Scalable Input Data Processing for Resource-Efficient ML

Ana Klimovic

ETH Zürich

SoCC Keynote
November 2022
It’s an exciting time for ML systems
Large growth in...

- ML use-cases
- ML model sizes
- Training data volume
Large growth in...

- ML use-cases
- ML model sizes
- Training data volume
- FLOPS provided by specialized ML hardware accelerators

Example: Google TPU pod
Large growth in...

- ML use-cases
- ML model sizes
- Training data volume
- FLOPS provided by specialized ML hardware accelerators
- Cost!!!

→ Training ML models consumes many GPU/TPU-hours and $$$
How much does it cost...

...to train a 100 Trillion parameter model for 1 day on the cloud?

A. $4,000
B. $40,000
C. $400,000
How much does it cost...
...to train a 100 Trillion parameter model for 1 day on the cloud?
A. $4,000
B. $40,000
C. $400,000

Used:
• 3,000 CPU cores
• 64 A100 GPUs
• 360 TB of RAM
How much does it cost...

...to train a 100 Trillion parameter model for 1 day on the cloud?

A. $4,000
B. $40,000
C. $400,000

Used:
- 3,000 CPU cores
- 64 A100 GPUs
- 360 TB of RAM

ML has a cost & resource efficiency problem
ML has a cost & resource efficiency problem

• Another example:

https://www.mosaicml.com/blog/gpt-3-quality-for-500k
How can we reduce the cost of ML?
How can we reduce the cost of ML?

Many complementary approaches...

Improve:

• Resource efficiency
• Resource cost
• Model efficiency
• Data efficiency
• ...

Systems
ML
How can we reduce the cost of ML?

Many complementary approaches...

Improve:

• Resource efficiency  ➔ maximize ML hardware (GPU/TPU) utilization
• Resource cost
• Model efficiency
• Data efficiency
• ...

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

• Resource efficiency → maximize ML hardware (GPU/TPU) utilization
• Resource cost → use cheap, transient resources (e.g., spot VMs)
• Model efficiency
• Data efficiency
• …
How can we reduce the cost of ML?

Many complementary approaches...

Improve:

• Resource efficiency → maximize ML hardware (GPU/TPU) utilization
• Resource cost → use cheap, transient resources (e.g., spot VMs)
• Model efficiency → sparsely activate models, sparse architectures
• Data efficiency
• ...

How can we reduce the cost of ML?

Many complementary approaches...

Improve:
- Resource efficiency $\rightarrow$ maximize ML hardware (GPU/TPU) utilization
- Resource cost $\rightarrow$ use cheap, transient resources (e.g., spot VMs)
- Model efficiency $\rightarrow$ sparsely activate models, sparse architectures
- Data efficiency $\rightarrow$ train on the most important/relevant data
- ...

How can we reduce the cost of ML?

Many complementary approaches…

Improve:

• Resource efficiency $\rightarrow$ maximize ML hardware (GPU/TPU) utilization

If we can ensure a job makes “good use” of ML hardware, the job will finish faster and we will pay for less time on that hardware.

Focus on maximizing GPU/TPU utilization $\rightarrow$ most $$$ component
What hinders high GPU/TPU utilization?

• Feeding GPUs/TPUs with input data is often a bottleneck
What hinders high GPU/TPU utilization?

• Feeding GPUs/TPUs with input data is often a bottleneck
  • Need to read large volumes of data from storage and preprocess data
What hinders high GPU/TPU utilization?

- Feeding GPUs/TPUs with input data is often a bottleneck
  - Need to read large volumes of data from storage and preprocess data
Input data ingestion for ML

Before we can feed training data to a model, we need to preprocess data.
Before we can feed training data to a model, we need to preprocess data.

**Offline preprocessing**
- Extract features
- Clean data
- Validate data
- Normalize data
Input data processing for ML

Before we can feed training data to a model, we need to preprocess data.
Input data processing for ML

Before we can feed training data to a model, we need to preprocess data.

Online preprocessing
- Filter features
- Sample elements
- Randomly augment
- Shuffle & batch

“Last mile” data processing
Input processing impacts training time & cost
Input processing impacts training time & cost

- Feeding data-hungry GPUs/TPUs is challenging
  - Input data processing on host CPU is often a bottleneck
Input processing consumes high CPU/energy
Input processing consumes high CPU/energy

• At Google, data processing consumes $\sim 30\%$ of compute time in training jobs [1]
• At Meta, data processing consumes more power than training for some jobs [2]

How to optimize ML input data processing?

1. **Autotune** the input data pipeline
2. **Disaggregate** and **distribute** data processing
3. **Multi-tenant** data processing **as a service**
How to optimize ML input data processing?

1. **Autotune** the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
How to optimize ML input data processing?

1. **Autotune** the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
tf.data: ML input data processing framework

• **API** provides generic operators that can be composed & parameterized:
  • Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

tf.data: ML input data processing framework

- **API** provides generic operators that can be composed & parameterized:
  - Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

```
read(file) → map(parse) → filter(cond) → map(crop) → shuffle() → batch() → prefetch()
```

tf.data: ML input data processing framework

- **API** provides generic operators that can be composed & parameterized:
  - Consists of stateless *datasets* (to define pipeline) and stateful *iterators* (to produce elements)

```
read(file) → map(parse) → filter(cond) → map(crop) → shuffle() → batch() → prefetch()
```

- **Runtime** efficiently executes input pipelines by applying:
  - Software pipelining and parallelism
  - Static optimizations (e.g., operator fusion)
  - Dynamic optimizations (autotuning parallelism & prefetch buffer sizes)

import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.
tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)

model = ...
model.fit(dataset, epochs=10)

batch data for training efficiency
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.
tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)

overlap data processing and loading
```python
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

*train model with tf.data dataset*
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord", num_parallel_readers=Z)
dataset = dataset.map(preprocess, num_parallel_calls=Y)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```python
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord", num_parallel_readers=Z)
dataset = dataset.map(preprocess, num_parallel_calls=Y)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=X)

model = ...
model.fit(dataset, epochs=10)
```

**tf.data runtime applies optimizations to the input pipeline under the hood**

- **tf.data.AUTO_TUNE**: This feature allows tf.data to automatically tune CPU/mem allocations to minimize output latency, modelled by M/M/1/k queue at each iterator.
Plumber: input pipeline perf debug/tuning

• Identify which op of the input pipeline is the bottleneck
• Adjust op CPU/memory/storage resource allocations to alleviate bottlenecks:
  • Measure resource accounted rate (i.e., “cost”) for each operator
  • If Operator B is twice as “expensive” as Operator A, give Operator B twice the resources
  • Cast resource allocation as an integer linear programming problem

Training speedup with tf.data optimizations

Baseline is input pipeline logic with no software parallelism or graph optimizations.

Training speedup with tf.data optimizations

Baseline is input pipeline logic with no software parallelism or graph optimizations.
How to optimize ML input data processing?

1. **Autotune** the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service

Autotuning tries to make best use of CPU and RAM available on the training node for high-throughput data processing.
How much CPU/RAM to provision per GPU/TPU?

It is hard to determine the right resource ratio for a ML training node.

→ Ideal resource allocation depends on the model and input pipeline

Training jobs benefit differently when given more CPU for data processing per accelerator core
How much CPU/RAM to provision per GPU/TPU?

It is hard to determine the right resource ratio for a ML training node.

Example of normalized CPU and RAM usage CDF, from ~73K ML training jobs at Google:

We need a scalable data processing architecture

Need to adjust resource allocation over time. ML training is increasingly data-hungry.

At Meta, storage and bandwidth has grown over 2x and 4x over the past 2 years.

Mark Zhao et al. “Understanding data storage and ingestion for large-scale deep recommendation model training”, ISCA 2022.
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
Solution: disaggregate data processing

• Independently scale resources for input data processing & model training
Solution: disaggregate data processing

- Independently scale resources for input data processing & model training
- Approach taken at Google (tf.data service), Meta (DPP), …

A case for disaggregating of ML data processing

Andrew Audibert
Google

Yang Chen
Google

Dan Graur
ETH Zurich

Ana Klomovic
ETH Zurich

Chandramohan A. Thekkath
Google

Jiri Simha
Google

Abstract

Machine Learning (ML) computation requires feeding input data for the models to ingest. Traditionally, input data processing happens on the same host as the ML computation [8, 25]. The input data processing can however become a bottleneck of the ML computation if there are insufficient resources to process data quickly enough. This slows down the ML computation and wastes valuable and scarce ML hardware (e.g., GPUs and TPUs) used by the ML computation.

In this paper, we present a data service, a disaggregated input data processing service built on top of tf.data. Our work goes beyond describing the design and implementation of a new system which disaggregates preprocessing from ML computation and presents: (1) empirical evidence based on production workloads for the need of disaggregation, as well as quantitative measurement of the impact disaggregation has on the performance and cost of production workloads, (2) benefits of disaggregation beyond horizontal scaling, (3) analysis of data service’s adoption at Google, the lessons learned during building and deploying the system and potential future lines of research opened up by our work.

We demonstrate that horizontally scaling data processing using tf.data service helps remove input bottlenecks, achieving speedups of up to 110x and job cost reductions of up to 89%. We further show that if data service can support computation reuse through data sharing across ML jobs with iden-

Understand Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training

Mark Zhao¹, Niket Agarwal¹, Aarti Basant¹, Bujra Gedik¹, Satadru Pun¹, Mustafa Onal², Rakesh Komuravelli¹, Jerry Pan¹, Tianhua Bao¹, Hao Lui¹, Sundaram Narayanan¹, Jack Langman¹, Kevin Wilfing¹, Harsha Rastogi¹, Carole-Jean Wu¹, Christos Kozyrakis¹, Parik Pol³

¹Meta, Stanford University

ABSTRACT

Datacenter-scale AI training clusters consisting of thousands of domain-specific accelerators (DSAs) are used to train increasingly complex deep learning models. These clusters rely on a data storage and ingestion (DSI) pipeline, responsible for loading end-to-end training data and serving it at tens of terabytes per second. As DSAs continue to push training efficiency and throughput, the DSI pipeline is becoming the dominating factor that constrains the overall training performance and capacity. Innovations that improve the performance of DSIs systems and hardware are urgent, demanding a deep understanding of DSI characteristics and infrastructure at scale.

This paper presents Meta’s end-to-end DSI pipeline, composed of a central data warehouse built on distributed storage and a Data Processing Service that scales to eliminate data stalls. We characterize how hundreds of models are collaboratively trained across geographically distributed datacenters via diverse and continuous training jobs. These training jobs read and heavily filter massive and evolving datasets, resulting in popular features and samples used across training jobs. We measure the network, memory, and compute resources required for each training job to prepare samples during training. Finally, we synthesize key takeaways based on our production infrastructure characterization. These include identifying hardware bottlenecks, discussing opportunities for heterogeneous DSI hardware, motivating research in datacenter scheduling and benchmark datasets, and outlining lessons learned in optimalizing DSI infrastructure.

1 INTRODUCTION

Domain-specific accelerators (DSAs) for deep neural networks (DNNs) have become ubiquitous because of their superior performance per watt over traditional general-purpose processors [46]. Industry has rapidly deployed DSAs for both DNN training and inference. These DSAs include both traditional technologies, such as GPUs and FPGAs, as well as application-specific integrated circuits (ASICs) (e.g., Habana [37], Graphcore [45], and NumWorks [47]), and others. DSAs are increasingly deployed in massive scale-out systems to train increasingly complex and computationally-demanding DNNs using massive datasets. For example, the latest MLPerf Training round 0.1-0.5 contains submissions from Azure and NVIDIA on training BERT using 248 and 4320 A100 GPUs, respectively. While Google submitted...
tf.data service: disagg ML data processing
tf.data service: disagg ML data processing

Users register ML data processing job with the tf.data service dispatcher.
tf.data service: disagg ML data processing

The dispatcher distributes data processing across remote workers
tf.data service: disagg ML data processing

Clients fetch processed data from workers in time for the next training step
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(preprocess)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)

register input pipeline with dispatcher
Benefits of disaggregated ML data processing

Remove input bottlenecks
Benefits of disaggregated ML data processing

Remove input bottlenecks → up to 110x speedup

Training time speedup

Benefits of disaggregated ML data processing

Remove input bottlenecks → up to 110x speedup, 89x cost reduction

How to optimize ML input data processing?

1. Autotune the input data pipeline
2. Disaggregate and distribute data processing
3. Multi-tenant data processing as a service
ML data processing as a service
ML data processing as a service

Can we leverage a global view of data processing across jobs?
Why leverage knowledge across jobs?

- Input data pipeline are often re-executed across jobs
  - e.g., hyperparameter tuning
Cachew: ML data processing as a service

The dispatcher *autoscales* workers → just enough workers to avoid data stalls
Cachew: ML data processing as a service

The dispatcher decides which datasets to cache in fast, distributed storage.
Challenges for ML data processing service

1. How to efficiently **autoscale resources** for input data processing?

![Graph showing epoch time vs. CPU cores per GPU/TPU accelerator](image)

**Training jobs benefit differently when given more CPU for data processing per accelerator core.**
Challenges for ML data processing service

1. How to efficiently **autoscale resources** for input data processing?
2. How/when to efficiently **cache and re-use** (transformed) datasets?
Challenges for ML data processing service

1. How to efficiently **autoscale resources** for input data processing?
2. How/when to efficiently **cache and re-use** (transformed) datasets?

Caching does not always improve performance…

- Input data reading may not be the training bottleneck
- Transformed dataset may be much larger than source dataset, saturating cache I/O bandwidth
- Reusing non-deterministically transformed data can hurt ML model accuracy (removes randomness)
Challenges for ML data processing service

1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

Scaling & caching are difficult optimization decisions for users.
Opportunity for ML data processing service

1. How to efficiently autoscale resources for input data processing?
2. How/when to efficiently cache and re-use (transformed) datasets?

Scaling & caching are difficult optimization decisions for users. → Need a data processing service that automates these decisions.

https://github.com/eth-easl/cachew
Autocaching policy

How to decide whether to read/write a dataset in faster, more $ storage?
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.map(parse).filter(filter_func).map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset("../my.tfrecord")
dataset = dataset.map(parse).filter(filter_func).map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()  
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)
import tensorflow as tf

def preprocess(record):
    ...

dataset = tf.data.TFRecordDataset(".../*.tfrecord")
dataset = dataset.autocache().map(parse).filter(filter_func).autocache().map(rand_augment)
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch()
dataset = dataset.distribute(dispatcher_IP)

model = ...
model.fit(dataset, epochs=10)

Cachew users can apply `autocache` ops to hint where it is viable (from an *ML perspective*) to cache/reuse data.

Cachew will decide which `autocache` op is an optimal dataset to cache from a *throughput perspective*. Caching will only be applied at 1 location, if at all.
Autocaching policy

- During first epoch, at each autocache op, infer compute vs. cache read throughput:

  - Cachew selects the autocache op with max throughput (i.e. min TotalCacheExecTime)
  - Compare with the throughput of pure compute (TotalComputeTime)
  - Select option with highest throughput → at most one autocache selected
Autoscaling policy

How to decide how many workers to allocate for a job?
Autoscaling policy

• **Intuition**: scale up data workers until no additional benefit to end-to-end training time.

• How to estimate end-to-end training time as we scale workers?
  • Leverage the iterative nature of ML training: monitor batch time
Autoscaling policy

• **Intuition**: scale up data workers until no additional benefit to end-to-end training time.

**Batch time**

- **Fetch batch from local buffer**
  - If buffer has data → approx. 0 wait time
  - If buffer empty → wait for batch from Cachew

- **Model training step on batch**
  - Depends on the model & HW accelerator → constant
Autoscaling policy

- **Intuition**: scale up data workers until no additional benefit to end-to-end training time.

**Batch time**

- Fetch batch from local buffer
- Model training step on batch
- Fetch
- Model training step on batch
- Fetch
- Model training step on batch
- Model training step on batch
- Model training step on batch

# workers = 1
# workers = 2
# workers = 3
# workers = 4
# workers = 5
Autoscaling policy

- **Intuition**: scale up data workers until no additional benefit to end-to-end training time.

<table>
<thead>
<tr>
<th>Batch time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch batch from local buffer</td>
</tr>
<tr>
<td>Fetch</td>
</tr>
<tr>
<td>Fetch</td>
</tr>
<tr>
<td>Model training step on batch</td>
</tr>
</tbody>
</table>

- \# workers = 1
- \# workers = 2
- \# workers = 3
- \# workers = 4

Remove worker
Autoscaling policy

- **Intuition**: scale up data workers until no additional benefit to end-to-end training time.

Batch time

- Fetch batch from local buffer
- Fetch batch
- Fetch
- Model training step on batch
- Model training step on batch
- Model training step on batch
- Model training step on batch

Converged:
# workers = 4
Cachew autoscaling & caching for multiple tenants

Converge to 3 workers
Cachew autoscaling & caching for multiple tenants

![Graph showing worker count over elapsed time with Compute and Put in Cache phases and converging to 3 workers](image-url)
Cachew autoscaling & caching for multiple tenants

![Graph showing the comparison of 'Compute', 'Put in Cache', and 'Read from Cache' with worker count over elapsed time [s].]
Cachew autoscaling & caching for multiple tenants
Cachew autoscaling & caching for multiple tenants

Compute | Put in Cache | Read from Cache

Worker count | Elapsed time [s]

Second job reads from cache directly
Cachew autoscaling & caching for multiple tenants

Converge to 4 workers; job2 client can ingest data 2x faster than job1’s client
Future directions for ML data services

How to leverage knowledge across jobs to improve data and model quality?

• Training data discovery service
  • Recommend “relevant” source datasets used by other jobs

• Data auto-augmentation service
  • Recommend data augmentations

• Data importance service
  • Recommend training examples that are most relevant for the task at hand
ML with *dynamic* input datasets

- Many practical ML use-cases involve training on dynamic data:
  - New data streaming in, some older data needs to be deleted
  - Model needs to adapt; learn from new data + recall “important” old data

- Need system support for:
  - Efficiently mixing new (streaming) & old (stored) data
  - Data importance aware data storage/caching & training
  - Data drift aware model retraining and deployment strategies

- To stimulate research in this area, we are building an open-source benchmark and system architecture for ML training on dynamic datasets.
  - early stage, collaborators welcome!
Thanks to great collaborators 😊

Dan Graur  Damien Aymon  Chandu Thekkath  Jiří Šimša  Derek Murray

Dan Kluser  Andrew Audibert  Michael Kuchnik  George Amvrosiadis  Virginia Smith
ML has a cost problem

...to train a 100 Trillion parameter model for 1 day on the cloud?
A. $4,000
B. $40,000
C. $400,000

Disaggregating data processing can eliminate data stalls

→ Up to 110x speedup, 89x cost reduction on production model

Input data processing is often a bottleneck, leaving expensive GPUs/TPUs idle

Disaggregating data processing can eliminate data stalls

→ Up to 110x speedup, 89x cost reduction on production model

Cachew: multi-tenant ML data processing service → autoscale & autocache

https://github.com/eth-easl/cachew