DADS7305: MLOPs

Northeastern University

Instructor: Ramin Mohammadi

September 7, 2025

These materials have been prepared and sourced for the course **MLOPs** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

`r.mohammadi@northeastern.edu`

Thank you for your understanding and collaboration.

High Performance Modeling

Distributed Training

Rise in Computational Requirements

- At first, training models is quick and easy
- Training becomes more time-consuming
  - With more data
  - With larger models
- Longer training $\rightarrow$ more epochs $\rightarrow$ less efficient
- Use distributed training approaches

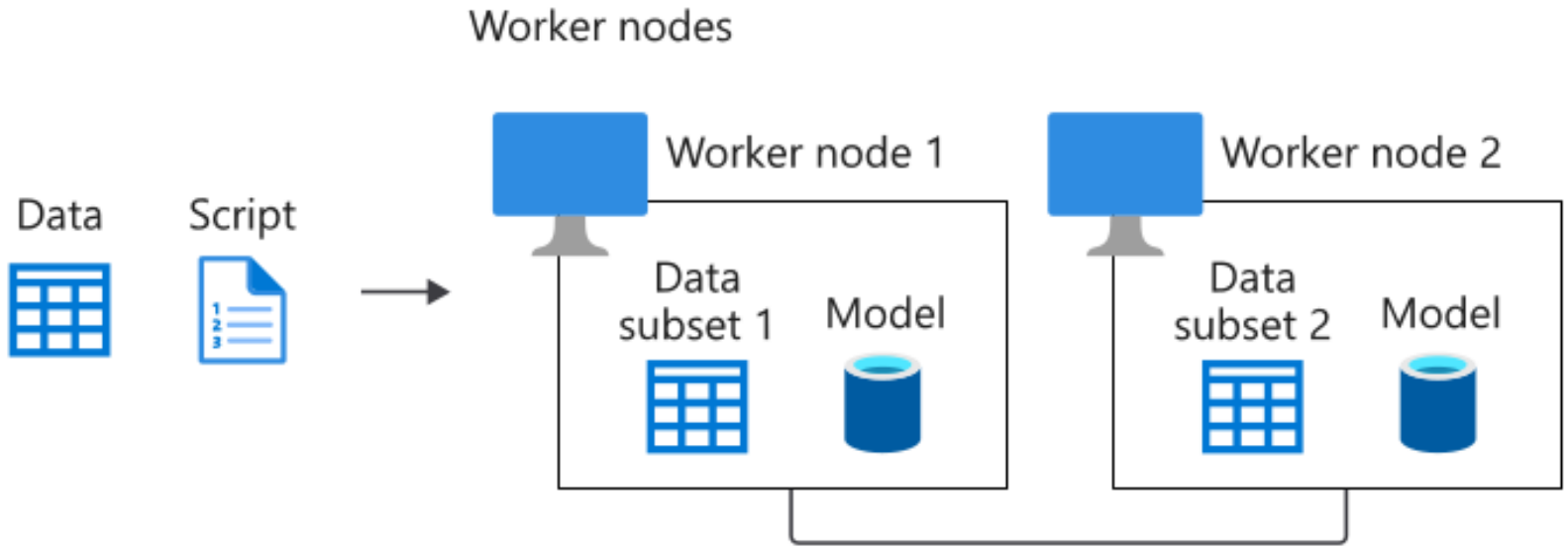
Types of distributed training

- Data parallelism: In data parallelism, models are replicated onto different accelerators (GPU/TPU) and data is split between them
- Model parallelism: When models are too large to fit on a single device then they can be divided into partitions, assigning different partitions to different accelerators

Distributed Training for LLMs

- **Data Parallelism:** Needed due to massive datasets (trillions of tokens).
- **Model Parallelism:** Required since LLMs (e.g., 70B–1T parameters) cannot fit in a single GPU memory.
- **Pipeline Parallelism:** Model layers split across devices; minibatches flow through in stages (like assembly line).
- **Tensor Parallelism:** Splits large matrix multiplications (e.g., attention layers) across multiple devices for efficiency.
- **Hybrid Parallelism:** Combination of data, model, pipeline, and tensor parallelism to scale training efficiently.

Data Parallelism



Distributed training using data parallelism

Synchronous training

- All workers train and complete updates in sync
- Supported via all-reduce architecture

Asynchronous Training

- Each worker trains and completes updates separately
- Supported via parameter server architecture
- More efficient, but can result in lower accuracy and slower convergence

Making your models distribute-aware

- If you want to distribute a model:
  - Supported in high-level APIs such as Keras/Estimators
  - For more control, you can use custom training loops

tf.distribute.Strategy

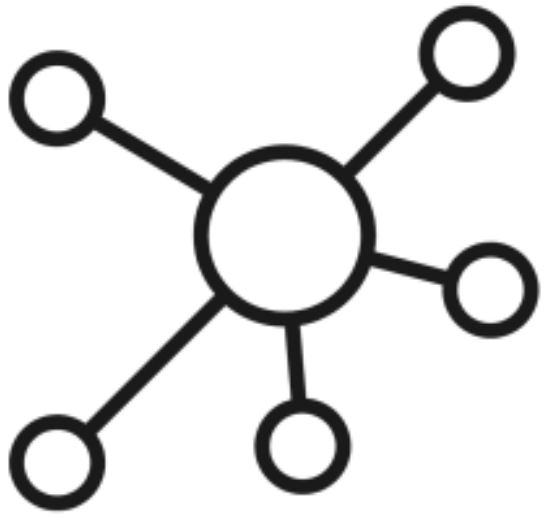
- Library in TensorFlow for running a computation in multiple devices
- Supports distribution strategies for high-level APIs like Keras and custom training loops
- Convenient to use with little or no code changes

Distributed Training in PyTorch

- `torch.distributed` is the core package for distributed training.
- `DistributedDataParallel (DDP)` is the most widely used API.
- Works by replicating models across devices and synchronizing gradients.
- Supports both data parallelism and model parallelism with custom setups.
- Flexible for custom training loops and large-scale LLM training.

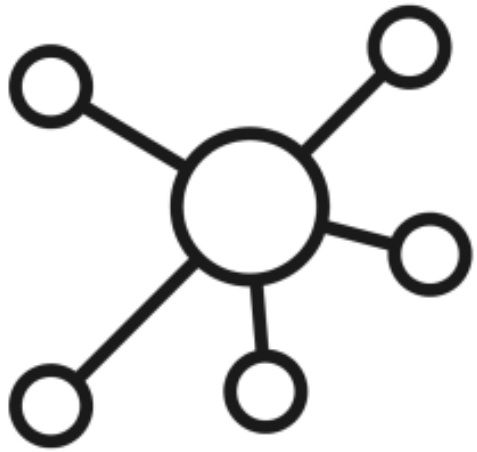
Distribution Strategies supported by tf.distribute.Strategy

- One Device Strategy
- Mirrored Strategy
- Parameter Server Strategy
- Multi-Worker Mirrored Strategy
- Central Storage Strategy
- TPU Strategy



Distribution Strategies supported by PyTorch

- **Single-Process Multiple-Devices (SPMD)**
- **FullyShardedDataParallel (FSDP)**
- **DistributedDataParallel (DDP)**
- **Tensor Parallelism (TP)**
- **Pipeline Parallelism**



One Device Strategy

- Single device – no distribution
- Typical usage of this strategy is testing your code before switching to other strategies that actually distribute your code

Mirrored Strategy

- This strategy is typically used for training on one machine with multiple GPUs
  - Creates a replica per GPU $\leftrightarrow$ Variables are mirrored
  - Weight updating is done using efficient cross-device communication algorithms (all-reduce algorithms)

Parameter Server Strategy

- Some machines are designated as workers and others as parameter servers
  - Parameter servers store variables so that workers can perform computations on them
- Implements asynchronous data parallelism by default

Fault Tolerance

- Catastrophic failures in one worker would cause failure of distribution strategies.
- How to enable fault tolerance in case a worker dies?
  - By restoring training state upon restart from job failure
  - Keras implementation: `BackupAndRestore` callback

Single-Process Multiple-Devices (SPMD)

SPMD takes a single-device program, shards it, and executes it in parallel.

- Run on one GPU/CPU.
- No distribution involved.
- Useful for debugging and small-scale experiments.

Fully Sharded Data Parallel (FSDP)

FSDP is designed for models that are too large to fit on a single GPU. It shards the model's parameters, gradients, and optimizer states across multiple devices. This reduces the memory footprint on individual GPUs, allowing for the training of much larger models. FSDP can also be combined with other parallelism strategies.

- Shards parameters, gradients, and optimizer states across GPUs instead of replicating them.
- Reduces per-GPU memory usage, enabling training of very large models.
- Can overlap communication and computation to improve efficiency.
- Often combined with data parallelism and pipeline/tensor parallelism in LLM training.

DistributedDataParallel (DDP)

This is a widely used strategy for data parallelism when the model fits within a single GPU's memory. DDP replicates the model on each process (typically one process per GPU) and distributes different mini-batches of data to each replica. Gradients are synchronized and averaged across all processes during the backward pass.

- Most common strategy in PyTorch.
- Efficient gradient synchronization across multiple GPUs/nodes.
- Reduces communication overhead with optimized all-reduce algorithms.
- Standard choice for scaling deep learning models.

Tensor Parallelism (TP)

Tensor Parallelism involves sharding individual tensors (such as large weight matrices) across multiple devices. This is particularly useful for very large layers (e.g., attention or feed-forward layers) that cannot fit on a single device.

- Splits large tensor operations (e.g., matrix multiplications in attention/MLPs) across multiple GPUs.
- Reduces per-GPU memory usage for extremely large layers.
- Requires communication between devices to exchange partial results of tensor operations.
- Often combined with pipeline parallelism and data parallelism for LLM training.

Pipeline Parallelism (PP)

Pipeline Parallelism involves splitting a model's layers into sequential stages, where each stage is placed on a different device. Data is passed through these stages in a pipeline, enabling parallel execution across devices.

- Divide the model into ordered stages across GPUs.
- Mini-batches are split into micro-batches that flow through the pipeline in assembly-line fashion.
- Overlaps computation across devices to improve utilization.
- Enables training of much larger models by distributing layers across accelerators.

Hybrid Parallelism

- Combines multiple strategies: DDP + Model/ Pipeline Parallelism.
- Critical for training Large Language Models (LLMs).
- Balances memory usage, communication, and compute efficiency.

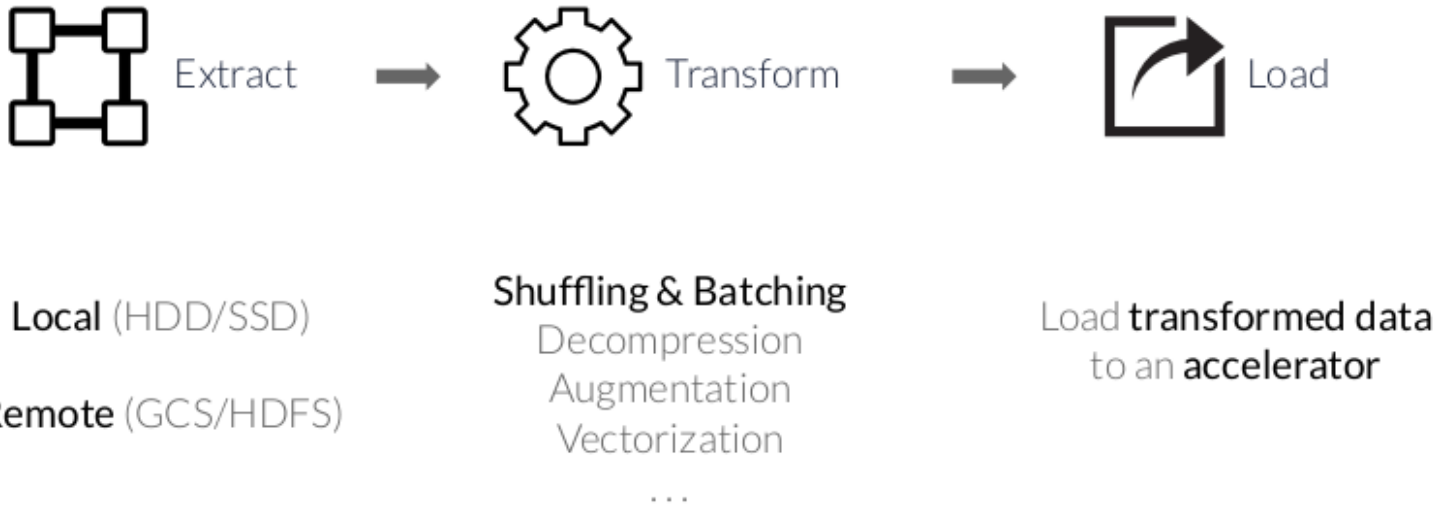
High Performance Modeling

High-performance Ingestion

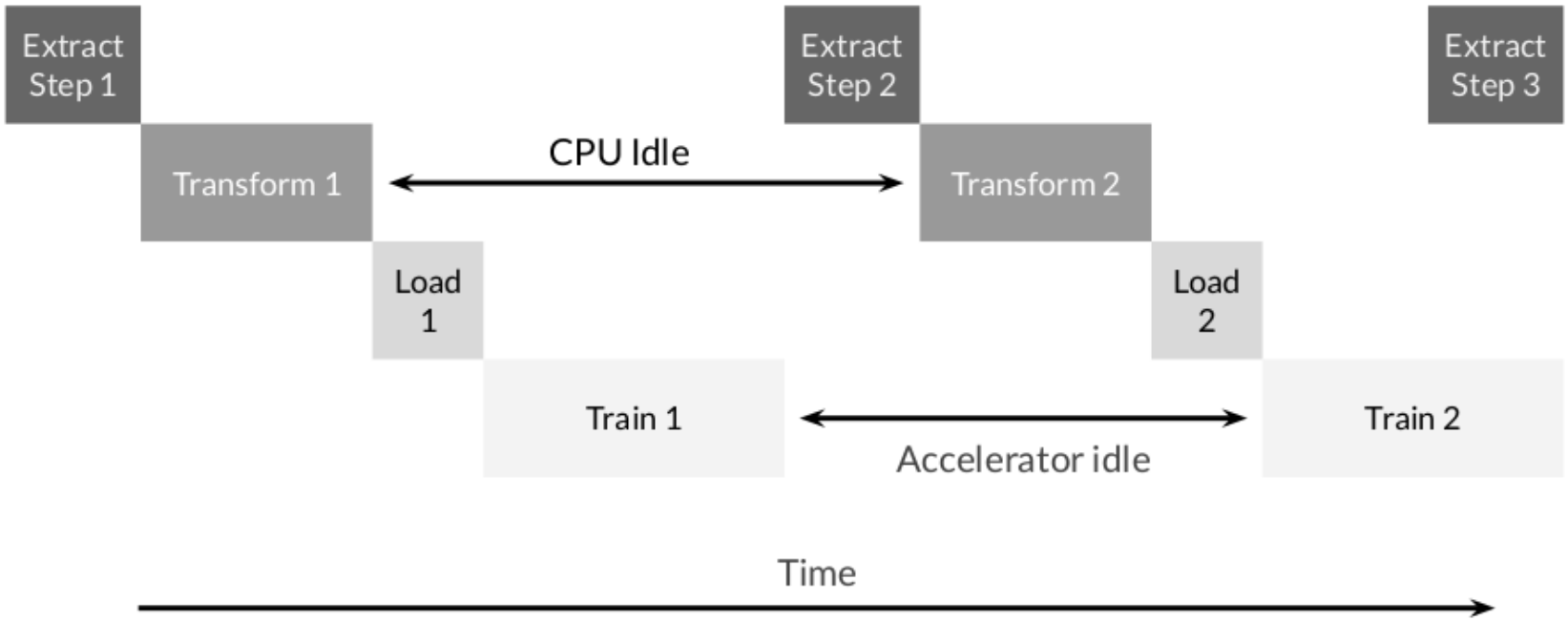
Why Input Pipelines?

- Data at times can't fit into memory.
- CPUs are under-utilized in compute intensive tasks like training a complex model.
- You should avoid these inefficiencies so that you can make the most of the hardware available.
- $\Rightarrow$ Use input pipelines

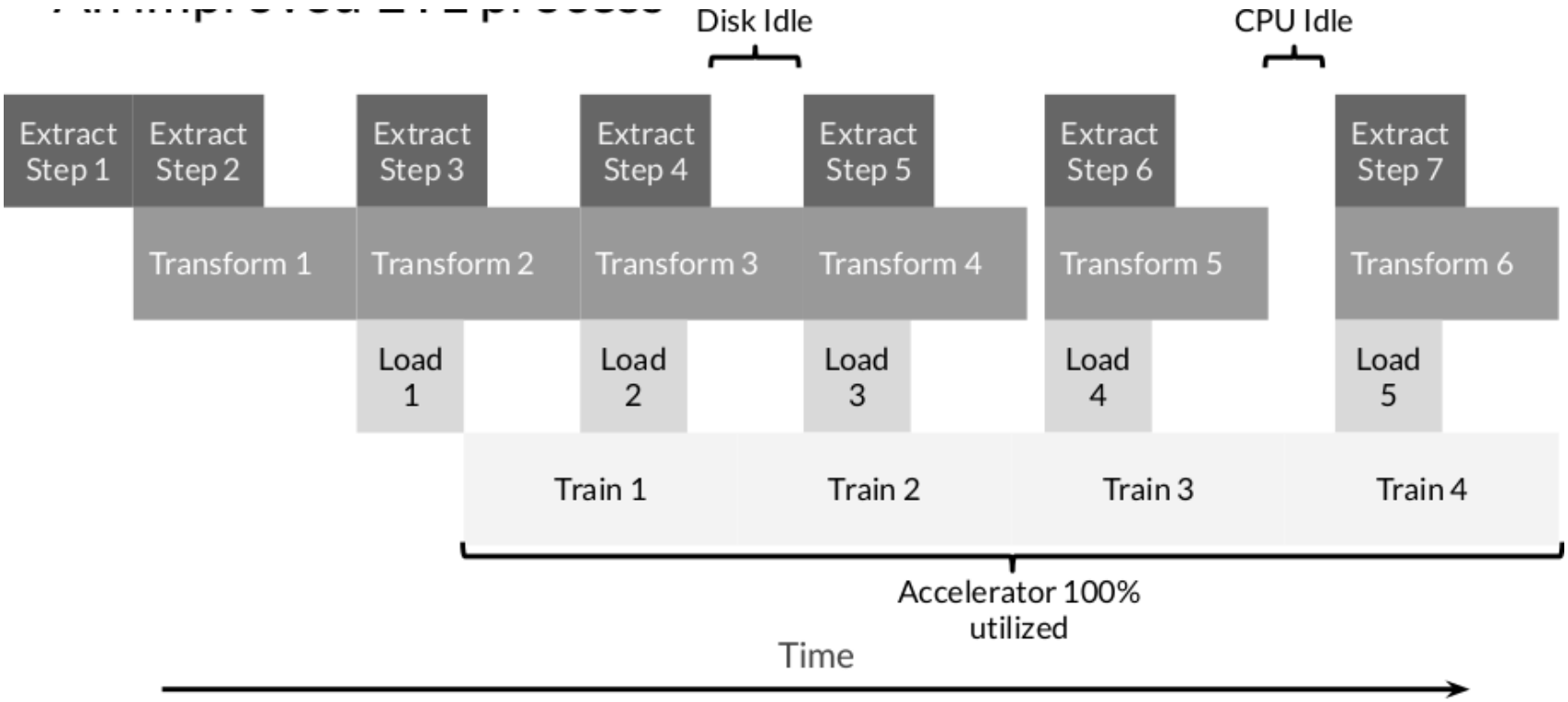
tf.data / pt.data: Input Pipeline



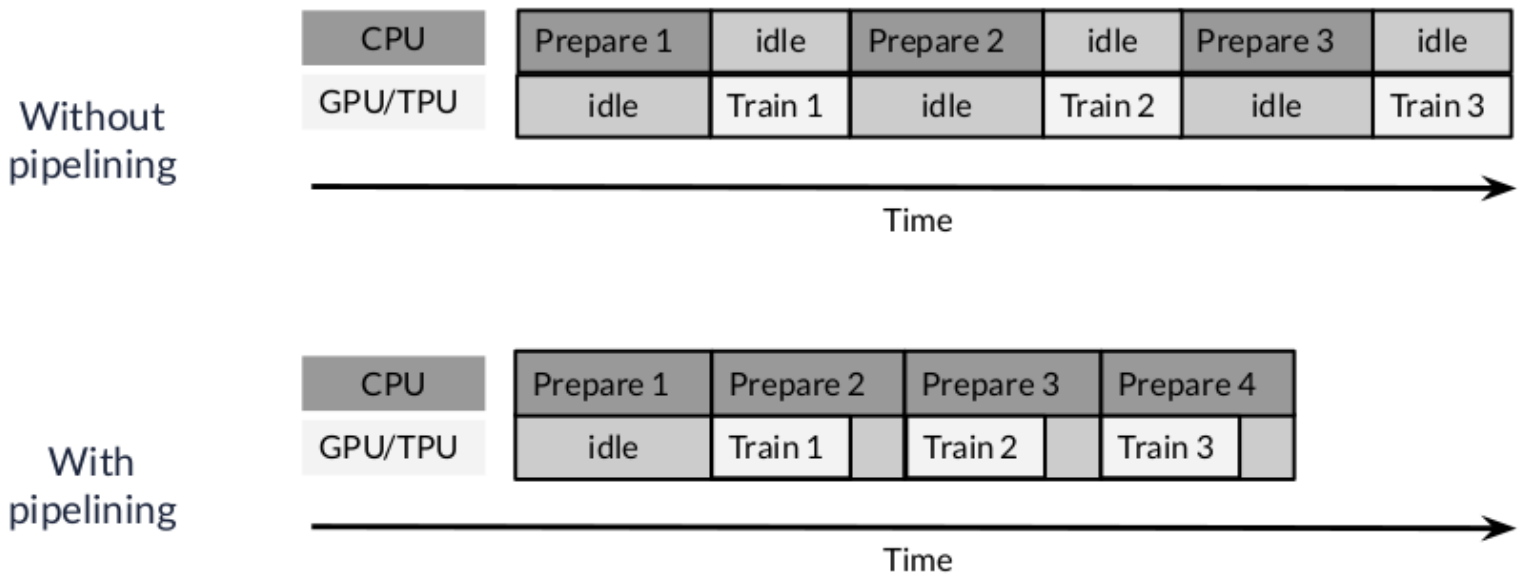
Inefficient ETL process



Inefficient ETL process



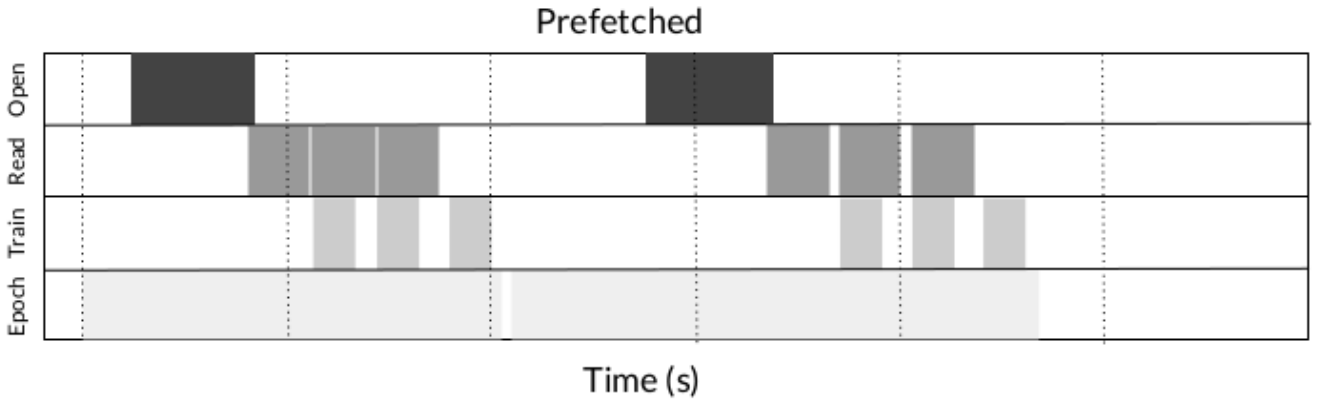
Pipelining


Without pipelining
CPU
GPU/TPU
Prepare 1
idle
Prepare 2
idle
Prepare 3
idle
idle
Train 1
idle
Train 2
idle
Train 3
Time
With pipelining
CPU
GPU/TPU
Prepare 1
Prepare 2
Prepare 3
Prepare 4
idle
Train 1
Train 2
Train 3
Time

How to Optimize Pipeline Performance?

- **Prefetching**: Prepare the next batch of data while the current one is being processed on the accelerator.
- **Parallelize Data Extraction and Transformation**: Use multiple CPU workers/threads to load and preprocess data in parallel.
- **Caching**: Cache datasets or intermediate transformations to avoid recomputation and repeated I/O.
- **Reduce Memory**: Optimize memory usage by streaming data, using smaller data formats, or avoiding unnecessary copies.

Optimize with prefetching

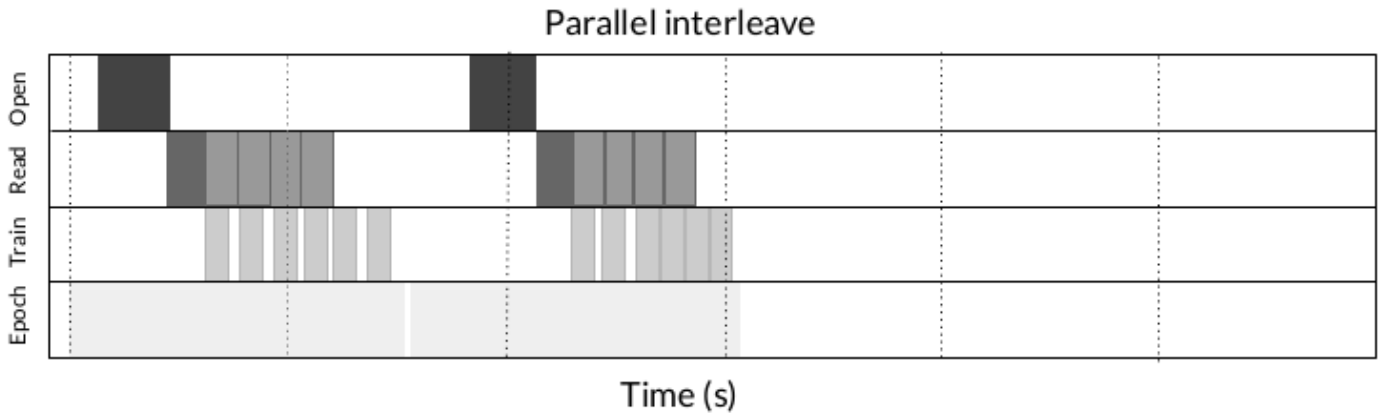



```
benchmark(
    ArtificialDataset()
    .prefetch(tf.data.experimental.AUTOTUNE)
)
```

Parallelize Data Extraction

- **Time-to-first-byte**: Prefer local storage as it takes significantly longer to read data from remote storage.
- **Read throughput**: Maximize the aggregate bandwidth of remote storage by reading more files in parallel.

Parallel interleave




```
benchmark(
    tf.data.Dataset.range(2)
    .interleave(
        ArtificialDataset,
        num_parallel_calls=tf.data.experimental.AUTOTUNE
    )
)
```

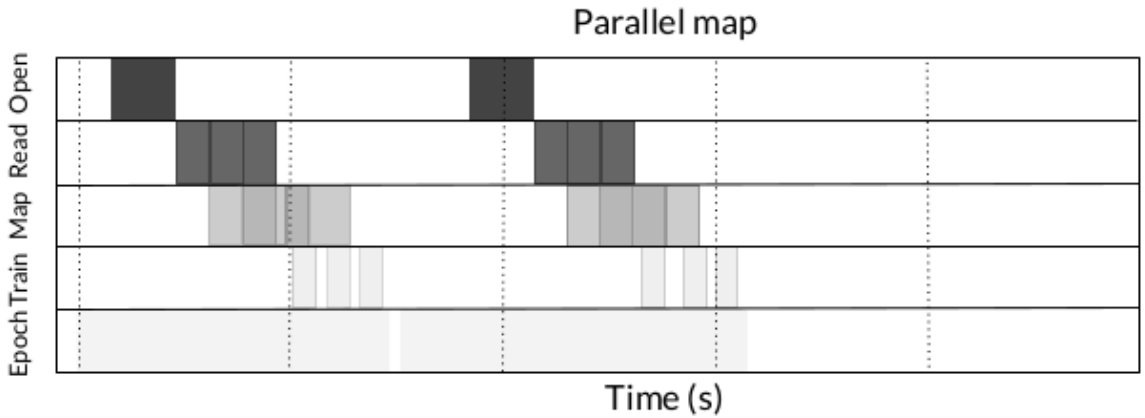
Parallelize Data Transformation

- Post data loading, the inputs may need preprocessing.
- Element-wise preprocessing can be parallelized across CPU cores.
- The optimal value for the level of parallelism depends on:
  - Size and shape of training data
  - Cost of the mapping transformation
  - Current CPU load
- With `tf.data`, you can use `AUTOTUNE` to set parallelism automatically.

Parallelize Data Transformation in PyTorch

- Preprocessing typically defined in `Dataset __getitem__()`.
- Libraries like `torchvision.transforms` and `torchaudio.transforms` help with image/audio preprocessing.
- For NLP/LLMs, Hugging Face `datasets` supports streaming and transformations.
- Use `DataLoader` to parallelize:
  - `num_workers`: number of CPU processes for preprocessing.
  - `prefetch_factor`: controls how many batches each worker preloads.
  - `pin_memory=True`: enables faster CPU $\rightarrow$ GPU transfer.
- Optimal values depend on data size, CPU load, and transformation cost.

Parallel mapping




```
benchmark(
    ArtificialDataset()
    .map(
        mapped_function,
        num_parallel_calls=tf.data.AUTOTUNE
    )
)
```

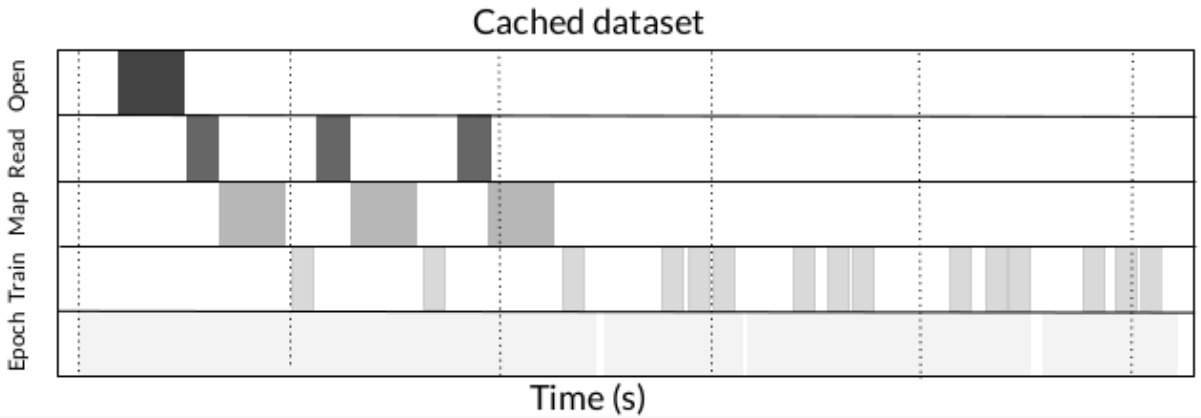
Improve Training Time with Caching

- **In-memory:** `tf.data.Dataset.cache()`
- **Disk:** `tf.data.Dataset.cache(filename=...)`

Improve Training Time with Caching in PyTorch

- PyTorch does not have a direct `.cache()` API like TensorFlow.
- Common caching strategies:
  - **In-memory:** Preload dataset into RAM (e.g., wrap in a custom `Dataset` that loads once and reuses).
  - **Disk:** Preprocess and save data as tensors (e.g., `.pt` files) or use formats like LMDB / WebDataset.
  - **TorchData / HuggingFace Datasets:** Provide built-in caching to disk and memory.
- Combine caching with `DataLoader` optimizations (`num_workers`, `pin_memory`) for maximum performance.

Caching



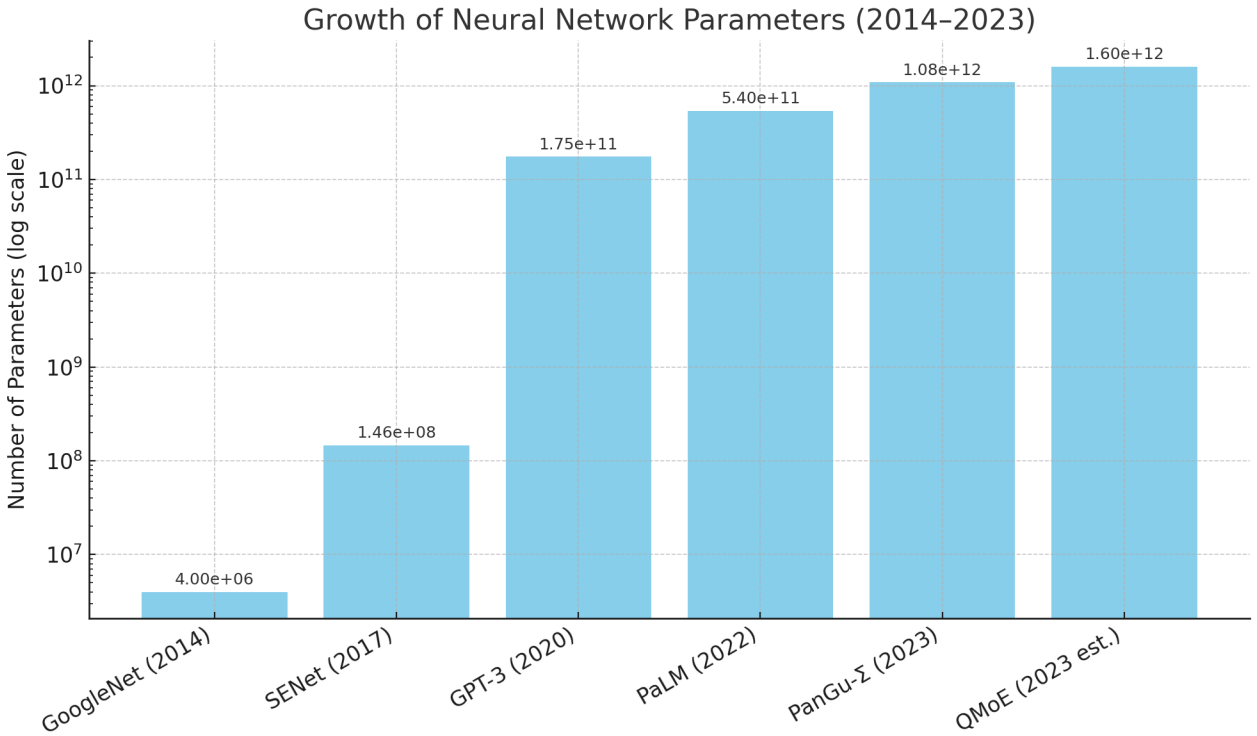

```
benchmark(
    ArtificialDataset().map(mapped_function).cache(),5
)
```

High Performance Modeling

Training Large Models - The Rise of Giant Neural Nets and Parallelism

Rise of Giant Neural Networks

- 2014: GoogleNet wins ImageNet with $\sim$ million parameters, achieving 74.8% top-1 accuracy.
- 2017: Squeeze-and-Excitation Networks reach 82.7% top-1 accuracy using $\sim$ million parameters-a 36$\times$ increase in just 3 years.
- Present Day (2025):
  - GPT-3 (2020): $\sim$ billion parameters.
  - PaLM (2022–2023): up to $\sim$ billion parameters.
  - Huawei PanGu$-\Sigma$ (2023): $\sim$ .085 trillion parameters.
  - QMoE: Estimated at $\sim$1.6 trillion parameters.


Growth of Neural Network Parameters (2014–2023)
Number of Parameters (log scale)
10^{12}
10^{11}
10^{10}
10^{9}
10^{8}
10^{7}
4.00e+06
1.46e+08
1.75e+11
5.40e+11
1.08e+12
1.60e+12
GoogleNet (2014)
SENet (2017)
GPT-3 (2020)
PaLM (2022)
PanGu-Σ (2023)
QMoE (2023 est.)

Issues Training Larger Networks

- GPU memory only increased by a factor of ~ 3.
- Saturated the amount of memory available in Cloud TPUs.
- Need for large-scale training of giant neural networks.

Overcoming Memory Constraints

Strategy #1 - Gradient Accumulation

- Split batches into mini-batches.
- Perform forward and backward passes on each mini-batch.
- Accumulate gradients and only update weights after the full batch.

Strategy #2 - Memory Swap

- Copy activations between CPU and GPU memory.
- Swap tensors back and forth during training to fit larger models.

Activation Checkpointing (Gradient Checkpointing)

- Saves memory by not storing all intermediate activations.
- During backpropagation, activations are recomputed instead of retrieved from memory.
- Reduces memory footprint at the cost of additional compute.
- Widely used in training LLMs like GPT-3 and PaLM.

Mixed Precision Training (FP16 / BF16)

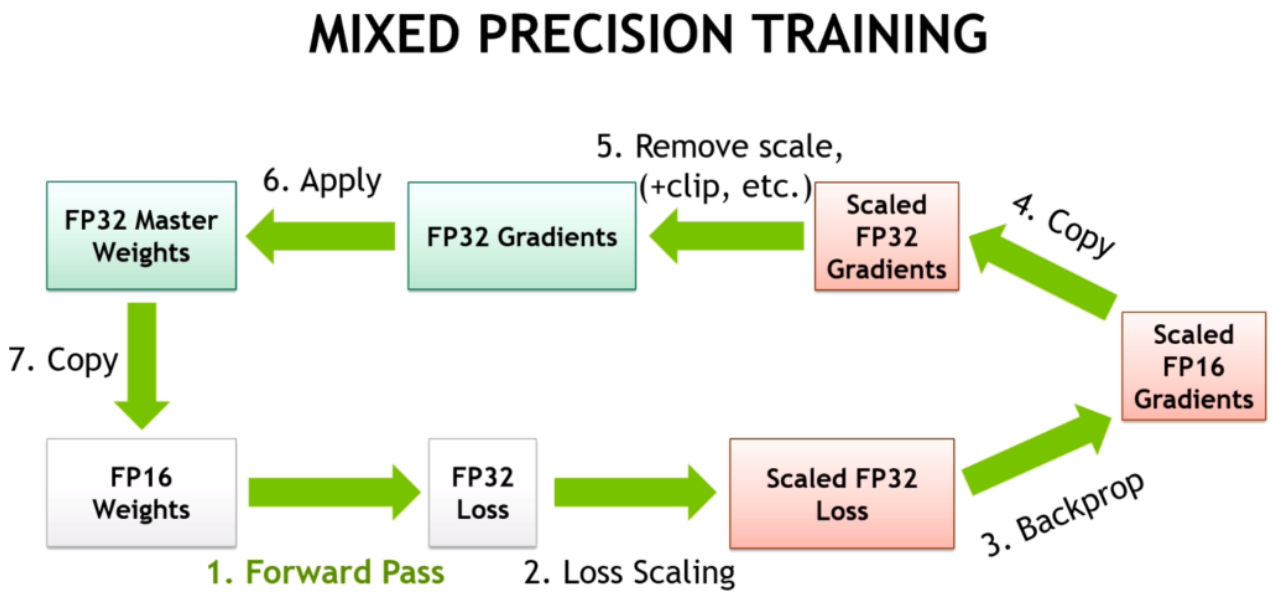


Figure: Video

- Use lower-precision formats (FP16 or BF16) instead of FP32.
- Reduces memory usage and speeds up training on GPUs/TPUs.
- FP16 improves efficiency but may require loss scaling for stability.
- BF16 is preferred for numerical stability in large-scale LLM training.

ZeRO: Zero Redundancy Optimizer (DeepSpeed)

Shards optimizer states, gradients, and parameters across GPUs.

- **ZeRO-1**: Shard optimizer states.
- **ZeRO-2**: Shard optimizer states + gradients.
- **ZeRO-3**: Fully shard parameters as well.
- Enables training of trillion-parameter models efficiently.

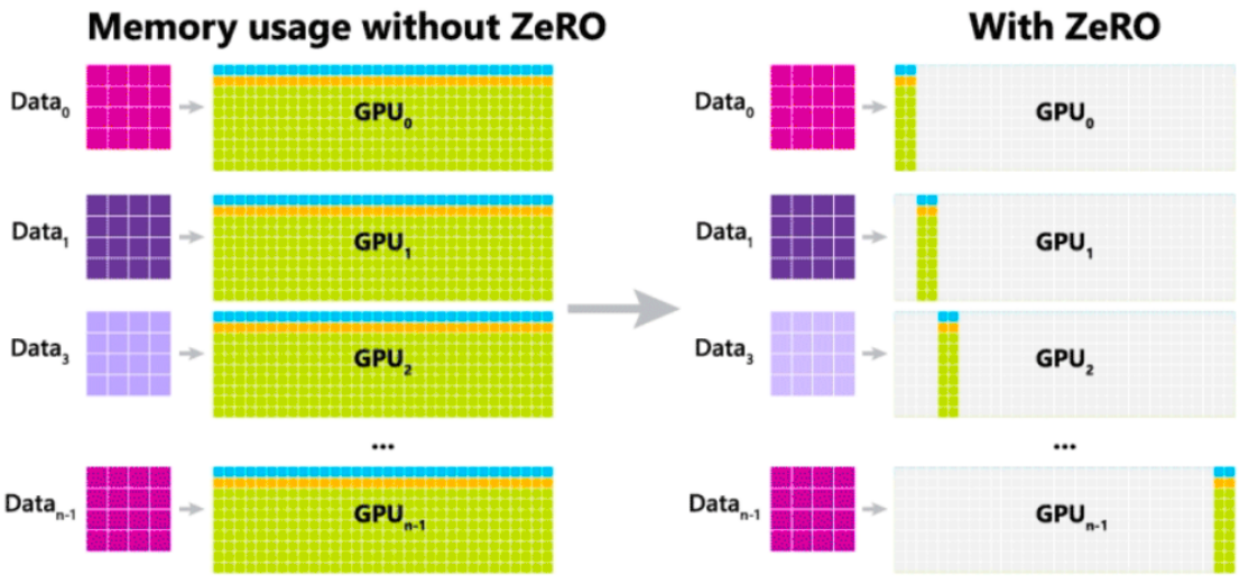


Figure: video

Offloading (CPU / NVMe)

NVMe offloading uses high-speed NVMe SSDs to extend system memory for demanding workloads like training very large AI models, storing less frequently accessed data to free up RAM and VRAM

- Move optimizer states, gradients, or parameters to CPU or NVMe storage.
- Useful when GPU memory is insufficient.
- DeepSpeed and PyTorch FSDP support CPU/NVMe offloading.
- Enables training beyond GPU memory limitations.

Tensor & Pipeline Parallelism

- **Tensor Parallelism**: Split large matrix multiplications across GPUs.
- **Pipeline Parallelism**: Divide model layers into stages across GPUs.
- Both strategies reduce per-GPU memory footprint.
- Essential for scaling to very large LLMs.

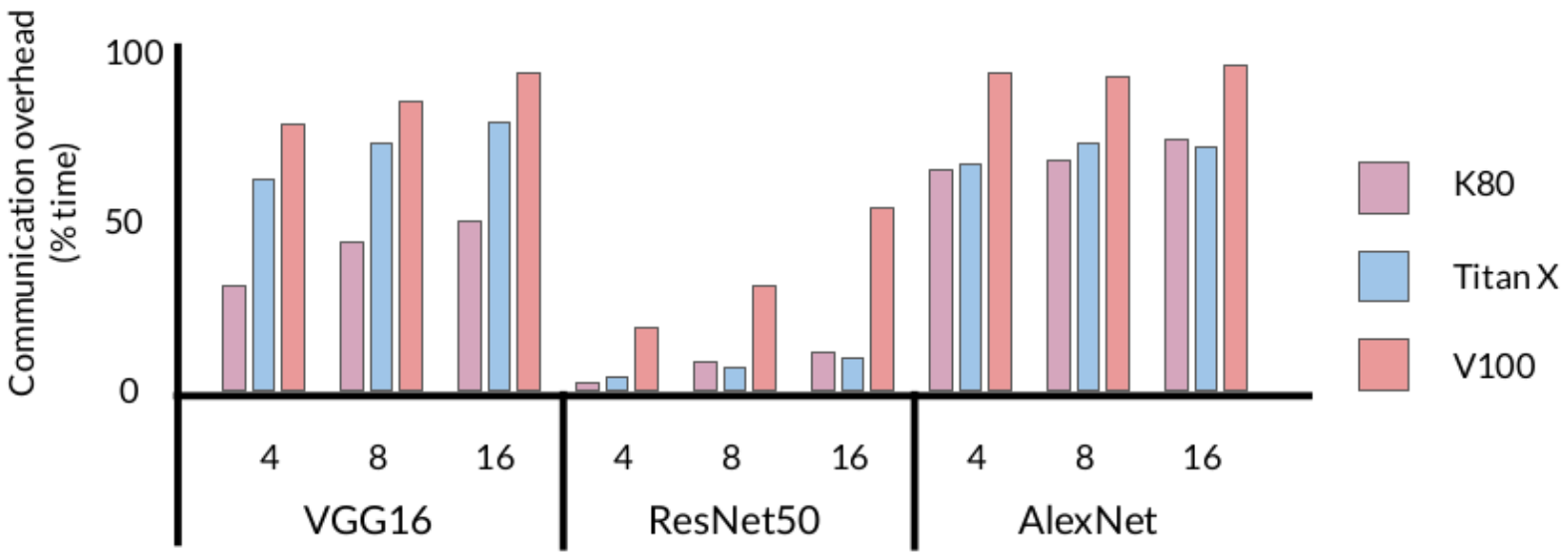
Quantization for Memory Efficiency

- Use reduced precision representations (8-bit, 4-bit) for weights and activations.
- Significantly reduces memory consumption.
- Accelerates inference and sometimes training.
- Post-training quantization and quantization-aware training are both used.

Parallelism Revisited

- **Data Parallelism:** In data parallelism, models are replicated onto different accelerators (GPU/TPU) and data is split between them.
- **Model Parallelism:** When models are too large to fit on a single device, they can be divided into partitions, assigning different partitions to different accelerators.

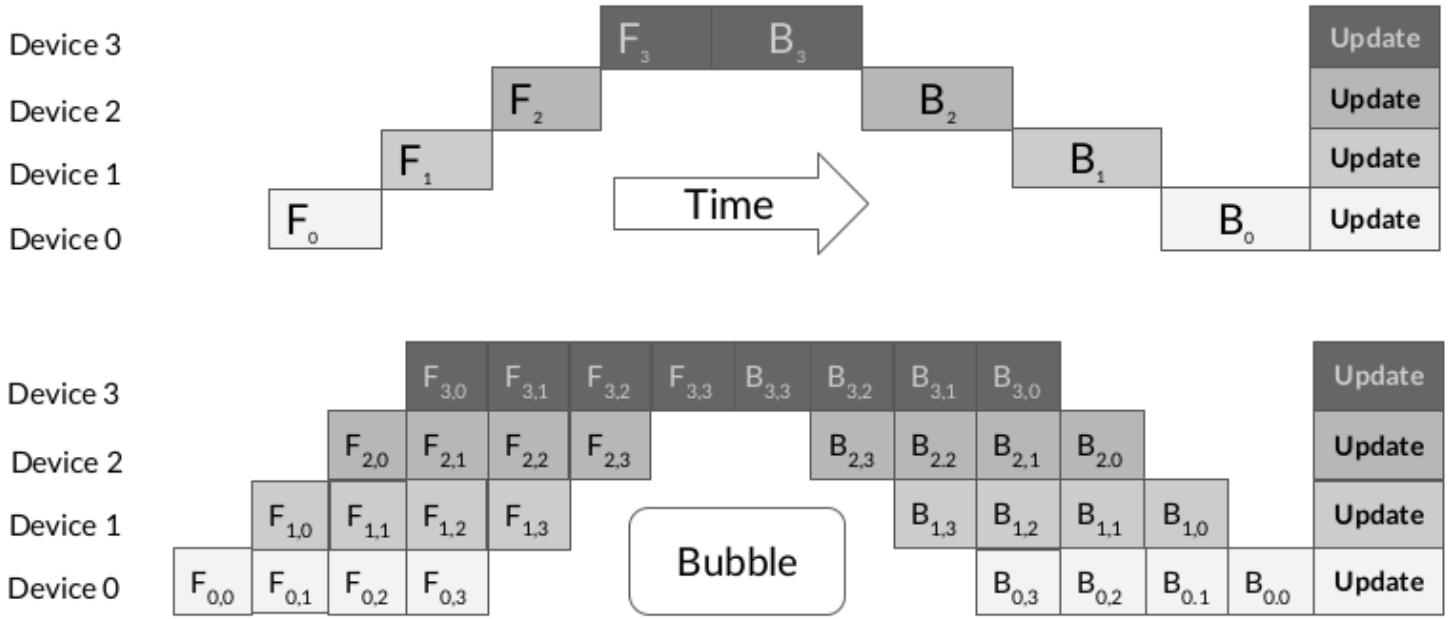
Challenges in data parallelism



Challenges Keeping Accelerators Busy

- Accelerators have limited memory.
- **Model Parallelism:** Large networks can be trained, but accelerator compute capacity is often underutilized.
- **Data Parallelism:** Same model is trained with different input data, but the maximum model size an accelerator can support is limited.

Pipeline parallelism



Pipeline Parallelism

- Integrates both data and model parallelism.
- **Divide mini-batch data into micro-batches.**
- Different workers process different micro-batches in parallel.
- Allows ML models to scale to significantly more parameters.

GPipe - Key Features

- Open-source TensorFlow library built on Lingvo framework.
- Inserts communication primitives at model partition boundaries.
- Implements pipeline parallelism with automatic splitting of model layers.
- Uses gradient accumulation across micro-batches to preserve model quality.
- Applies heuristic-based partitioning to balance compute across accelerators.

DeepSpeed - Pipeline Parallelism (Industry Standard)

- Open-source library by Microsoft Research, built for PyTorch.
- Supports 3D parallelism: hybrid of data, model, and pipeline parallelism.
- Efficient layer partitioning across multiple GPUs (pipeline parallelism).
- Integrates with ZeRO optimizer (ZeRO-1/2/3) for memory-efficient training of trillion-parameter models.
- Enables training extremely large models using CPU/NVMe offloading, mixed precision, and advanced parallelism.

torchgpipe - PyTorch GPipe Implementation

- PyTorch implementation of GPipe.
- Enables pipeline parallelism by splitting models into sequential stages.
- Uses micro-batching to improve accelerator utilization.
- Automatic backpropagation through pipeline partitions.
- Compatible with existing PyTorch models with minimal code changes.

Comparison of Pipeline Parallelism Libraries

Library	Framework	Pipeline Parallelism	Key Strengths
GPipe	TensorFlow	Yes	Easy partitioning, micro-batching, and gradient accumulation; heuristic-based partitioning.
DeepSpeed	PyTorch	Yes (with 3D parallelism)	Combines data/model/pipeline parallelism, integrates ZeRO optimizer, supports CPU/NVMe offloading, enables trillion-parameter training.
torchgpipe	PyTorch	Yes	PyTorch-native GPipe implementation, supports micro-batching and automatic backpropagation through partitions, minimal code changes.

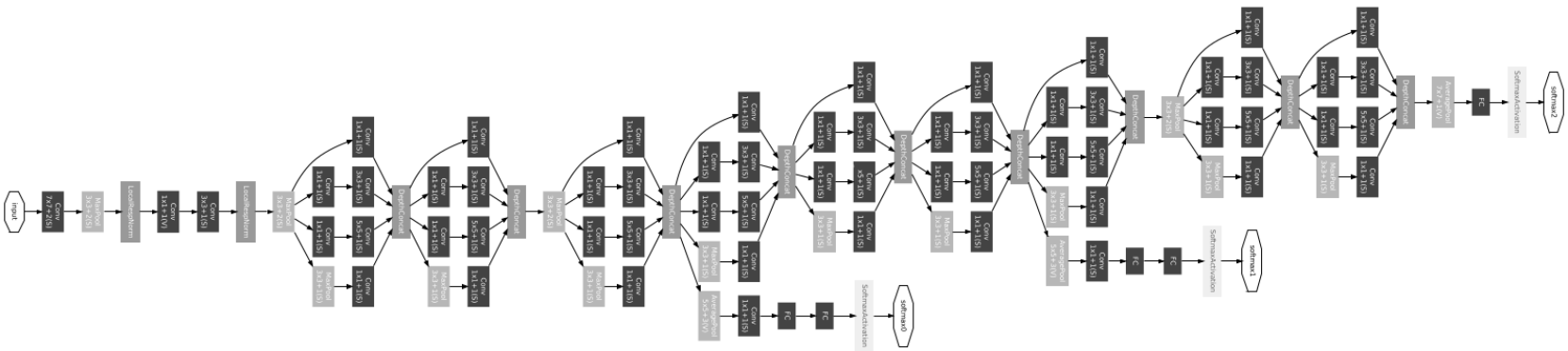
Knowledge Distillation

Teacher and Student Networks

Sophisticated Models and Their Problems

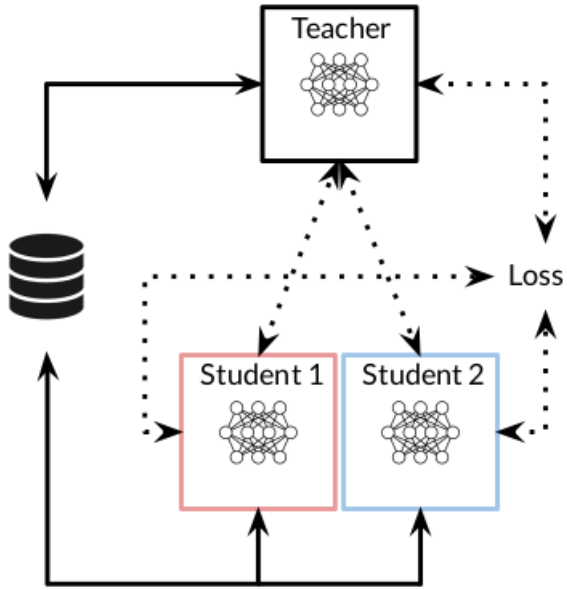
- Larger sophisticated models become increasingly complex.
- Complex models are capable of learning complex tasks.
- Question: Can we express this learning more efficiently?
- Is it possible to **distill** or concentrate this complexity into smaller networks?

GoogleNet



Knowledge Distillation

- Duplicate the performance of a complex model in a simpler model.
- Idea: Create a simple *student* model that learns from a complex *teacher* model.



Knowledge Distillation

Knowledge Distillation Techniques

Teacher and Student

- Training objectives of the models vary.
- **Teacher (normal training)**
  - Maximizes the actual metric.
- **Student (knowledge transfer)**
  - Matches the probability distribution (p-distribution) of the teacher's predictions to form *soft targets*.
  - *Soft targets* tell us about the knowledge learned by the teacher.


Dark Knowledge

Transferring "dark knowledge" to the student

- Improve softness of the teacher's distribution with *softmax temperature* (T).
- As T grows, you get more insight about which classes the teacher finds similar to the predicted one.

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

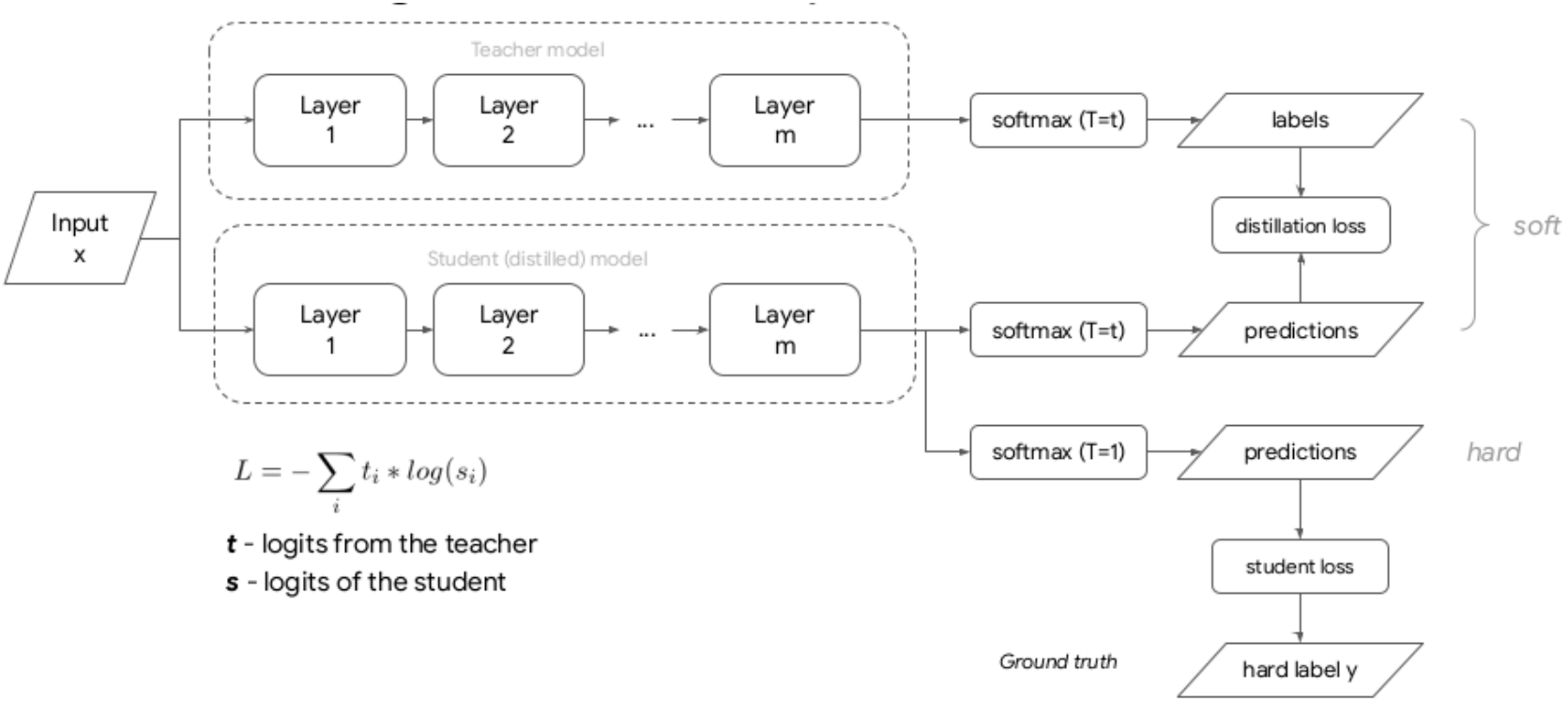
Techniques

- **Approach #1:** Weigh objectives (student and teacher) and combine during backprop.
- **Approach #2:** Compare distributions of the predictions (student and teacher) using KL divergence.

KL Divergence

$$L = (1 - \alpha)L_H + \alpha L_{KL}$$

How knowledge transfer takes place



First quantitative results of distillation

Model	Accuracy	Word Error Rate (WER)
Baseline	58.9%	10.9%
10x Ensemble	61.1%	10.7%
Distilled Single Model	60.8%	10.7%

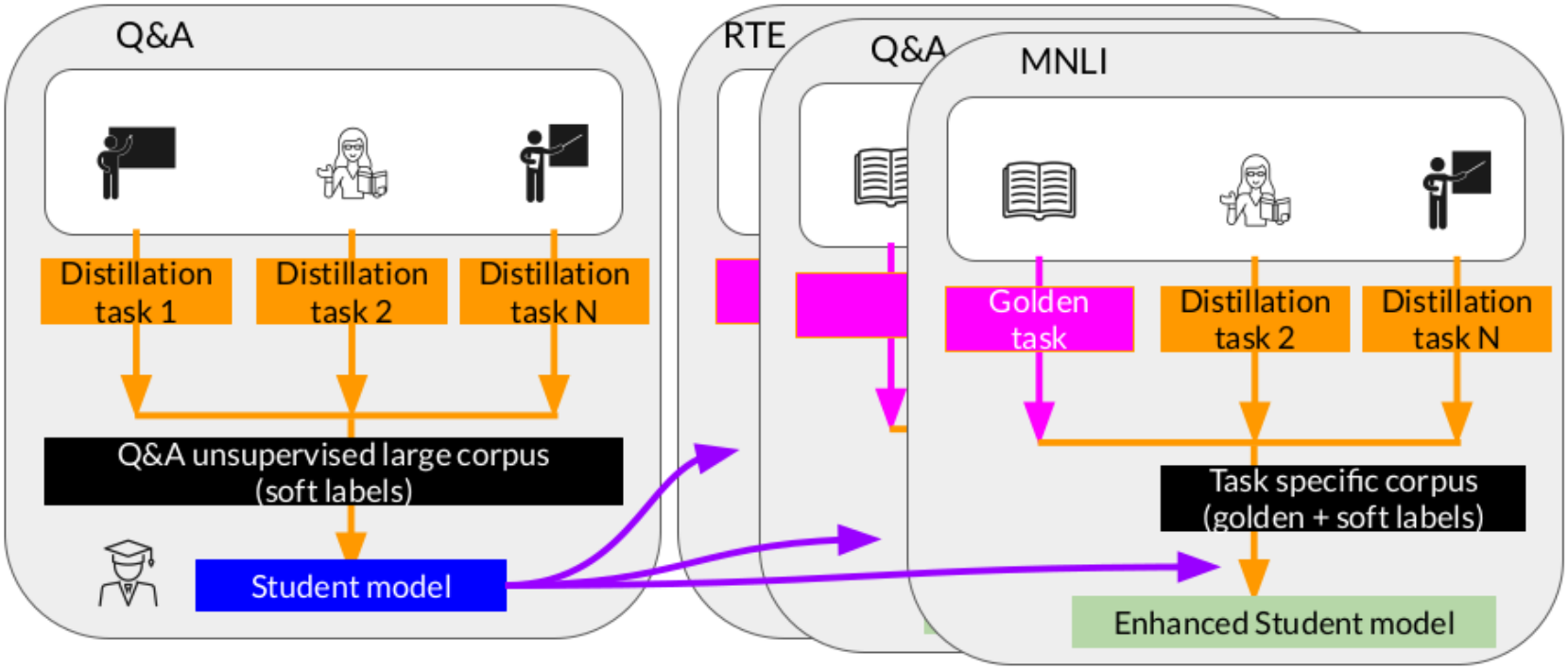
DistilBERT


20000
15000
10000
5000
0
Numbers of Parameters (in Millions)
Ai2
ELMo
94
OpenAI
GPT
110
BERT
340
Ai2
Transformer
ELMo
465
OpenAI
GPT-2
1500
MT-DNN
330
XLM
665
340
XLNET
Carnegie Mellon University
UNIVERSITY of WASHINGTON
Grover
1500
RoBERTa
355
NVIDIA
MegatronLM
8300
DistilBERT
66
11000
T5
BART
400
XLM-R
550
17000
Turing-NLG
2600
Meena
July 2018
January 2019
July 2019
January 2020

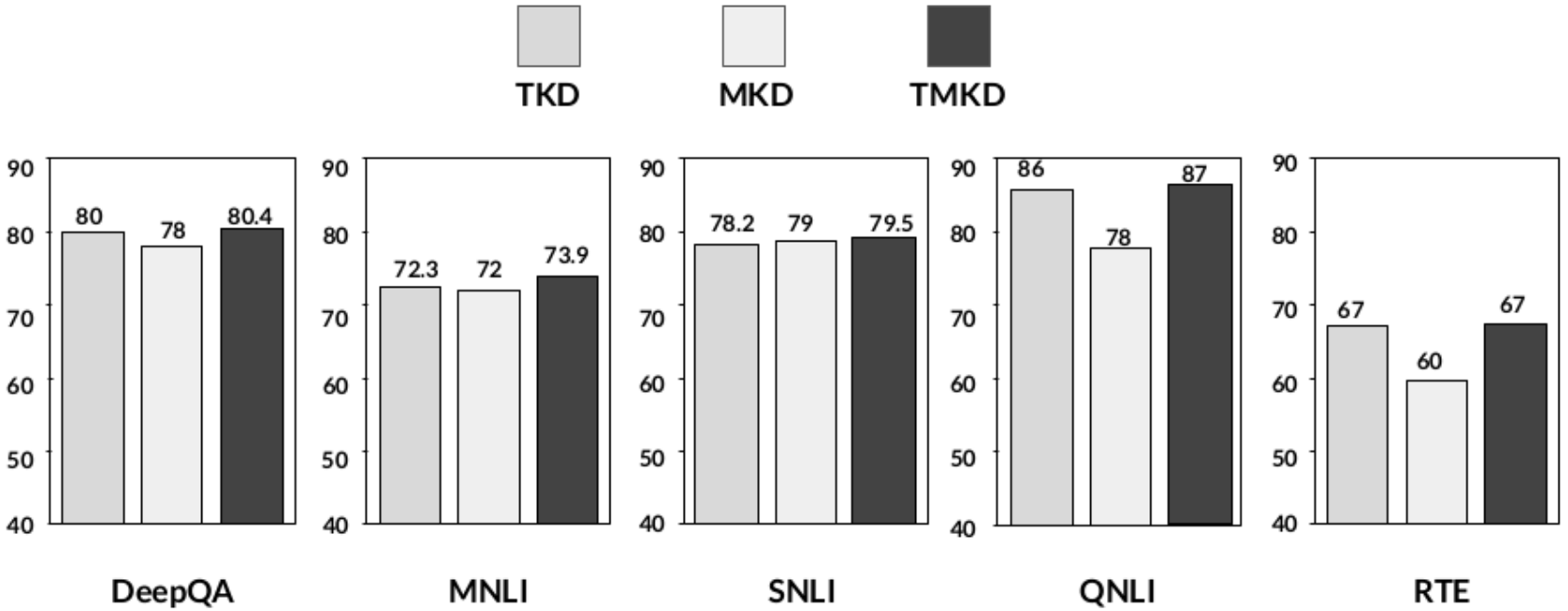
Knowledge Distillation

Case Study - How to Distill Knowledge for a Q&A Task

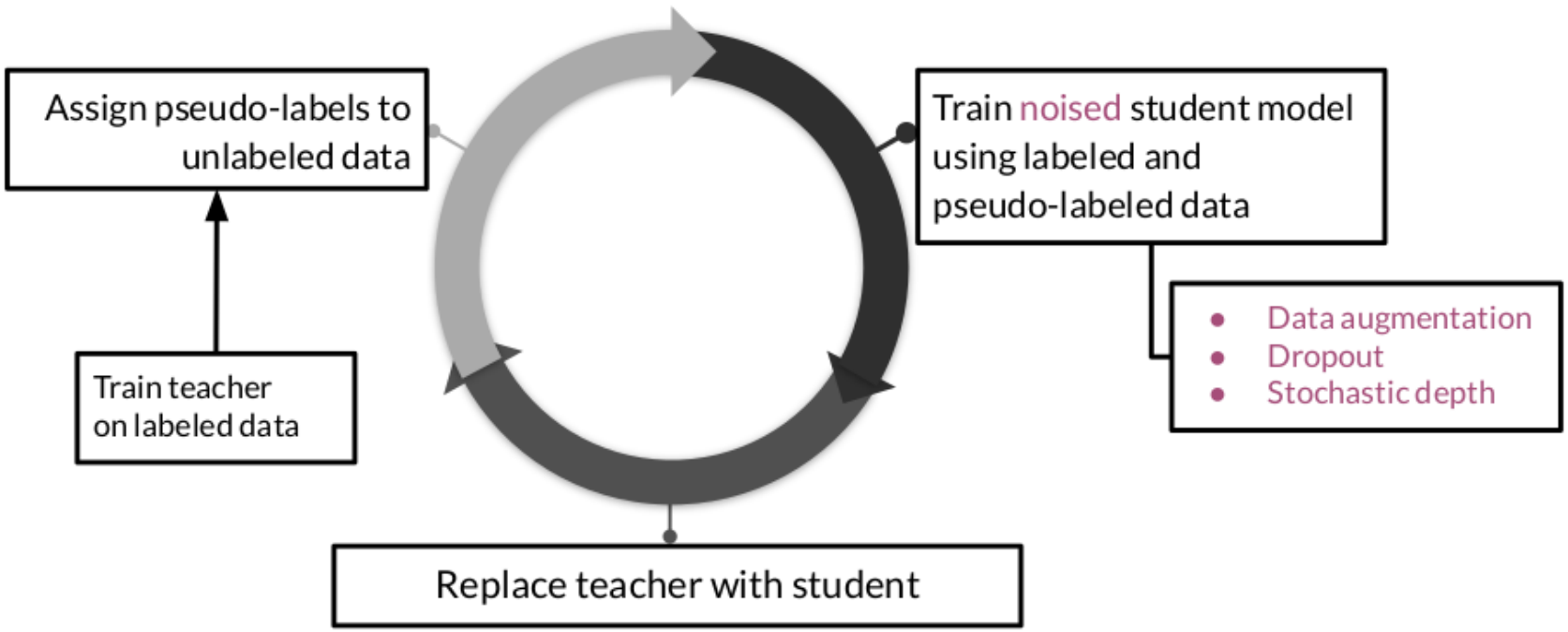
Two-stage multi-teacher distillation for Q&A



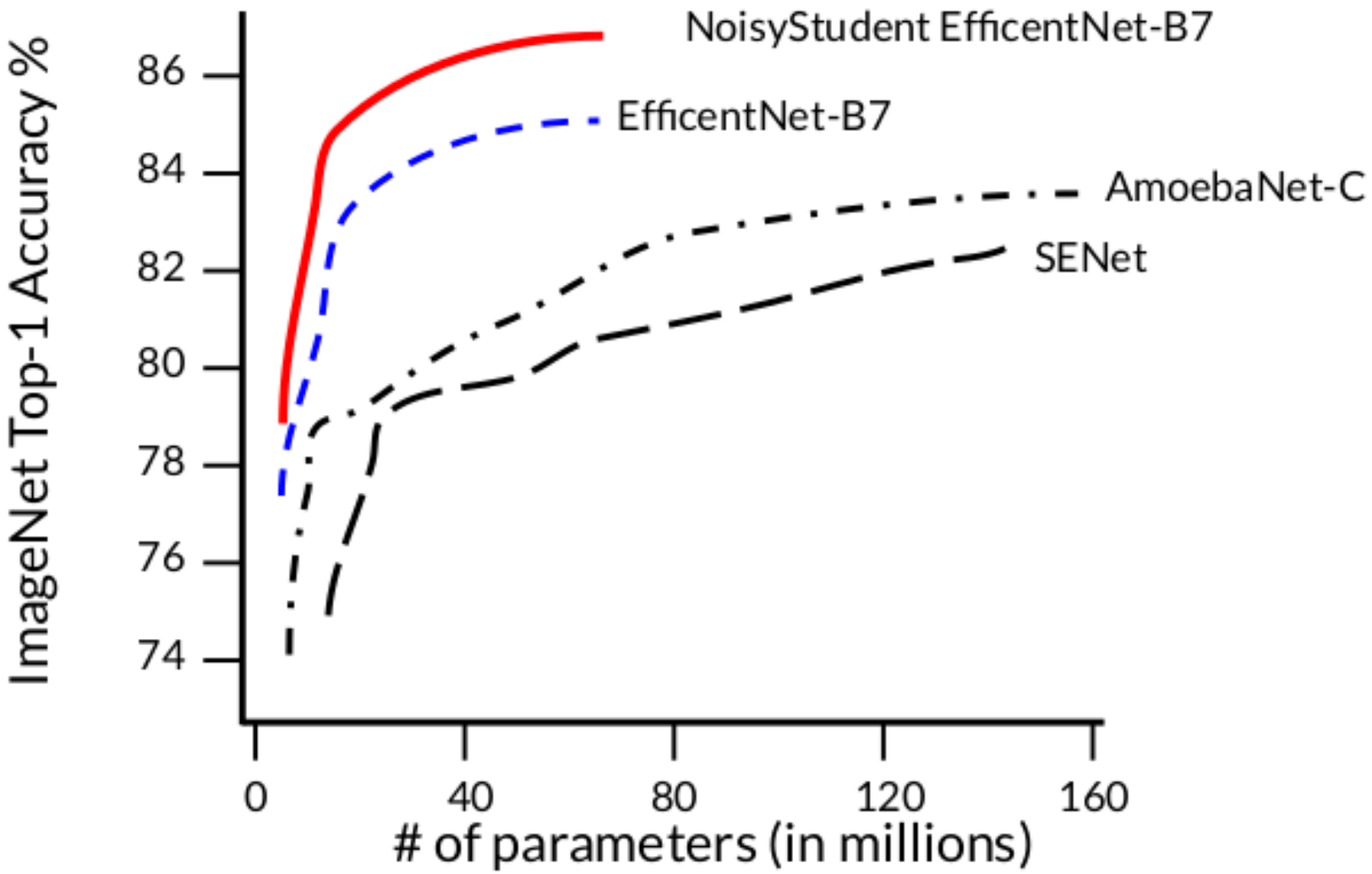
Impact of two-stage knowledge distillation



Make EfficientNets robust to noise with distillation



Results of noisy student training



Labs for This Week

Objective

Briefly describe the learning goal for this week's lab(s).

Lab Activities:

- Lab 10: [KD] - [Knowledge Distillation Lab]
- Lab 10: [FSDP] - [Model Parallelism]
- Lab 10: [DDP] - [Data Parallelism]
- Lab 10: [Mixed Precision] - [Model Parallelism]
- Lab 10: [DDP] - [Data Parallelism]

Submission Deadline: [Before the next class]

Choose two

- Assignment 10: Your first lab
- Assignment 10: Your second lab

Reading Materials

This Week's Theme

Topic focus: [People + AI Guidebook - Data Collection + Evaluation.pdf]
You should use the worksheet related to this pdf to your project and submit it when its requested.

Required Readings:

- [On the Reliable Detection of Concept Drift from Streaming Unlabeled Data]

Be prepared to discuss highlights and open questions in class.

Sources

 DeepLearning.AI

 The People + AI Guidebook