Input Data Processing for Machine Learning

Ana Klimovic

Joint work with Jiri Simsa*, Derek Murray+, Ihor Indyk*

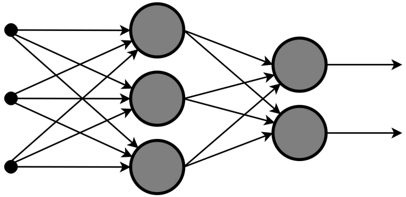
FastPath 2021
March 28, 2021

*Google

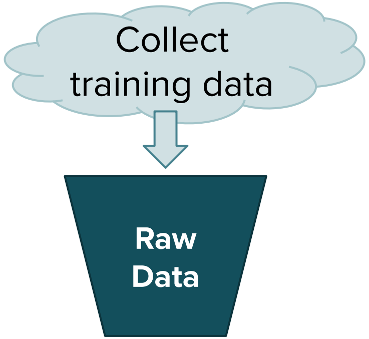
Systems@ETH Zürich

Data is critical for ML training and inference

Train/serve model



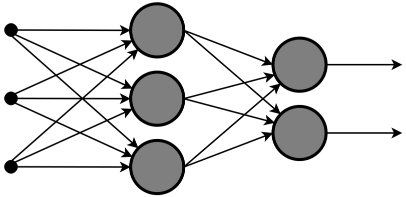
Data is critical for ML training



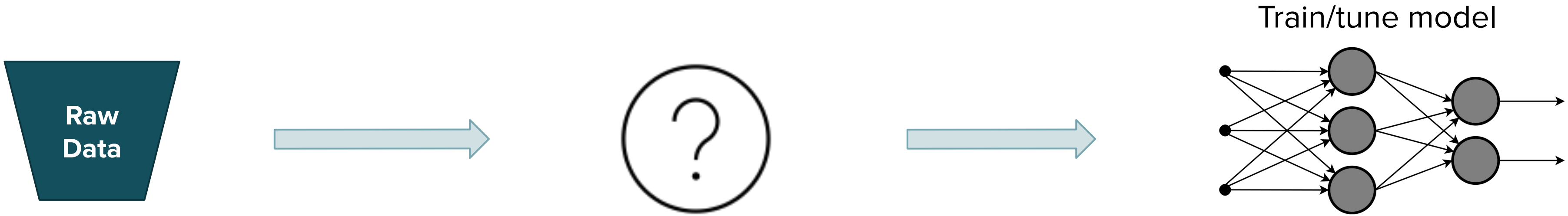
Collect training data

Raw Data

Train/tune model



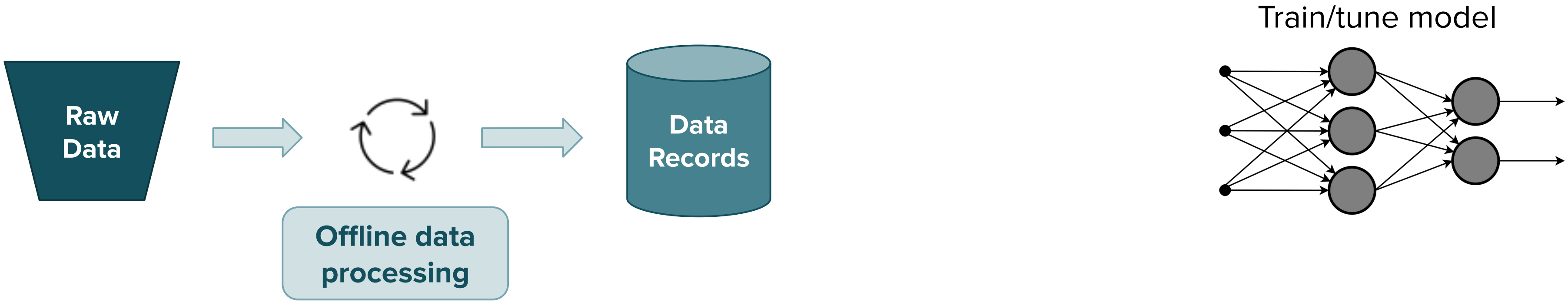
Data processing is critical



Raw Data

Train/tune model

Some data processing can be done offline...



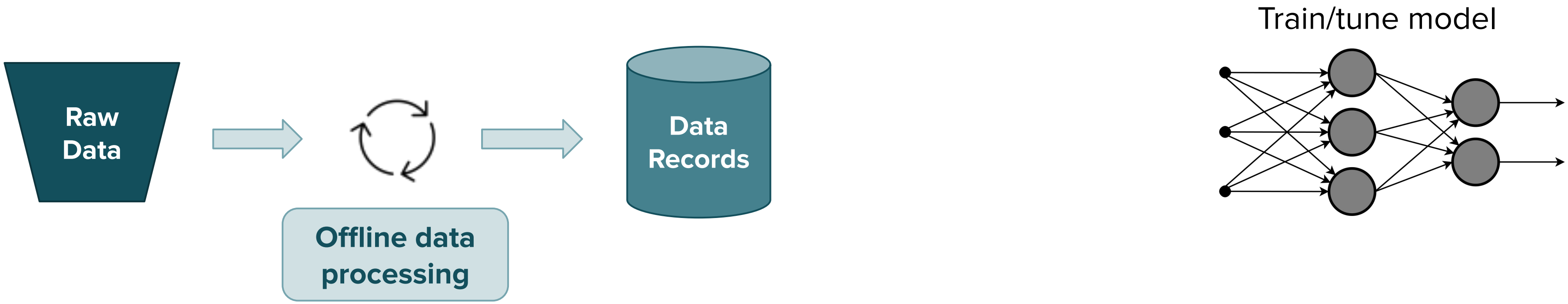
Raw Data

Offline data processing

Data Records

Train/tune model

Some data processing can be done offline...



Raw Data

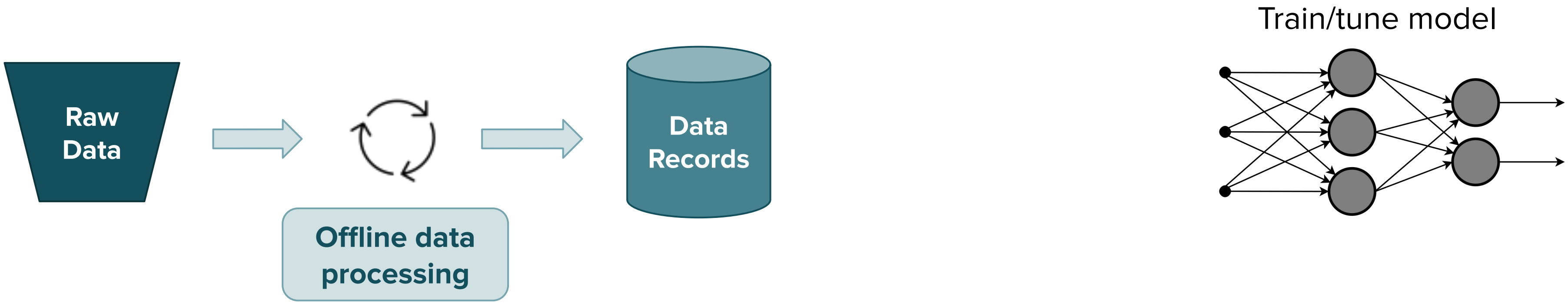
Offline data processing

Data Records

Train/tune model

- *Extract features*
- *Clean data*
- *Validate data*
- *Normalize data*
- *Convert data to binary format*

Some data processing can be done offline...



Offline data processing

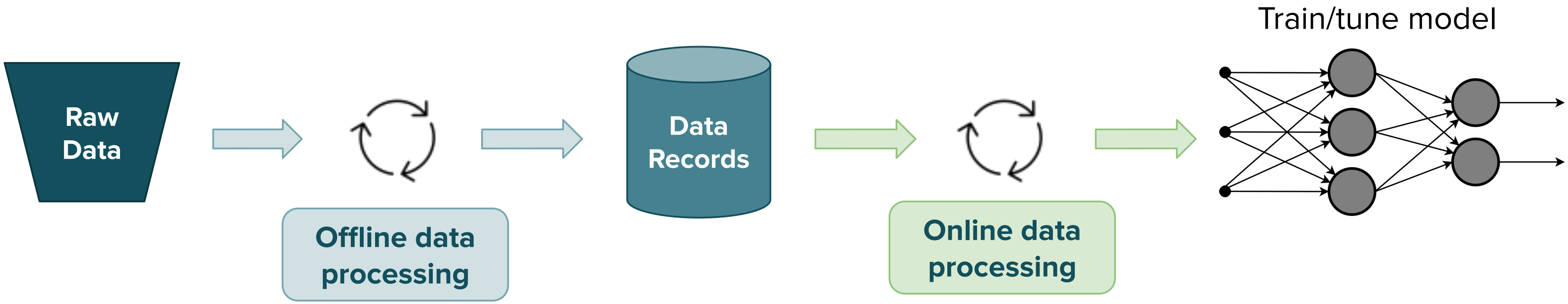
Apache Beam

APACHE Spark

FLUME

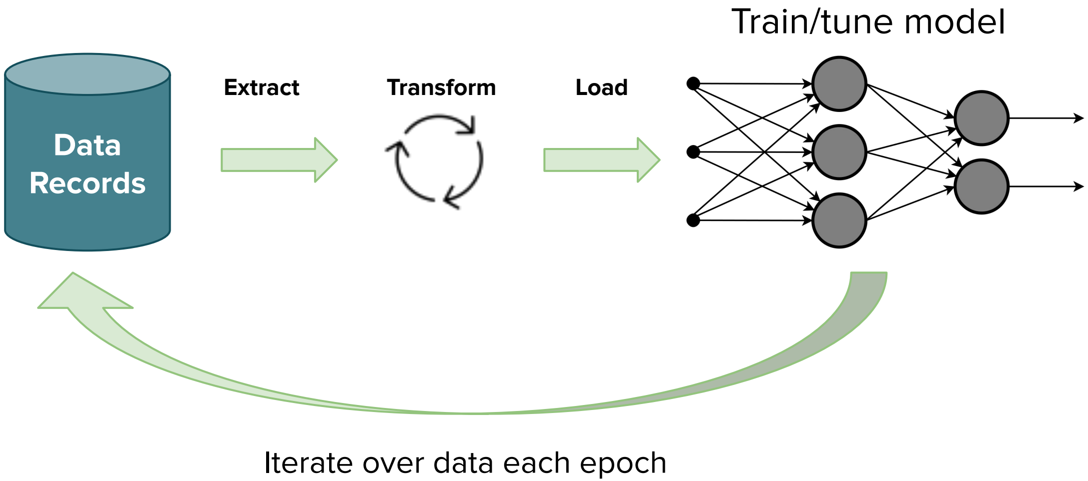
- *Extract features*
- *Clean data*
- *Validate data*
- *Normalize data*
- *Convert data to binary format*

Some data processing must be done online...



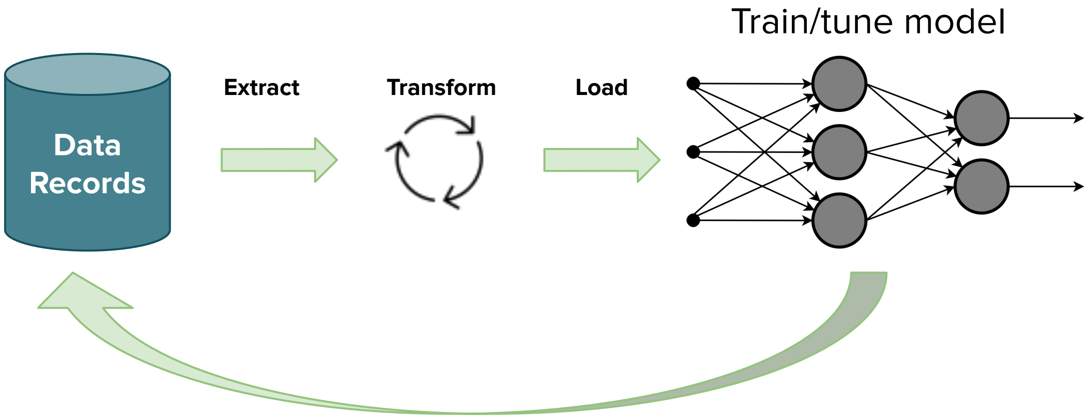
Some data processing must be done online...

During training:



“Last mile” data processing

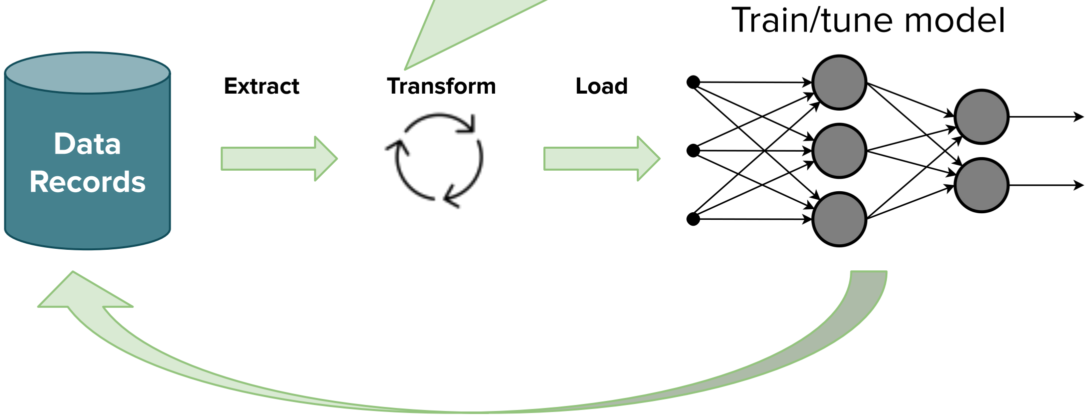
The ***Extract, Transform, Load*** input data pipeline that runs during ML training.



“Last mile” data processing

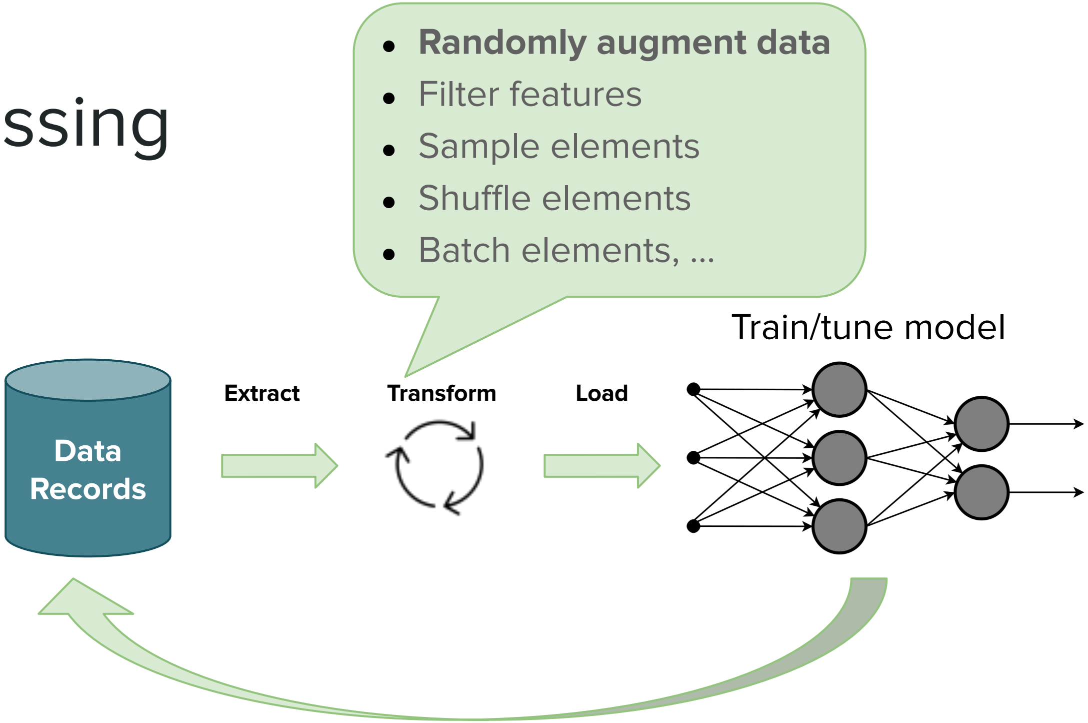
The ***Extract, Transform, Load*** input data pipeline that runs during ML training.

- Randomly augment data
- Filter features
- Sample elements
- Shuffle elements
- Batch elements, ...

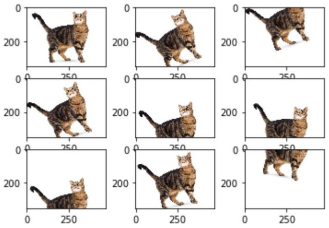


“Last mile” data processing

The ***Extract, Transform, Load*** input data pipeline that runs during ML training.



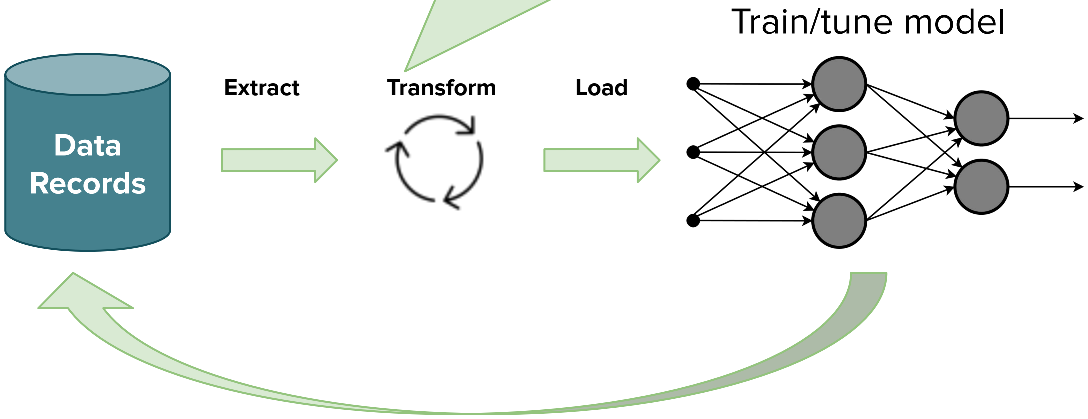
- **Randomly augment data**
- Filter features
- Sample elements
- Shuffle elements
- Batch elements, …



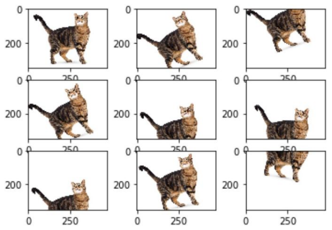
“Last mile” data processing

The ***Extract, Transform, Load*** input data pipeline that runs during ML training.

- **Randomly augment data**
- Filter features
- Sample elements
- Shuffle elements
- Batch elements, …

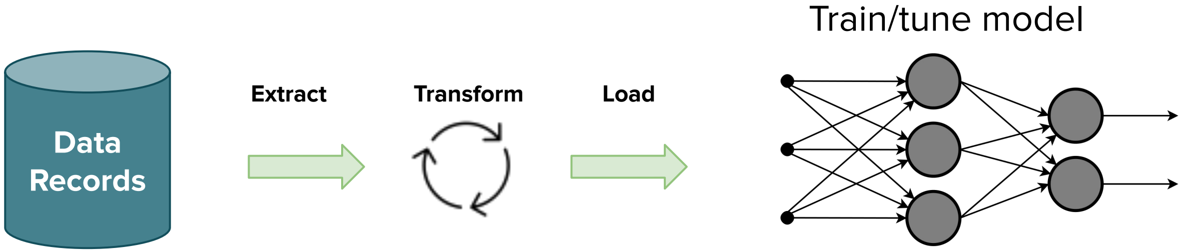


Transformations that ***add randomness*** *and/or* that users want to ***tune at runtime*** (e.g., batch size, feature selection)



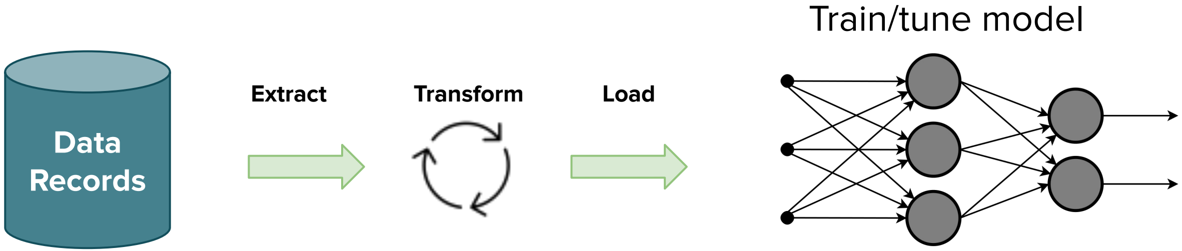
Why care about last mile data processing?

1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)



Why care about last mile data processing?

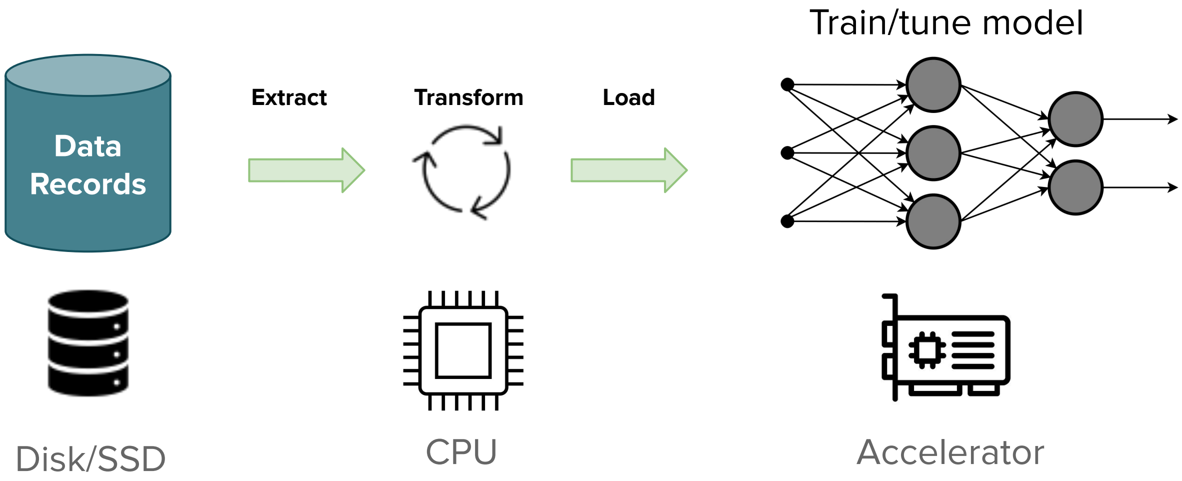
1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
    - For high performance and accelerator utilization, need Rate$_{load\_data}$ >= Rate$_{train\_model}$



Why care about last mile data processing?

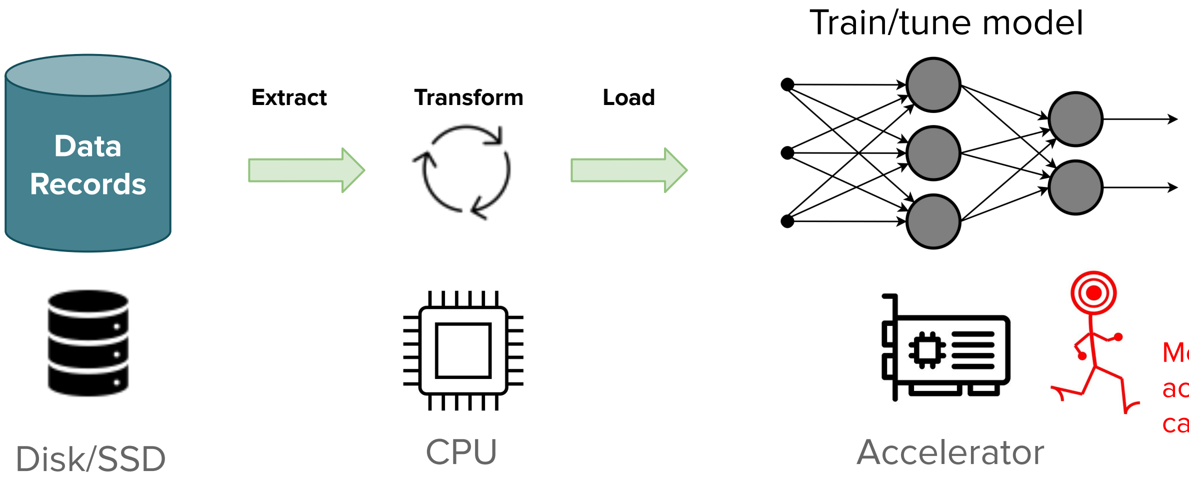
1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)

   - For high performance and accelerator utilization, need Rate$_{load\_data}$ >= Rate$_{train\_model}$



Why care about last mile data processing?

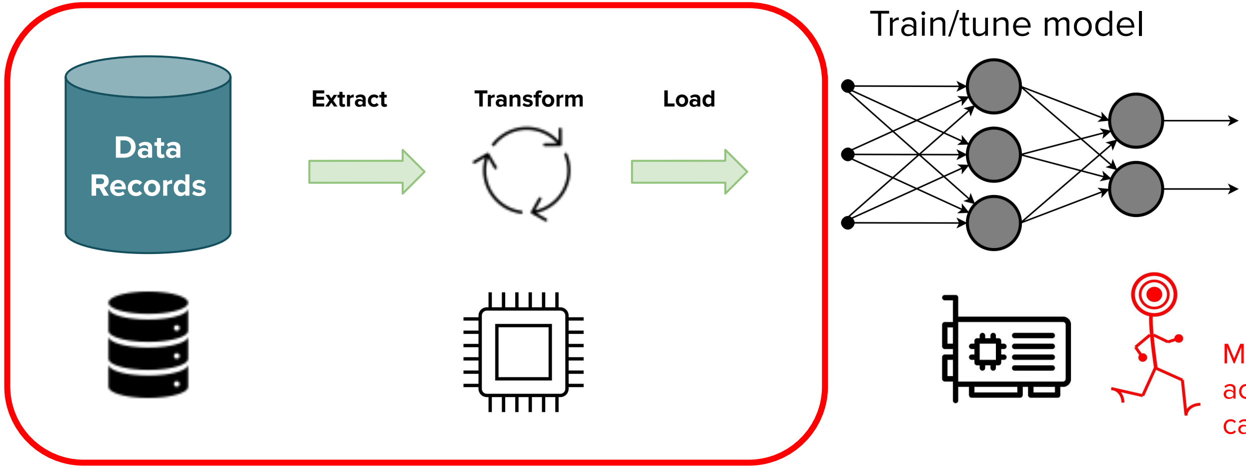
1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
   - For high performance and accelerator utilization, need Rate$_{\text{load\_data}}$ >= Rate$_{\text{train\_model}}$



Moving target…
accelerator compute
capabilities keep increasing

Why care about last mile data processing?

1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
   - For high performance and accelerator utilization, need Rate$_{load\_data}$ >= Rate$_{train\_model}$

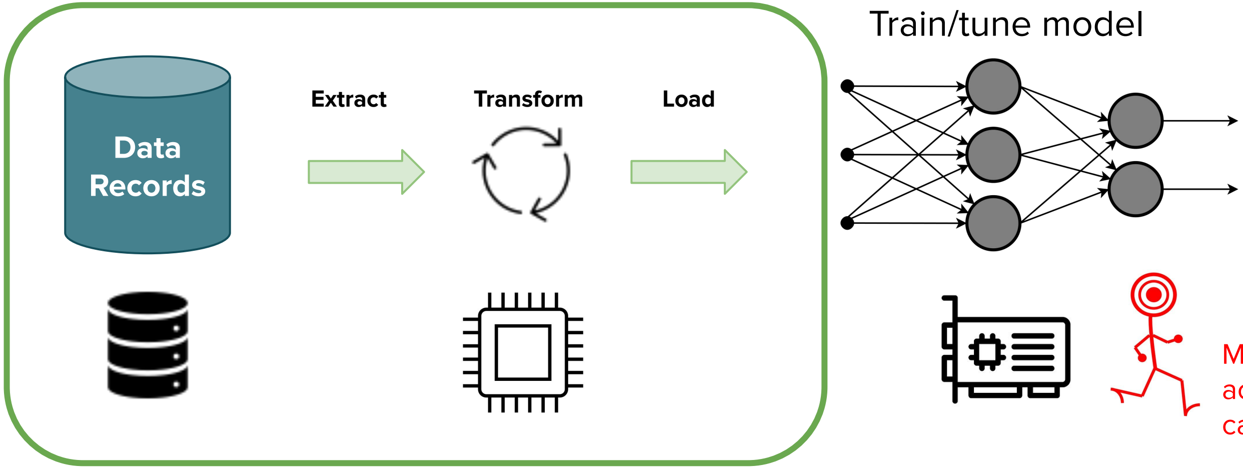


Performance Bottleneck

Moving target… accelerator compute capabilities keep increasing

Why care about last mile data processing?

1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
    - For high performance and accelerator utilization, need Rate$_{\text{load\_data}}$ >= Rate$_{\text{train\_model}}$
    - e.g., **removing input data bottlenecks** can improve ResNet-50 training time by over **20x**



Moving target…
accelerator compute
capabilities keep increasing

Why care about last mile data processing?

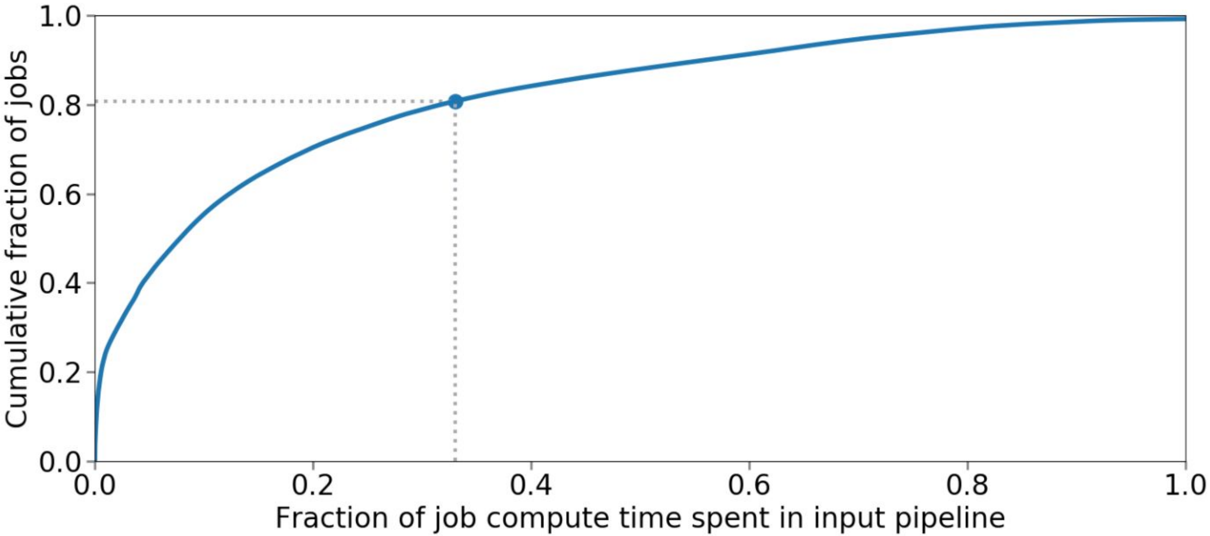
1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
2. Consumes **significant compute resources** in ML jobs

Why care about last mile data processing?

1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
2. Consumes **significant compute resources** in ML jobs
    - On average, 30% of total ML training compute time is spent on last mile data processing

Why care about last mile data processing?

1. Greatly impacts **end-to-end training time** and **accelerator utilization** ($$)
2. Consumes **significant compute resources** in ML jobs
   - On average, 30% of total ML training compute time is spent on last mile data processing



tf.data: framework for input data processing

Framework for building and executing efficient input data processing for ML jobs.

tf.data API provides generic operators that can be composed and parameterized:

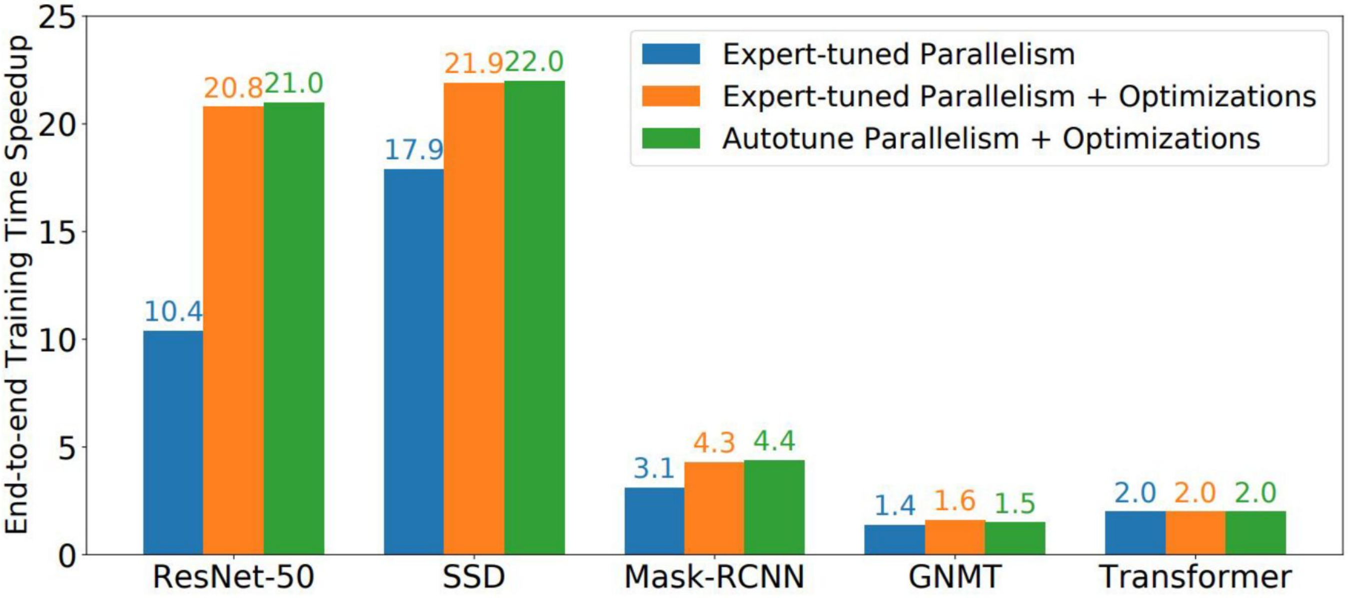
- Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

tf.data runtime efficiently executes input pipelines by applying:

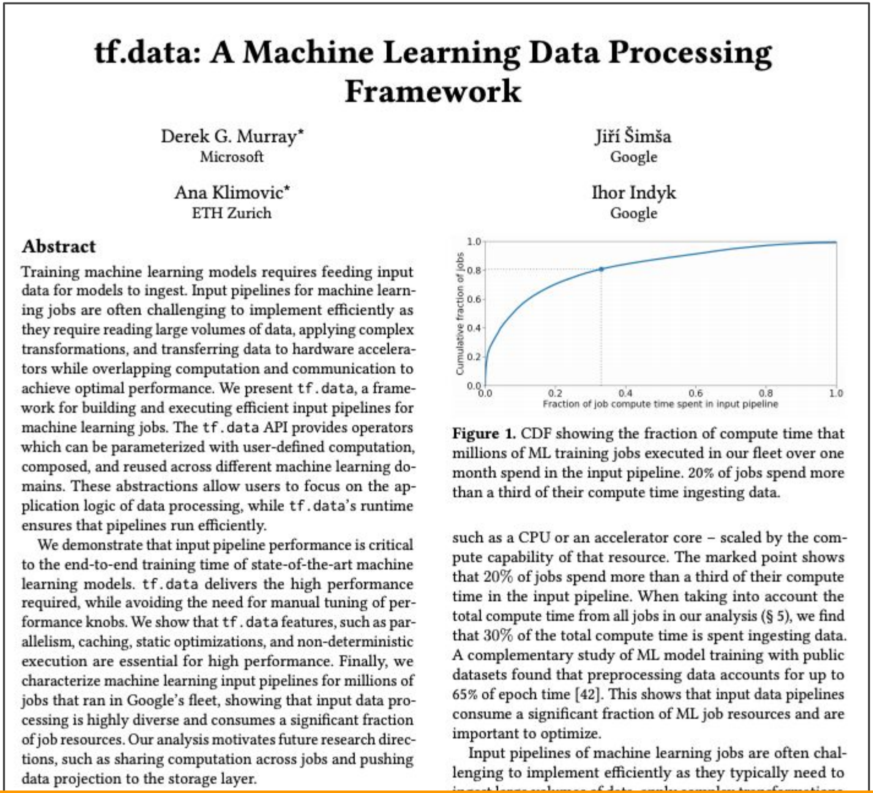
- Parallelism, software pipelining, prefetching
- Static optimizations (e.g., operator fusion)
- Dynamic optimizations (autotuning the degree of parallelism and prefetch buffer sizes)

Impact of tf.data performance optimizations

Up to 22x speedup on MLPerf training with tf.data optimizations vs. baseline tf.data with no parallelism or optimizations



More about tf.data in our paper



tf.data: A Machine Learning Data Processing Framework

Derek G. Murray*
Microsoft

Jiří Šimša
Google

Ana Klimovic*
ETH Zurich

Ihor Indyk
Google

Abstract

Training machine learning models requires feeding input data for models to ingest. Input pipelines for machine learning jobs are often challenging to implement efficiently as they require reading large volumes of data, applying complex transformations, and transferring data to hardware accelerators while overlapping computation and communication to achieve optimal performance. We present `tf.data`, a framework for building and executing efficient input pipelines for machine learning jobs. The `tf.data` API provides operators which can be parameterized with user-defined computation, composed, and reused across different machine learning domains. These abstractions allow users to focus on the application logic of data processing, while `tf.data`'s runtime ensures that pipelines run efficiently.

We demonstrate that input pipeline performance is critical to the end-to-end training time of state-of-the-art machine learning models. `tf.data` delivers the high performance required, while avoiding the need for manual tuning of performance knobs. We show that `tf.data` features, such as parallelism, caching, static optimizations, and non-deterministic execution are essential for high performance. Finally, we characterize machine learning input pipelines for millions of jobs that ran in Google's fleet, showing that input data processing is highly diverse and consumes a significant fraction of job resources. Our analysis motivates future research directions, such as sharing computation across jobs and pushing data projection to the storage layer.

1.0
0.8
0.6
0.4
0.2
0.0
Cumulative fraction of jobs
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of job compute time spent in input pipeline

Figure 1. CDF showing the fraction of compute time that millions of ML training jobs executed in our fleet over one month spend in the input pipeline. 20% of jobs spend more than a third of their compute time ingesting data.

such as a CPU or an accelerator core – scaled by the compute capability of that resource. The marked point shows that 20% of jobs spend more than a third of their compute time in the input pipeline. When taking into account the total compute time from all jobs in our analysis (§ 5), we find that 30% of the total compute time is spent ingesting data. A complementary study of ML model training with public datasets found that preprocessing data accounts for up to 65% of epoch time [42]. This shows that input data pipelines consume a significant fraction of ML job resources and are important to optimize.

Input pipelines of machine learning jobs are often challenging to implement efficiently as they typically need to

https://arxiv.org/pdf/2101.12127.pdf

What can we learn from Google's ML input pipelines?

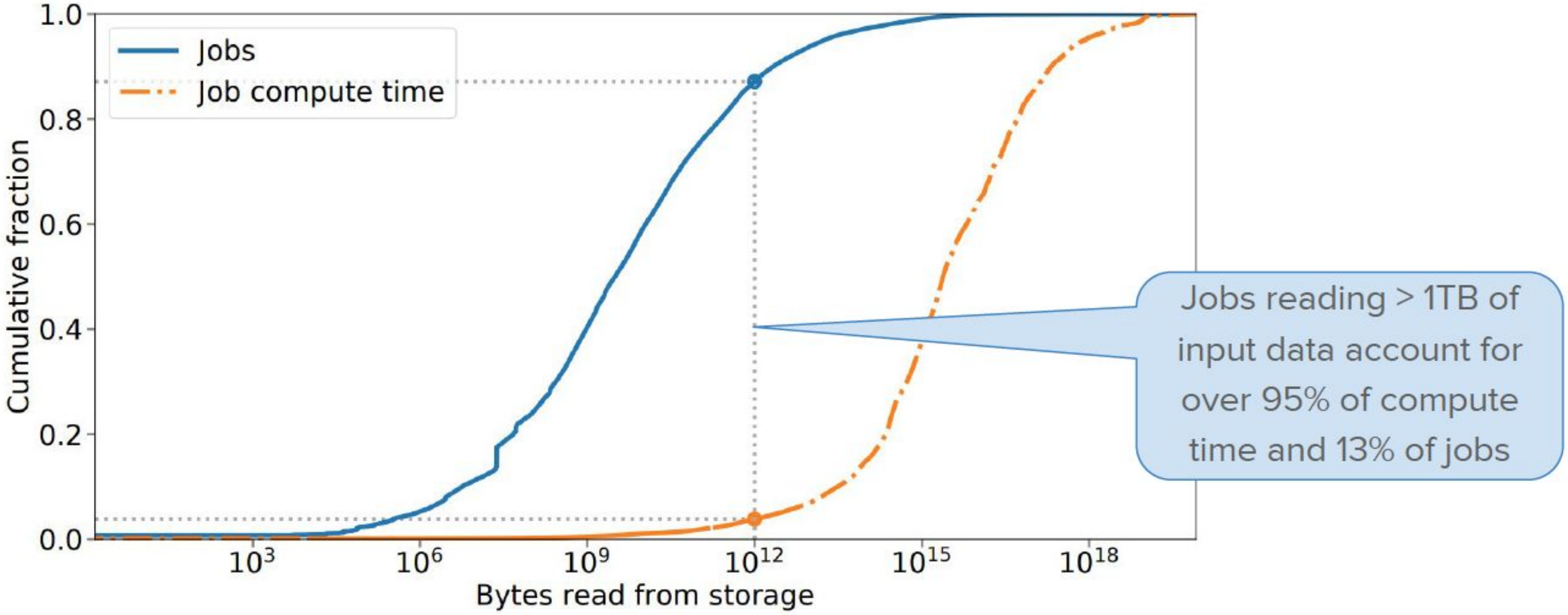
Fleetwide analysis of millions of tf.data jobs during one month in 2020 at Google.

Our goals:

- Characterize last mile input data processing
  - How much input data do ML jobs ingest?
  - How does last mile data processing affect data volume?
  - What is the opportunity to reuse computation in the last mile?
- Identify opportunities to further optimize input data processing

How much input data do jobs read?

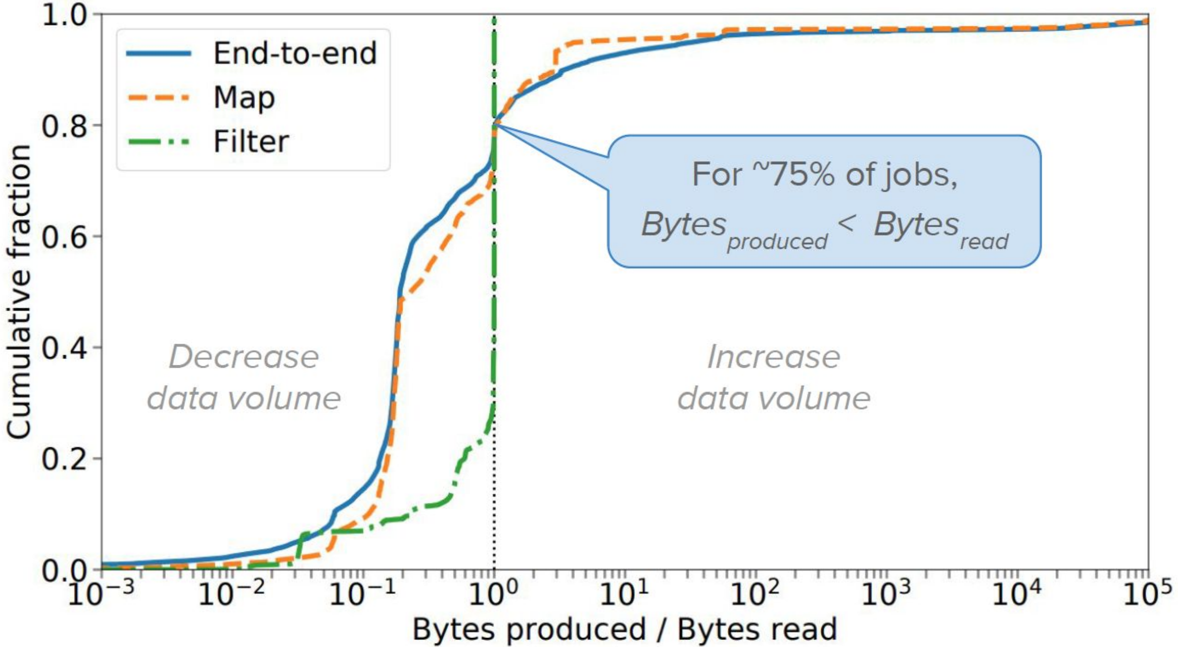
Takeaway: for a significant fraction of jobs, input data does not fit in memory.



Jobs reading > 1TB of input data account for over 95% of compute time and 13% of jobs

How does last mile processing affect data volume?

Takeaway: input data processing can significantly reduce or expand data. Overall, it is more common for input pipelines to produce less bytes than they read from sources.

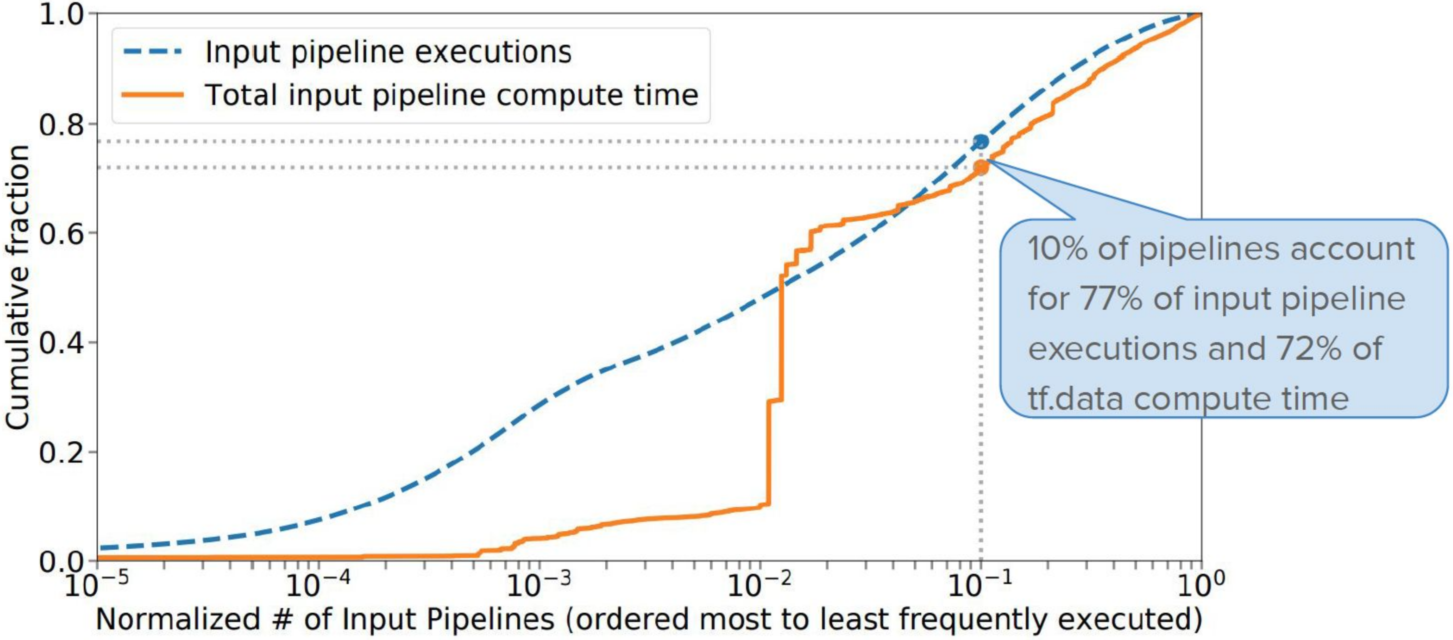


For ~75% of jobs,

$Bytes_{produced} < Bytes_{read}$

How frequently are input pipelines re-executed?

Takeaway: Input pipelines are frequently re-executed. There is a large opportunity to reuse computation in the last mile within and across jobs.



10% of pipelines account for 77% of input pipeline executions and 72% of tf.data compute time

Insights from Google’s ML input data pipelines

- Input data processing consumes **significant compute resources**
- **Larger-than-memory datasets** are common
- Data processing involves diverse transformations, however overall many transformations **reduce the data volume**
- Many input data pipelines are **repeatedly re-executed**

What are the implications for system design?

We can optimize last mile data processing by building an **input data service** with a **global view** of data processing **across jobs**.

What are the implications for system design?

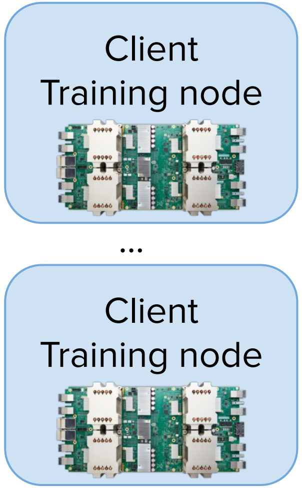
We can optimize last mile data processing by building an **input data service** with a **global view** of data processing **across jobs**.

The input data service should *automatically*:

- Scale distributed resources for data processing to eliminate input data bottlenecks
- Cache and reuse the outputs of “expensive” input data pipelines
- Execute parts of input data pipelines close to storage vs. close to accelerators

tf.data service

Distribute input data processing across multiple CPU workers to avoid bottlenecks.



Client
Training node

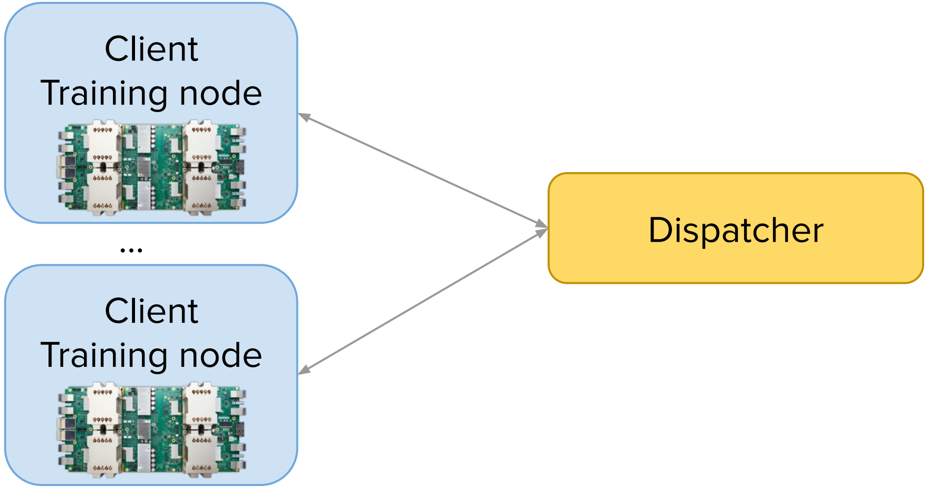
...

Client
Training node

Ongoing work by: tf.data team at Google

tf.data service

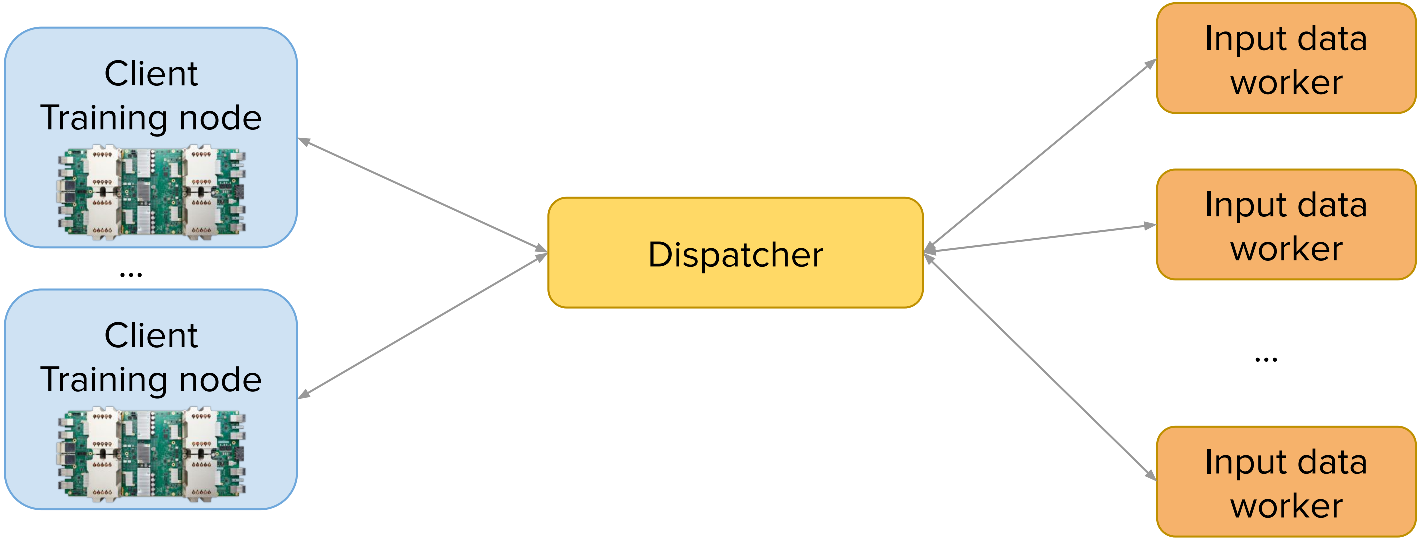
Distribute input data processing across multiple CPU workers to avoid bottlenecks.



Ongoing work by: tf.data team at Google

tf.data service

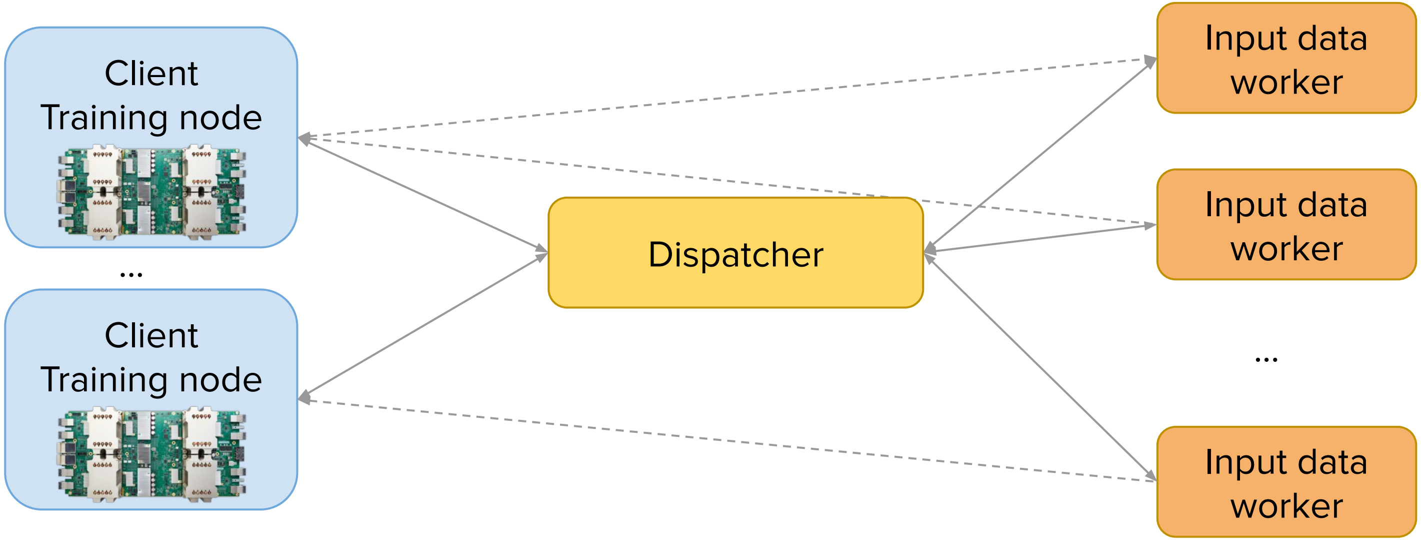
Distribute input data processing across multiple CPU workers to avoid bottlenecks.



Ongoing work by: tf.data team at Google

tf.data service

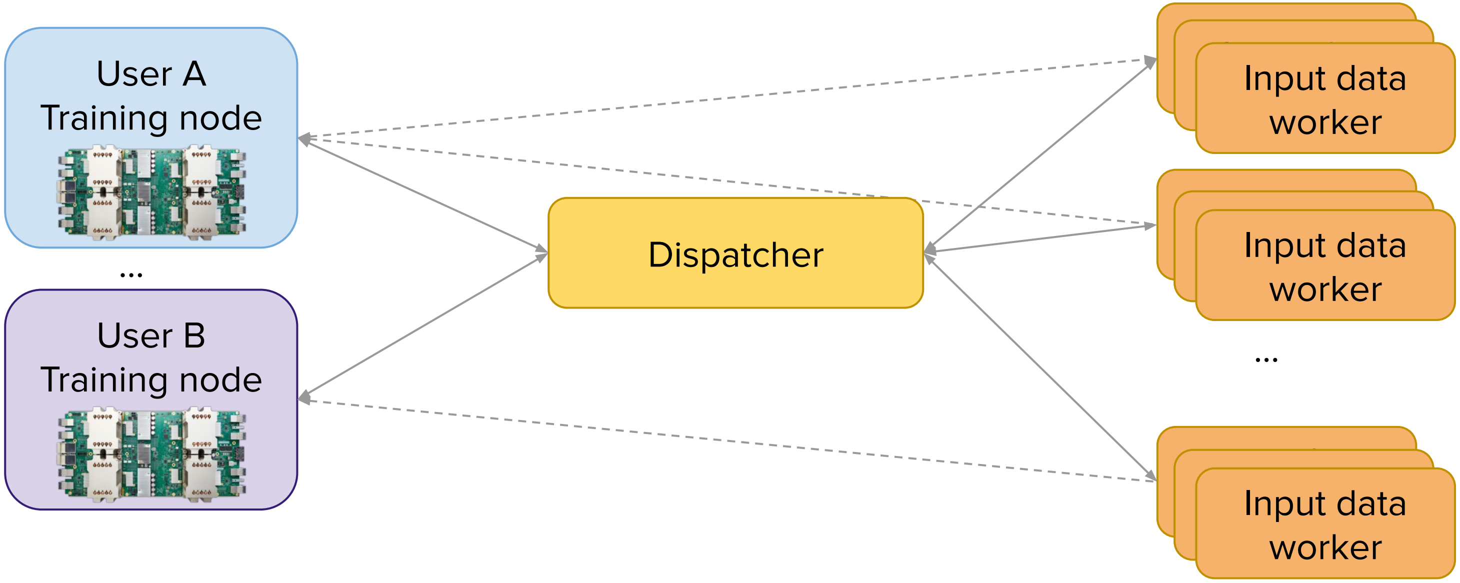
Distribute input data processing across multiple CPU workers to avoid bottlenecks.



Ongoing work by: tf.data team at Google

Multi-tenant input data service

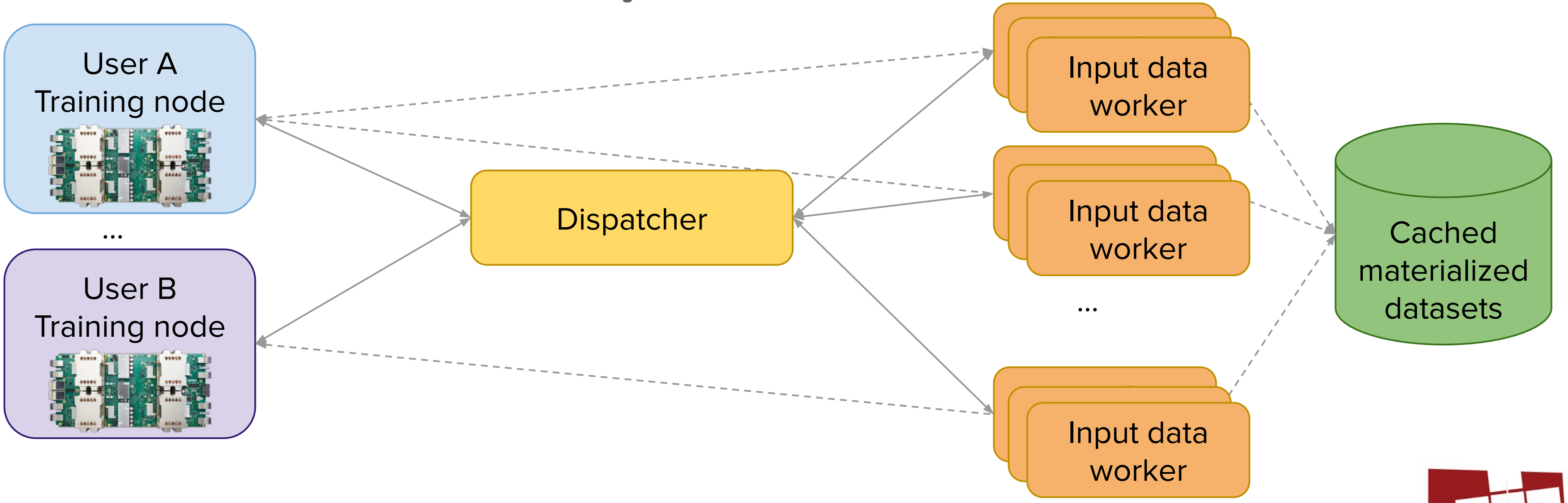
Elastically scale input data workers to eliminate input data bottlenecks



Ongoing work by: Damien Aymon, Dan Graur, Tanguy Albrici, Julia Bazinska, Chandu Thekkath

Multi-tenant input data service

Elastically scale input data workers to eliminate input data bottlenecks and reuse cached datasets within and across jobs



Ongoing work by: Damien Aymon, Dan Graur, Tanguy Albrici, Julia Bazinska, Chandu Thekkath

Why is reusing last mile computation challenging?

- Caching and reusing outputs removes randomness in the input data pipeline
  ➜ may negatively impact training accuracy.

Why is reusing last mile computation challenging?

- Caching and reusing outputs removes randomness in the input data pipeline
  ➜ may negatively impact training accuracy.
- The cost-benefit analysis for caching vs. recomputing is non-trivial:
  - How compute-intensive is the input pipeline?
  - How frequently is the input pipeline executed within and across jobs?
  - Is this input pipeline a performance bottleneck in training jobs?
  - How large is the materialized dataset and how long does it take to read/deserialize?
  - How does reuse affect model training dynamics?

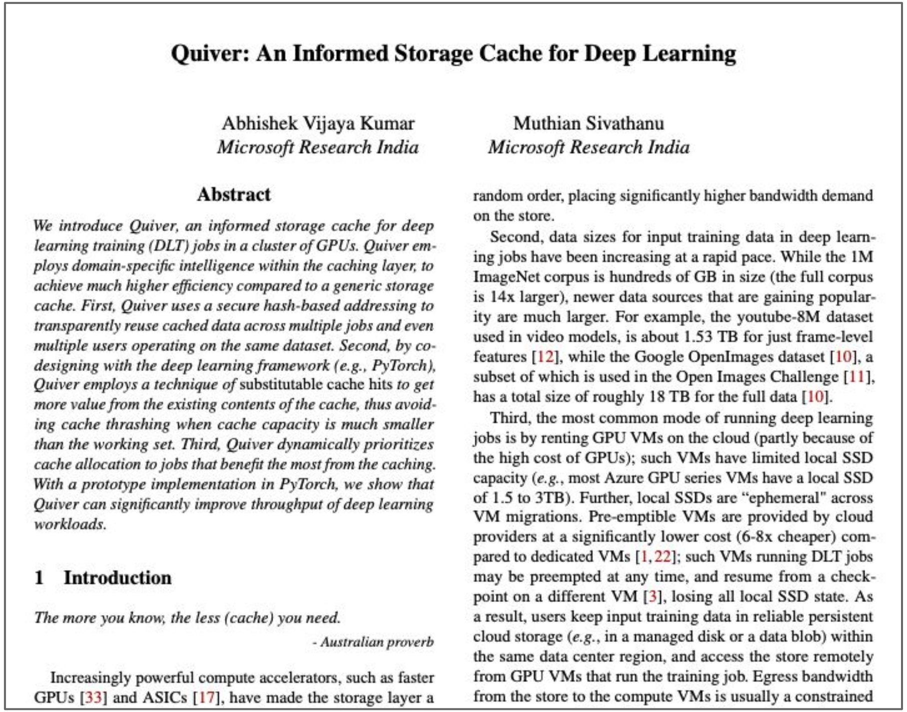
Why is reusing last mile computation challenging?

- Caching and reusing outputs removes randomness in the input data pipeline
  ➔ may negatively impact training accuracy.
- The cost-benefit analysis for caching vs. recomputing is non-trivial:
  - How compute-intensive is the input pipeline?
  - How frequently is the input pipeline executed within and across jobs?
  - Is this input pipeline a performance bottleneck in training jobs?
  - How large is the materialized dataset and how long does it take to read/deserialize?
  - How does reuse affect model training dynamics?
- How to detect cache hit? ➔ consider graph isomorphism, access control

Why is reusing last mile computation challenging?

- Caching and reusing outputs removes randomness in the input data pipeline
  ➔ may negatively impact training accuracy.
- The cost-benefit analysis for caching vs. recomputing is non-trivial:
  - How compute-intensive is the input pipeline?
  - How frequently is the input pipeline executed within and across jobs?
  - Is this input pipeline a performance bottleneck in training jobs?
  - How large is the materialized dataset and how long does it take to read/deserialize?
  - How does reuse affect model training dynamics?
- How to detect cache hit? ➔ consider graph isomorphism, access control
- Considering sub-graphs leads to large search space ➔ need to prune

Other recent work optimizing last mile data processing...



Quiver: An Informed Storage Cache for Deep Learning

Abhishek Vijaya Kumar
Microsoft Research India

Muthian Sivathanu
Microsoft Research India

Abstract

We introduce Quiver, an informed storage cache for deep learning training (DLT) jobs in a cluster of GPUs. Quiver employs domain-specific intelligence within the caching layer, to achieve much higher efficiency compared to a generic storage cache. First, Quiver uses a secure hash-based addressing to transparently reuse cached data across multiple jobs and even multiple users operating on the same dataset. Second, by co-designing with the deep learning framework (e.g., PyTorch), Quiver employs a technique of substitutable cache hits *to get more value from the existing contents of the cache, thus avoiding cache thrashing when cache capacity is much smaller than the working set. Third, Quiver dynamically prioritizes cache allocation to jobs that benefit the most from the caching. With a prototype implementation in PyTorch, we show that Quiver can significantly improve throughput of deep learning workloads.*

1 Introduction

The more you know, the less (cache) you need.

- Australian proverb

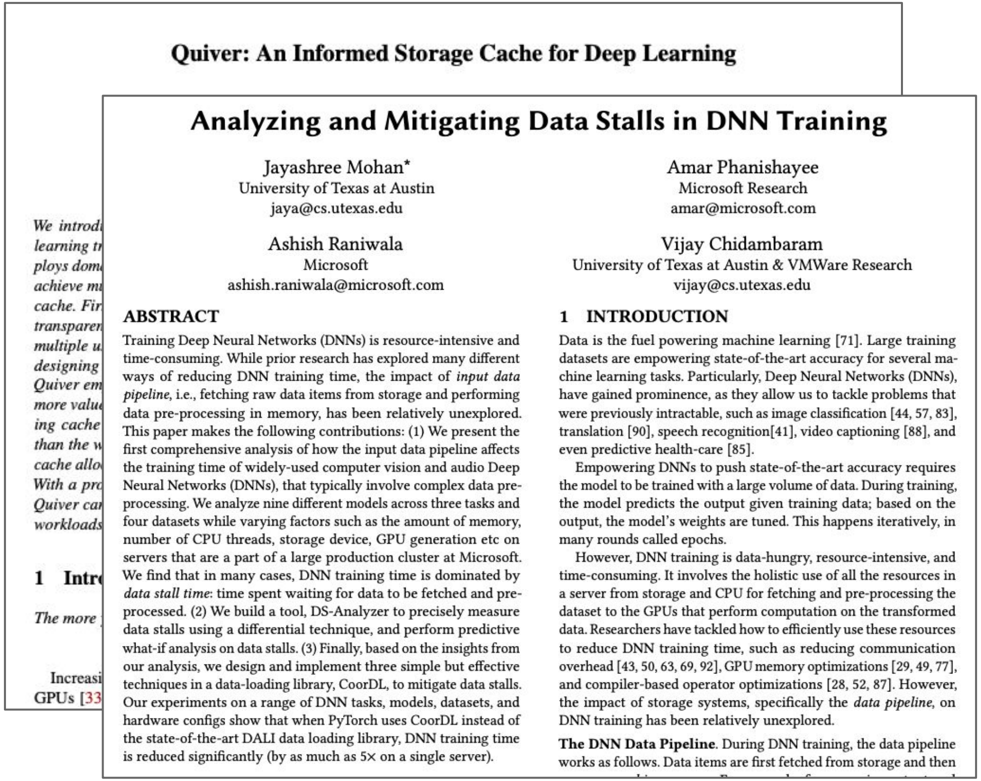
Increasingly powerful compute accelerators, such as faster GPUs [33] and ASICs [17], have made the storage layer a

random order, placing significantly higher bandwidth demand on the store.

Second, data sizes for input training data in deep learning jobs have been increasing at a rapid pace. While the 1M ImageNet corpus is hundreds of GB in size (the full corpus is 14x larger), newer data sources that are gaining popularity are much larger. For example, the youtube-8M dataset used in video models, is about 1.53 TB for just frame-level features [12], while the Google OpenImages dataset [10], a subset of which is used in the Open Images Challenge [11], has a total size of roughly 18 TB for the full data [10].

Third, the most common mode of running deep learning jobs is by renting GPU VMs on the cloud (partly because of the high cost of GPUs); such VMs have limited local SSD capacity (*e.g.*, most Azure GPU series VMs have a local SSD of 1.5 to 3TB). Further, local SSDs are “ephemeral" across VM migrations. Pre-emptible VMs are provided by cloud providers at a significantly lower cost (6-8x cheaper) compared to dedicated VMs [1, 22]; such VMs running DLT jobs may be preempted at any time, and resume from a checkpoint on a different VM [3], losing all local SSD state. As a result, users keep input training data in reliable persistent cloud storage (*e.g.*, in a managed disk or a data blob) within the same data center region, and access the store remotely from GPU VMs that run the training job. Egress bandwidth from the store to the compute VMs is usually a constrained

Other recent work optimizing last mile data processing…



Quiver: An Informed Storage Cache for Deep Learning

We introdu
learning tr
ploys dom
achieve m
cache. Fir
transparen
multiple u
designing
Quiver em
more valu
ing cache
than the w
cache allo
With a pro
Quiver ca
workloads

1 Intro

The more

Increasi
GPUs [33

Analyzing and Mitigating Data Stalls in DNN Training

Jayashree Mohan*
University of Texas at Austin
jaya@cs.utexas.edu

Amar Phanishayee
Microsoft Research
amar@microsoft.com

Ashish Raniwala
Microsoft
ashish.raniwala@microsoft.com

Vijay Chidambaram
University of Texas at Austin & VMWare Research
vijay@cs.utexas.edu

ABSTRACT

Training Deep Neural Networks (DNNs) is resource-intensive and time-consuming. While prior research has explored many different ways of reducing DNN training time, the impact of *input data pipeline*, i.e., fetching raw data items from storage and performing data pre-processing in memory, has been relatively unexplored. This paper makes the following contributions: (1) We present the first comprehensive analysis of how the input data pipeline affects the training time of widely-used computer vision and audio Deep Neural Networks (DNNs), that typically involve complex data pre-processing. We analyze nine different models across three tasks and four datasets while varying factors such as the amount of memory, number of CPU threads, storage device, GPU generation etc on servers that are a part of a large production cluster at Microsoft. We find that in many cases, DNN training time is dominated by *data stall time*: time spent waiting for data to be fetched and pre-processed. (2) We build a tool, DS-Analyzer to precisely measure data stalls using a differential technique, and perform predictive what-if analysis on data stalls. (3) Finally, based on the insights from our analysis, we design and implement three simple but effective techniques in a data-loading library, CoorDL, to mitigate data stalls. Our experiments on a range of DNN tasks, models, datasets, and hardware configs show that when PyTorch uses CoorDL instead of the state-of-the-art DALI data loading library, DNN training time is reduced significantly (by as much as 5× on a single server).

1 INTRODUCTION

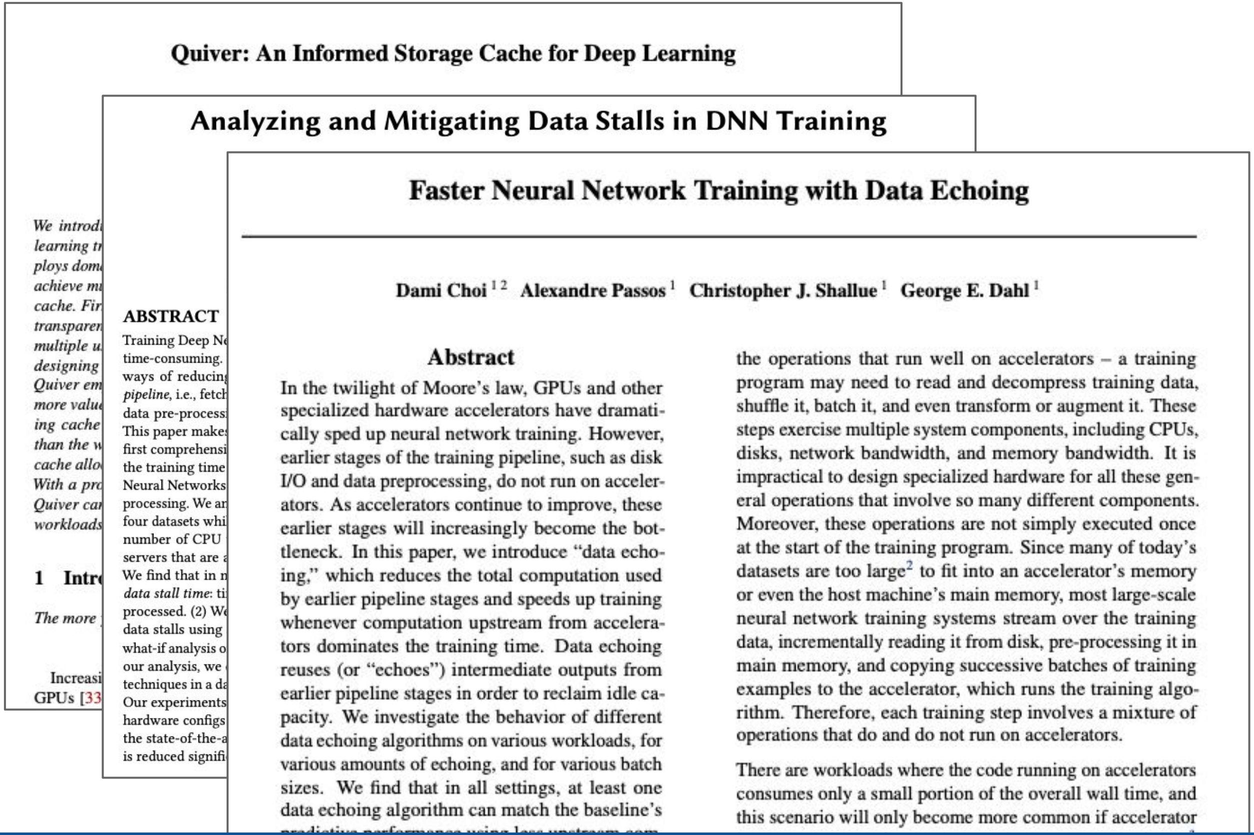
Data is the fuel powering machine learning [71]. Large training datasets are empowering state-of-the-art accuracy for several machine learning tasks. Particularly, Deep Neural Networks (DNNs), have gained prominence, as they allow us to tackle problems that were previously intractable, such as image classification [44, 57, 83], translation [90], speech recognition[41], video captioning [88], and even predictive health-care [85].

Empowering DNNs to push state-of-the-art accuracy requires the model to be trained with a large volume of data. During training, the model predicts the output given training data; based on the output, the model's weights are tuned. This happens iteratively, in many rounds called epochs.

However, DNN training is data-hungry, resource-intensive, and time-consuming. It involves the holistic use of all the resources in a server from storage and CPU for fetching and pre-processing the dataset to the GPUs that perform computation on the transformed data. Researchers have tackled how to efficiently use these resources to reduce DNN training time, such as reducing communication overhead [43, 50, 63, 69, 92], GPU memory optimizations [29, 49, 77], and compiler-based operator optimizations [28, 52, 87]. However, the impact of storage systems, specifically the *data pipeline*, on DNN training has been relatively unexplored.

The DNN Data Pipeline. During DNN training, the data pipeline works as follows. Data items are first fetched from storage and then

Other recent work optimizing last mile data processing…



Quiver: An Informed Storage Cache for Deep Learning

We introd
learning tr
ploys dom
achieve m
cache. Fir
transparen
multiple u
designing
Quiver em
more valu
ing cache
than the w
cache allo
With a pro
Quiver ca
workloads

1 Intr

The more

Increasi
GPUs [33

Analyzing and Mitigating Data Stalls in DNN Training

ABSTRACT

Training Deep N
time-consuming.
ways of reducin
pipeline, i.e., fetch
data pre-process
This paper make
first comprehensi
the training time
Neural Networks
processing. We ar
four datasets whi
number of CPU
servers that are
We find that in m
data stall time: ti
processed. (2) We
data stalls using
what-if analysis o
our analysis, we
techniques in a da
Our experiments
hardware configs
the state-of-the-a
is reduced signifi

Faster Neural Network Training with Data Echoing

Dami Choi [1 2] **Alexandre Passos** [1] **Christopher J. Shallue** [1] **George E. Dahl** [1]

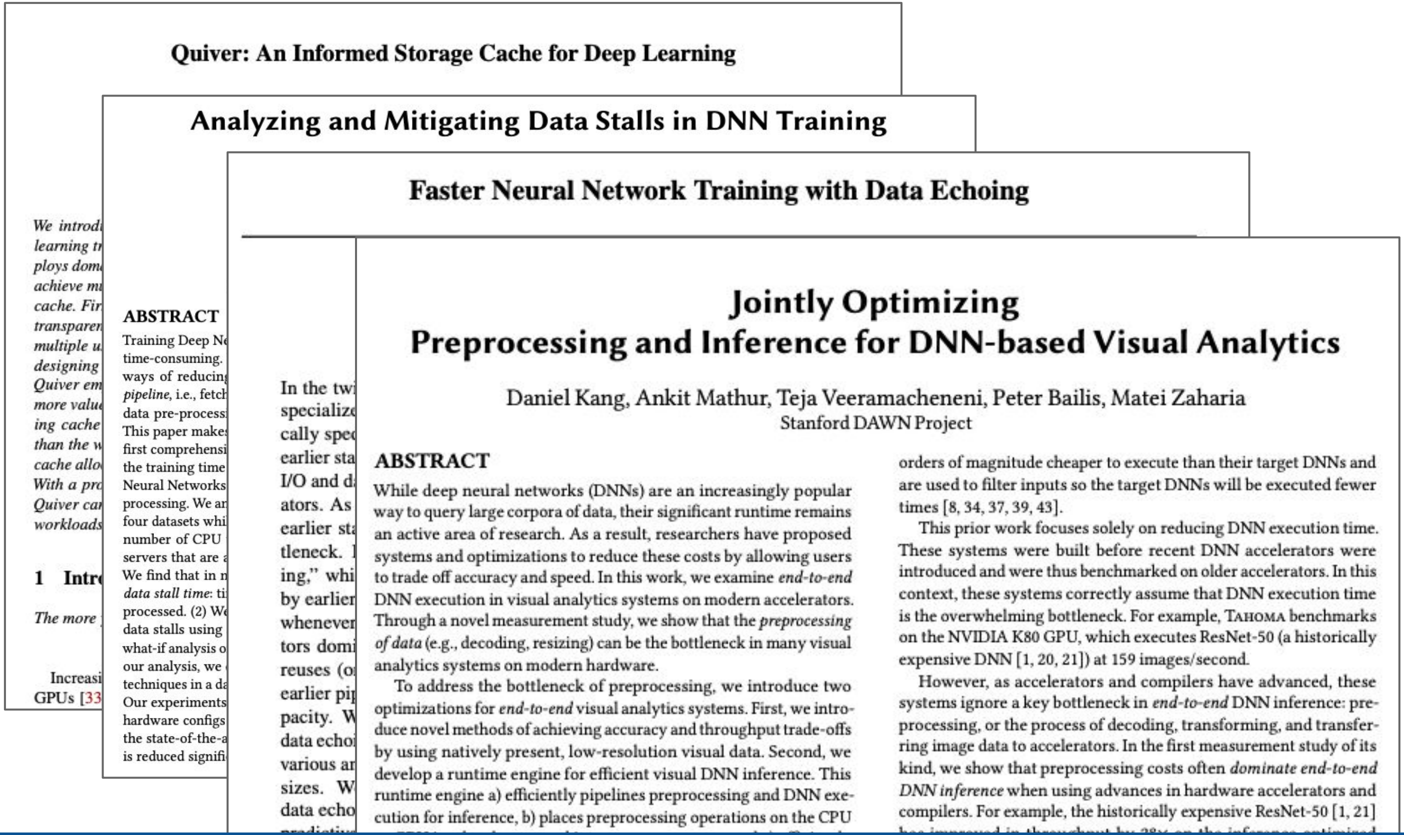
Abstract

In the twilight of Moore's law, GPUs and other specialized hardware accelerators have dramatically sped up neural network training. However, earlier stages of the training pipeline, such as disk I/O and data preprocessing, do not run on accelerators. As accelerators continue to improve, these earlier stages will increasingly become the bottleneck. In this paper, we introduce "data echoing," which reduces the total computation used by earlier pipeline stages and speeds up training whenever computation upstream from accelerators dominates the training time. Data echoing reuses (or "echoes") intermediate outputs from earlier pipeline stages in order to reclaim idle capacity. We investigate the behavior of different data echoing algorithms on various workloads, for various amounts of echoing, and for various batch sizes. We find that in all settings, at least one data echoing algorithm can match the baseline's

the operations that run well on accelerators – a training program may need to read and decompress training data, shuffle it, batch it, and even transform or augment it. These steps exercise multiple system components, including CPUs, disks, network bandwidth, and memory bandwidth. It is impractical to design specialized hardware for all these general operations that involve so many different components. Moreover, these operations are not simply executed once at the start of the training program. Since many of today's datasets are too large[2] to fit into an accelerator's memory or even the host machine's main memory, most large-scale neural network training systems stream over the training data, incrementally reading it from disk, pre-processing it in main memory, and copying successive batches of training examples to the accelerator, which runs the training algorithm. Therefore, each training step involves a mixture of operations that do and do not run on accelerators.

There are workloads where the code running on accelerators consumes only a small portion of the overall wall time, and this scenario will only become more common if accelerator

Other recent work optimizing last mile data processing...



Quiver: An Informed Storage Cache for Deep Learning

We introd
learning tr
ploys dom
achieve m
cache. Fir
transparen
multiple u
designing
Quiver em
more valu
ing cache
than the w
cache allo
With a pr
Quiver ca
workloads

1 Intr

The more

Increasi
GPUs [33

Analyzing and Mitigating Data Stalls in DNN Training

ABSTRACT

Training Deep N
time-consuming.
ways of reducin
pipeline, i.e., fetch
data pre-process
This paper make
first comprehensi
the training time
Neural Networks
processing. We a
four datasets whi
number of CPU
servers that are
We find that in
data stall time: ti
processed. (2) W
data stalls using
what-if analysis o
our analysis, we
techniques in a d
Our experiments
hardware configs
the state-of-the-
is reduced signifi

Faster Neural Network Training with Data Echoing

In the tw
specializ
cally spe
earlier sta
I/O and d
ators. As
earlier st
tleneck.
ing," whi
by earlier
whenever
tors domi
reuses (o
earlier pi
pacity. W
data echo
various a
sizes. W
data echo

Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics

Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, Matei Zaharia
Stanford DAWN Project

ABSTRACT

While deep neural networks (DNNs) are an increasingly popular way to query large corpora of data, their significant runtime remains an active area of research. As a result, researchers have proposed systems and optimizations to reduce these costs by allowing users to trade off accuracy and speed. In this work, we examine *end-to-end* DNN execution in visual analytics systems on modern accelerators. Through a novel measurement study, we show that the *preprocessing of data* (e.g., decoding, resizing) can be the bottleneck in many visual analytics systems on modern hardware.

To address the bottleneck of preprocessing, we introduce two optimizations for *end-to-end* visual analytics systems. First, we introduce novel methods of achieving accuracy and throughput trade-offs by using natively present, low-resolution visual data. Second, we develop a runtime engine for efficient visual DNN inference. This runtime engine a) efficiently pipelines preprocessing and DNN execution for inference, b) places preprocessing operations on the CPU

orders of magnitude cheaper to execute than their target DNNs and are used to filter inputs so the target DNNs will be executed fewer times [8, 34, 37, 39, 43].

This prior work focuses solely on reducing DNN execution time. These systems were built before recent DNN accelerators were introduced and were thus benchmarked on older accelerators. In this context, these systems correctly assume that DNN execution time is the overwhelming bottleneck. For example, TAHOMA benchmarks on the NVIDIA K80 GPU, which executes ResNet-50 (a historically expensive DNN [1, 20, 21]) at 159 images/second.

However, as accelerators and compilers have advanced, these systems ignore a key bottleneck in *end-to-end* DNN inference: preprocessing, or the process of decoding, transforming, and transferring image data to accelerators. In the first measurement study of its kind, we show that preprocessing costs often *dominate end-to-end DNN inference* when using advances in hardware accelerators and compilers. For example, the historically expensive ResNet-50 [1, 21]

Conclusion

- Last mile input data processing is a common bottleneck in ML training
  - Impacts end-to-end training performance and accelerator utilization ($)
- ML input data pipeline characterization at Google:
  - Datasets often exceed the size of main memory
  - Transformations can reduce or augment data, but overall often reduce
  - Many identical input data pipelines are repeatedly re-executed
- Opportunities to optimize last mile data processing:
  - Elastically scale data processing workers to eliminate input data bottlenecks
  - Reuse last mile computation across jobs
  - Run data reducing ops close to storage and data augmenting ops close to training node

tf.data design and fleetwide analysis paper: https://arxiv.org/pdf/2101.12127.pdf