

tf.data: A Machine Learning Data Processing Framework

Derek G. Murray
Lacework*
derek.murray@lacework.net

Ana Klimovic
ETH Zurich*
aklimovic@ethz.ch

Jiří Šimša
Google
jsimsa@google.com

Ihor Indyk
Google
iindyk@google.com

ABSTRACT

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

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

PVLDB Reference Format:

Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. `tf.data`: A Machine Learning Data Processing Framework. PVLDB, 14(12): 2945-2958, 2021.

doi:10.14778/3476311.3476374

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/tensorflow/tensorflow>.

*Work done while at Google.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476374

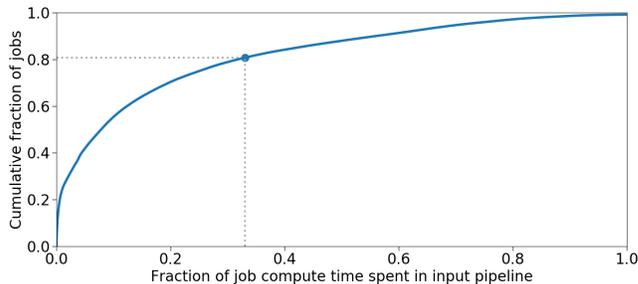


Figure 1: CDF showing the fraction of compute time that millions of ML training jobs executed in our fleet over one month spend in the input pipeline. 20% of jobs spend more than a third of their compute time ingesting data.

1 INTRODUCTION

Data is the lifeblood of machine learning (ML). Training ML models requires steadily pumping examples for models to ingest and learn from. While prior work has focused on optimizing the accuracy and speed of model training and serving, how we store and preprocess data for machine learning jobs has received significantly less attention. Across the millions of ML jobs we run in Google's datacenters every month, we observe that the input data pipeline accounts for significant resource usage and can greatly impact end-to-end performance. Figure 1 shows how the fraction of compute time that jobs spend in the input pipeline varies, where we define *compute time* as the time spent on a hardware resource – such as a CPU or an accelerator core – scaled by the compute capability of that resource. The marked point shows that 20% of jobs spend more than a third of their compute time in the input pipeline. When taking into account the total compute time from all jobs in our analysis (§ 5), we find that 30% of the total compute time is spent ingesting data. A complementary study of ML model training with public datasets found that preprocessing data accounts for up to 65% of epoch time [44]. This shows that input data pipelines consume a significant fraction of ML job resources and are important to optimize.

Input pipelines of machine learning jobs are often challenging to implement efficiently as they typically need to ingest large volumes of data, apply complex transformations, overlap communication and computation, and shuffle and batch data with various data ordering guarantees. For example, some jobs require that each example is visited exactly once before any example is visited a second time during training. Moreover, to achieve good performance and avoid input pipeline stalls, the data preprocessing should leverage parallelism and pipelining to overlap preprocessing with model

training computations. Determining the optimal degree of parallelism and amount of data to prefetch is often challenging as it depends on the nature of the workload and the hardware resources available.

Hardware accelerators used for ML training further increase the need for efficient input pipelines. Today’s accelerators, such as GPUs and TPUs, are tailored towards executing the linear algebra operations that are common in ML computations, but have limited support for common data preprocessing operations. Hence, input data is commonly processed on the CPU and feeding an accelerator with data at a sufficient rate to saturate its compute capabilities is becoming increasingly challenging. The high cost of accelerators compared to their CPU hosts makes it particularly important to ensure that accelerators operate at high utilization [6, 20].

We present `tf.data`, an API and a runtime for building and executing efficient input data pipelines for machine learning jobs. The `tf.data` API provides generic operators that can be parameterized by user-defined functions, composed, and reused across ML domains. Inspired by the programming models of relational databases [21, 24], declarative collection libraries [29, 42], and data-parallel big-data systems [66, 67], the `tf.data` API consists of stateless *datasets*, which are an abstraction for users to define their input pipeline, and stateful *iterators*, which produce a sequence of elements and maintain the current position within a dataset. These abstractions allow users to focus on the application logic of their input pipeline and leave the task of executing the pipeline efficiently to the `tf.data` runtime. In particular, `tf.data` internally represents an input pipeline dataset as a graph and applies static optimizations using graph rewrites. Furthermore, `tf.data` can automatically tune parameters such as the degree of parallelism and data prefetch buffer sizes, which are critical for performance yet often challenging for an average ML user to tune by hand.

Our evaluation demonstrates that 1) input pipeline performance is critical to end-to-end training time of state-of-the-art ML benchmarks, 2) `tf.data` is capable of improving input pipeline latency through a combination of software pipelining, parallelization, and static optimizations, and 3) `tf.data` dynamic optimizations avoid the need to manually tune performance knobs. For example, we show that introducing parallelism and software pipelining to the input pipeline of a RESNET50 model training on the IMAGENET dataset results in a 10.4× decrease in time to convergence. Applying further optimizations with `tf.data`, such as caching and static optimizations, improves training time by an additional 2×. We also demonstrate that `tf.data`’s auto-tuning matches the performance of expert hand-tuned input pipelines.

The `tf.data` API and runtime are open source and integrated in TensorFlow [3]. At Google, `tf.data` has been in production use since 2017 for various ML training jobs, such as supervised learning, federated learning, and reinforcement learning; with different data modalities, including text, image, and video data. The system is used daily by hundreds of thousands of ML jobs in our fleet.

We conduct a fleet-wide analysis of `tf.data` jobs to characterize the input pipelines of millions of real machine learning jobs and identify opportunities for future work in data preprocessing systems. We find that the set of transformations applied in input pipelines varies greatly across jobs. For 75% of jobs, the materialized dataset is smaller in size than the raw input data read from storage,

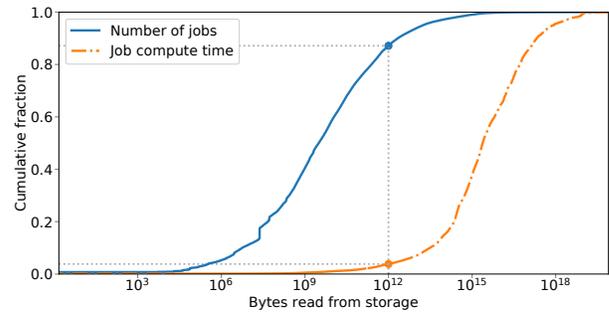


Figure 2: CDF of input data size across ML training jobs. 13% of jobs read more than 1 TB of data. These jobs consume over 96% of total compute resources.

which implies that preprocessing commonly decreases the volume of data. Most notably, we observe that identical input pipelines are frequently re-executed within and across jobs, suggesting that caching materialized datasets is a promising future direction to explore to improve the performance and efficiency of input data processing for ML. Our findings motivate several other directions for future research, such as processing data closer to storage and disaggregating input data processing from model training to avoid host resource bottlenecks.

2 INPUT PIPELINE REQUIREMENTS

Raw input data, such as images, audio, and text files, undergo both offline and online preprocessing before being ingested for model training. Offline data preprocessing involves extracting features from raw data, validating data [12], and converting data to binary formats, such as Avro [8], Parquet [9], or TFRecord [58], to enable higher throughput data ingestion. Batch computing frameworks such as Apache Spark [67], Beam [1], and Flume [2] are well suited to offline preprocessing. While some data transformations, such as normalization, can be applied during offline preprocessing, ML training also requires applying transformations online as examples are fed to the model. For instance, image models commonly rely on data augmentation, e.g. randomly distorting images, to improve accuracy [17, 54]. Data augmentation multiplies the size of the original dataset, making it prohibitive to store outputs in intermediate files. Our work focuses on online data preprocessing, which executes as part of the input pipeline of ML training jobs.

The input pipeline of ML training can be characterized as a three-stage extract, transform, load (ETL) process. The first stage reads input data from a storage system. Machine learning jobs commonly train on large data sets. Figure 2 shows that 13% of jobs, out of the millions of jobs we analyzed, read at least 1 TB of input data. This means that for a non-trivial fraction of training jobs, the input data cannot fit in memory. Furthermore, over 96% of total compute resources across jobs are spent in jobs that read over 1 TB of data.

The second stage transforms the data to a format amenable to ML training computation. It applies transformations to the input data, such as sampling, permuting, and filtering data to extract the subset of most relevant features. When training image models, it is common practice to apply data augmentation such as clipping,

resizing, flipping, and blurring images. For text pipelines, training examples commonly need to be grouped and batched based on sequence length. Finally, the third stage loads the data onto the accelerator device that executes the training computation.

ML training imposes unique requirements for input data pipelines. We describe these requirements below and summarize why they are not adequately addressed by other systems.

Data ordering Unlike many data-parallel data processing platforms [16, 66, 67], ML training is sensitive to the order in which records are delivered. The most common training algorithms are derived from *stochastic* gradient descent [50], which accesses the input examples pseudo-randomly. Empirically, convergence is more rapid when the algorithm makes multiple passes over input examples (called *epochs*), and uses a different random permutation of the input data on each pass (or equivalently, samples examples without replacement within each epoch) [10]. Furthermore, to improve system efficiency via vectorization and reduced communication, the input pipeline typically concatenates consecutive examples into a batch that is processed in a single training step.

The final parameters of a trained model can be sensitive to the exact order in which the input examples were consumed. To aid in debugging, especially when porting models between different hardware architectures, `tf.data` must be able to produce random results in a deterministic order, according to a given seed. While such a feature is useful for debugging, it is in tension with high performance, since any variability in the element processing time could lead to head-of-line blocking. Therefore, while `tf.data` defaults to deterministic execution, a user can disable it to mitigate the effect stragglers have on end-to-end performance.

Finally, both the end-to-end training computation and the individual epochs can take a long time to complete. To provide ordering guarantees in the presence of preemptions – commonplace in our data centers – the data processing computation for ML training jobs must be checkpointable.

Performance A single training step consumes a batch of input elements and updates the current weights of the model. Often, the step computation runs on an accelerator device – such as a GPU or TPU [30] – that can compute vector floating point operations efficiently, although the computation may also run on a multi-core CPU. Ideally, the data processing computation is pipelined with the training computation, minimizing the likelihood that the training computation is blocked waiting for the next batch of elements and hence maximizing the utilization of valuable accelerator resources.

The input pipeline is responsible for fetching the raw input data from storage and transforming it into input features for the model. For example, the raw input for an image classification model might be a protocol buffer [19] containing a JPEG-encoded image, and the input pipeline must convert the raw input into a dense three-dimensional array of floating point values corresponding to the RGB values of each pixel. Along the way, the input pipeline must extract and decode the JPEG and apply additional transformations such as affine transformations and colorspace changes to augment the training data [54]. These activities are CPU-intensive, and must make efficient use of available CPU resources to maximize input pipeline throughput.

Ease of use Machine learning workloads in a typical large organization span different domains, storage systems, data formats, and accelerator hardware. Therefore, it must be possible to combine pipeline stages in unanticipated ways, and extend the system with new data sources and transformations. To emphasize the importance of flexibility, in our fleet-wide analysis of ML jobs, we classified transformations into categories – such as reading input data from storage, caching, batching, or shuffling – and recorded the combination of transformation categories used by each job. While the 10 most common combinations of transformations account for over 75% of jobs, there is a heavy tail with over 1000 combinations of transformations in total. In addition to supporting diverse input pipelines, we also require the input pipeline framework to address the tension between performance and ease-of-use. Optimizing an input pipeline can require expertise in how to structure operations and tune performance-related parameters, such as degrees of parallelism and pipeline buffer sizes. Hence, we require that `tf.data` can optimize an input pipeline automatically.

Before designing `tf.data`, we evaluated several existing input pipeline implementations, and found that they did not meet our requirements in all of the above areas: 1) PyTorch’s `DataLoader` API [14] is easy to use (it provides a simple Python interface), but its reliance on Python on the critical path – despite the use of multiprocessing to work around the interpreter lock bottleneck – and assumption of uniform random access to all input data, do not satisfy our performance requirement, especially for multi-terabyte datasets. 2) MXNet’s `DataIter` API [47] uses a native C++ implementation for greater performance than PyTorch, but it requires users to add native extensions in order to handle new preprocessing schemes. Therefore it does not help our users with diverse data processing needs, who tend to prefer writing Python, and who are often restricted to memory-safe programming languages for security reasons. 3) NVIDIA’s `Data Loading Library (DALI)` API [22] enables some preprocessing, such as image decoding, to be offloaded to a GPU. This offloading partially fulfils our performance requirement, but it lacks the flexibility to support heterogeneous preprocessing workloads and different types of accelerators.

In the next section, we present the `tf.data` programming model, which is based on chaining higher-order functional transformations, and inspired by LINQ [42]. Several data processing systems offer a similar programming model, including DryadLINQ [66], Spark [67], and Naiad [46]. We discuss them in more detail in § 6. For pragmatic reasons, we did not consider using any of these systems, because the impedance mismatch with TensorFlow’s C++ codebase would severely limit performance. Furthermore, these systems are designed to optimize data parallel computations, with a large number of independent values in each batch. This makes it difficult or inefficient for them to produce values sequentially, to fulfill the sequential ordering requirement. While one could use a system like Spark Streaming [68] for online preprocessing and pass data to the ML framework through an in-memory buffer, the additional copies would have significant overhead due to the short step times in ML training workloads. In the training workloads we have analyzed, step times less than 1 ms are not uncommon and most workloads have step times less than 10ms. The extra copy overhead

would be especially significant in the common case where memory bandwidth is the bottleneck. By integrating `tf.data` directly into TensorFlow, and sharing the same threadpools and memory allocators, we avoid this overhead.

3 DESIGN AND IMPLEMENTATION

In § 3.1, we present `tf.data`'s API which enables users to compose and parameterize operators. In § 3.2 and § 3.3 we discuss key aspects of `tf.data`'s runtime.

3.1 Datasets and Iterators

The `tf.data` DATASET represents the stateless definition of an input pipeline as a (potentially infinite) sequence of elements. A dataset can either be a *source dataset* that is created from primitive values (e.g. a matrix of floating-point numbers representing input examples, or a vector of strings representing filenames), or a *transformed dataset* that transforms one or more input datasets into a new sequence of elements. The elements of a dataset are statically typed, and valid element types include tensors (with a specific element type and optional shape) and composite types (such as tuples, optionals, and nested datasets). Together, source and transformed datasets form an expression tree that represents the entire input pipeline. Table 1 shows the DATASET interface.

`tf.data` includes source datasets that support common file formats, and transformed datasets that implement functional transformations and may be parameterized by user-defined functions (UDFs). The UDFs can be written in Python, and `tf.data` uses TensorFlow's Autograph library to convert them into dataflow graphs [45]. Table 2 summarizes the most common `tf.data` transformations.

The `tf.data` ITERATOR represents the current state of traversing a DATASET. An iterator provides sequential access to the elements of a dataset via the `get_next` operation, which either returns a typed element, or an error status such as "out-of-range" (EOF). In `tf.data`, implementations of the ITERATOR interface are thread-safe, so multiple threads can call `get_next` concurrently to improve throughput, at the expense of determinism. The interface also includes `save` and `restore` methods to support checkpointing.

The iterator interface (Table 3) abstracts all details of how the elements are produced, including internal buffering and parallelism. Before applying optimizations, there is a one-to-one correspondence between dataset and iterator objects, but the optimizations in § 3.3 exploit the iterator abstraction to change the underlying dataset graph, and optimize how elements are produced, while presenting the same interface.

Method	Description
<code>make_iterator</code>	Creates a new iterator over the dataset.
<code>serialize</code>	Converts the dataset to a serialized expression.
<code>element_spec</code>	Returns the type signature of dataset elements.

Table 1: DATASET interface

Dataset	Description
<code>batch</code>	Concatenates multiple elements into a single element.
<code>cache</code>	Stores the input data in memory.
<code>concatenate</code>	Concatenates two datasets.
<code>from_file</code>	Reads elements from a file, e.g. <code>TextFileDataset</code> .
<code>from_tensors</code>	Creates a singleton dataset from data in memory.
<code>filter</code>	Returns elements matching a predicate.
<code>flat_map</code>	Maps elements to datasets and flattens the result.
<code>interleave</code>	Like <code>flat_map</code> , but mixes outputs from input elements.
<code>map</code>	Transforms individual elements.
<code>prefetch</code>	Adds a buffer to pipeline input production.
<code>reduce</code>	Reduces a dataset to a single element.
<code>repeat</code>	Produces the input dataset multiple times.
<code>shard</code>	Selects a subset of elements from the dataset.
<code>shuffle</code>	Randomizes the order of elements.
<code>unbatch</code>	Splits input elements on the 0th dimension.
<code>zip</code>	Combines elements of multiple datasets into tuples.

Table 2: Common `tf.data` source and transformed datasets.

The example in Figure 3 illustrates a training loop that uses a `tf.data` input pipeline to read elements from files, apply user-defined processing logic on each element and combine the processed elements into a mini-batch.

3.2 Parallel and Distributed Execution

To efficiently utilize available host resources, `tf.data` provides transformations that enable software pipelining, and parallel execution of computation and I/O. The `prefetch` transformation decouples the producer and consumer of data using an internal buffer, making it possible to overlap their computation. Input pipelines can use this transformation to overlap host computation, host-to-device transfer, and device computation. The `map` transformation takes an optional argument that specifies the degree of parallelism to use for applying the user-defined computation to input elements concurrently. The `interleave` transformation provides a similar optional argument that specifies the degree of parallelism to use for fetching data from input elements concurrently. In particular, the `interleave` transformation can parallelize I/O by interleaving data read from multiple files. By default, `tf.data` transformations produce elements in a deterministic order. However, as deterministic ordering can lead to head-of-line blocking, the parallel `map` and `interleave` transformations provide an option for enabling non-deterministic ordering, which can result in better performance at the expense of reproducibility.

To illustrate the benefits of the above transformations, we revisit the example presented in Figure 3. Let us assume that it takes 5ms to read an element from the file, 2ms to apply the user-defined logic to an element, and 1ms to batch 10 elements. The accelerator would

Method	Description
<code>get_next</code>	Returns the next element, or raises EOF.
<code>save</code>	Writes the iterator state to a file.
<code>restore</code>	Reads the iterator state from a file.

Table 3: ITERATOR interface

```

ds = tf.data.TextFileDataset(["a.txt", ...])
ds = ds.map(parse).batch(batch_size=10)
for elem in ds:
    train_step(elem)

```

Figure 3: Example of a training loop using a `tf.data` input pipeline that reads from a list of text files. `parse` is a user-defined function for data preprocessing.

```

ds = tf.data.Dataset.from_tensors(["a.txt", ...])
ds = ds.interleave(
    tf.data.TextFileDataset, cycle_length=2,
    num_parallel_calls=2)
ds = ds.map(parse, num_parallel_calls=10)
ds = ds.batch(batch_size=10)
ds = ds.prefetch(buffer_size=1)
for elem in ds:
    train_step(elem)

```

Figure 4: Optimized version of Figure 3 with a `tf.data` input pipeline that employs parallelism (for I/O and parsing) and software pipelining.

be idle for $(5+2) * 10 + 1 = 71$ ms at the start of each iteration before data for the training computation becomes available.

The `tf.data` input pipeline in Figure 4 is semantically equivalent to that of Figure 3. However, it uses 1) the optional `num_parallel_calls` argument of `interleave` and `map` to parallelize I/O and computation respectively, and 2) `prefetch` to overlap the input pipeline computation with the training computation. As a result, the input pipeline in Figure 4 will take $\max(10 * 5 / 2, 10 * 2 / 10, 1) = 25$ ms to produce a batch (assuming a sufficiently slow consumer) and the input pipeline computation (of the next batch) will be overlapped with the training computation on the accelerator (for the current batch). If the training computation takes more than 25ms, the data for each iteration of the training loop will be ready by the time the iteration starts. In § 3.3.2 we describe a mechanism for auto-tuning parallelism and buffer sizes so that users do not have to tune them manually.

While `interleave` is typically used to parallelize I/O, it can also be used for parallel execution of multiple copies of an arbitrary input pipeline (operating over different shards of the input data). We have found this mechanism useful to speed up input pipelines bottlenecked by inherently sequential transformations, such as `filter` or `unbatch`.

In addition to supporting efficient single-host execution, we also designed `tf.data` for distributed ML training computation use-cases, such as data parallel synchronous training, across multiple hosts (and accelerators per host). In this setup, each host has a `tf.data` input pipeline providing data for the accelerators attached to the host. To provide for clean separation of epochs, the input data can be sharded across multiple files and the `shard` transformation ensures that different hosts operate over different shards of the data. The sharded input pipelines do not communicate with each other.

3.3 Automatic Optimization

`tf.data`'s functional programming model enables it to provide multiple different implementations for a single input pipeline. Automatic *static* (§3.3.1) and *dynamic* (§3.3.2) optimizations improve `tf.data`'s performance and usability.

3.3.1 Static Optimizations. At run-time, `tf.data` can reflect on the expression tree of any dataset and replace it with a more efficient version. We implemented static optimizations as a virtual dataset transformation that converts the input dataset to an expression tree, applies a suite of rewriting rules, and then evaluates the rewritten expression tree to produce an output dataset. The current implementation uses TensorFlow's GraphDef protocol buffer as the representation and the Grappler optimization framework [56] to manipulate these expression trees. We are investigating the use of MLIR [37] as a richer representation that will enable us to reuse optimizations from other domains.

As we gained experience with `tf.data`, we created several custom transformation that fuse commonly adjacent transformations for performance reasons. These transformations are: `map` + `batch` fusion, `shuffle` + `repeat` fusion, `map` + `map` fusion, `map` + `filter` fusion, and `filter` + `filter` fusion. For example, the `map` + `batch` fusion transforms `d.map(f).batch(b)` into `map_and_batch(f, b)`, which is functionally equivalent but the implementation of the fused operator parallelizes and pipelines the copies of each element into the output batch with the processing of other batch elements. Many of the fusion optimizations in `tf.data` are inspired by deforestation in functional languages [60]. As the simplest example, the `map` + `map` fusion transforms `d.map(f).map(g)` expression into `d.map(g ∘ f)`. This eliminates the per-element overhead of an iterator—a virtual call to `get_next` and one of two function dispatches—and the composition $g \circ f$ may be optimized further by Grappler's standard optimization passes, such as arithmetic optimization and dead code elimination.

`tf.data` static optimizations are not limited to fusions. *Map vectorization* is an optimization that transforms `d.map(f).batch(b)` into the more efficient `d.batch(b).map(pfor(f))`. In the transformed expression, `pfor(f)` applies `f` to every slice of the batch in parallel [4]. This increases the efficiency of the resulting code by converting multiple invocations of a per-element operation (e.g. `tf.matmul()`) into a single invocation of a batched operation (e.g. `tf.batch_matmul()`) that itself has an efficient vectorized implementation. It also reduces the framework-induced overhead by replacing `b` function invocations with a single invocation.

3.3.2 Dynamic Optimizations. In many cases, the optimal configuration for a `tf.data` pipeline depends on properties of the input data (e.g. raw image sizes) and the available resources (e.g. number of CPU cores, RAM, and network bandwidth). Hence, `tf.data` provides configuration parameters such as the degree of parallelism for `map` transformations and the size of the buffer for the `prefetch` transformation.

To avoid the need for users to manually tune performance-related knobs, the `tf.data` runtime contains an *auto-tuning* mechanism that allocates CPU and RAM resources across various parts of the input pipeline in a way that minimizes the (expected) latency of the input pipeline producing an element. In the rest of this section,

we refer to the time it takes for an iterator to produce an element as its *output latency* and the output latency of an input pipeline is the output latency of the iterator for its final transformation.

To perform auto-tuning, `tf.data` executes the input pipeline in a light-weight harness that maintains a tree representation of the iterators currently executing as part of the input pipeline, and measures the processing time spent in each of the iterators. The root of the tree is the iterator producing data for training computation, the leaves of the tree correspond to source dataset iterators, and edges are implied by the input-output relationship between transformed datasets' iterators and their inputs. The tree structure can change over time as transformations such as `interleave` or `repeat` create multiple iterators during their lifetime.

The auto-tuning implementation uses the processing time and the input pipeline structure to build an analytical model that is used to estimate how input pipeline parameters affect end-to-end latency. The estimating function is a composition of the output latencies of individual iterators as functions of tunable parameters, iterator's processing time and inputs' output latency. The outermost function of the composition is the one for the final iterator. For synchronous transformations (i.e. transformations that do not decouple producer and consumer), the output latency of an iterator is a linear function of the output latencies of its inputs and the processing time spent in the iterator. For asynchronous transformations, such as `prefetch`, and the parallel `map` and `interleave`, the output latency of an iterator is no longer linear and additionally depends on the parallelism, buffer size, and the rate of the consumer. In particular, the expected output latency of the iterator is computed as the output latency of its input(s) multiplied by the probability that the buffer is empty, which we model using an M/M/1/k queue [53] and estimate as:

$$P_{empty} = \begin{cases} \frac{1}{n+1} & \text{if } x = y \\ \frac{1 - \frac{x}{y}}{1 - \left(\frac{x}{y}\right)^{n+1}} & \text{otherwise} \end{cases} \quad (1)$$

where n is the buffer size, x is the producer rate (computed from the output latency of the input iterator(s)), and y is the consumer rate (computed from the frequency of `get_next` calls). Note that the producer rate, x , in general depends on upstream computation, while the consumer rate, y , in general depends on downstream computation. We traverse the iterator tree depth first to estimate both x and y in a single traversal.

To illustrate how the estimation works, let's revisit the example from Figure 4, additionally assuming that 1) the `num_parallel_calls` and `buffer_size` arguments are set to the special `AUTOTUNE` value to enable auto-tuning, 2) the training computation requests data every 10ms on average, and 3) the auto-tuning harness is estimating the following combination: `interleave` parallelism 1 and buffer size 1, `map` parallelism 5 and buffer size 5, and `prefetch` buffer size 2. Figure 5 gives an example of how `tf.data` computes the output latency for such a pipeline.

`tf.data` creates a background thread that periodically uses the estimation process above to evaluate different combinations of parallelism and buffer sizes for tunable transformations. Parameters are chosen to minimize the expected output latency of the input

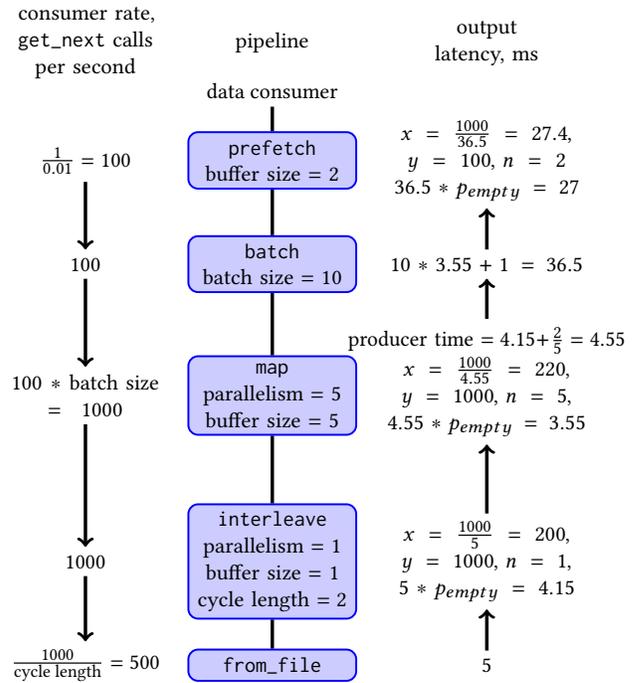


Figure 5: Output latency estimation: the downward traversal computes the consumer rate starting with the root adjusting it by the number of concurrent `get_next` calls from the consumer and the number of iterators. The upward traversal can compute the output latency of each iterator in the tree since by the time the traversal returns to an iterator, the output latency of its inputs is known. Asynchronous transformations `prefetch`, parallel `map` and `interleave` use (1) to estimate the output latency, whereas a synchronous `batch` produces an estimate with a linear function of its own processing time and output latency of its input.

pipeline subject to CPU and RAM budget constraints. The optimization uses a gradient descent algorithm and is depicted in Figure 6. The optimization period ranges from milliseconds to seconds and is determined automatically based on changes to the input pipeline structure and execution time.

An important aspect of the optimization is its ability to minimize output latency of the end-to-end input pipeline as opposed to minimizing the output latency of individual transformations. As different transformations share the same CPU and RAM resources, locally optimal decisions may lead to excessive parallelism and buffering, which in turn lead to inefficient thread scheduling and poor cache locality, negatively affecting end-to-end performance.

The ability to perform the optimization analytically is essential; it allows `tf.data` to quickly find a good configuration without affecting the performance of the real input pipeline while evaluating sub-optimal configurations. Once the background thread identifies a configuration to use, it updates the parallelism and buffer sizes of the actual input pipeline accordingly. For most input pipelines the optimization takes less than a few milliseconds to complete.

```

while True:
    model = pipeline.get_analytical_model()
    params = model.get_tunable_parameters()
    best_latency = INFINITY
    latency = model.latency()
    while (best_latency - latency >= EPS and
           model.resource_usage() <= BUDGET):
        best_latency = latency
        params -= DELTA * model.latency_grad()
        latency = model.latency()
    pipeline.set_tunable_parameters(params)
    sleep(OPTIMIZATION_PERIOD)

```

Figure 6: Periodic optimization of tunable parameters.

4 EVALUATION

To evaluate `tf.data` we seek to answer the following questions: 1) how do `tf.data`'s performance-related features affect input pipeline throughput, 2) how do input pipeline optimizations impact the end-to-end time to reach a target accuracy when training state-of-the-art ML models, and 3) how does `tf.data` performance compare to other systems?

For our evaluation, we used the open-source MLPerf benchmark suite [41], which is the de facto standard for evaluating ML software and hardware systems by measuring how fast a system can train models to a target quality metric. We use `tf.data` to implement the input pipelines for each of the MLPerf benchmarks. Our evaluation considers the following combinations of model architectures and input data: 1) RESNET50 [26] with IMAGENET [17], 2) SSD [40] with COCO [39], 3) MASK-RCNN [25] with COCO [39], 4) GNMT [64] with WMT16 [62], and 5) TRANSFORMER [59] with WMT17 [63].

Table 4 summarizes the attributes of the MLPerf datasets, which range from 135 MB to 140 GB in size. Though these public datasets fit in memory before decompression and/or data augmentations, in Section 5.1 we discuss our experience with production workloads which commonly preprocess larger-than-memory datasets (Figure 2). When dealing with such datasets, `tf.data`'s prefetching and software pipelining optimizations become even more critical for end-to-end performance.

Table 5 shows the performance-related features of `tf.data` used in the input pipelines of our MLPerf benchmark implementations. All input pipelines used the `map`, `interleave`, and `prefetch` transformations for parallel computation, parallel I/O, and software pipelining, respectively. All pipelines also used non-deterministic ordering to mitigate the effect of stragglers. With the exception of TRANSFORMER, the input pipelines used static `tf.data` optimizations to benefit from transformation fusion and the cache transformation to materialize intermediate preprocessing artifacts in memory to avoid their recomputation across epochs. Note that intermediate artifacts cannot always be materialized as they may be a result of a randomized transformation which produces a different result each epoch. Finally, the image-based input pipelines (RESNET50, SSD, and MASK-RCNN) disabled intra-op parallelism for `tf.data` computation. Intra-op parallelism makes it possible to parallelize execution of individual TensorFlow ops, such as `tf.matmul`, but this comes at the expense of increased CPU usage. For `tf.data`

Dataset	Domain	Artifacts	Size
IMAGENET	image classification	1.3M images	140GB
COCO	object detection	330K images	19GB
WMT16	translation	4M pairs	1.3GB
WMT17	translation	4.5M pairs	720MB

Table 4: MLPerf input data overview.

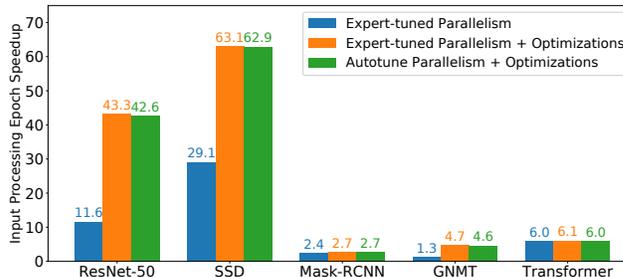


Figure 7: Speedup of input pipeline processing time with different configurations, relative to a sequential input pipeline.

input pipelines, intra-op parallelism generally provides little benefit, as there is plenty of inter-op and inter-element parallelism. Superfluous fine-grained parallelism can actually hurt performance due to poorer cache locality and additional synchronization.

4.1 Input Pipeline Experiments

Methodology: To evaluate the effect of `tf.data` performance-related features on input pipeline throughput, we executed the input pipeline portion of our MLPerf benchmark implementations in a tight loop (with no model training computation) and measured the time it takes to process an epoch's worth of data. We used a single machine with 56 Intel Xeon 2.60 GHz CPU cores, 128 GB of RAM, and the input data stored on a 1 TB Samsung SM961 SSD. We limited the RESNET50 experiment to only use 60% of the IMAGENET data to make sure that an epoch's worth of data can be cached in memory. For each of the input pipelines we ran the following experiments: 1) a baseline which does not use any `tf.data` performance features (i.e. sequential reading and processing), 2) a version that uses expert-tuned¹ parallelism for I/O and compute, 3) a version that uses all `tf.data` performance features in Table 5 with expert-tuned parallelism, and 4) a version that uses all `tf.data` performance features with auto-tuned parallelism. Note that even though the baseline does not use input pipeline parallelism, TensorFlow may still parallelize the user-defined computation in `map`.

Results: Figure 7 shows the mean duration of a single epoch, normalized to the epoch duration of the baseline, which does not use any `tf.data` performance-related features. On the 56-core machine used for the experiment, the speedups ranged from 2.7 \times (MASK-RCNN) to 63.1 \times (SSD). Since we are parallelizing both compute and I/O we could achieve speedup greater than 56 \times .

¹Expert-tuned parallelism sets `map` parallelism to the number of CPU cores available on the machine, `interleave` parallelism to a constant between 10 and 64 tuned based on available I/O bandwidth, and the `prefetch` buffer size to an empirically tuned multiple of batch size.

	Parallel computation	Parallel I/O	Software pipelining	Non-deterministic	Caching	Static Optimization	No intra-op parallelism
RESNET50	✓	✓	✓	✓	✓	✓	✓
SSD	✓	✓	✓	✓	✓	✓	✓
MASK-RCNN	✓	✓	✓	✓	✓	✓	✓
GNMT	✓	✓	✓	✓	✓	✓	✓
TRANSFORMER	✓	✓	✓	✓	✓	✓	✓

Table 5: `tf.data` features used by different MLPerf benchmarks.

The performance of RESNET50 and SSD input pipelines benefits significantly from `tf.data`'s performance optimizations, and the input pipelines can fully utilize the available CPU. In particular, `map + batch` fusion yields the most significant speedup among static optimizations for these two benchmarks, as it enables computing multiple batches in parallel. In contrast, the performance of MASK-RCNN, GNMT, and TRANSFORMER input pipelines sees less benefit from the `tf.data` optimizations. For MASK-RCNN, the reason for the limited speedup is two-fold: 1) the baseline employs parallelism as the user-defined computation applied to each element can be parallelized by TensorFlow and 2) the input pipeline is bottlenecked by batching, which is performed sequentially because of an intermediate step between `map` and `batch` in the pipeline that prevents `map + batch` fusion. Similarly, the text pipelines (GNMT and TRANSFORMER) did not benefit from `map + batch` fusion as elements need to be grouped based on size after the `map` operation before they are batched, but the `tf.data` runtime does not currently support `map + groupby + batch` fusion. Most benchmarks saw less than 4% improvement in training time with non-deterministic vs. deterministic data ordering, however RESNET50 benefited more (approx. 40% higher throughput) as its dataset (IMAGENET) has a wide distribution of image sizes, and non-deterministic ordering avoids head-of-line blocking.

For all of the input pipelines, using auto-tuned parallelism results in comparable performance to the expert-tuned versions. This demonstrates that the algorithm described in § 3.3 is able to automatically configure performance knobs similar to a human expert.

4.2 End-to-End Experiments

Methodology: To evaluate how input pipeline performance optimizations with `tf.data` translate to end-to-end performance benefits when training state-of-the-art ML models, we measured the time it takes to reach target accuracy with our `tf.data`-based implementation of the MLPerf benchmarks. We executed each benchmark using 8 hosts with 112 CPU cores, 100 GB of RAM, and 8 TPUv3 accelerators each [30]. For the MASK-RCNN benchmark, we used 400 GB RAM per host to ensure that intermediate artifacts can be fully cached in memory. We ran the following experiments for each benchmark: 1) a baseline that trains the MLPerf model with sequential reading and processing of input data, 2) a version that uses expert-tuned parallelism for I/O and compute in the input pipeline, 3) a version that uses all `tf.data` performance features with expert-tuned parallelism, and 4) a version that uses all `tf.data` performance features with auto-tuning.

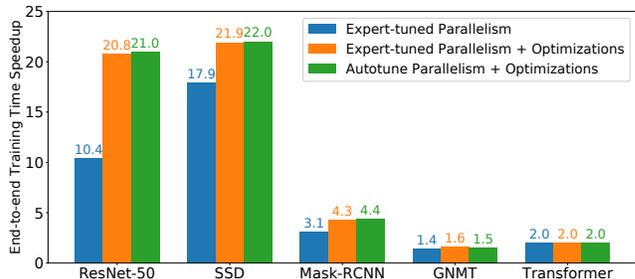


Figure 8: Speedup of the time to convergence for MLPerf workloads with `tf.data` optimizations, relative to execution with a sequential input pipeline.

Results: Figure 8 shows the end-to-end training time speedup (relative to the model training time with a sequential input pipeline) for each MLPerf benchmark. We draw several insights from these results. First and foremost, the performance of the input pipeline significantly affects the end-to-end training performance. Second, computation and I/O parallelism is necessary but not sufficient to match the rate at which accelerators perform training computation. Compared to using a sequential input pipeline as the baseline, adding software pipelining and parallelism in the input pipeline improved end-to-end training time by 7× on average across the five MLPerf benchmarks. For image-based input pipelines (RESNET50, SSD, and MASK-RCNN), the end-to-end performance benefited further from the application of `tf.data` performance-oriented features, providing an additional 2×, 1.2×, 1.4×, speedup respectively. For text-based input pipelines (GNMT and TRANSFORMER), parallelism and software pipelining alone were sufficient to match the rate at which data was consumed by the training computation.

Figure 8 also compares the training time with expert-tuned `tf.data` configuration to training time with auto-tuned configuration. Similarly to the input pipeline experiments, we find that using `tf.data`'s dynamic optimizations to select parameters such as the degree of parallelism and prefetch buffer sizes leads to similar performance compared to the expert-tuned pipelines. The end-to-end time to convergence with dynamic tuning is within 1% of the time to convergence with expert-tuned input pipelines for RESNET50, SSD, MASK-RCNN, and TRANSFORMER and within 4% for GNMT. This demonstrates that `tf.data` can effectively relieve users from the burden of hand-tuning input pipeline configurations.

Finally, we also verified that `tf.data` optimizations enable input pipelines to match the rate at which accelerators perform training

Input data framework	Hardware	Epoch duration (s)
PyTorch DataLoader	CPU-only	213
NVIDIA DALI	CPU-only	777
NVIDIA DALI	CPU + 1 GPU	172
NVIDIA DALI	CPU + 2 GPUs	107
tf.data	CPU-only	110

Table 6: IMAGENET-RESNET50 input data processing time with tf.data vs. NVIDIA DALI and PyTorch DataLoader.

computations for state-of-the-art models. For each MLPerf benchmark, we measured the time it takes to ingest a batch of data and perform the model computation when using 1) an optimized tf.data input pipeline versus 2) an artificial input pipeline that produces data as fast as possible (by caching a single batch and repeating it infinitely). The artificial pipeline does not perform any data processing and hence serves as an upper bound on input pipeline performance. Step times with optimized tf.data pipelines match the upper-bound performance, hence the MLPerf benchmarks are no longer input bound after tf.data optimizations.

4.3 Comparison to Other Systems

To evaluate how tf.data compares to other input data processing systems for ML, we implement a standard IMAGENET pipeline using tf.data, PyTorch DataLoader [14], and NVIDIA DALI [22]. Table 6 shows the average time to process an epoch’s worth of data with each framework running on a 64 core server (n2-standard-64 on Google Cloud) with 256 GB of RAM, 500 GB local SSD, and NVIDIA Tesla T4 GPUs. The tf.data pipeline is 1.9× faster than DataLoader, thanks to tf.data’s static and dynamic optimizations. For example, if we disable map + batch fusion in tf.data, performance drops to 448 seconds per epoch. Table 6 also shows that tf.data outperforms DALI on CPU or even with one GPU. When offloading computation to multiple GPUs, DALI achieves higher throughput, however the use of GPUs adds to the cost of input data processing and consumes GPU cores and memory that could otherwise be dedicated to model training.

In addition to comparing input pipeline throughput, it is useful to compare end-to-end model training time with different input data frameworks across heterogeneous platforms. The MLPerf Training competition provides the fairest comparison across ML systems as each submission is optimized by experts familiar with their performance knobs. For each benchmark, a cluster ranging from 8 accelerators to over 1000 accelerators was used to train the model to a target accuracy. Table 7 summarizes the top MLPerf v0.7 training times achieved, categorized by the input pipeline framework used [43]. The end-to-end training times in Table 7 do not provide an apples-to-apples performance comparison of input data frameworks, since the competition entries used different software frameworks (TensorFlow, PyTorch, MXNet) and hardware (TPUs, GPUs) to run model training computations. However, we can still draw two important takeaways from the end-to-end training times in Table 7. First, tf.data is the only input processing framework that was used across all MLPerf benchmarks, including image and

	RESNET50	SSD	MASK-RCNN	GNMT	TRANSFORMER	BERT
tf.data	28.8	27.6	487.8	77.4	15.6	23.4
DataLoader	-	-	627.6	42.6	37.2	48.6
DALI	49.8	49.2	-	-	-	-

Table 7: Best end-to-end MLPerf v0.7 competition training times (in seconds), categorized by the input data framework used. Entries with tf.data, DataLoader, and DALI input pipelines use TensorFlow, PyTorch, and MXNet, respectively, for model training.

text workloads. This attests to tf.data’s flexibility. Other frameworks only achieved competitive results for a subset of benchmarks (e.g., DALI for image workloads and DataLoader for text workloads). Second, tf.data is fast enough to avoid input bottlenecks across state-of-the-art models and hardware configurations, enabling training RESNET50, SSD, TRANSFORMER, and BERT in under 30 seconds. As shown in § 4.2, the MLPerf workloads are not input-bound after applying tf.data optimizations. In particular, the higher end-to-end training time with GNMT, is due to the TensorFlow model computation being slower than the PyTorch implementation; the tf.data part of the computation is not on the critical path.

5 EXPERIENCE

Our colleagues at Google have been using tf.data in training research and production ML models since 2017. As of today, the system implementation consists of over 21K lines of Python and over 56k lines of C++ (excluding test code). The tf.data framework is used for data processing by the majority of TensorFlow training jobs in Google’s fleet. These jobs run in production clusters, spanning a variety of application domains (e.g., image classification, translation, and video content recommendation) and using various types of ML training algorithms (e.g., supervised learning, reinforcement learning, and federated learning). tf.data’s generality has also facilitated novel research. For example, a creative approach to working around limited I/O bandwidth when training models and was implemented using three standard tf.data transformations [13]. tf.data was also used to automatically generate a data augmentation policy that achieved state-of-the-art results on image classification tasks [15].

To understand the characteristics of machine learning input data pipelines at scale, we studied millions of tf.data jobs in Google’s fleet over a one month period in 2020. We show that input pipelines are highly diverse and frequently re-executed. We also identify several future research directions motivated by our findings, such as the opportunity to re-use input pipeline computation across jobs.

5.1 Fleet-wide Input Pipeline Analysis

Methodology: We instrument tf.data to collect metrics such as the set of transformations applied in each job’s input pipeline. For each job, we also record the bytes consumed and produced by each transformation in its input pipeline. 71% of jobs define their input pipeline as a single tf.data dataset, while the remaining jobs define their input processing logic across two or more tf.data datasets. When an iterator is created for a tf.data dataset, we

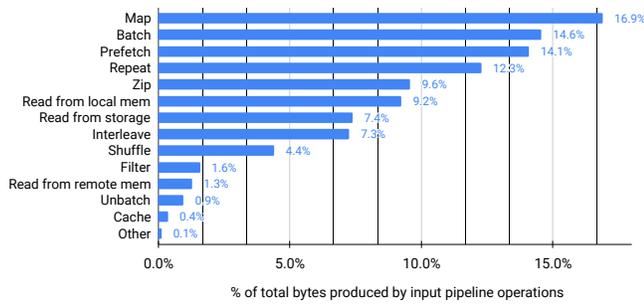


Figure 9: Types of input data pipeline operations and their prevalence, based on the bytes produced by each type of op.

fingerprint the dataset by computing a hash of its dataflow graph. We include the list of input file names in the hash calculation and exclude random seed values. We track the number of iterator creations for each unique hash over time. We also measure the total compute time for jobs and the compute time that jobs spend in `tf.data`. The compute time is measured in normalized compute units and is the product of the time spent on a hardware resource – such as a CPU or an accelerator core – scaled by the compute capability of that resource. Our compute time metric is analogous to AWS’s Elastic Compute Units (ECUs) [5]. We collect the metrics described above with full coverage for all `tf.data` jobs, with one exception. Measuring the fraction of compute time spent in `tf.data` requires a configuration flag to be set when jobs are launched. Due to configuration differences across jobs, we measured the fraction of compute time spent in `tf.data` for 66% of jobs, accounting for 75% of total compute time across `tf.data` jobs. For the remaining jobs, we assume that each job spends 10% of its total compute time in `tf.data`, as this is the median time that jobs spend in the input pipeline (see Figure 1).

Our analysis focuses on three key questions: 1) how frequently are various transformations used in an input pipeline, 2) how does the “shape” of data change as data flows through an input pipeline, and 3) how much computation is shared across input pipeline executions in our fleet?

Which datasets are most common? Figure 9 plots the relative frequency of `tf.data` transformations across jobs, based on the number of bytes each transformation is applied on. The `map`, `batch`, `prefetch`, `repeat`, and `zip` transformations are the five most commonly applied types of transformations, followed by reading input data from local memory and storage. We also study how many input pipelines rely on various `tf.data` optimizations. On average, 77% of input pipelines rely on parallel I/O optimizations by using the `interleave` transformation, 87% of input pipelines rely on pipeline parallelism with the `prefetch` transformation, and 40% of pipelines rely on parallelizing compute with the `map` transformation (and its fusion with `batch`). Only 19% of jobs use the `cache` transformation to cache input data in memory, though we later show that many more jobs could benefit from caching since many input pipelines are re-executed.

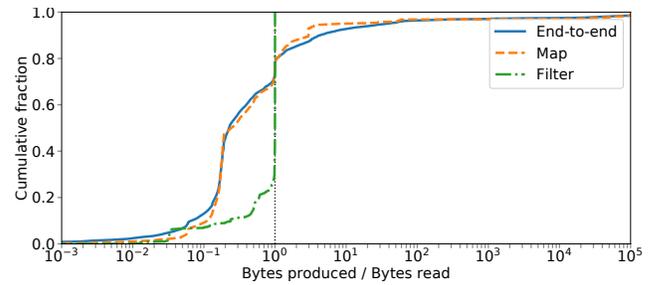


Figure 10: CDF showing how the ratio of bytes produced vs. bytes read varies for end-to-end input pipelines, `map`, and `filter` transformations. For 75% of jobs, preprocessing reduces the volume of data end-to-end.

How does preprocessing affect data volume? Although some transformations, such as filtering, decrease the size of input data, machine learning jobs also commonly apply transformations that increase the size of data, such as decompressing and augmenting images to train image understanding models. To understand how the volume of data flowing through ML input pipelines varies with different transformations, we measure each input pipeline’s ratio of bytes produced versus the bytes read from inputs sources. We compute this ratio for the end-to-end input pipeline of each job, as well as for each type of transformation applied in the job’s input pipeline. When the bytes produced over bytes consumed ratio is less than one, it means that the input pipeline or transformation in this job decreases the data volume, whereas a ratio greater than one implies that the volume of data increases.

Figure 10 plots the CDF of the bytes produced over bytes consumed ratio across jobs for their end-to-end input pipeline, `map` transformations, and `filter` transformations. For approximately 75% of jobs, the volume of data produced by the input pipeline and fed to the model training stage is less than the volume of input data read. In other words, for most jobs, the materialized dataset used for training is smaller than the raw input data. For some jobs, decompressing and augmenting data results in high expansion of source data. Figure 10 shows that user-defined `map` transformations, while preserving dataset cardinality, can decrease or expand data by over an order of magnitude for 13% of jobs. `filter` transformations, which can modify dataset cardinality, discard more than 10% of input data volume for approximately 23% of jobs. For 8% of jobs, more than half of the bytes fed into `filter` transformations are discarded. `filter` is also used to sanitize data, hence in 70% of jobs, the transformation reduces the data by less than 1%.

How often are input pipelines re-executed? We observe that input pipelines are commonly re-executed and there is a sizeable opportunity to reuse input pipeline computation both within and across jobs. Some jobs rely on different permutations of the same dataset across iterators to improve convergence. To conservatively estimate the opportunity for computation reuse across input pipeline executions, we have excluded datasets that use the `shuffle` transformation (57% of `tf.data` jobs) in this part our analysis.

An input pipeline iteration begins by creating an iterator for a dataset definition. We record the number of iterator creations at

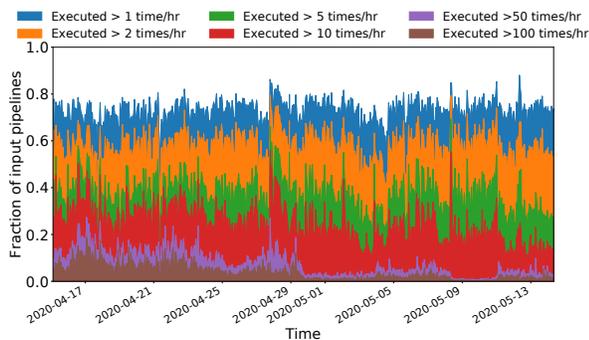


Figure 11: Fraction of input pipelines executed more than x times per hour, over time. Approx. 75% of input pipelines are executed more than once in the same hour.

the granularity of one hour time intervals for each dataset fingerprint (computed by hashing its dataflow graph). Figure 11 plots the fraction of input pipelines that are executed more than x times in the same hour, over time. Approximately 75% of input pipelines are executed more than once within the same hour and 5% of input pipelines are executed more than 100 times within an hour. Re-execution of input pipelines can occur across epochs of a training job and also across jobs. For example, neural architecture search [69] and hyper-parameter tuning both require training multiple models using the same input pipeline.

Having found that many input pipelines are re-executed, we next quantify the opportunity for reusing input pipeline computation by caching materialized datasets. Figure 12 plots the cumulative distribution of input pipeline executions over the one month time span of our study, with input pipelines ordered from most to least frequently executed. We also show the CDF of the compute resources spent executing these pipelines. Figure 12 shows that by caching the top 10% of materialized datasets, we can capture 72% of CPU resources used for computing `tf.data` datasets across all jobs that executed in the one month period. The steepness of the CDF curves indicates that some datasets are particularly frequently executed and consumed significant resources. Only 10% of input pipelines are re-executed across multiple jobs. 1% of input pipelines are executed by more than 25 different jobs and the largest cross-job sharing we observed was approximately 50,000 jobs executing the same input pipeline. However, our analysis conservatively estimates the opportunity for reuse since it only counts re-executions of pipelines with identical end-to-end transformation graphs. We anticipate further opportunities to reuse computation across jobs by considering input pipeline sub-graphs.

5.2 Implications for Future Research

We discuss the implications of our fleet-wide analysis and future research directions based on our experience with `tf.data`.

Datasets as a service We showed that input pipelines are frequently re-executed, yet only 19% of jobs in our analysis used the cache transformation. It is often challenging for users to decide if and where to apply caching as there are several factors to consider: the cost-benefit of caching the data – spending RAM to save

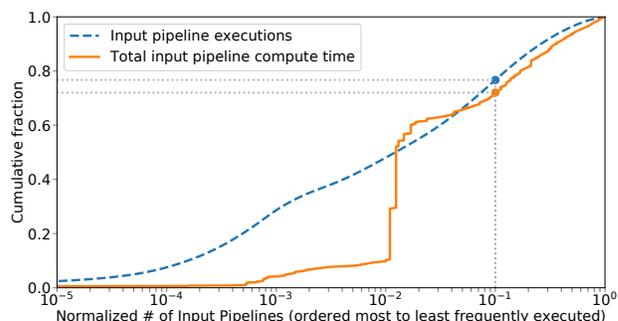


Figure 12: CDF of input pipeline executions over a one month period. 10% of pipelines account for 77% of total input pipeline executions and 72% of compute resources.

CPU and possibly improve throughput – and the impact of caching on training quality – in general, results of randomized transformation (such as `shuffle`) should not be cached. Navigating the compute-storage trade-off and estimating the benefit of caching on end-to-end performance and downstream accelerator utilization for ML training jobs is a complex task for users [23]. Hence, automating cache insertion in input pipelines is important [65]. Furthermore, since input pipelines can be shared across jobs, designing a dataset caching service to re-use input pipeline computations across jobs is a promising future direction. Quiver [36] and CoordL [44] already optimize source dataset caching for ML training. Several systems have shown that caching data across jobs greatly improves performance for big data analytics [7, 18, 23, 38, 49].

Processing data closer to storage Figure 10 showed that data preprocessing reduces the volume of data for 75% of jobs. For 14% of jobs, the volume of data fed into the model for training is less than 10% of bytes read from storage. As input data for ML jobs commonly resides in remote storage, such as a distributed file system or cloud object store, this means that more data than necessary is sent over the network during ML training. Designing ML data processing systems that apply projections closer to storage is a promising way to reduce data transfers. Using columnar data formats is another well-known approach to enable reading only the relevant fields in a record [9]. We are exploring this approach to improve data ingestion efficiency for ML jobs.

Addressing host bottlenecks Some input pipelines require significant CPU and memory resources to produce their data. When a host machine isn't powerful enough to generate input data at the rate the attached accelerator(s) consume the data, the accelerator(s) idle and slow down model training. To solve this problem, we are currently exploring the disaggregation of data processing from model training, by enabling users feed accelerators from input workers distributed across multiple hosts [57]. The number of input workers can scale up or down as needed to keep up with the accelerators, independent of the number of accelerators attached to one host. Another approach to address host resource bottlenecks for input data processing is to offload data preprocessing computations to accelerators [22].

Data processing for online inference We designed `tf.data` to support ML training. However, efficient input processing is also critical for ML inference. Inference involves less model computation per input element since only a forward pass of the model is required, whereas training also requires backpropagation. Hence, although not all input transformations applied during training – such as data augmentations – are applied when serving a model, the input pipeline for inference still presents a significant fraction of total work. Inference jobs need a different input pipeline design compared to training jobs as the primary performance objective is to optimize the latency of individual requests rather than overall throughput. This implies less buffering, no shuffling, and a different approach to batching to balance request latency with accelerator efficiency. Kang *et al.* propose jointly optimizing inference and data preprocessing for DNN-based visual analytics [33].

6 RELATED WORK

Kakarapathy *et al.* argue for building a single, unified system for data loading that could be shared between multiple machine learning jobs, and potentially between different frameworks as well [31]. Their observation that much I/O and preprocessing work can be shared between jobs agrees with our findings in § 5.2. By contrast, our work on `tf.data` has focused on a more general programming model, enabling users to build different preprocessing schemes.

Our inspiration for `tf.data`'s programming model drew from the successful application of LINQ [42] to parallel processing with PLINQ [55], big-data cluster processing with DryadLINQ [66], and stream processing with Naiad [46]. Many `tf.data` transformations have direct equivalents in LINQ, though we added order-sensitive transformations (e.g., `batch`, `shuffle`, and `interleave`) to support ML training algorithms. Optimus [35], which added dynamic graph rewriting support to DryadLINQ, is similar to the automatic optimization framework that we described in § 3.3. Optimus focused on reducing network I/O in distributed big-data queries, whereas the bottleneck in `tf.data` tends to be the host CPU, and our optimizations aim to reduce the wall-clock execution time of code within a single machine. Dandelion extended LINQ with the ability to run on GPU and FPGA accelerators [52], using the PTask abstraction to manage the accelerators [51]. Dandelion and PTask provide a simple programming model that hides data movement between the host and accelerator devices, similar to how `tf.data` uses `prefetch` to mask copies. Dandelion goes further than `tf.data` in using functional transformations to represent all computation – not just the input pipeline – while `tf.data` interoperates with existing ML frameworks such as TensorFlow [3], Pytorch [48], and JAX [11] by using their existing programming models for the training loop.

The design, implementation, and optimization of `tf.data` all bear similarities to how SQL is used in a relational database management system (RDBMS). A related strand of work has investigated how to push machine learning computations into SQL, and optimize across the boundary between relational data and linear algebra. The MADlib analytics library pushes various learning algorithms into an existing RDBMS [27]. MADlib uses existing SQL constructs for orchestration – i.e. defining both the input pipeline and the “driver program” (or training loop) – and provides a C++ abstraction layer for plugging in user-defined functions that call high-performance

numerical libraries. By building `tf.data` into TensorFlow and using its Tensor type to represent values, we achieved efficient interoperability for free. More recently, Karanasos *et al.* introduced Raven, which integrates the ONNX Runtime for machine learning into Microsoft SQL Server [34]. Raven focuses on ML inference for SQL-based analytic pipelines, achieving better performance by pushing linear algebra operators into earlier stages of the query plan and using ONNX Runtime to offload computation to accelerators. The model-related optimizations in `tf.data` are more conservative than Raven's, because the model is mutable at training time, but the ideas in Raven would be useful for applications like knowledge distillation [28] that perform inference at training time.

Several related projects have investigated the problem of automatically tuning dataflow workloads. SEDA dynamically allocates threads to stages, using a simple scheme that adds threads to a stage when its queue length exceeds a threshold, and removes them when they idle for a period [61]. By contrast, `tf.data` tunes the performance of each stage based on the predicted end-to-end performance. The DS2 scaling controller for dataflow-based stream processing attempts to find the minimum parallelism for each stage in a dataflow graph that will enable it to consume data at the rates of all the sources [32]. Like DS2, `tf.data` uses lightweight instrumentation of “useful” processing time in each transformation to make scaling decisions, but we additionally model memory consumption as a possible bottleneck resource to avoid excessive buffering.

7 CONCLUSION

We presented `tf.data`, a framework for building and executing efficient input data processing pipelines for machine learning jobs at scale. `tf.data`'s programming model enables users to build diverse input pipelines by composing and customizing operators. `tf.data` executes input pipelines as dataflow graphs and applies static optimizations that improve end-to-end training time for state-of-the-art models. For example, input pipeline parallelism and software pipelining improve RESNET50 training time by over 10× and other `tf.data` optimizations such as operator fusion provide an additional 2× improvement. We developed an analytical approach to automatically tune internal buffer sizes and the degree of parallelism in input pipelines. These dynamic optimizations achieve comparable performance to expert-tuned input pipelines while relieving users from the burden of manually tuning parameters.

The widespread deployment of `tf.data` across millions of ML training jobs at Google enabled us to quantify several aspects of ML data processing, namely its resource footprint, diversity, and the extent of redundant computation. On average, ML training jobs spend 30% of their total compute time in the input pipeline. Our findings motivate future work on sharing input pipeline computation across jobs and pushing data projection to the storage layer.

ACKNOWLEDGEMENTS

We thank Paul Barham, Chandu Thekkath, Vijay Vasudevan, Martín Abadi, Sudip Roy, Dehao Chen, and our anonymous reviewers for their helpful feedback on this work. We gratefully acknowledge Andrew Audibert, Brennan Saeta, Fei Hu, Piotr Padlewski, Rachel Lim, Rohan Jain, Saurabh Saxena, and Shivani Agrawal for their engineering contributions to `tf.data`.

REFERENCES

- [1] 2020. Apache Beam: An advanced unified programming model. <https://beam.apache.org/>.
- [2] 2020. 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 *Proceedings of OSDI*. 265–283.
- [4] Ashish Agarwal. 2019. Static Automatic Batching In TensorFlow. In *Proceedings of ICML*. 92–101.
- [5] Amazon. 2020. Amazon EC2 FAQs. <https://aws.amazon.com/ec2/faqs>.
- [6] Amazon. 2020. Amazon EC2 Pricing. <https://aws.amazon.com/ec2/pricing/>.
- [7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruva Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of NSDI*. 20.
- [8] Apache Software Foundation. 2012. Avro. <https://avro.apache.org/docs/1.2.0>.
- [9] Apache Software Foundation. 2018. Parquet. <https://parquet.apache.org/>.
- [10] Leon Bottou. 2009. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proceedings of the Symposium on Learning and Data Science*.
- [11] 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>
- [12] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *Proceedings of Machine Learning and Systems (MLSys) 2019*.
- [13] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. 2019. Faster Neural Network Training with Data Echoing. [arXiv:1907.05550](https://arxiv.org/abs/1907.05550) [cs.LG]
- [14] Torch Contributors. 2019. PyTorch Docs: torch.utils.data. <https://pytorch.org/docs/stable/data.html>.
- [15] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2019. RandAugment: Practical automated data augmentation with a reduced search space. [arXiv:1909.13719](https://arxiv.org/abs/1909.13719) [cs.CV]
- [16] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*. 137–150.
- [17] Jia Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of CVPR*.
- [18] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. 2016. Quartet: Harmonizing Task Scheduling and Caching for Cluster Computing. In *Proceedings of HotStorage*.
- [19] Google. [n.d.]. Protocol Buffers. <https://developers.google.com/protocol-buffers>.
- [20] Google. 2020. Google Cloud: All Pricing. <https://cloud.google.com/compute/all-pricing>.
- [21] Goetz Graefe. 1994. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowledge and Data Engineering* 6, 1 (Feb 1994), 120–135.
- [22] Joaquin Anton Guirao, Krzysztof Łęcki, Janusz Lisiecki, Serge Panev, Michał Szolucha, Albert Wolant, and Michał Zientkiewicz. 2019. Fast AI Data Preprocessing with NVIDIA DALI. <https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali>.
- [23] Pradeep Kumar Gunda, Lenin Ravindranath, Chandu Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of OSDI*.
- [24] Donald J. Haderle and Robert D. Jackson. 1984. IBM Database 2 overview. *IBM Systems Journal* 23, 2 (1984), 112–125.
- [25] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. *CoRR* (2017). <http://arxiv.org/abs/1703.06870>
- [26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of CVPR*. IEEE Computer Society, 770–778.
- [27] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1700–1711.
- [28] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. [arXiv:1503.02531](https://arxiv.org/abs/1503.02531) [stat.ML]
- [29] Java. 2020. Stream API. <https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>.
- [30] 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 (June 2020), 67–78.
- [31] Aarati Kakaraparthi, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *Proceedings of HotCloud*. Renton, WA.
- [32] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of OSDI*. 783–798.
- [33] Daniel Kang, Ankit Mathur, Teja Veeramacheni, Peter Bailis, and Matei Zaharia. 2020. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics. [arXiv:2007.13005](https://arxiv.org/abs/2007.13005) [cs.DB]
- [34] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *Proceedings of CIDR*.
- [35] Qifa Ke, Michael Isard, and Yuan Yu. 2013. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of EuroSys*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). 15–28.
- [36] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of FAST*. 283–296.
- [37] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zimenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore’s Law. *CoRR* (2020). <https://arxiv.org/abs/2002.11054>
- [38] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of SoCC*. 1–15.
- [39] 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 *Proceedings of ECCV*.
- [40] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. SSD: Single shot multibox detector. In *Proceedings of ECCV*. Springer, 21–37.
- [41] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2019. MLPerf training benchmark. [arXiv preprint arXiv:1910.01500](https://arxiv.org/abs/1910.01500) (2019).
- [42] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of SIGMOD*. 706.
- [43] MLPerf Training v0.7 Results. 2020. Designing Efficient Data Loaders for Deep Learning. <https://mlperf.org/training-results-0-7/>.
- [44] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proceedings of VLDB Endow.* 14, 5 (Jan. 2021), 771–784.
- [45] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zack Nado, D. Sculley, Tiark Rompf, and Alexander B Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *SysML*.
- [46] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (proceedings of the 24th acm symposium on operating systems principles (sosp) ed.). ACM.
- [47] MXNET. 2018. Designing Efficient Data Loaders for Deep Learning. https://mxnet.apache.org/api/architecture/note_data_loading.
- [48] 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.
- [49] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of OSDI*. 401–417.
- [50] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *Ann. Math. Statist.* 22, 3 (09 1951), 400–407.
- [51] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of SOSP*. 233–248.
- [52] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of SOSP*. 49–68.
- [53] John E. Shore. 1980. The lazy repairman and other models: Performance collapse due to overhead in simple, single-server queuing systems. *ACM SIGMETRICS Performance Evaluation Review* 9, 2 (1980), 217–224.
- [54] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *Proceedings of ICDAR*. IEEE Computer Society, 958.
- [55] Roy Patrick Tan, Pooja Nagpal, and Shaun Miller. 2009. Automated Black Box Testing Tool for a Parallel Programming Library. In *Proceedings of ICST*. IEEE Computer Society, 307–316.
- [56] TensorFlow. 2019. TensorFlow Graph Optimizations. <https://research.google/pubs/pub48051.pdf>.

- [57] TensorFlow. 2021. tf.data service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service.
- [58] TensorFlow. 2021. TFRecord and tf.Example. https://www.tensorflow.org/tutorials/load_data/tfrecord.
- [59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*. 5998–6008.
- [60] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of the Second European Symposium on Programming*. North-Holland Publishing Co., NLD, 231–248.
- [61] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 230–243.
- [62] WMT. 2016. 1st Conference on Machine Translation. <http://statmt.org/wmt16>.
- [63] WMT. 2017. 2nd Conference on Machine Translation. <http://statmt.org/wmt17>.
- [64] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
- [65] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Accelerating Human-in-the-Loop Machine Learning. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1958–1961.
- [66] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of OSDI*. 1–14.
- [67] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of HotCloud*.
- [68] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of SOSP*. 423–438.
- [69] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *Proceedings of ICLR*.