tf.data service: A Case for Disaggregating ML Input Data Processing

Andrew Audibert
Google

Yang Chen
Google

Dan Graur
ETH Zürich

Ana Klimovic
ETH Zürich

Jiří Šimša
Google

Chandramohan A. Thekkath
Google

arXiv:2210.14826v3 [cs.LG] 2 Jan 2024

ABSTRACT

Machine learning (ML) computations commonly execute on expensive specialized hardware, such as GPUs and TPUs, which provide high FLOPs and performance-per-watt. For cost efficiency, it is essential to keep these accelerators highly utilized. This requires preprocessing input data at the rate at which the accelerators can ingest and perform ML computations on the data. To avoid data stalls, the host CPU and RAM required for input data processing per accelerator core used for ML computations varies across jobs. Hence, the traditional approach of processing input data on ML accelerator hosts with a fixed hardware ratio leads to either under-utilizing the accelerators or the host CPU and RAM. In this paper, we address these concerns by building a disaggregated ML data processing system.

We present *tf.data service*, an open-source disaggregated input data processing service built on top of tf.data in TensorFlow. We show that disaggregating data preprocessing has three key advantages for large-scale ML training jobs. First, the service can horizontally scale-out to right-size CPU/RAM host resources for data processing in each job, saving 32× training time and 26× cost, on average. Second, the service can share ephemeral preprocessed data results across jobs, to optimize CPU usage and reduce redundant computations. Finally, the service supports coordinated reads, a technique that avoids stragglers due to different input sizes in distributed training, reducing training time by 2.2×, on average. Our design is inspired by lessons learned from deploying tf.data service in production, including relaxing data visitation guarantees without impacting model accuracy.

1 INTRODUCTION

The ubiquity of machine learning (ML) has led to widespread deployment of specialized hardware to accelerate ML computations (e.g., GPUs and TPUs). While vastly improving the performance per watt of ML training and inference [39, 40], hardware accelerators are significantly more expensive than traditional CPU servers [4, 23]. Hence, achieving energy and cost efficiency benefits requires keeping hardware accelerators highly utilized.

Operating ML accelerators at high utilization requires feeding fresh batches of preprocessed data at the rate at which ML computations executing on accelerators request new data. Data preprocessing typically executes on CPU host resources of ML servers, as data transformations consist of user-defined functions [56]. This makes it right-sizing host CPU/RAM (for data preprocessing) to ML accelerator cores (for ML model computations) imperative, in order to avoid data preprocessing stalls and maximize resource efficiency.

However, ML jobs require diverse ratios of host CPU/RAM and ML accelerator resources [30, 55]. Figure 1 shows the distributions of host CPU and memory resource requirements for data preprocessing across ML jobs at Google, normalized to the peak usage for each resource. As the distributions are heavy-tailed, picking a particular CPU and memory configuration for ML accelerator hosts satisfy the requirements of a limited set of jobs at a particular point p on the x-axes of the CDFs. This would leave many jobs on the left of p underutilizing CPU/RAM and many jobs on the right of p with insufficient CPU/RAM for data preprocessing, idling ML accelerators. Hence, provisioning ML jobs with a one-size-fits-all ratio of host CPU/RAM and ML accelerator resources is inefficient for most jobs. Meanwhile, datacenter providers
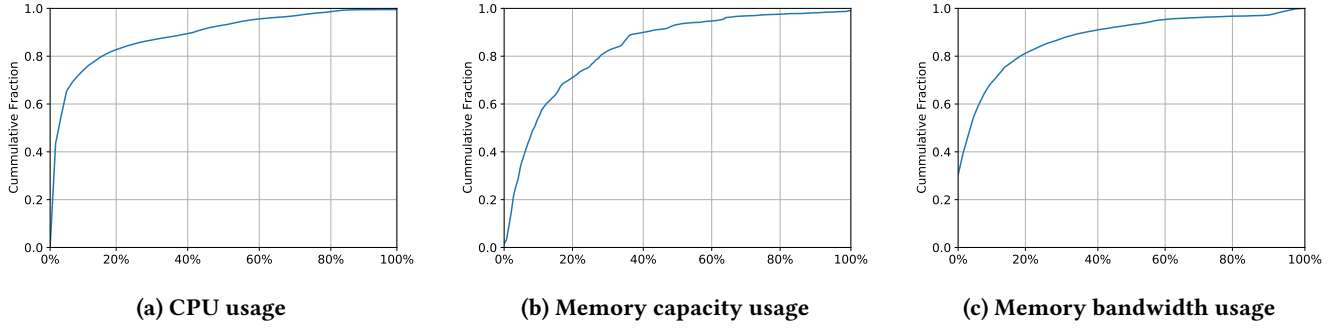
SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

(a) CPU usage   (b) Memory capacity usage   (c) Memory bandwidth usage

Figure 1: CDFs of normalized ML host resource usage for over 73k colocated processing jobs running in Google datacenters over a 24 hour period. The takeaway is that host resource requirements vary widely across ML jobs.
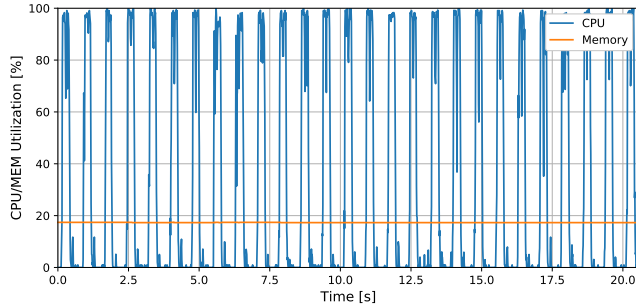
Figure 2: RetinaNet [48] CPU/MEM usage when training on COCO [49] using a GCP TPU v2-8 VM [24].

must typically limit server heterogeneity to simplify resource management and server maintenance at scale [42].

As today's ML accelerator servers are typically provisioned with generous amounts of CPU and RAM [5, 24], one could improve resource utilization by loaning spare CPU/RAM on ML accelerator hosts to other CPU workloads. However, colocating workloads with ML data preprocessing is challenging. Figure 2 shows that data preprocessing has highly bursty CPU usage as batches of data are processed and loaded to the accelerator. Even if the average CPU utilization on many ML accelerator hosts may be low, the bursty nature of ML data preprocessing makes it difficult to share host resources with other jobs without significant interference.

An alternative approach is to *disaggregate* resources for data preprocessing and ML computations. Disaggregation enables allocating CPU/RAM and ML accelerators independently for each job to meet its unique requirements. Just as disaggregating storage from compute is known to improve resource efficiency in datacenters [17, 42, 60], disaggregating ML accelerators from CPU/RAM enables right-sizing resources for ML workloads. For example, Meta has reported that their internal closed-source ML data preprocessing system, DPP [77], relies on scaling out data preprocessing to disaggregated CPU hosts to meet the data ingestion demands of large-scale recommender system model training [51]. However, disaggregating ML data processing does not come for

free: it requires recruiting potentially many remote nodes for data processing, dealing with failures of these distributed nodes, and sending large volumes of preprocessed data over the network. There is a need for a data processing platform that optimizes these tradeoffs to support resource-efficient ML input data preprocessing.

We present the design and implementation of the first open-source disaggregated data processing framework, *tf.data service*, available through the Tensorflow GitHub project [68]. We evaluate tf.data service on a variety of production ML workloads at Google from vision and natural language processing (NLP) domains. We show that the primary advantage of disaggregation is the ability to *horizontally scale-out* data processing to remote workers to remove data stalls, which improves training throughput by 31.7× and reduces cost by 26.2×, on average, for input-bound jobs. The cost savings of maximizing accelerator utilization and reducing training time significantly outweigh the cost of using extra CPU hosts for data processing.

We also show that the disaggregated architecture of tf.data service has two additional advantages. First, enabling ML jobs to fetch preprocessed data from remote hosts enables sharing intermediate preprocessing results between a set of jobs that execute in overlapping time windows. This *ephemeral data sharing* feature saves CPU cycles by avoiding redundant computations. Second, the disaggregated system architecture of tf.data service enables coordinating data ingestion across clients in distributed ML training jobs to reduce stragglers and improve training time and cost. For example, NLP models typically ingest data with a wide variety of input sizes, which can lead to stragglers among training clients. The *coordinated reads* feature in tf.data service allows all ML accelerator clients in a job to fetch data from remote workers that prepare batches of data with similar size per training iteration, resulting in up to 2× overall speedup due to less data padding and synchronization overheads.

Contrary to the conventional wisdom of enforcing strict data ordering guarantees and ensuring that each example is

visited exactly once per training epoch [21, 54, 55], we design the data sharing and coordinated reads features of tf.data service with more relaxed data visitation guarantees. We find that relaxing data visitation to at-most-once (instead of exactly-once) guarantees generally has negligible impact on model accuracy, as models are typically trained with larger datasets than in academic settings. Relaxing data visitation guarantees simplifies the design of ephemeral data sharing, coordinated reads, and fault tolerance mechanisms (e.g., compared to other ML data caching systems [30, 45, 55]).

We present the first open-source disaggregated data processing framework for ML workloads and show it has three key advantages for large-scale ML training jobs: 1) horizontal scale-out to right-size CPU/RAM host resources for data processing, 2) ephemeral data sharing to avoid redundant preprocessing among fully or partially concurrent jobs, and 3) coordinated reads to avoid stragglers in distributed training that can arise due to differences in input data sizes. Our design is inspired by lessons from deploying tf.data service in production, including the ability to relax data visitation guarantees without impacting model accuracy. Since launching tf.data service open source in 2020, the system has already been used as the foundation for several research projects, including data echoing [13], Cachew [30], and FastFlow [71].

2 BACKGROUND AND RELATED WORK

ML input data processing. Input data processing involves an "extract transform load" (ETL) pipeline. Jobs read source data from storage, process data on-the-fly, and load batches of transformed data to ML accelerators for model training or inference. Common data transformations include decompressing data, parsing file formats, extracting features, and batching elements. It is also common to add randomness to input data (e.g., randomly sample, augment, and shuffle elements) to improve model generalization [15, 16, 18, 64].

While model training or inference typically executes on expensive, specialized hardware, user-defined input data transformations execute on general purpose CPUs. Their relative cost difference makes it particularly important to ensure that accelerators operate at high utilization [4, 23]. If the ML computation does not have to wait for data, the job is considered *model-bound*. Otherwise, the job is considered *input-bound*. Input-bound jobs are more problematic as they leave valuable ML hardware underutilized, however model-bound jobs can also be costly by leaving CPUs underutilized. Removing input bottlenecks is critical as input-bound jobs hog valuable hardware for extended periods of time, incurring significant costs and significant delays for the job itself as well as other jobs that are waiting for the hardware resources to free up.

Data processing frameworks. Generic data processing frameworks, such as Beam [1], Flume [2] and Spark [76] are often used for *offline* ML data processing (i.e. processing that takes place prior to any ML compute and handles data cleaning, feature engineering, normalization, etc.). These generic frameworks are not suitable for *online* ML data processing (i.e. on-the-fly data processing during a training job that handles data augmentation, shuffling, batching, etc.), primarily due to the overhead they impose. For instance, Spark Streaming recommends batching work at 50ms granularity [65] while ML training jobs often have step times below 1ms. Other issues stem from API mismatches between generic data processing frameworks and ML training frameworks, a lack of holistic optimization opportunities to improve runtime performance (as the preprocessing and ML compute frameworks are disjoint), and the difficulty of accommodating ML-specific preprocessing requirements (such as relaxed data visitation guarantees, which we describe below).

ML-specific data frameworks, used for online data preprocessing, include PyTorch DataLoader [14], NVIDIA DALI [31], and tf.data [56]. We focus on tf.data as it is widely used internally at Google and in open-source Tensorflow programs. tf.data provides an API and a runtime for creating and executing efficient ML data processing pipelines. With tf.data, users can express and execute input data pipelines using a variety of libraries that integrate with ML frameworks like PyTorch [58] and Tensorflow [3]. It provides generic operators (e.g. `map`, `filter`, `prefetch`, etc.) and an autotuning feature that automatically adjusts runtime configuration knobs (e.g. parallelism, prefetching, etc.) [22].

Colocated vs. disaggregated data processing. Traditionally, ML frameworks have *colocated* input data processing and ML computations on the same machine [14, 57, 67]. However, feeding powerful modern accelerators with data at sufficient throughput requires more CPU and memory resources than are often locally available [30, 77]. This motivates a *disaggregated* processing mode, where data processing executes on separate machines whose resources can be scaled independently from expensive ML accelerators. The ability to right-size resource allocations is critical to satisfy the wide variety of resource requirements in ML jobs (see Figure 2) and ensure that all allocated resources — both costly ML accelerators and CPU/MEM — remain highly utilized.

Data visitation guarantees. In the ML community, it is customary to train models with *exactly-once* data visitation semantics for each epoch [7, 8, 21, 54]. This means that every sample in the dataset is visited exactly once before any sample is re-visited during training. For small datasets, deviating from exactly-once guarantees can skew the data distribution, potentially leading to a less generalizable or lower accuracy model [21, 54]. In §3, we will discuss how relaxing data visitation guarantees is possible for production ML jobs, as

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

they generally train on significantly larger datasets that are continuously updated [36].

Prior work on optimizing input processing. Several approaches have been proposed to alleviate input data stalls in ML training jobs. Plumber [44] and tf.data's autotuning harness [67] dynamically tune software parallelism and memory buffer sizes to maximize performance on a given training node. However, this tuning does not scale resources beyond a single node. NVIDIA DALI supports offloading data processing to GPUs [31]. This alleviates CPU bottlenecks but may lead to GPU resource contention among input data transformation tasks and model training tasks. Other works cache and reuse (transformed) data across epochs [13, 46] or across jobs [30, 45, 55], trading storage capacity to save CPU cycles for repeated data processing. However, caching solutions are still not guaranteed to eliminate data stalls. In-memory caching solutions have capacity limitations since ML datasets often exceed the size of a training node's RAM [67]. SSD-based caching requires high parallelism to avoid I/O bottlenecks when reading large volumes of data from storage.

Meta's closed source Distributed Data Processing (DPP) service [77] proposes horizontally scaling data workers with disaggregated data processing. Zhao et. al [77] characterize the compute, memory, and network resource requirements of data processing compared to model training for recommender system models at Meta and advocated for disaggregation to scale-out data processing and avoid input bottlenecks. However, they do not quantify the performance and cost benefits of disaggregated data processing compared to colocated data processing. In contrast, we provide the first quantitative analysis of how much disaggregated data processing actually improves the overall performance and cost of production ML workloads. We also go beyond DPP's design and show the benefits of disaggregation besides horizontal scaling, such as enabling ephemeral data sharing and coordinated reads.

FastFlow [71] builds on top of tf.data service, leveraging its ability to preprocess data on both local and remote resources. FastFlow extends our disaggregation mechanism to support splitting data preprocessing between local and remote workers at specific points in an input pipeline. The system selects a pipeline split that maximizes throughput in a fixed-size tf.data service deployment, i.e., the system decides which portion of an input pipeline to process locally vs. remotely. FastFlow is complementary to our work and further shows the benefits that flexible resource allocation with disaggregation can bring to ML data processing.

3 TF.DATA SERVICE DESIGN

We present tf.data service, a system that enables disaggregating and distributing data processing for ML workloads. tf.data service integrates seamlessly with ML frameworks,

such as TensorFlow or PyTorch, and is particularly designed to meet the data processing needs of ML training jobs. We design tf.data service with three key principles in mind:

(1) Eliminate input data stalls. Eliminating data stalls is key to maximizing throughput, which is an important performance metric for ML training jobs. As we will demonstrate in §4, using extra CPU/RAM resources to alleviate data preprocessing stalls is often cost-efficient for input-bound jobs as it helps them complete faster and hence consume expensive hardware accelerators for less time.

(2) Avoid redundant data preprocessing. Input data pipelines are often repeatedly executed in ML clusters, for example in hyperparameter tuning or model search workflows [41, 56]. Since data preprocessing can be compute, memory, and power-intensive [56, 77], reducing redundant data preprocessing (i.e., sharing intermediate results) across jobs is important for hardware and energy efficiency.

(3) Simplify the design by relaxing constraints. While guaranteeing exactly-once data visitation is customary when benchmarking ML training on academic datasets [54, 55], we find that production ML training jobs allow for more relaxed data visitation guarantees with negligible impact on accuracy. These jobs train on vast volumes of data that are continuously updated [21, 36]. Some datasets are large enough that training on all data may not be practical as the model can safely converge on a subset of the entire dataset without the risk of overfitting [21]. Hence, we find that *at-most-once* data visitation is sufficient for most jobs (i.e., each sample is visited at most once per epoch) [1]. We can leverage relaxed data ordering guarantees to simplify fault-tolerance.

With the above principles in mind, we design tf.data service as a disaggregated system that enables: eliminating input data bottlenecks by horizontally scaling out data workers (§3.1), reducing CPU usage for data processing by sharing ephemeral data transformations between jobs (§3.5), and avoiding stragglers in distributed training by coordinating reads between multiple clients (§3.6).

3.1 System Architecture

Figure 3 shows the tf.data service architecture. The core system is composed of a *dispatcher*, which manages various metadata, and a pool of *workers*, which perform data preprocessing. The dispatcher and workers are managed by an ML job *orchestrator*. *Clients* execute ML computations (e.g., model training) on servers equipped with accelerators. Distributed training jobs consist of multiple clients. Source data

---

[1] Further relaxing visitation guarantees to *zero-once-or-more* can produce a high quality model in some cases, as long as the random transformations applied to the source data to ensure a sufficiently diverse and representative distribution [13, 46].

is stored in a distributed storage system (e.g. Colossus [26]) or object storage (e.g. GCS [27]) accessible by all workers.

Life of a job. As shown in Figure 3, the ML developer first submits their ML job to the orchestrator. The orchestrator spins up the tf.data service dispatcher and workers. Clients then register their input data pipeline with the dispatcher, using the API we describe in §3.2. The dispatcher distributes a tf.data computation graph representing the input data pipeline to all available workers. This informs workers which source data to read from the storage layer and how to preprocess data into batches. For each job, each worker processes the same input data pipeline but on different partitions of the source dataset, determined by the dispatcher's sharding policy. The dispatcher informs clients about the IP addresses of workers and clients request batches from workers over the network. All communication between the clients, the dispatcher, and the workers is done via gRPC, which uses HTTP/2, and multiplexes multiple calls on a single TCP connection [28].

Dispatcher. The dispatcher manages metadata about registered workers, clients, and input pipelines. It receives heartbeats from clients and workers, assigns dataset processing tasks to workers for partitions of the input dataset using a sharding policy (see §3.3), and notifies clients of worker pool updates. It also keeps track of states of dataset processing tasks and notifies clients of completed tasks. To avoid bottlenecks, the dispatcher does not perform any data processing.

Workers. Workers execute data processing tasks defined in a tf.data computation graph. Workers read data from storage, apply requested transformations, and store the samples in a buffer. Workers respond to client requests for buffered samples. The number of workers and their CPU/RAM resources can be specified manually or tuned automatically by integrating tf.data service with an auto-scaling framework [30, 59].

Workers are logically disaggregated components, however they can be deployed on remote CPU servers or local host resources on clients. Remote workers serialize data before sending it to clients over the network, hence clients need to deserialize the data before copying it to accelerator memory. In bandwidth constrained environments, workers can also compress the data before serialization and clients can decompress the data. In deployments where bandwidth is abundant, we find that compression is not helpful as it occupies unnecessary CPU cycles for both clients and workers. In contrast to remote workers, local workers execute as part of the client's main TensorFlow process and hence can communicate directly via function calls with the client. Remote workers are useful for scaling out data preprocessing to eliminate input bottlenecks when local host resources on client machines are not sufficient. Local workers are useful for jobs with relatively low data ingestion rate requirements
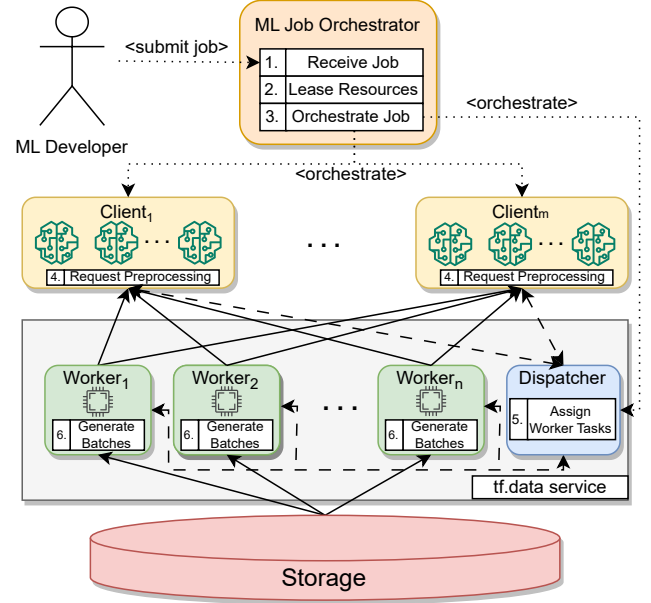
Figure 3: Architecture and workflow. Solid lines represent the data path, dashed lines represent the control path, and dotted lines the execution flow.

that run on client machines with abundant CPU and memory. For jobs that rely on many remote workers to eliminate input data bottlenecks, local CPU and memory resources on clients are consumed by deserialization, decompression, and data loading to accelerators, leaving few spare resources for preprocessing data with local workers.

Clients. Without tf.data service, each client would run its own local input pipeline for data preprocessing. With tf.data service, clients instead send gRPC requests to workers to fetch preprocessed data. To maximize data ingestion, clients can request data from multiple workers in parallel. Each client stores results in a client-side buffer and returns batches to the downstream ML computation when requested.

Orchestrator. At Google, we use Borg [70, 73] as the tf.data service orchestrator. Borg deploys clients, workers, and the dispatcher as containers on machines in the same Borg cell. In our fleet, client machines with accelerators are dedicated to the ML job whereas the tf.data service workers run on multi-tenant machines with fungible resources. Borg uses Autopilot [59] to horizontally scale the number of worker nodes based on user hints and CPU utilization. Autopilot can also vertically scale the resources allocated to each worker or dispatcher container using machine learning and other statistical techniques that leverage raw hardware utilization signals [59]. tf.data service can also be deployed with other orchestration systems, such as Kubernetes [43]. Kubernetes supports horizontal scaling [35] vertical scaling [74]. Users

```
1  ds = make_dataset()
2  ds = ds.distribute(
3      processing_mode=ShardingPolicy.OFF,
4      service_address=...)
5  for batch in ds:
6      train_step(batch)
```

Figure 4: tf.data service API example.

and researchers can also implement custom autoscaling policies for tf.data service. For example, Cachew [30] autoscales workers based on batch time processing monitoring.

3.2 API and Pipeline Optimizations

API. Figure 4 shows an example of a simple tf.data service pipeline. Here, `make_dataset()` (line 1) is a user-defined function that generates tf.data dataset object using the standard tf.data API. To indicate that a tf.data dataset should use tf.data service, we apply the `distribute` transformation to the dataset definition (lines 2-4). This serializes the upstream dataset definition and sends it to the dispatcher. The dispatcher shares the definition with the pool of workers and sends the worker IP addresses to the client. During input pipeline execution – using the `for` loop (lines 5-6) – the client sends RPCs to the workers to request data. This simple Python API abstracts communication between clients and workers. User programs simply iterate over the Python dataset object the same way they would for colocated processing with tf.data.

Via the `distribute` transformation users specify several deployment parameters, such as the compression method to use (if any) when sending data over the network, whether clients should fetch batches from local and/or remote workers (by default, they read from both), and the communication protocol between clients and workers (by default gRPC).

Pipeline Optimizations. tf.data service pipelines benefit from all optimizations available in tf.data, such as static graph optimization passes and dynamic autotuning [56]. Before the client registers an input pipeline with the dispatcher, it undergoes a number of optimization stages that, among others, try to eliminate dead transformations, inject transparent prefetching, and fuse operators (e.g. `map-filter` fusion) to reduce software overhead. Besides the optimizations that act on the input pipeline's dataflow graph prior to execution, input pipelines also transparently benefit from automatic tuning of operator-specific parameters at runtime (e.g., the number of software threads to use), through the `AUTOTUNE` mechanism [56]. These optimizations transparently tune the level of parallelism for each operator, as well as other parameters (e.g. size of a prefetch buffer) that explicitly use the `tf.data.AUTOTUNE` token. These optimizations take place

at runtime since they are dependent on the target hardware configuration and the underlying data distribution.

3.3 Source Data Sharding

When using tf.data service, users can specify a sharding policy for the `distribute` transformation (line 3 in Figure 4). Sharding policies dictate how source data should be partitioned across workers. Source data is typically stored in several files, hence, each file constitutes a source data shard. It is also possible to change the granularity of sharding to individual samples in the source data or to sets of files [25]. tf.data service offers several sharding policies, including: (1) no sharding via the `OFF` policy (2) disjoint first-come-first-served sharding via the `DYNAMIC` policy, and (3) several static sharding policies that pre-allocate shards across workers.

With the `OFF` sharding policy the dispatcher does not shard the dataset and each worker processes the entire dataset independently. Each worker typically processes data in a different random order. This sharding policy is suitable for models that are robust to the most relaxed visitation guarantees, *zero-once-or-more*. The benefit of this approach is that it does not require dataset partitioning and coordination of partition assignments across workers.

The `DYNAMIC` sharding policy targets jobs which require some data visitation guarantees. With dynamic sharding, the dispatcher partitions the source dataset into disjoint shards. The workers request shards from the dispatcher whenever they run out of data to process. Workers process one shard at a time. Since the shards are completely disjoint, each source data element contributes to at most one batch which is seen by at most one client. By using a larger number of shards than the number of workers, the dispatcher can load balance across the workers. If workers are preempted or fail, dynamic sharding provides *at-most-once* visitation guarantees, as distributed shards are not recovered until the following epoch (see §3.4). In the absence of preemptions and failures, dynamic sharding provides *exactly-once* visitation guarantees.

tf.data service also supports static sharding strategies, which assign source data partitions to the available workers upfront at the beginning of a job. Users are also free to implement their own custom sharding policies.

3.4 Fault tolerance

Disaggregation allows tf.data service to horizontally scale data processing across multiple machines, which introduces a new failure domain. The likelihood of a data worker failing increases as we distribute data processing to more workers. We use a simple recovery mechanism for workers, enabled by relaxed visitation guarantees tolerated by production ML jobs. We also discuss dispatcher fault tolerance.
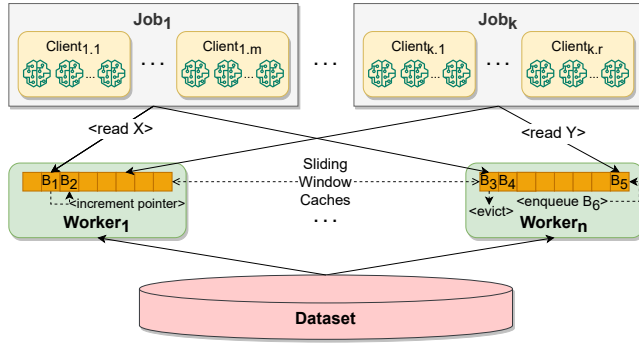
Figure 5: Ephemeral data sharing workers serve requests from different jobs via sliding window caches.

Worker fault tolerance. To simplify fault tolerance, tf.data service workers are designed to be stateless. Hence, a restarted worker will register with the dispatcher and retrieve the dataset definition it needs to process, like any new worker would. Worker failure recovery depends on the sharding policy. With no sharding, restarted workers re-process the entire dataset in a newly randomized order. In the dynamic sharding case, the restarted worker queries the dispatcher for the next split to process. The dispatcher assigns each split to exactly one worker. If the worker fails when it is processing a split, the remaining data in that split is lost. Hence with dynamic sharding, tf.data service guarantees samples are seen at-most-once. We opted for this design due to its simplicity and performance. Exactly-once visitation guarantees can be implemented if desired. This would require the dispatcher to log the distribution of shards and the workers to persist how much of the shard has been processed (at operator granularity). The same or a new worker can then be restarted with the same shards and the computation can be resumed with the same state of the original worker.

Dispatcher fault tolerance. The dispatcher writes its state changes to a write-ahead journal, including registered datasets, active workers, and active clients. On restart, the dispatcher replays the journal to restore its previous state. When down, the dispatcher cannot register new jobs, and, in case any sharding is used, cannot distribute new shards to workers. Workers will continue to produce batches for their active jobs until their data shards are fully consumed, or if no sharding is used, workers will process the entire dataset. Clients will continue to train on any incoming batches. If the dispatcher downtime is short, active jobs will remain unaffected.

3.5 Ephemeral Data Sharing

Disaggregating data preprocessing enables sharing intermediate preprocessing results between multiple training clients.

Sharing is useful as input data pipelines are frequently re-executed across jobs, for example in hyperparameter tuning or model search training workflows [41]. For example, a previous study at Google found that 10% of unique input data pipelines executed over a one month period in the fleet accounted for 77% of input pipeline executions [56].

To capitalize on the opportunity of reusing preprocessed data across jobs that execute the same input data pipeline, we implement the *ephemeral data sharing* functionality that allows concurrently executing jobs to use a shared tf.data service deployment instead of reading and preprocessing the same data in separate deployments. Figure 5 illustrates the data sharing process. Each worker stores a sliding window cache of batches it produces. The worker additionally stores a pointer in this cache for each job that it supplies data to. When a job requires a data batch from a worker, the worker returns the batch in the cache at the location pointed to by the job's pointer, then increments the pointer. For example, in Figure 5, when Job_1's client issues read X, it fetches batch B_1 from $Worker_1$'s sliding window cache. After the read, $Worker_1$ increments Job_1's pointer, moving it to batch B_2.

Jobs whose pointers are at the front of the cache dictate the production of new data and eviction of old data. In this case, the cache functions like a queue. When such jobs ask for a batch, they implicitly cause the worker to compute a new batch that is added to the front of the cache after the read is complete. To limit the RAM used for the cache, the batch at the back of the cache is evicted. Slower jobs pointing to the back of the cache will not be able to see the discarded batch. After the eviction, their pointers remain unchanged and implicitly point to the end of the queue, while all other pointers are decremented in order to continue pointing to the correct batches. For instance, in Figure 5, when Job_k's read Y operations fetches batch B_5, this causes $Worker_n$ to compute and enqueue a new batch B_6, and push every other batch back. Due to the limited cache space, B_3 is evicted and B_4 is pushed to the end of $Worker_n$'s sliding window cache. Job_1 will not get to train on B_3, as it is permanently evicted, and will implicitly point to B_4.

In extreme cases, one job can severely lag behind another, thus facing high eviction rates. This could degrade accuracy for models trained on very small datasets, due to overfitting. In practice, we have not seen this issue as datasets are rarely that small; many production datasets are even too large to fully visit during training, thus comfortably accommodating evictions. Moreover, random augmentations and shuffling, which are frequently present in input pipelines, artificially increase the diversity and size of the dataset, making such scenarios even more rare [19, 46].

To analyze the benefits of data sharing, let us assume that the data processing cost for each job in a hyperparameter tuning experiment using the same input data pipeline is C. If

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

all jobs run at the same speed, then the data processing cost with sharing is C. Without sharing, the data processing cost would be $k \times C$. If the jobs run at different speeds, slower jobs may miss some preprocessed batches due to cache evictions needed to accommodate faster jobs. In the worst case, the jobs run sequentially and each job only shares the final window populated by the previous job (the preprocessing for the rest of the batches needs to be re-executed since it is no longer available in the buffer). In this case, the processing cost with sharing is $k \times C - (k-1) \times \frac{\text{cache size}}{\text{dataset size}} \times C$ (assuming constant processing cost per element).

Ephemeral data sharing differs from related work on ML data caching in several ways. First, as we relax data visitation guarantees, a large number of concurrently running jobs can share a single service deployment without stalling for slower jobs. In contrast, with exactly-once visitation guarantees, the fastest jobs would need to wait for the slowest jobs. Second, our data sharing algorithm supports partially-concurrent, asynchronous jobs with different ML models attached (i.e., ephemeral data sharing supports jobs whose execution overlaps only partially and no synchronization is required between jobs). This contrasts with the caching approach proposed in CoorDL [55], which is designed for fully-concurrent, synchronous, hyperparameter tuning jobs with exactly-once visitation guarantees. Third, unlike Cachew [30], which caches source datasets and preprocessed datasets on SSD storage, tf.data service caches data in memory with a shorter time to live. This improves cache performance, but requires that job execution overlaps at least partially.

3.6 Coordinated Reads

Distributing ML model training across multiple clients is increasingly common as model and dataset sizes continue to scale [47, 72]. Clients in a distributed ML training job usually synchronize model parameter updates, to ensure convergence [37]. Synchronization between training clients introduces another type of potential bottleneck that can leave expensive ML accelerators idle: waiting for stragglers. While stragglers can occur for a variety of reasons in distributed ML jobs, here we focus on stragglers that occur due to uneven input data sizes across clients. We show how we can leverage disaggregated data preprocessing in tf.data service to coordinate reads in a way that evens out input data sizes across clients to avoid stragglers in distributed ML training.

Some ML models (particularly in NLP), train on variable-sized source data and produce variable-sized preprocessed samples. A common approach is to pad the variable-sized samples to a fixed size, thus producing homogeneously shaped batches that fit the expected input shapes of the ML model. tf.data and TensorFlow offer a more efficient approach than padding: (1) tf.data can produce variable sized batches by

(a) No coordinated reads   **(b) With coordinated reads**
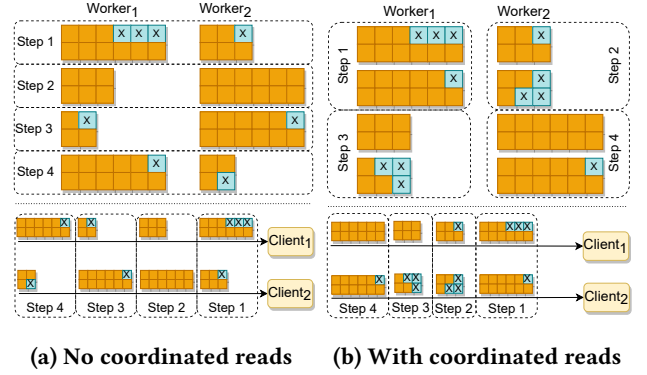
Figure 6: Batch distribution to two clients with and without coordinated reads. The deployment has two workers generating batches. In (a) the batches used in a step are generated by two workers every time while in (b) workers take turns to supply batches for each step. Each batch consists of two samples. Each ■ represents a scalar value in a sample. Batch samples are padded (⊠ blocks) to have a length equal to the maximum length sample in the batch. In (b), samples are grouped into buckets based on their unpadded length. In each step, the designated worker sends batches from the same bucket to the clients.

padding to the longest sample in each batch (as opposed to a batch-agnostic static size) and (2) TensorFlow can dynamically kernelize its ML operations for the variable input tensor shapes it sees during training. This feature improves training time by not relying on fixed-size padding, however it is susceptible to synchronization overheads in distributed synchronous training when, within the same training step, different clients receive batches of different sizes and hence take different times to train on the batches.

In such settings, synchronous training requires careful coordination of input data batches to avoid stragglers. Our goal is to ensure that batches in each training step are roughly the same size and thus incur similar training times. Consider the example of training an NLP model on two clients (Client_1 and Client_2) in Figure 6 with two preprocessing workers, (Worker_1 and Worker_2). As there are two clients, each training step requires two batches. Without coordinating reads, each training step, clients receive differently sized batches in the same training step. In contrast, with coordinated reads, each worker is tasked with generating m similarly sized batches (where m is the number of clients; $m = 2$ in Figure 6b). In the example, we configure coordinated reads to group samples of length $(0, 3]$ into one bucket, and of length $(3, 6]$ into another. Workers generate batches for each bucket, padding samples to the length of the longest sample in each batch. At each training step, only one of the

```
1  ds = ds.bucket_by_sequence_length(
2      bucket_boundaries=[128, 256])
3  ds = ds.group_by_window(window_size=
       num_consumers)
4  ds = ds.flat_map(lambda x: x)
5  ds = ds.distribute(
6      job_name="coordinated_reads_job",
7      num_consumers=num_consumers,
8      consumer_index=i, ...)
```

Figure 7: Dynamic-sequence-length training with tf.data service and coordinated read.

workers supplies batches to all clients, from the same bucket. Workers take turns in round-robin fashion to supply batches for a training step. Hence, each worker supplies m batches every n steps (where n is the number of workers). This scales well as each worker has $n-1$ steps to prepare m batches. To avoid encountering network bottlenecks, workers can send batches to clients ahead of time in predetermined round-robin client-side buffer slots. Overall, the coordinated reads feature ensures that clients get similarly sized batches with minimal padding, thus spending similar amounts of time on ML computation. As we show in §4.4, this reduces synchronization overheads and improves training time and cost.

Disaggregation is essential for enabling coordinated reads as it allows clients to communicate with and selectively read from a different worker each training step. Disaggregation also enables the service to scale out when workers are not able to keep up with the data ingestion rate of clients. To ensure high scalability to a large number of workers and maintain simple fault tolerance mechanisms, we design the service such that coordination only takes place within workers, not across them. This avoids additional dependencies and communication between workers.

Figure 7 shows how users can configure coordinated reads in the tf.data service API. Users define the desired bucket boundaries and call `bucket_by_sequence_length` (lines 1-2) to bucketize the training data by sequence length. Here the bucket boundaries are $(0, 128]$, $(128, 256]$, and $(256, \infty)$. The `group_by_window` call (line 3) will group batches from the same bucket into windows of `num_consumers` length. Clients will receive data from the same window, hence with batches of similar sequence length. The tf.data service documentation provides more details on the API [25].

4 EVALUATION

In this section we quantify the performance and cost benefits that tf.data service achieves for input-bound ML jobs by horizontally scaling out workers to eliminate data stalls. We also evaluate ephemeral data sharing and coordinated reads, which bring benefits even for model-bound ML jobs.

4.1 Methodology

Workloads. We evaluate tf.data service service on eight production models, which we refer to as $M_1, M_2, ..., M_8$. Models M_1 to M_4 are heavy users of tf.data service internally from the computer vision domain, models M_5 to M_8 are from the NLP domain. While these models are not publicly available, the logic of the input pipelines uses standard preprocessing operations for vision and NLP domains [62, 63], similar to the open-source input pipelines available in TF Model Garden [69]. Models M_1, M_2 and M_3 are input-bound with colocated data preprocessing on our machines (and hence can benefit from horizontal scale-out), whereas the other models are not input-bound. We also include a canonical open-source model in our evaluation: ResNet50 [33] trained on the ImageNet [18] dataset with its default input pipeline augmented with AutoAugment [15]. We use the open-source implementations offered in TF Model Garden for both models and input pipelines [69].

We use M_1, M_2, M_3, and the open-source ResNet50 model to evaluate benefits of horizontal scale-out, as these are examples of models that are input-bound with colocated data processing on our hardware and can benefit from this feature of tf.data service. We use M_4 to evaluate ephemeral data sharing, as it is a model often trained internally with the same input data pipeline for hyperparameter tuning and hence benefits from the ephemeral data sharing feature. We evaluate coordinated reads on the NLP models (M_5 to M_8) as they are not input-bound, but suffer from stalls in distributed training due to variations in input data sizes that lead to stragglers, which is what the coordinated reads feature is designed to alleviate. While we demonstrate the benefits of tf.data service features on particular categories of models, benefits can extend beyond the types of models we sample in this study.

Hardware. For experiments with production models, each client machine is equipped with TPU v4 accelerators [38], two 240 core AMD EPYC 7B12 processors, 32GB of accelerator memory, 400 GB of DRAM memory and 2TB of SSD storage [53], same as for Google's MLPerf v2 benchmark submissions [53]. The hardware configuration of workers and the dispatcher can vary for each model, as they are deployed on multi-tenant machines in the fleet, however they resemble general-purpose virtual machines available in public cloud-providers [5, 23]. Source data is stored in and read from Colossus, Google's internal distributed file system [26].

For externally reproducible experiments with open-source workloads, we use TPU v2-8 VMs from Google Cloud, equipped with 96 vCPUs and 335 GB or RAM [24]. Source data is stored in and read from GCS [27]. The dispatcher and workers run on `n2-standard-8` VMs. Unless specified otherwise, storage, preprocessing and ML compute are all colocated in the

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

same geographical region. Additionally, unless specified otherwise, neither the storage layer nor the network bandwidth are a bottleneck.

Software Frameworks. We use Tensorflow [3] to define and execute ML training computations, though tf.data service can also be used with other frameworks like JAX [9] and PyTorch [58]. We deploy experiments with production workloads using Borg [70, 73], Google's internal cluster management system, which autoscales resources with Autopilot [59]. For open-source experiments, we orchestrate the tf.data service deployment with Kubernetes [43] instead of Borg and manually tune the number of workers. We disable compression between workers and clients as it would require unnecessary CPU cycles and bandwidth is not a bottleneck.

Baselines. For the colocated baseline, we measure training throughput (in batches per second) using tf.data (without service) for input data preprocessing. tf.data provides state-of-the-art colocated data preprocessing performance [67].

We also show an *ideal* training throughput baseline for input-bound jobs. We obtain the ideal throughput by training the model with a `take(1).cache().repeat()` operation at the end of the input pipeline. This command retrieves the first element produced by the input pipeline, caches it, and returns it without any recomputation whenever the model requires a new batch. Hence, this baseline represents training time with an infinitely fast input pipeline. To avoid any potential network overheads, we measure ideal training time using tf.data without service.

We cannot compare to Meta's disaggregated data preprocessing service, DPP [77], as it is a closed-source proprietary system. To the best of our knowledge, while DPP does disaggregate and scale out data preprocessing, it does not offer features such as ephemeral data sharing or coordinated reads.

Metrics. We measure training throughput in batches per second as a sum across all accelerators and report speedups for each workload with versus without tf.data service. We quantify tf.data service's impact on cost using the cost model described below. To ensure disaggregation does not affect model quality, despite relaxed visitation guarantees, we compared the loss and accuracy of the trained models. We verified that the models achieved approximately the same loss and accuracy with and without tf.data service.

Cost Model. We compute the cost C of an ML job using the following formula:

$$
\begin{aligned}
C = t(&C_{CPU}(n_W \cdot \overline{CPU}_u^W + n_T \cdot CPU_a^T) \\
&+ C_{MEM}(n_W \cdot \overline{MEM}_u^W + n_T \cdot MEM_a^T) \\
&+ C_{ACC} \cdot n_T \cdot n_{ACC/T})
\end{aligned}
\tag{1}
$$

where t represents the job execution time, n_W, n_T and $n_{ACC/T}$ represent the number of tf.data service workers, clients and

accelerators per client respectively. C_{CPU}, C_{MEM} and C_{ACC} represent the normalized cost of one "unit" of CPU, memory and accelerator per unit of time respectively. $\overline{CPU}_u^W$ and $\overline{MEM}_u^W$ represent the average CPU and memory utilization across all tf.data service workers per unit of time. Finally, CPU_a^T and MEM_a^T represent the available CPU and memory for a client. We use CPU and memory utilization (as opposed to reservation) for tf.data service workers, as reserved but unused CPU and memory resources are released back to the global resource pool. In contrast, the CPU and memory of ML hosts are non-fungible, regardless of their true utilization, and are thus charged altogether. For experiments in the production cluster, we do not disclose the absolute or relative values of C_{CPU}, C_{MEM}, and C_{ACC} but they are consistent with the pricing of cloud VMs [4, 23]. For open-source experiments, we use the prices of Google Cloud VMs in the month of June 2023 for the region `us-central1`: 4.5$/h for a TPU v2-8 VM, and 0.08$/h for a `n2-standard-8` VM. The formula allows us compare the cost of different data preprocessing setups in our experiments.

We do not explicitly account for network bandwidth usage in the cost model. tf.data service is strongly recommended to run in the same datacenter as the ML accelerator clients, since intra-datacenter network bandwidth is cheap and abundant. For example, on Google Cloud, intra-zone network bandwidth is free, meaning it is already included in VM pricing [29]. Other cloud providers have similar policies.

4.2 Horizontal Scale-Out

To demonstrate the benefits of tf.data service for input-bound ML jobs, we compare training throughput and cost for models M_1, M_2, M_3, and ResNet50 with and without tf.data service. We use 32, 8, 16, and 1 accelerators to train the models, respectively. When trained without tf.data service, models M_1 and M_2 and ResNet50 fully utilize locally available CPU resources for data ingestion and processing, while M_3 makes partial use of the locally available CPU resources. We present this mix of models to demonstrate that tf.data service can address both hardware and software input bottlenecks.

Speedup. Figure 8a shows training throughput speedup with horizontal scale-out enabled by disaggregation in tf.data service. We observe speedups of 11.7× (from 0.55 batches/s to 6.47 batches/s) for M_1, 110.3× (from 4.7 batches/s to 518.4 batches/s) for M_2, 2.9× (from 22.2 batches/s to 63.8 batches/s) for M_3, and 2.57× (from 1.75 batches/s to 4.5 batches/s) for ResNet50. The average speedup across these jobs is **31.7×**. The service was scaled to 442, 421, 128, and 16 workers for the M_1, M_2, M_3, and ResNet50 experiments, respectively. For models M_1 M_3, and ResNet50, tf.data service was able to reach the ideal speedup, while for M_2 it fell 8% short. Upon closer inspection into model M_2's disaggregated performance, we
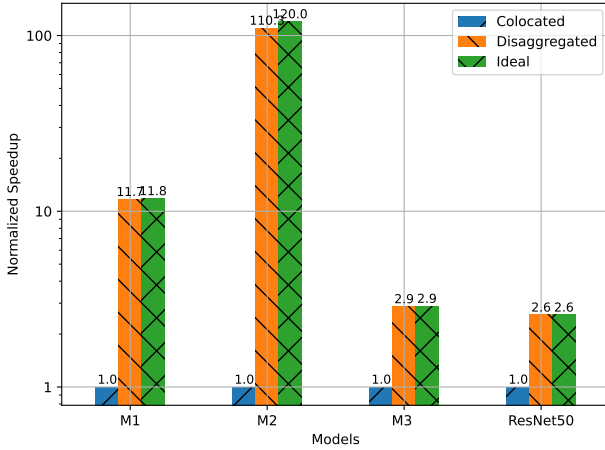
(a) Job speedups
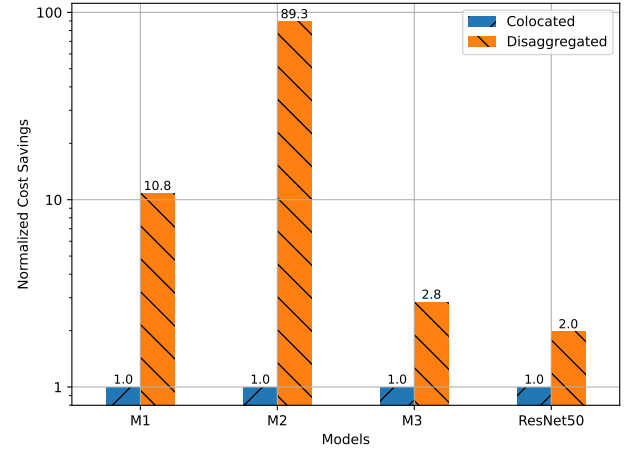
(b) Job cost savings

Figure 8: End-to-end job time speedups and cost reductions with disaggregated tf.data service.

found that the model receives data from tf.data service without delay upon request, suggesting that the penalty in performance stems from hardware limitations on the side of the trainer nodes. The remarkably high ingestion rate requirements of this model (up to 580 batches per second) in concert with the need to deserialize and copy all batches received from tf.data service introduce resource contention on the trainer nodes, causing the model to slow down. This contrasts with the ideal speedup where no deserialization and additional data copies are required.

Cost Savings. tf.data service achieved the aforementioned throughput benefits by using additional resources for data preprocessing. To better understand the end-to-end cost effectiveness of horizontal-scale out, we compare the cost of model training with and without tf.data service in Figure 8b. Notably, the cost savings are not much less than the speedups: 10.8×, 89.3×, 2.8×, and 1.97× for models M_1, M_2, M_3, and ResNet50, respectively. The average cost saving across these jobs is **26.2×**. The cost savings are a result of jobs completing faster and hence using expensive ML accelerator resources for less time. In the case of ResNet50, the total training cost for colocated training (i.e. only TPU VM) over 112320 steps at a batch size of 1024 is 80.2\$, while for disaggregated training with 16 workers and 1 dispatcher, the cost drops to 40.6\$ (9.4\$ for the service and 31.2\$ for the TPU VM).

Note that our cost equation does not capture possibly the most important cost – the opportunity cost. By allowing ML workloads to finish faster, the valuable and scarce ML hardware is released back into the general pool sooner and can be used by other ML computations. In other words, the cost savings estimated using Equation 1 are likely an underestimate of the actual economical impact of using disaggregated tf.data service to remove input bottlenecks of ML workloads.

Cross-region Scenario. To further illustrate the benefits of disaggregated tf.data service, we evaluate a scenario where the source dataset is stored in a different georgraphical region than data preprocessing and training. This situation occurs in practice, for example due to per-region resource quota availability or data regulations. Such situations can incur large performance penalties on the job time due to increased latency caused by extra network hops between storage and compute. For this experiment, we choose M_3 and store its data in a different continent than the region where data preprocessing and training take place. We measure that with "out-of-region" data, training with local tf.data preprocessing is 13.3× slower than ideal (compared to only 2.9× slower than ideal with the "in-region" scenario in Figure 8a). This is due input data fetching bottlenecks. We verify that with horizontal scaling, tf.data service is able to reach the ideal speedup even in the "out-of-region" case (not plotted in the figure), by using extra workers to hide data fetching latency.

Sweeping Worker Count. We train M_1 using a varying data preprocessing worker pool size of 8, 16, 32, 64, 128, 256, 512, and 640 workers. We choose the smallest worker pool size (8 workers) such that it has a comparable amount of CPU/RAM resources to the CPU/RAM resources available on training client nodes for colocated data preprocessing (remote workers have less resources than training clients hosts in our setup). We choose the largest worker pool size (640 workers) such that it easily eliminates the input bottleneck.

Figures 9a and 9b plot the training time speedup and job cost savings for each configuration respectively. We normalize to the training time and cost of the colocated case. The dotted horizontal line indicates the ideal job time speedup.

With 8 workers, training time and cost is 83% higher than with colocated processing time (0.3 batches/s versus 0.55
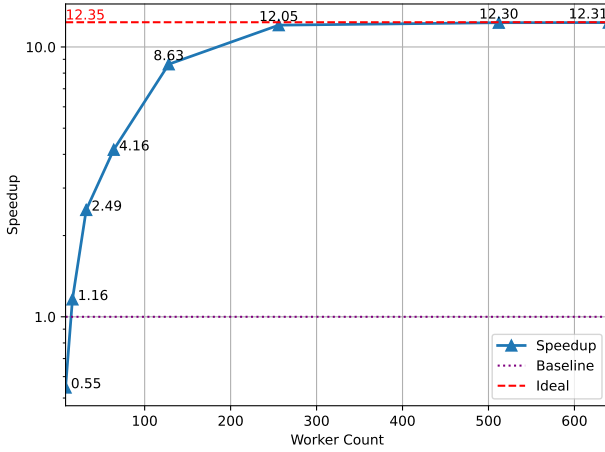
(a) Job time speedup
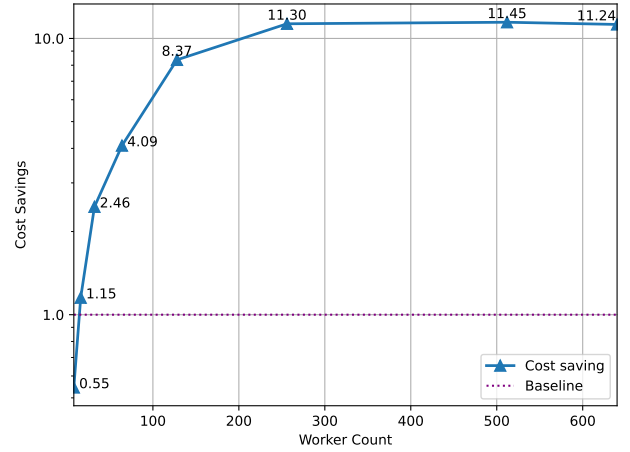
(b) Job cost savings

Figure 9: Job time speedup and cost savings for model M_1 across several tf.data service worker counts.

batches/s). This is because CPU/RAM resources are not only used for data preprocessing computations, but also for RPC processing to send and receive data over the network and auxiliary tasks such as serialization and deserialization. When we add 8 more workers, bringing the total to 16 workers, training throughput with tf.data service increases to 0.64 batches/s, which is 1.14× higher than the colocated baseline. At 64 workers, we reach a speedup of 4.1× (2.3 batches/s) and at 128 workers, we see a 8.6× speedup (4.77 batches/s) compared to the colocated case. With 512 workers, tf.data service reaches the ideal training time, achieving a training time speedup of 12.3× and corresponding cost savings of 11.4× compared to the colocated baseline. The job cost increases marginally with 640 workers, due to the superfluous 128 tf.data service workers, however the job time remains unaffected. Over-provisioning data preprocessing resources increases cost but has minimal impact on job time. Our observations are consistent for the other models (not plotted).

Network Impact. In our experiments, we deploy workers in the same region as client nodes. In such deployments, the available bandwidth is never below the client ingestion rates, hence we can achieve model-bound training time. In situations where the available network bandwidth bewteen workers and client is below the clients' ingestion rate requirements, disaggregating data preprocessing increases training time. In extreme scenarios, where the available bandwidth is below the colocated preprocessing throughput, disaggregation is strongly discouraged, as the colocated deployment would achieve better performance at lower costs.

We find that network latency is less of a concern for disaggregated data preprocessing, as we can hide high network latency by using more workers. For example, we observe in
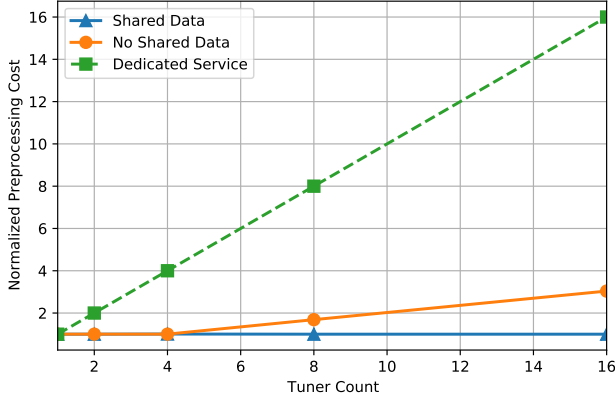
our "out-of-region" experiment that the higher the network latency between workers and clients, the higher the number of workers required to hide it. This means that higher network latency results in higher overall cost, as each remote worker contributes to cost. However, the dominant cost component in most deployments is still the TPU hardware. In extreme cases, where latency is exceptionally high, remote worker costs can outweigh the benefits of removing the input bottleneck. In these situations, the practitioner will need to decide if they care more about optimizing for training time or total cost.

Takeaway. tf.data service enables horizontally scaling out data processing to remote CPU servers, which effectively eliminates input data stalls for jobs that are input-bound with colocated data preprocessing. Even when accounting for the cost of the extra remote CPU servers in the service deployment, we observe significant overall cost savings for training jobs (26.2× on average) with disaggregated data preprocessing due to substantial speedups (31.7× on average) that come from ensuring ML accelerators do not stall for data. The speedups translate into proportional cost savings as jobs consume expensive ML accelerators for less time.

4.3 Ephemeral Data Sharing

We evaluate the performance of ephemeral data sharing on a set of hyperparameter tuning jobs across three different scenarios: (A) all hyperparameter tuning jobs share a single tf.data service deployment with data sharing enabled, (B) all hyperparameter tuning jobs share a single tf.data service deployment with no data sharing enabled, (C) each hyperparameter tuning job uses a dedicated tf.data service
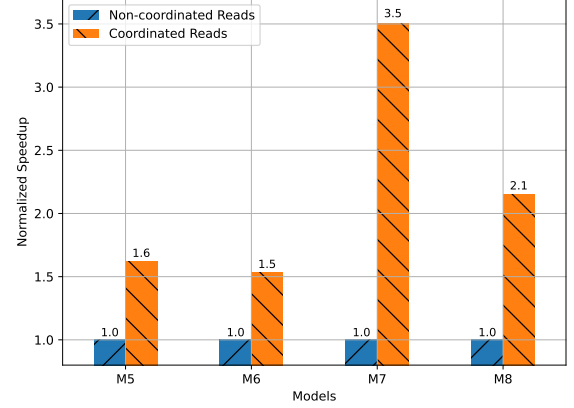
Figure 10: Preprocessing costs in various deployment modes for hyperparameter tuning jobs.

Figure 11: Job time speedup in NLP production models with dynamic sequence length support.

deployment. We evaluate data sharing using a hyperparameter tuning job, as these types of workloads most frequently employ this feature.

For modes (A) and (B), we deploy tf.data service with 128 workers, whereas for mode (C) each individual tf.data service deployment has 128 workers. We vary the number of M_4 model hyperparameter tuning jobs over the set $\{1, 2, 4, 8, 16\}$. A single tuning job for this model makes use of 16 accelerators. The model is not input-bound when there are at least 128 tf.data service workers: at this point, it reaches its ideal training throughput of 1.92 batches/s.

Our results show that mode (A) of employing a shared tf.data service incurs no performance overhead on the job time regardless of the number of tuning jobs. We test that this holds up to 64 tuning jobs (we did not test beyond this, since it is sufficient for our use case). With mode (B), which employs a single tf.data service with no data sharing, the preprocessing resources are sufficient to support up to four tuning jobs. For more concurrent jobs, the job time gradually increases by 1.75× for 8 tuning jobs (from 1.92 batches/s to 1.09 batches/s) and to 3× for 16 tuning jobs (0.64 batches/s). The longer job time maps directly to increased cost. For mode (C), which uses several dedicated tf.data service deployments, the job time is not affected, however the processing cost grows linearly with the number of tuning jobs. Figure 10 presents the normalized preprocessing cost of each of the presented modes. We normalize relative to a training run of the model without any hyperparameter tuning and with a dedicated tf.data service deployment.

Besides the job time and cost benefits, tf.data service with data sharing helps reduce storage and network bandwidth usage by keeping the number of connections to and the number of bytes read from storage layer constant with respect to the number of tuning jobs. This contrasts with the other two modes, in which the number of connections and bytes read

scales linearly in the number of tuning jobs. This feature also helps free up significant CPU and memory resources that can be used immediately by other jobs being scheduled. This reduces scheduling times and increases cluster efficiency.

Takeaway. Ephemeral data sharing in the context of concurrent jobs with identical input data pipelines (e.g. hyperparameter tuning or model search training workflows) can linearly save preprocessing resources (tested up to 64 M_4 jobs) without affecting the average end-to-end time of the jobs. This feature helps reduce network and storage bandwidth usage by avoiding redundant data transfers, and saves CPU/MEM resources, improving cluster efficiency.

4.4 Coordinated Reads

To quantify the benefits of the coordinated reads feature, we evaluate the performance of models M_5, M_6, M_7, and M_8. Since none of these models are input-bound with colocated data preprocessing, all observed performance benefits stem from the ability of the coordinated reads feature to reduce imbalance of input data batch sizes fed to clients each training step. We deploy the models on 64, 8, 64 and 4 accelerators, respectively. The maximum sequence length for each of the models is 512. We use tf.data with no service for the baseline training time. We deploy tf.data service with a similar amount of preprocessing resources as colocated data preprocessing to measure the coordinated reads training time. The service deployment has 4, 1, 4, and 1 worker for models M_5 through M_8, respectively. The bucket boundaries are defined at every multiple of 64 (i.e. $(0, 64]$, $(64, 128]$, etc.) for M_5 and M_7 and 128 for M_6 and M_8, up to the maximal sequence length.

Figure 11 shows the training time speedups. With tf.data service coordinated reads, we observe speedups ranging from 1.5× to 3.5×, with an average of 2.2× across the four jobs. More specifically the jobs performance improves by: (M_5)

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

1.62× (3.18 batches/s to 5.15 batches/s) (M_6) 1.53× (11.9 batches/s to 18.3 batches/s) (M_7) 3.5× (2 batches/s to 7 batches/s) (M_8) 2.15× (5.9 batches/s to 12.7 batches/s). These speedups correspond to equivalent job cost savings.

Takeaway. Coordinated reads helps ensure that all batches, at every training step, have similar size and employ minimal padding, leading to uniform processing time across the training clients at every step. Coordinated reads help synchronization overheads due to stragglers, realizing speedups of 2.2× on average for the four NLP jobs in our evaluation and equivalent cost savings.

4.5 Fleetwide Usage

We conclude our evaluation with some more general insights of tf.data service usage across the Google fleet. We launched tf.data service internally in Q2 2020 and open-sourced it in Q3 2020 [66]. Its adoption has since been growing steadily both internally and externally (e.g. for Cloud TPU workloads).

Deployment sizes. Figure 12a shows the distribution of tf.data service deployment sizes (i.e. the number of data processing workers) for internal ML workloads over the last year. While most training jobs deploy between 2 and 32 workers, the largest model uses more than 5K workers. In our fleet, tf.data service workers are heterogeneous (i.e., they have different amounts of CPU and RAM resources) as Borg schedules workers on multi-tenant machines with fungible resources. The optimal number of data processing workers and their size depends on the model, as shown by the wide distribution in Figure 12a.

Scale-out CPU usage. We compare the total CPU usage with versus without tf.data service for the top 10 most CPU-intensive tf.data service jobs in the fleet. Figure 12b shows the CPU usage for jobs with disaggregated data preprocessing, normalized to their CPU usage with colocated data preprocessing. The tf.data service workers use up to 25× more CPU cores than available locally on ML hosts with colocated preprocessing. Without tf.data service, data processing would incur large input bottlenecks, slowing down the ML computation and limiting utilization of valuable ML hardware.

5 DISCUSSION

We discuss insights from the internal adoption of tf.data service and the implications ML disaggregation has on future research and preprocessing dynamics.

Target workloads. ML jobs can benefit from our work on disaggregation in three different ways: (1) eliminating input bottlenecks via horizontal-scale out (2) avoiding recomputation via ephemeral data sharing and (3) avoiding stragglers in distributed training via coordinated reads. We expect ML jobs in the image, video and audio domains to benefit the most from (1) as their input pipelines often employ complex transformations that manipulate large volumes of data, making them more likely to be input-bound. NLP jobs often have less compute intensive input pipelines and are thus less likely to benefit from (1). All types of ML workloads can benefit from (2) when their execution overlaps in time and they have the same input pipeline. For instance, this scenario often occurs for hyperparameter tuning jobs. Finally, NLP models frequently benefit from (3), as their input data has irregular sequence length, which can lead to straggler trainer nodes in synchronous distributed training. We are exploring how coordinated reads may also benefit other ML domains with variable input sizes (e.g. different image or audio resolutions). We note that jobs can benefit from a combination of the features brought forward by our work, and are not bound to one. For instance, a set of hyperparameter tuning NLP jobs could benefit from tf.data service to reduce redundant computation via ephemeral data sharing and eliminate straggler clients via coordinated reads.

Implications for ML accelerator host design. The current generation of Cloud TPU hosts provide 96 CPU cores and 335 GB of RAM per TPU accelerator, which provides significant local resources for data processing. In production, we find that approximately 20% of accelerator usage benefits from distributed data processing since it needs even more CPU and/or memory resources than what is available on TPU hosts to avoid input data processing bottlenecks. Why not build even beefier accelerator hosts to satisfy the needs of the most data processing intensive jobs and colocate other workloads on any spare CPU/memory of hosts with less data processing intensive ML jobs? As shown in Figure 2, sharing the host resources of ML accelerator nodes is challenging because CPU usage tends to be extremely bursty. CPU utilization oscillates from high when preprocessing a batch of data to low while the accelerator computes the model training step on this batch, making ML data processing a challenging workload to colocate with other jobs. Thus, to avoid performance interference, TPU hosts are dedicated machines – they are not shared between users like other machines in Google's cluster [73].

With the support for disaggregated data processing provided by tf.data service, it is worth exploring the design of ML accelerator nodes with wimpier resources to reduce costs. For example, the job in Figure 2 under-utilizes host memory. However, other jobs that will use the same server configuration may need more CPU and/or memory. Disaggregation has the potential to ensure all ML jobs are provisioned with the right CPU/MEM to accelerator ratio. Just as disaggregating data storage from computation in cloud applications has led to new hardware trends such as SmartNIC-attached
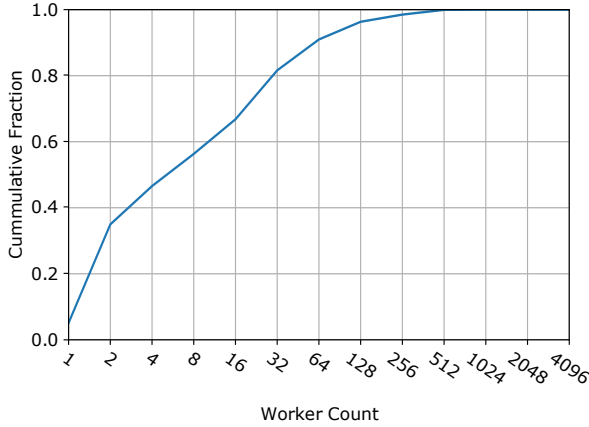
(a) CDF of worker counts in production jobs.
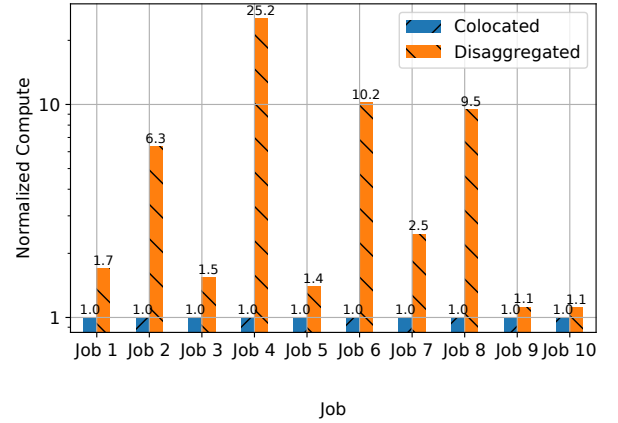
(b) Client node CPU limit and tf.data worker CPU usage.

Figure 12: tf.data service fleetwide usage.

storage device arrays [10, 52], it is interesting to explore hardware deployments that completely separate ML accelerators from the CPU and RAM resources used for traditional data processing. How to design SmartNIC-attached ML accelerator arrays and to what extent can such hardware improve the performance and cost of ML jobs remain open questions.

ML data processing as a multi-tenant service. While tf.data service is currently deployed per job or set of jobs submitted by a user, we are exploring how to run tf.data service as a long-lived, multi-tenant service that is shared across an organization. Such a deployment would provide a global view of all jobs executed within an organization, which can help optimize job performance and cluster resources efficiency. This setup goes hand in hand with novel policy development which exploits global information to remove cumbersome decisions from practitioners, helping both preprocessing performance and model quality. The advantage of features that exploit the organization's view is that they can help make globally optimal decisions, in contrast to practitioners which have a limited view of only their jobs. For example, Cachew [30] is a system built on top of tf.data service that caches common input data pipelines across jobs from different tenants to optimize training throughput. Cachew also autoscales the number of remote workers for each job to maximize throughput-per-dollar. We see further opportunities to leverage global knowledge across jobs: for dataset discovery [11, 75] and automatic input pipeline augmentations [15, 16]. Sharing preprocessing at a finer granularity within the input pipeline (similar to shared query execution [20, 32, 50]) and combining this with semantic transformations of the input pipeline (similar to semantic query optimization [12, 34, 61]) are other promising directions. Disaggregation enables a rich design space for resource and data

management policies, to optimize preprocessing throughput and hardware utilization in concert with training dynamics.

6 CONCLUSION

We presented tf.data service and showed that disaggregating ML input data preprocessing from ML training allows us to eliminate input data stalls with horizontal scale-out, achieving up to 31.7× speedups and 26.2× cost savings, on average. We also showed that disaggregation is useful beyond horizontal scaling, as it enables other features in tf.data service, such as ephemeral data sharing, which improves resource efficiency by avoiding redundant computations in concurrent jobs with no performance loss, and coordinated reads, which helps eliminate straggler clients in synchronous distributed training (2.2× average speedup for NLP jobs). Our work motivates future research in ML hardware and data processing service design to further improve cluster efficiency and model quality.

ACKNOWLEDGMENTS

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

REFERENCES

[1] 2022. Apache Beam: An advanced unified programming model. https://beam.apache.org/.

[2] 2022. Apache Flume. https://flume.apache.org/.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. of OSDI*. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[4] Amazon. 2022. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/.

[5] Amazon. 2022. Amazon EC2 Pricing. https://aws.amazon.com/ec2/instance-types/.

[6] Rohan Anil, Battulga Bayarsaikhan, Ryan Doherty, and Emanuel Taropa. 2021. Distributed computing pipeline processing. https://patents.google.com/patent/WO2021177976A1.

[7] Leon Bottou. 2009. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proc. of the Symposium on Learning and Data Science*.

[8] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 177–186.

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[10] Broadcom. 2019. *Broadcom Stingray PS250 SmartNIC*. https://docs.broadcom.com/doc/PS250-PB

[11] Tianshi Cao, Sasha (Alexandre) Doubov, David Acuna, and Sanja Fidler. 2021. Scalable Neural Data Server: A Data Recommender for Transfer Learning. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.).

[12] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15, 2 (jun 1990), 162–207. https://doi.org/10.1145/78922.78924

[13] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. 2019. Faster Neural Network Training with Data Echoing. arXiv:1907.05550 [cs.LG]

[14] Torch Contributors. 2022. PyTorch Docs: torch.utils.data. https://pytorch.org/docs/stable/data.html.

[15] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*.

[16] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. 2020. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.). 18613–18624.

[17] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*.

[18] Jia Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. of CVPR*.

[19] Jonas Geiping, Micah Goldblum, Gowthami Somepalli, Ravid Shwartz-Ziv, Tom Goldstein, and Andrew Gordon Wilson. 2023. How Much Data Are Augmentations Worth? An Investigation into Scaling Laws, Invariance, and Implicit Regularization. arXiv:2210.06441 [cs.LG]

[20] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared workload optimization. *Proceedings of the VLDB Endowment* 7, 6, 429–440.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[22] Google. 2022. *Better performance with the tf.data API*. https://www.tensorflow.org/guide/data_performance

[23] Google. 2022. Google Cloud: All Pricing. https://cloud.google.com/compute/all-pricing.

[24] Google. 2022. Google Cloud: TPU regions and zones. https://cloud.google.com/tpu/docs/regions-zones.

[25] Google. 2022. *tf.data service API documentation*. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service

[26] Google. 2023. *Colossus under the hood: a peek into Google's scalable storage system.*

[27] Google. 2023. Google Storage. https://cloud.google.com/storage.

[28] Google. 2023. *gRPC Documentation*.

[29] Google. 2023. Network Pricing. https://cloud.google.com/vpc/network-pricing.

[30] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *Proc. of USENIX ATC*.

[31] Joaquin Anton Guirao, Krzysztof Łęcki, Janusz Lisiecki, Serge Panev, Michał Szołucha, Albert Wolant, and Michał Zientkiewicz. 2019. Fast AI Data Preprocessing with NVIDIA DALI. https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali.

[32] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 383–394.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of CVPR*. IEEE Computer Society. https://doi.org/10.1109/CVPR.2016.90

[34] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

[35] Kubernetes HPA. 2023. *Kubernetes Horizontal Pod Autoscaler Documentation*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[36] Chip Huyen. 2022. *Designing Machine Learning Systems*. O'Reilly Media, USA.

[37] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[38] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433* (2023).

[39] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (2020).

[40] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. of ISCA* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3079856.3080246
[41] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.
[42] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proc. EuroSys (EuroSys '16)*. Article 29.
[43] Kubernetes. 2023. *kubernetes Documentation*. https://kubernetes.io/docs/home/
[44] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. In *Proc. of Machine Learning and Systems*, Vol. 4. 33–51.
[45] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *Proc. of FAST*.
[46] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. 2021. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *Proc. of USENIX ATC*.
[47] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.
[48] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.
[49] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Proc. of ECCV* (2014-01-01). Zürich. /se3/wp-content/uploads/2014/09/coco_eccv.pdf,http://mscoco.org Oral.
[50] Renato Marroquin, Ingo Müller, Darko Makreshanski, and Gustavo Alonso. 2018. Pay one, get hundreds for free: Reducing cloud costs through shared query execution. In *Proceedings of the ACM Symposium on Cloud Computing*. 439–450.
[51] Meta. 2022. Scaling data ingestion for machine learning training at Meta. https://engineering.fb.com/2022/09/19/ml-applications/data-ingestion-machine-learning-training-meta/.
[52] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling

[53] MLCommons. 2022. *ML Perf v2 Google Hardware Configurations*. https://github.com/mlcommons/training_results_v2.0/tree/main/Google/systems
[54] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.
[55] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. arXiv:2007.06775 [cs.DC]
[56] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *Proc. VLDB Endow.* 14, 12 (2021).
[57] MXNET. 2018. Designing Efficient Data Loaders for Deep Learning. https://mxnet.apache.org/api/architecture/note_data_loading.
[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[59] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *Proc. of the Fifteenth European Conference on Computer Systems*.
[60] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. of OSDI*.
[61] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. 1987. A System for Semantic Query Optimization. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 181–195. https://doi.org/10.1145/38713.38736
[62] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of big data* 6, 1 (2019), 1–48.
[63] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. 2021. Text data augmentation for deep learning. *Journal of big Data* 8 (2021), 1–34.
[64] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *Proc. of ICDAR (ICDAR '03)*. IEEE Computer Society, USA, 1 pages.
[65] Apache Spark. 2023. Spark Streaming Programming Guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html.
[66] TensorFlow. 2022. Module: tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service.
[67] TensorFlow. 2022. tf.data: Build TensorFlow input pipelines. https://www.tensorflow.org/guide/data.
[68] TensorFlow. 2023. Tensorflow. https://github.com/tensorflow/tensorflow.
[69] TensorFlow. 2023. TensorFlow Model Garden. https://github.com/tensorflow/models.
[70] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the fifteenth European

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

Audibert A., Chen Y., Graur D., Klimovic A., Šimša J., and Thekkath C.

conference on computer systems. 1–14.

[71] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1086–1099.

[72] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. *ACM Comput. Surv.* 53, 2, Article 30 (mar 2020).

[73] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proc. of EuroSys*.

[74] Kubernetes VPA. 2023. *Kubernetes Vertical Pod Autoscaler Documentation.* https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

[75] Xi Yan, David Acuna, and Sanja Fidler. 2020. Neural Data Server: A Large-Scale Search Engine for Transfer Learning Data. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[76] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. of HotCloud* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

[77] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *Proc. of ISCA (ISCA '22)*.