queries
- **Incremental storage** of modified parts of the data states enables space efficiency
- Decentralized data and metadata distributed enables **scalable queries** and **fine-grain access**



EvoStore: Results

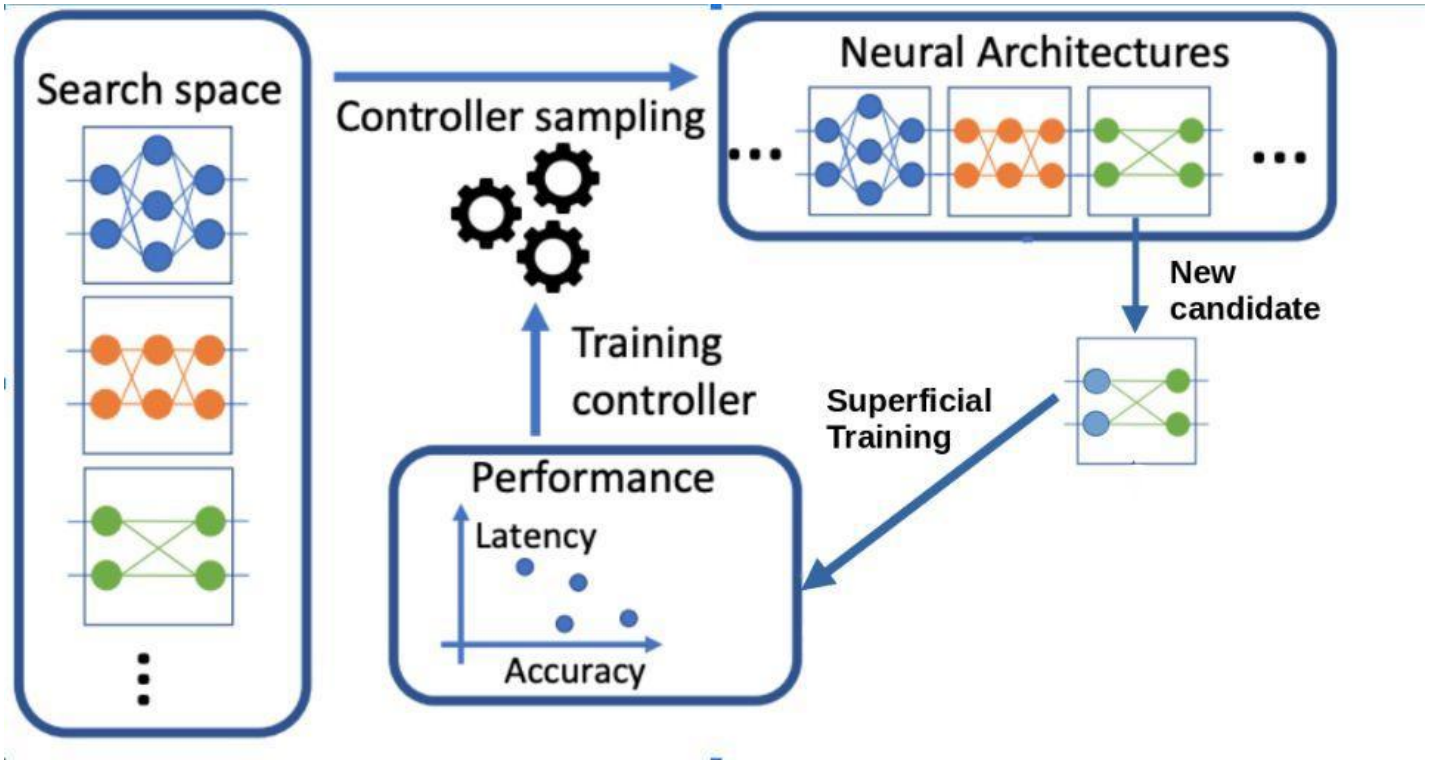


Conclusion: Reduction in load/store overhead 2x-60x compared with state-of-art

M. Madhyastha, R. Underwood, R. Burns, B. Nicolae. DStore: A Lightweight Scalable Learning Model Repository with Fine-Grain Tensor-Level Access. In ICS'23: The 2023 International Conference on Supercomputing

Basics of Network Architecture Search

One of the most challenging aspects of AI is finding a suitable learning model architecture

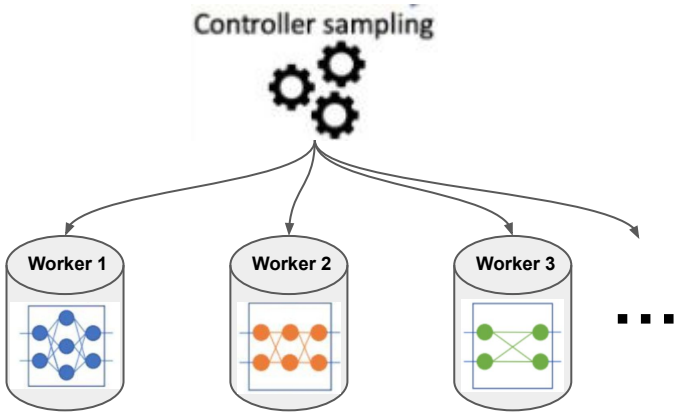


What are possible solutions?

- Naive strategy: random sampling from search space (defined as a meta-model with choices)
- More advanced evolution-based strategies (mutate a population of top-K models)
- Training new candidates from scratch (random weights) avoids intermediate data (only architecture and score retained)

DeepHyper: Network Architecture Search at Scale

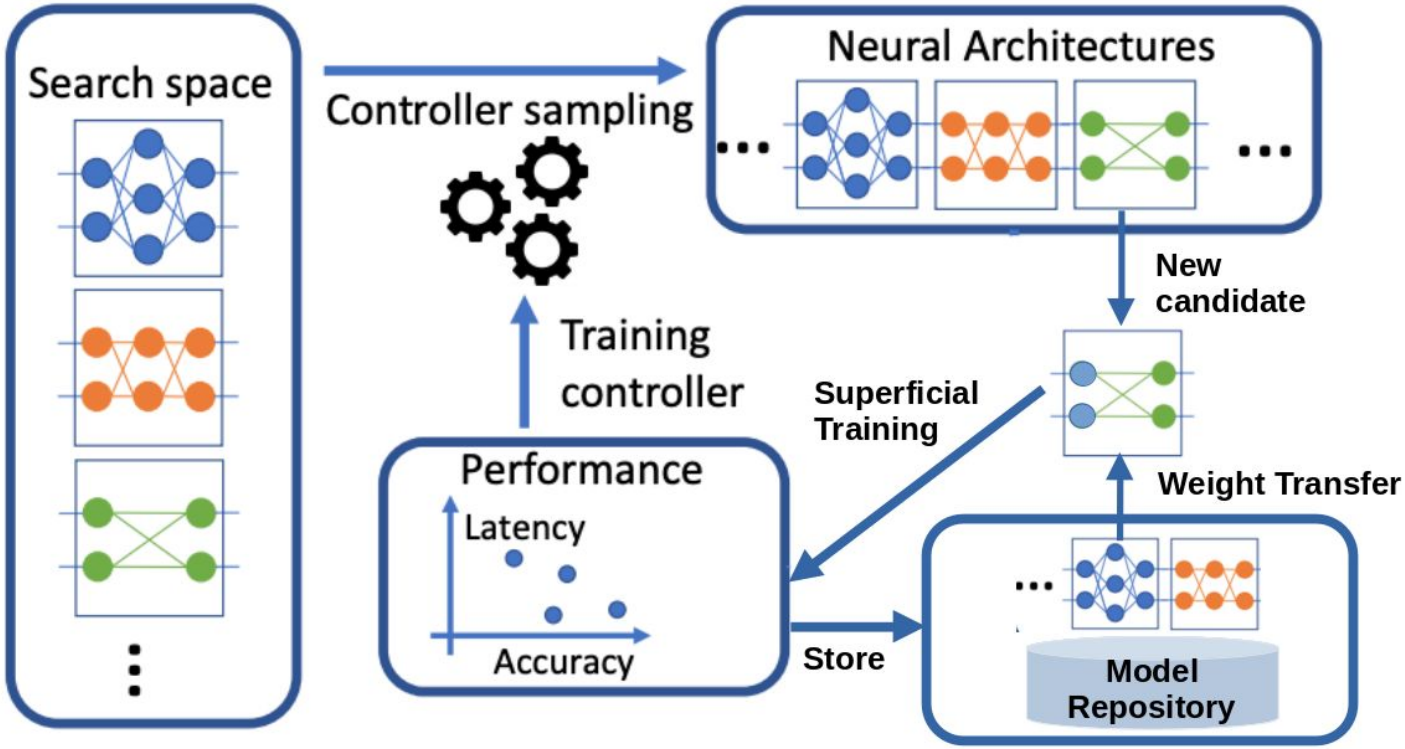
```
if __name__ == "__main__":
    from deephyper.evaluator import Evaluator
    from deephyper.evaluator.callback import TqdmCallback
    from deephyper.search.hps import CBO

    # we give a budget of 2 minutes for each search
    timeout = 120
    parallel_evaluator = Evaluator.create(
        black_box.run_ackley,
        method="process",
        method_kwargs={"num_workers": 5})
    parallel_search = CBO(problem, parallel_evaluator, random_state=42)
    results["parallel"] =parallel_search.search(timeout=timeout)
```



Master-Worker Distributed Pattern

Network Architecture Search with Transfer Learning

One of the most challenging aspects of AI is finding a suitable learning model architecture

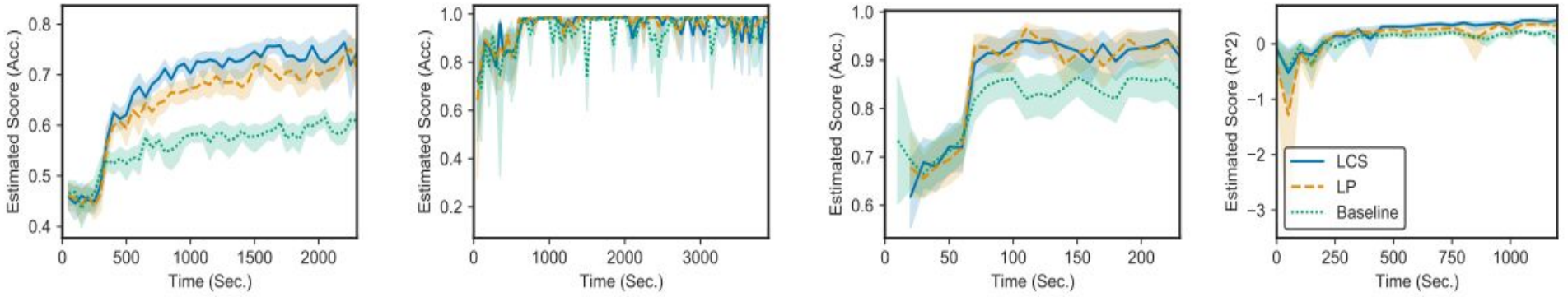


Better Strategy: Transfer Learning

- What if we don't start from random weights, but from weights of previous candidates?
- This makes sense because most candidates are similar and share a large part of the architecture
- This accelerates superficial training and/or produces better top-K candidates
- Needs a scalable repository for DL models (DataStates!)

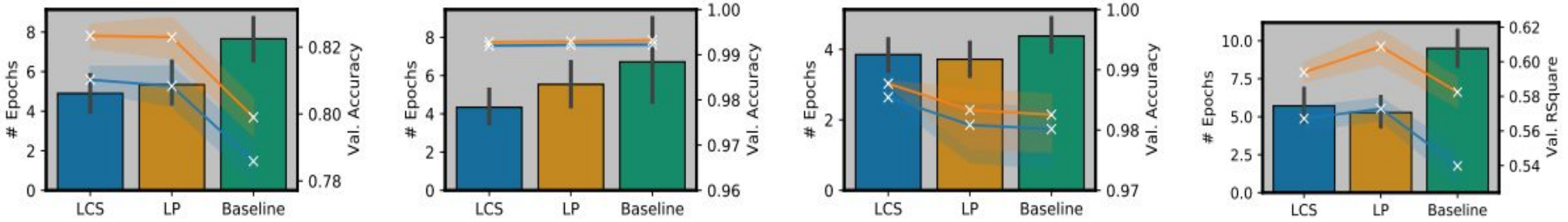
Hongyuan Liu, Bogdan Nicolae, Sheng Di, Franck Cappello, Adwait Jog. Accelerating DNN Architecture Search at Scale using Selective Weight Transfer. In CLUSTER'21: The 2021 IEEE International Conference on Cluster Computing.

Network Architecture Search with Transfer Learning



(a) CIFAR-10 (b) MNIST (c) NT3 (d) Uno

Fig. 7: Estimated objective metrics (scores) of the candidate models during NAS runtime. After the beginning phase, with our weight transfer techniques, the estimated objective metrics increase significantly compared to the baseline.



(a) CIFAR-10 (b) MNIST (c) NT3 (d) Uno

Fig. 8: LCS and LP achieve 1.5× and 1.4× speedup vs. training from scratch for full training (using early stopping). LCS and LP also achieve better or comparable objective metrics. The bars show the average number of epochs before early stopping. The blue lines show the objective metrics with early stopping. The orange lines show the objective metrics of the fully trained models.

Conclusion: weight transfer by capturing and reusing the model state speeds up NAS and improves model quality

NAS with Transfer Learning Using EvoStore

- Scalable lineage (incremental storage of frozen layers) using ancestry maps
- Scalable graph prefix query support (identify quickly best candidate for transfer learning)
- Caching/Prefetching optimizations based on inheritance patterns
- Application: CANcer Distributed Learning Environment (CANDLE)

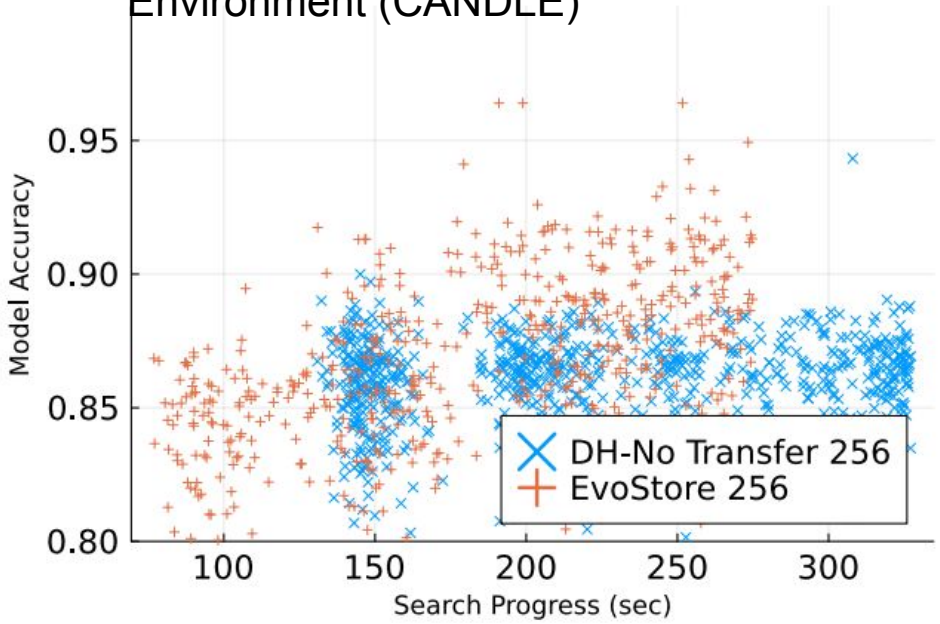
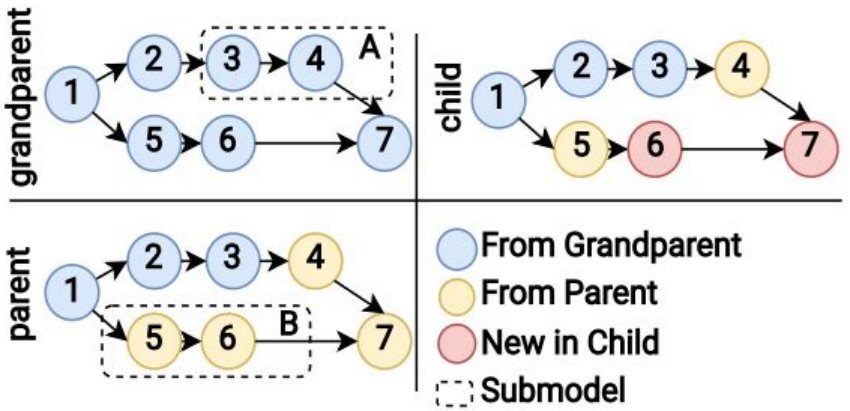
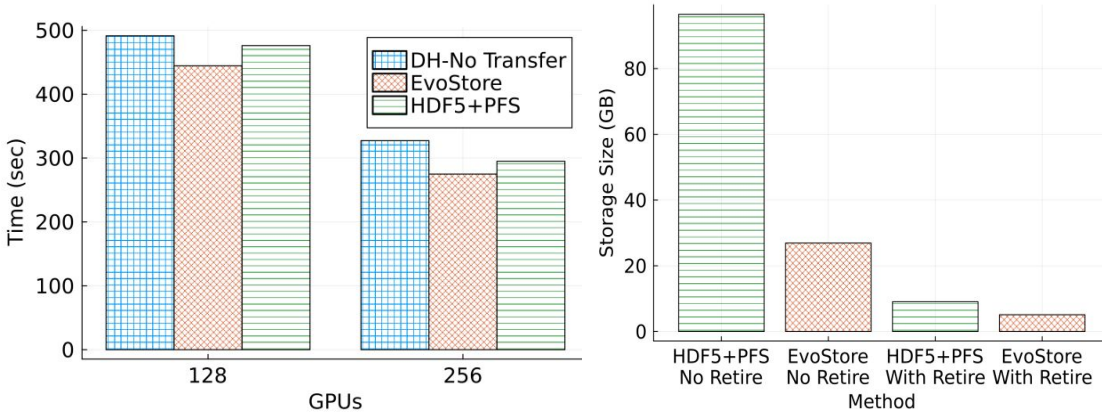


Figure 6: Accuracy of the DL model candidates over time: DeepHyper based on transfer learning facilitated by EvoStore produces candidates with high accuracy (>80%) much faster than DH-NoTransfer. Furthermore, the average accuracy of the candidates is significantly higher and the overall runtime is significantly shorter.





EvoStore accelerates the search by up to 20% (left) and reduces the storage space by up 80% (right) compared with HDF5=based model storage

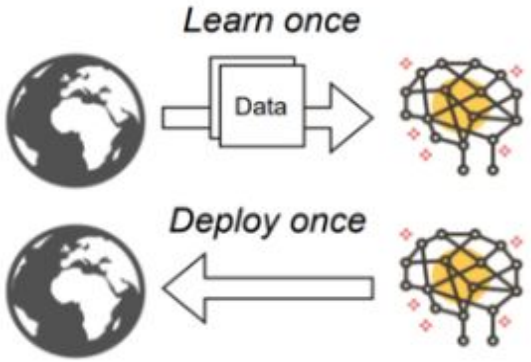
R. Underwood, M. Madhyastha, R. Burns, B. Nicolae. Understanding Patterns of Deep Learning Model Evolution in Network Architecture Search. In HiPC'23

Continual Learning

Offline Learning vs. Continual Learning

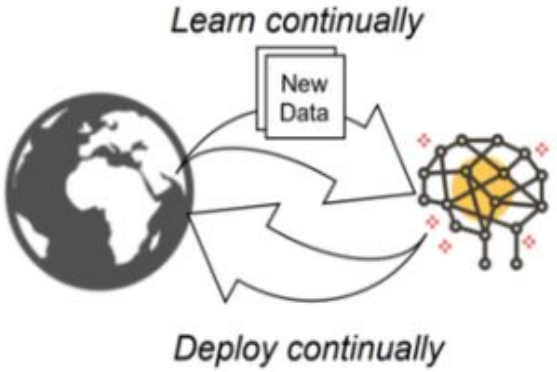
Offline Learning:

- Assumes all data is available in advance
- Trains the model once
- Runs all subsequent inferences on the same model instance



Continual Learning:

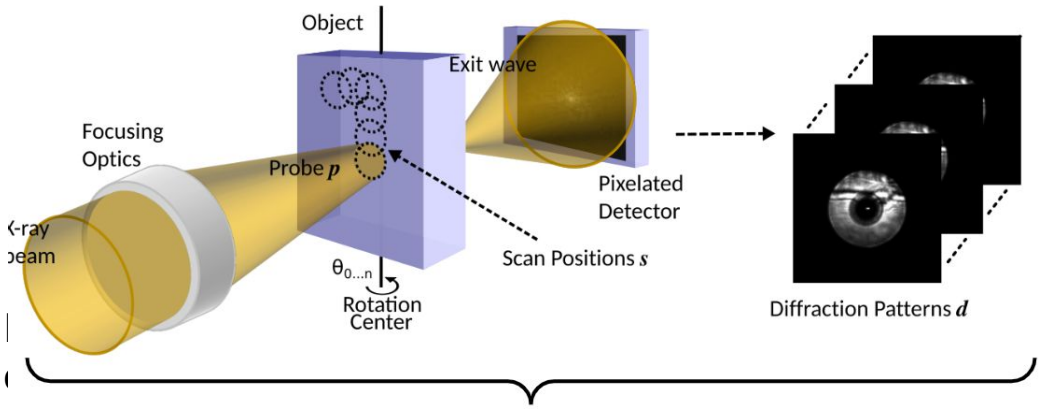
- New training data becomes available while we are running inferences
- Need to update the model in near-real time to accommodate shifts and new patterns of data



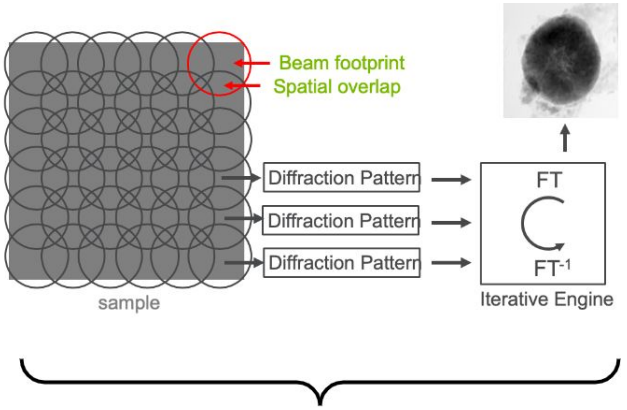
Why is this challenging:

- Cannot afford to retrain the DNN model from scratch (too slow, too much data that constantly accumulates)
- Cannot update the model by simply training it with the new samples (due to catastrophic forgetting: bias towards new samples)

Example: Ptychography

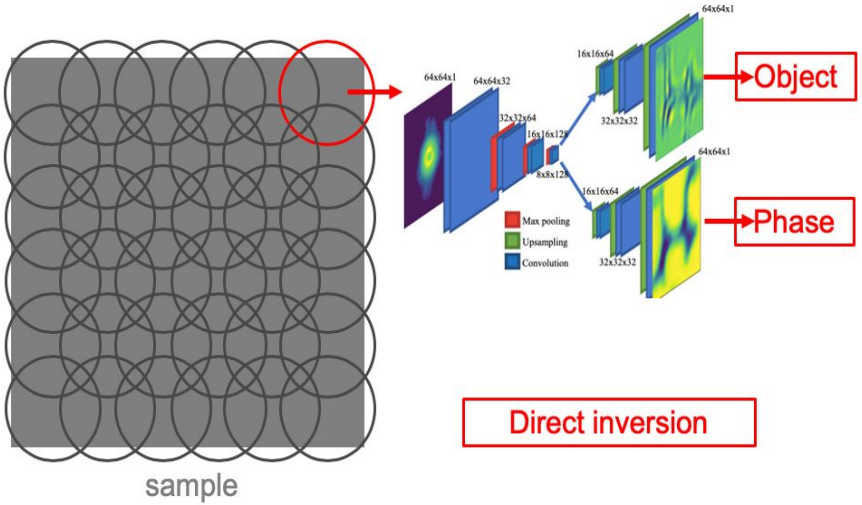


Data acquisition at the Advanced Photon Source (APS)



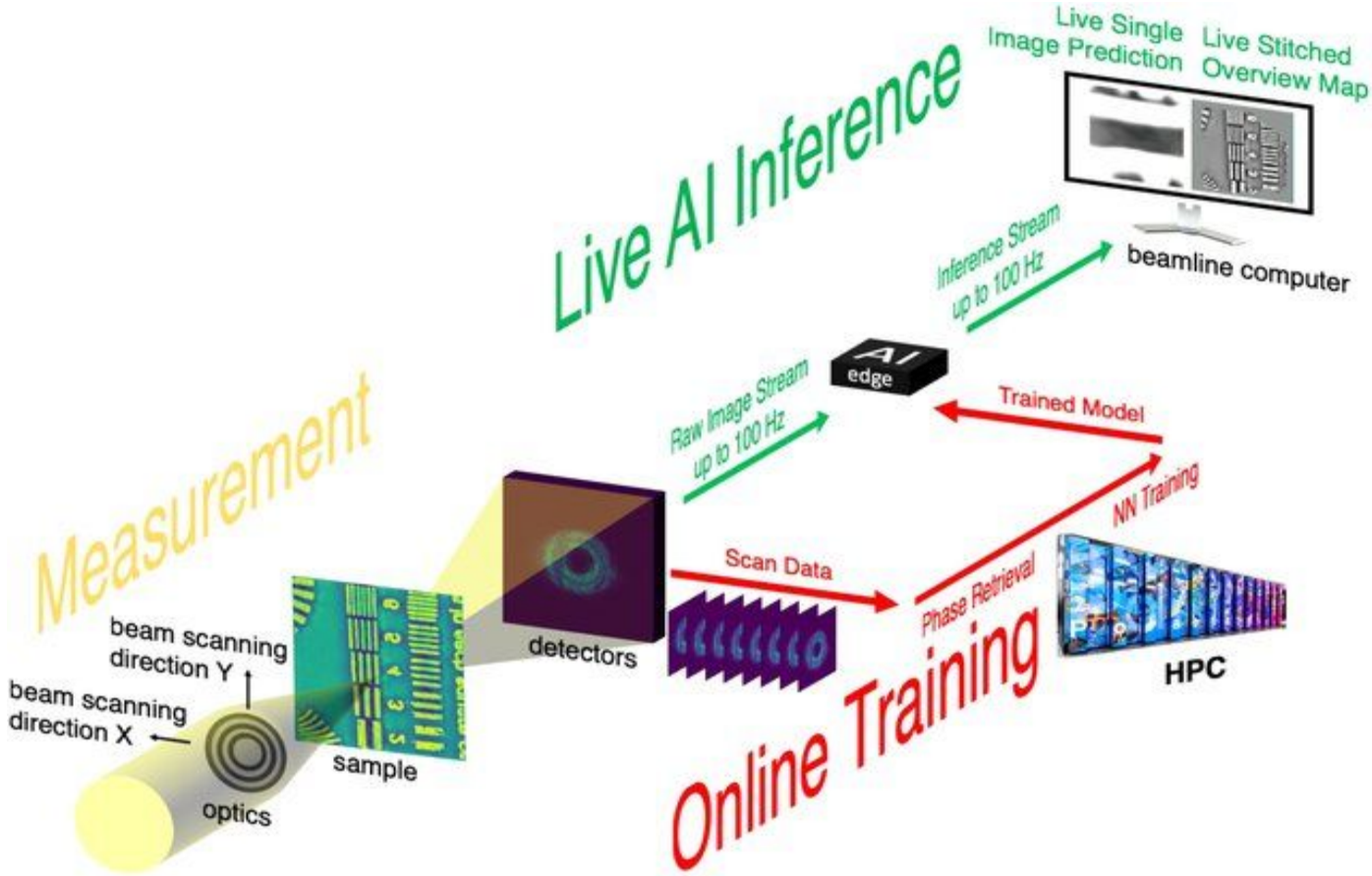
Ptychography reconstruction on
Theta (ground truth)

- Iterative reconstruction methods are computationally expensive ⇨ preclude real-time imaging
- Using a CNN-based autoencoder [Cherukara, 2020] that learns a direct mapping from diffraction patterns to output images is >100X faster and requires 25X less data

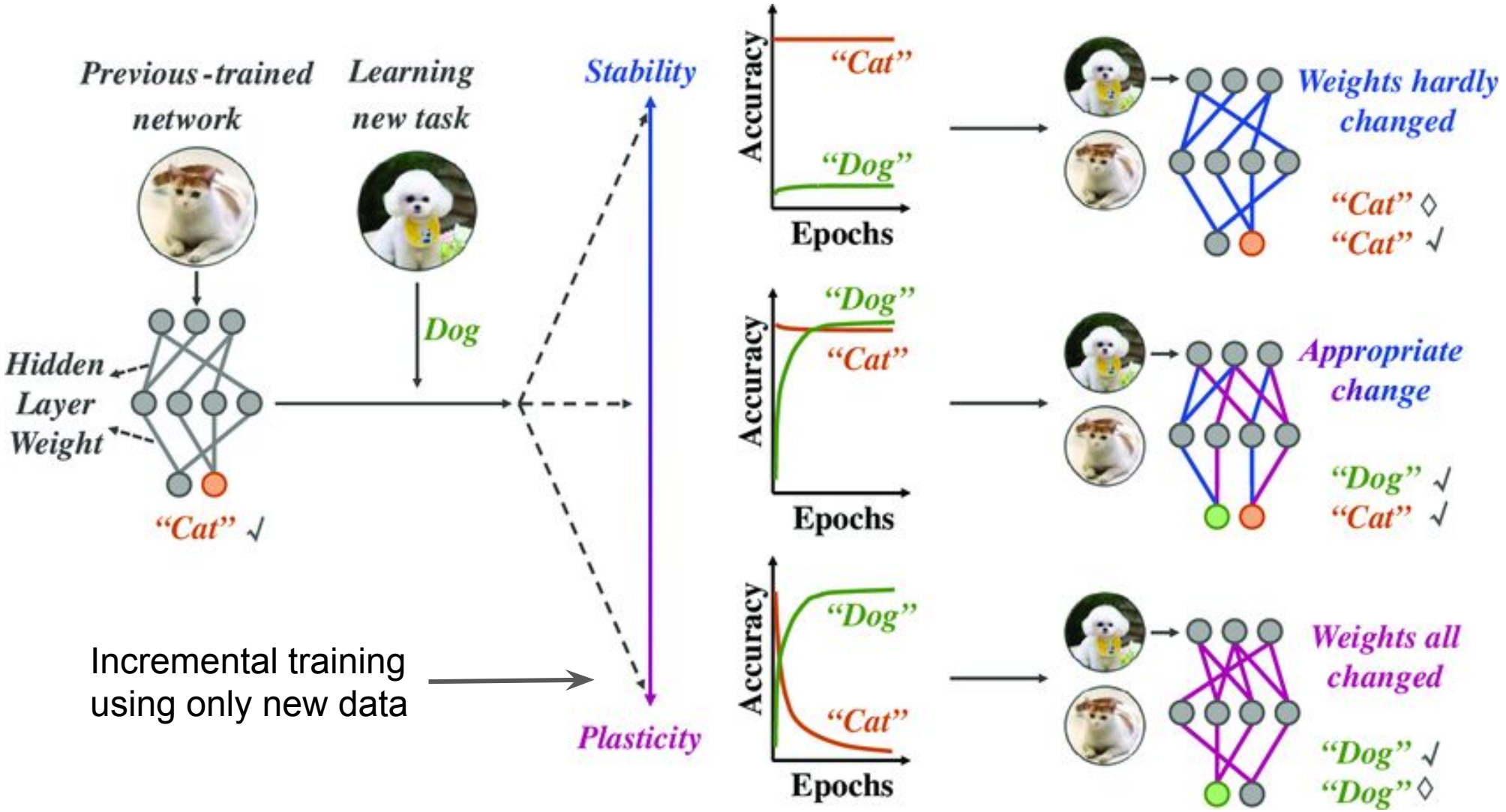


Accelerating Ptychography with Continual DL Learning

- We want to replace expensive iterative methods with DL model inferences
- The **DL model is trained in parallel** using the result of iterative methods as ground truth
- After the DL model is accurate enough, we switch to it
- Higher overhead in the beginning (both iterative method and DL training), but much lower overhead later
- **Continual learning**: the **data is acquired continuously**, thus offline training is not feasible



Why can’t we simply train a DL model incrementally?



- Catastrophic forgetting: Bias in favor of new training data
- Often undesirable and needs to be mitigated

Rehearsal: Mitigating Catastrophic Forgetting

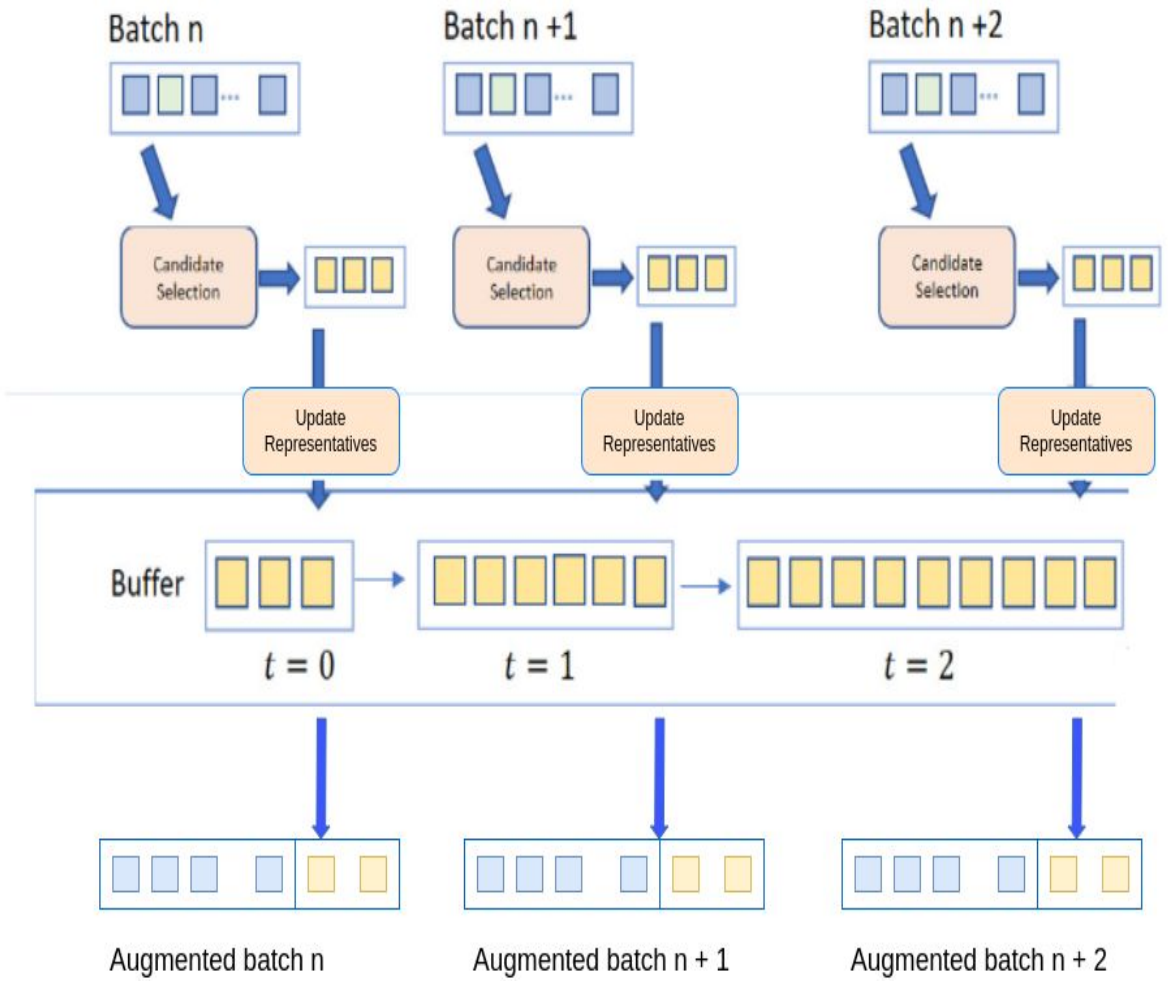
Naive Incremental Learning (NIL) `[Munoz, 2020]` is a naive strategy based on random selection and random update of representatives

1. Select data samples from incoming minibatches using a *selection policy*

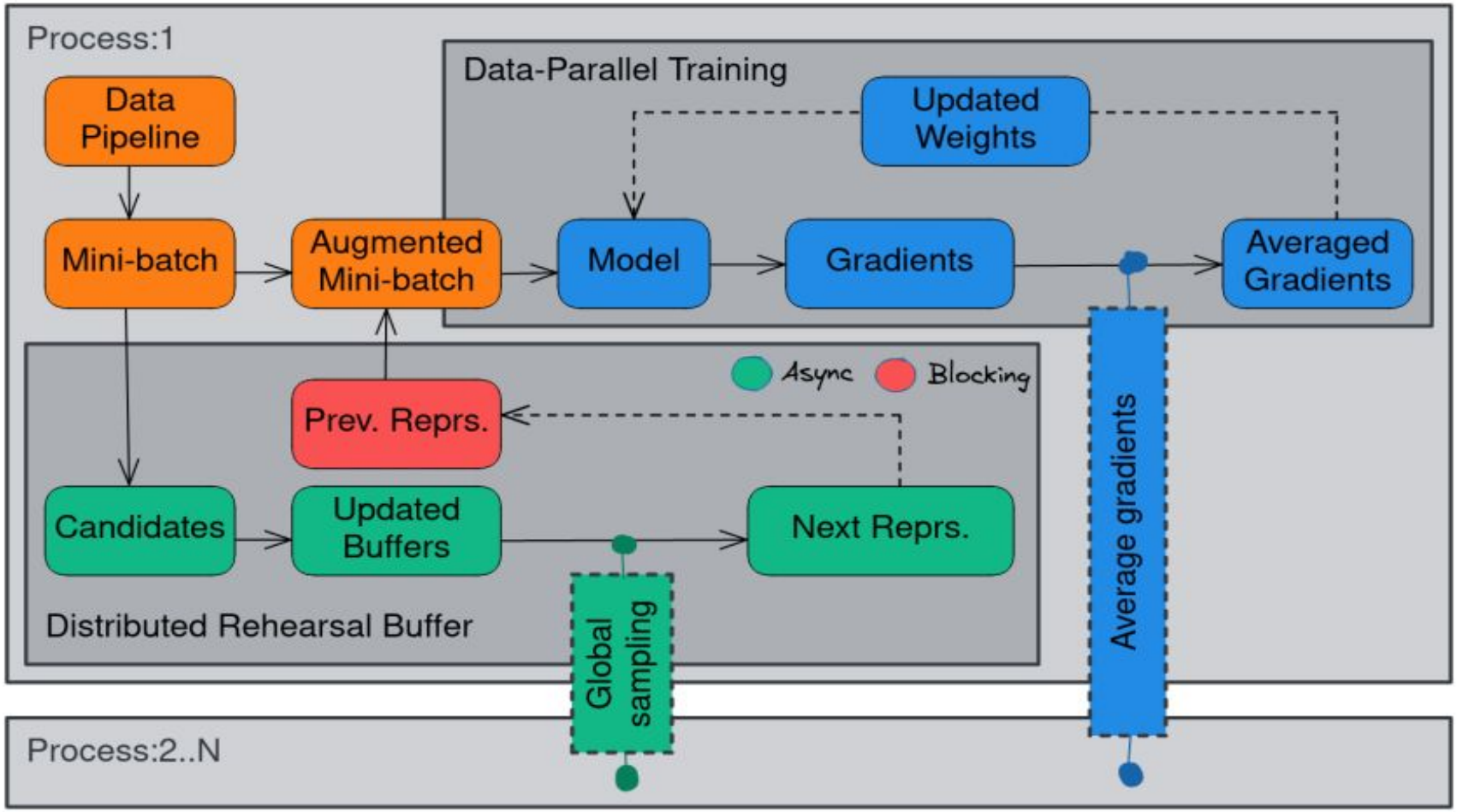
2. Store the selected samples in the ***rehearsal buffer***: we call such samples ***representatives***

3. ***Augment*** future minibatches with representatives stored in the rehearsal buffer

Retain information from past data increments



Distributed Data Pipelines with Rehearsal Support



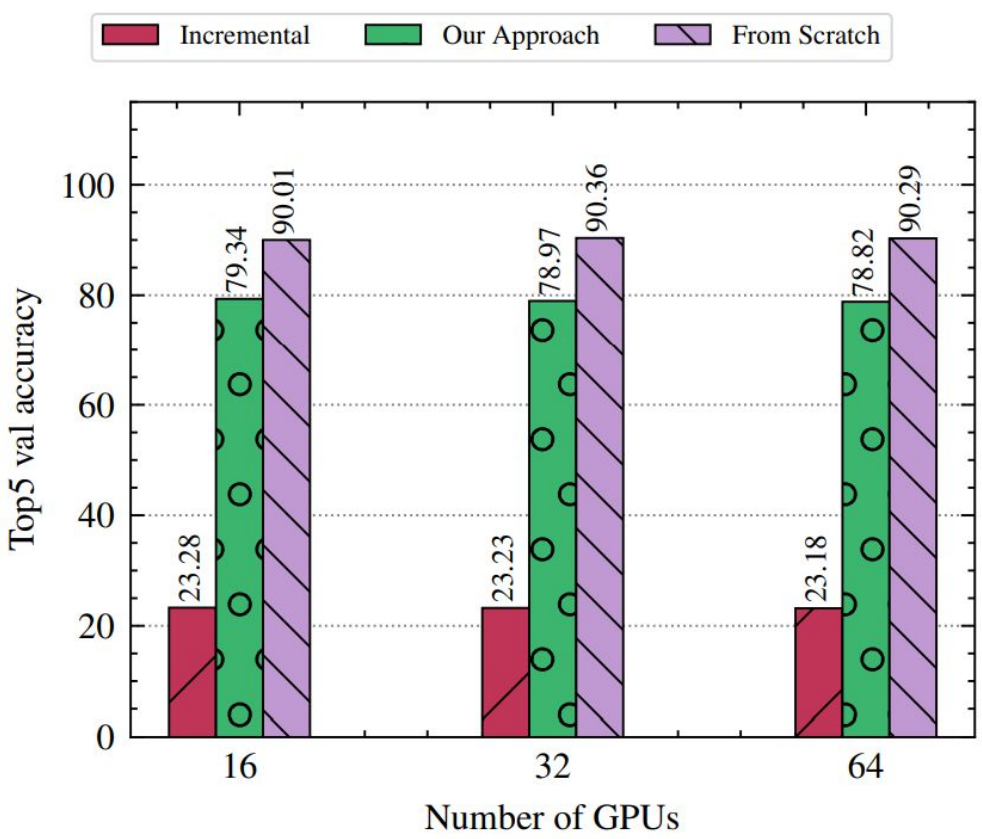
Key ideas:

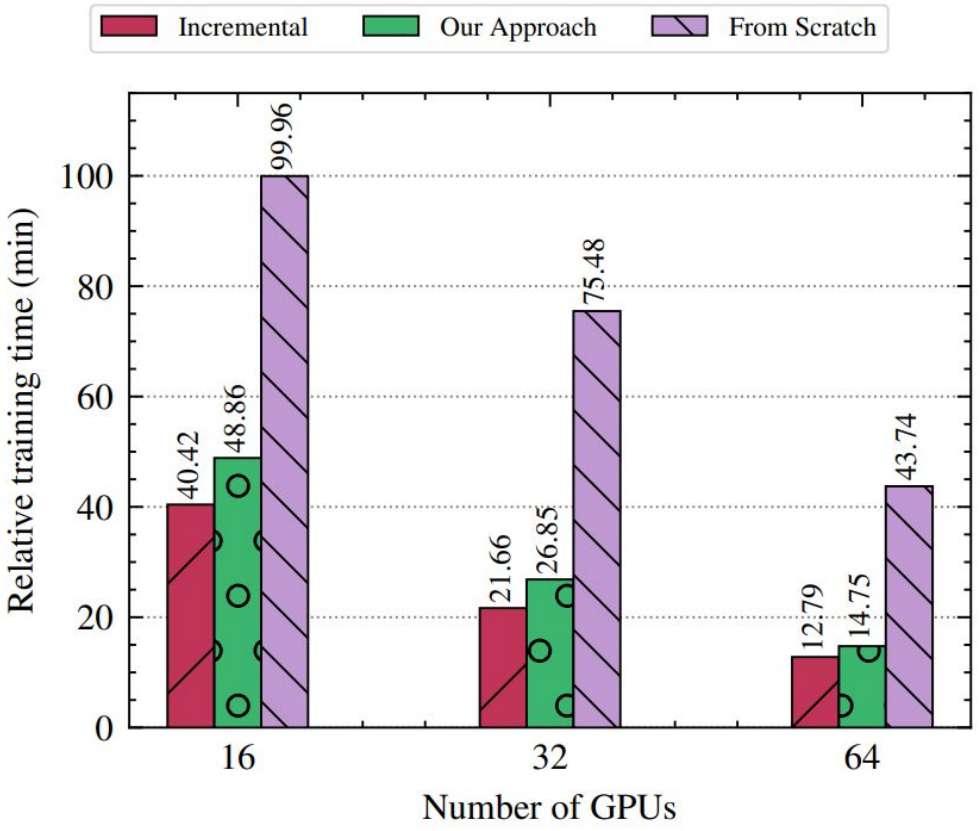
- Rehearsal for data-parallel training
- Extend data pipeline to transparently augment new mini-batches with representative training samples

Challenges at scale:

- How to build a distributed cache that efficiently indexes and revisits representative training samples
- Asynchronous augmentation that hides inter-node communication overheads
- High performance integration with Python

Data Pipelines with Rehearsal Support: Results





Experimental setup:

- ANL Polaris (A100 GPUs)
- Model: ResNet50, DataSet: ImageNet

T. Bouvier, B. Nicolae, A. Costan, I. Foster, G. Antoniu. Efficient Data-Parallel Continual Learning with Asynchronous Distributed Rehearsal Buffers. In CCGrid'24

Comparisons:

- Incremental training: fast and scalable but inaccurate (due to catastrophic forgetting)
- Training from scratch: best accuracy but slow and less scalable
- Our approach: fast, scalable and accurate

Acknowledgements

Co-authors and collaborators: *Robert Underwood, Franck Cappello, Justin Wozniak, Rob Ross, Matthieu Dorier, Tom Peterka, Orçun Yıldız, Ian Foster, Kyle Chard, Tekin Bicer (ANL); Kathryn Mohror, Greg Kosinovsky (LLNL), JiaLi Li, George Bosilca, Nigel Tan, Michela Taufer (UTK), Shu-Mei Tseng, Aparna Chandramowlishwaran (UCI), Mikaila Gossman, Jon Calhoun (Clemson), Hongyuan Liu, Await Jog (CWM), Avinash Maurya, Mustafa Rafique (RIT), Meghana Madhyastha, Randal Burns (JHU), Line Pouchard (BNL), Tanzima Islam (TSU), Jie Ye, Anthony Kougas, Xian-He Sun (IIT), T. Bouvier, A. Costan, G. Antoniu (INRIA), Kento Sato (RIKEN)*

Funding: *VELOC(ECP), DataStates (ECRP), BRAID (ASCR), Triple Convergence (ASCR), RECUP (ASCR)*

Conclusions

- The intersection of AI (Deep Learning) and Parallel/Distributed Systems covers a broad scope of actively researched problems
- We discussed only a small part of them (mostly related to data management aspects):
  - 3D parallelism: Data vs. pipeline vs. tensor parallelism
  - Data pipelines, asynchronous overlapping and adaptations for data parallelism
  - Checkpoint restart and adaptations for transformers
  - Lineage and versioning of DL models
  - Transfer learning, fine-tuning and network architecture search
  - Continual learning using rehearsal
- Many other exciting research directions, feel free to reach out!

Contact: bnicolae@anl.gov
Web: www.bnicolae.net