TensorSocket: Shared Data Loading for Deep Learning Training

TIES ROBROEK, IT University of Copenhagen, Denmark
NEIL KIM NIELSEN, IT University of Copenhagen, Denmark
PINAR TÖZÜN, IT University of Copenhagen, Denmark

Training deep learning models is a repetitive and resource-intensive process. Data scientists often train several models before landing on a set of parameters (e.g., hyper-parameter tuning) and model architecture (e.g., neural architecture search), among other things that yield the highest accuracy. The computational efficiency of these training tasks depends highly on how well the training data is supplied to the training process. The repetitive nature of these tasks results in the same data processing pipelines running over and over, exacerbating the need for and costs of computational resources.

In this paper, we present TensorSocket to reduce the computational needs of deep learning training by enabling simultaneous training processes to share the same data loader. TensorSocket mitigates CPU-side bottlenecks in cases where the collocated training workloads have high throughput on GPU, but are held back by lower data-loading throughput on CPU. TensorSocket achieves this by reducing redundant computations and data duplication across collocated training processes and leveraging modern GPU-GPU interconnects. While doing so, TensorSocket is able to train and balance differently-sized models and serve multiple batch sizes simultaneously and is hardware- and pipeline-agnostic in nature.

Our evaluation shows that TensorSocket enables scenarios that are infeasible without data sharing, increases training throughput by up to 100%, and when utilizing cloud instances, achieves cost savings of 50% by reducing the hardware resource needs on the CPU side. Furthermore, TensorSocket outperforms the state-of-the-art solutions for shared data loading such as CoorDL and Joader; it is easier to deploy and maintain and either achieves higher or matches their throughput while requiring fewer CPU resources.

CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**; **Concurrent computing methodologies**.

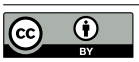
Additional Key Words and Phrases: Data Loading, Data Sharing, Work Sharing, Deep Learning Training, Systems For ML, Hardware Underutilization, Workload Collocation

ACM Reference Format:
Ties Robroek, Neil Kim Nielsen, and Pınar Tözün. 2025. TensorSocket: Shared Data Loading for Deep Learning Training. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 267 (September 2025), 26 pages. https://doi.org/10.1145/3749185

1 Introduction

The process of training a deep learning (DL) model is computationally expensive, mandating the use of powerful accelerators such as GPUs to match the computational needs. However, while the core of the training process can naturally be accelerated this way for many DL models, some training pipelines feature computationally expensive data pre-processing operations such as augmentation

Authors' Contact Information: Ties Robroek, IT University of Copenhagen, Denmark, , titr@itu.dk; Neil Kim Nielsen, IT University of Copenhagen, Denmark, , neilkimn@protonmail.com; Pınar Tözün, IT University of Copenhagen, Denmark, , pito@itu.dk.



ACM 2836-6573/2025/9-ART267
https://doi.org/10.1145/3749185

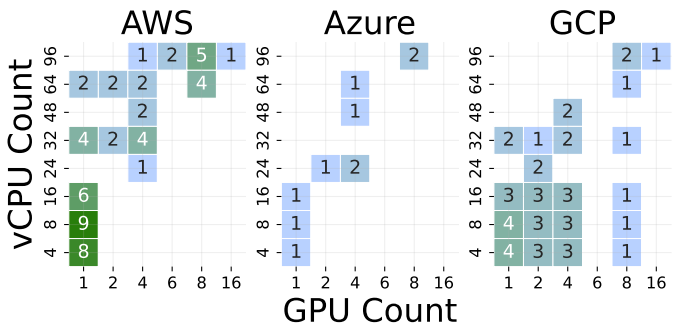


Fig. 1. Cloud instances by vCPU to GPU ratio of Amazon Web Services (AWS) [1], Microsoft Azure [33], & Google Cloud Platform (GCP) [15]. The color scale denotes the number of instances offered with the specified vCPU-GPU pairing.

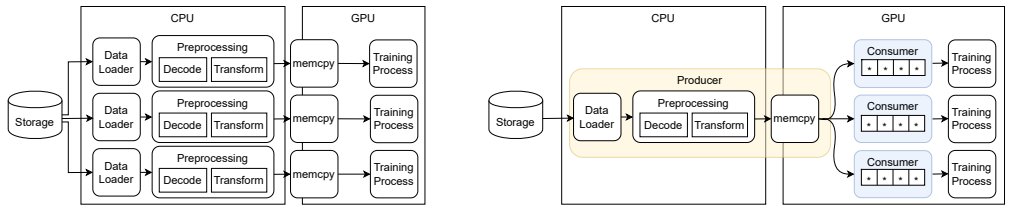
and decoding [9]. Such operations often cause bottlenecks on the host-, or input-side of the training pipeline, where the dominating processing unit is still the CPU [16, 21].

Compute offerings of cloud providers are popular for addressing the computational needs of deep learning training thanks to their on-demand availability. On the other hand, the range of CPU-to-GPU configurations is rather limited, as shown in Figure 1. Furthermore, an instance with a high vCPU to GPU ratio can cost up to 16 times as much as an instance with minimal vCPU count with the same GPU [1]. This high trade-off for the need for more CPU availability, combined with the wide range of DL workloads and their differing computational requirements, lead to several bottlenecks [32, 35, 36]. Specifically, DL training processes that are bottlenecked by their input processing render expensive high-performance accelerators underutilized [22, 64]. In turn, this wastes both CPU and GPU resources. Underutilization of cloud resources is financially wasteful for everyone as compute that has been paid for is not used effectively. Furthermore, it creates an unsustainable carbon footprint in order to address the demand for AI [3, 12, 45, 51, 55, 60].

In practice, it is common to train several models to accomplish a task. The feasibility of DL models heavily depends on finding a model architecture (neural architecture search) or a set of parameters (hyper-parameter tuning) that responds well to the data. This results in model training scenarios that exhibit shared tasks, especially in their input pipelines.

There has been recent work that has proposed ways to leverage such shared tasks [4, 24, 35, 66] and effectively demonstrated the premise of sharing. On the other hand, these works either focus on the cloud scale, paying less attention to the finer-grained cooperation across training processes on the same server, or put a heavy burden on the CPU resources, essentially locking the CPU into being a data feeder for a hardware accelerator instead of facilitating efficient resource utilization.

In this paper, we draw inspiration from these prior works on data sharing across DL training tasks in addition to the work on database systems that leverage the shared work done across concurrent database requests [14, 19, 47, 58]. Our goal is to increase opportunities for work sharing and collocation across DL training jobs while minimizing the hardware resource requirements for such jobs. Rather than viewing these jobs as big monolithic isolated tasks that have to get scheduled exclusively on some CPU and GPU resources, we propose TensorSocket, a novel data loader that is shared across models being trained on the same dataset. TensorSocket turns the *competition* for data and hardware resources into a *cooperation* allowing for more effective workload collocation across concurrent training tasks. As a result, it alleviates resource underutilization and the aggregate costs of DL model training.



(a) Conventional DL input pipeline.

(b) Shared input pipeline with TensorSocket.

Fig. 2. Collocated DL model training (a) without sharing the data loader and (b) with sharing the data loader using TensorSocket. The arrows denote the flow of data from storage to training processes. The shared data loading process (producer) is shown in yellow, and the training processes (consumers) in blue.

Our contributions are as follows:

- We present the design and implementation of a novel data loader, TensorSocket, that enables work and data sharing across collocated models training on the same dataset by detaching the data loading pipeline from the training process. TensorSocket employs techniques such as *batch buffering* and *flexible batch sizing* to support models joining at different times, models of varying complexity, and different batch sizes and order across models as the collocated training process continues.
- We demonstrate how to adopt TensorSocket as an alternative data loader in training pipelines implemented in PyTorch [44] in a plug-and-play manner and evaluate its benefits across a variety of training scenarios (image analysis, audio classification, generative AI) and hardware setups (powerful on-premises hardware, different cloud offerings). Our evaluation shows that TensorSocket enables scenarios that are infeasible without data sharing, doubles training throughput for some scenarios, and, most importantly, when utilizing cloud instances, can halve the cloud costs by reducing the CPU needs by up to four times.
- We compare TensorSocket to the state-of-the-art techniques for shared data loading, CoorDL [35] and Joader [66], and demonstrate that TensorSocket either outperforms or matches their training throughput while requiring less CPU resources and lower deployment effort.

The rest of this paper is structured as follows. Section 2 gives an overview of data loading in DL model training, before Section 3 presents TensorSocket and motivates concrete use case scenarios for it. Then, Section 4 demonstrates the multi-dimensional benefits of TensorSocket over a variety of training pipelines and hardware setups, and Section 5 discusses its further applicability. Finally, Section 6 surveys related work, and Section 7 concludes the paper.

2 Data Loading in Deep Learning

Characteristics and bottlenecks. Figure 2a shows the different training processes that deal with reading and transforming data. In general, a DL training process consists of a model configuration, a dataset partitioned for training and validation, and a training loop. The training process iterates over the full training partition of the dataset for a number of times, specified as *epochs*. The *data loader* is tasked with fetching and readying the data for the model to train on. During each iteration, a *batch* of data is prepared from either disk or memory. This preparation includes fetching, decoding, and transforming the samples. Furthermore, data may be augmented in order to improve the accuracy of models trained on it and their ability to generalize to future unseen data [9, 61].

Decoding, transforming, and augmenting data are all steps that handle and modify data during training and are collectively called *data pre-processing* operations. The more extensive the pre-processing, the higher the computational overhead during training, potentially introducing input-bound bottlenecks. In some real-world training, these operations can amount to half of the energy costs [72].

Furthermore, even though the data used for training is in fast local storage, such as main-memory or SSDs, this data commonly exceeds memory capacity. The result is that the data has to be repeatedly read from disk, swapping out the data that has already been trained on. The damage done by this swapping and OS thrashing depends on how much data can fit in memory as well as the storage backend. This introduces I/O as a potential bottleneck on the critical path [35].

Alleviating the bottlenecks. Data loaders can be configured to alleviate problems that may arise due to inadequate host-side resources that result in the training process idling the GPU [36]. For example, pre-fetching overlaps the work done for data preparation with model training by reading and processing data prior to when it is required during training. Similarly, scaling up the number of workers contributing to reading and pre-processing the data can help hide the input bottlenecks in training. While increasing the worker count does not speed up the pre-processing time of individual batches, it does increase the total batch throughput that can be fed to the model training. On the other hand, a high degree of pre-fetching and parallel workers incur higher host-side CPU utilization and memory consumption, increasing the resource cost and potentially causing contention for resources at the host side.

If the CPU is fully utilized while the GPU is not, another option is to offload pre-processing operations to the GPU. Tools like NVIDIA DALI [41] or techniques like FusionFlow [26] and FastFlow [59] offer methods for resolving data loading bottlenecks that may arise in such scenarios. However, offloading pre-processing to the GPU reserves compute resources that could otherwise be used for training the model itself and should therefore be done with care.

Opportunities for sharing. Developing an effective DL model often requires multiple models to be trained and evaluated in quick succession over the same or similar datasets. Model selection is a common practice where model architectures and training configurations are empirically compared. Hyper-parameter tuning evaluates different hyper-parameter settings, such as learning rate, weight decay, and optimizer settings. These types of tasks are essential for landing on the best performing model [29, 30, 37, 46], as the range of different model architectures and hyper-parameters available increase with the introduction of new models.

Our goal is to propose a mechanism to alleviate the data loading bottlenecks in deep learning training that is complementary to the ones listed above. More specifically, motivated by the repetitive nature of tasks during the model search and hyper-parameter tuning for deep learning training, we would like to leverage the shared data and work required by these tasks. In our solution, TensorSocket, we aim to dedicate the maximum amount of GPU resources to the training loop itself and minimize CPU resource requirements for data loading. Furthermore, by extending existing data loader implementations instead of replacing them, our solution is compatible with other optimizations that can be done for data pre-processing.

State-of-the-art sharing techniques. Motivated by these opportunities for sharing, prior work has also advocated for data sharing in DL [4, 24, 35, 66]. Here, we more specifically detail the proposals that are closest to TensorSocket, which we also compare TensorSocket against in Section 4.

CoorDL [35] is an extension to NVIDIA DALI that coordinates data pre-processing. It is designed for the cluster level and can be used to share data directly between training processes. It can distribute a batch of data to any number of training processes in the cluster. Once all training processes are done with the data, CoorDL continues to the next batch. CoorDL's focus on the cluster

level and design around DALI, however, surfaces some limitations. Firstly, CoorDL is designed for models training on separate GPUs and cannot utilize leftover GPU compute power to train multiple models on a single GPU in a collocated fashion. Secondly, CoorDL performs poorly when the models that train simultaneously are not very similar, as in this case the models that are faster have to wait for the slower ones. This rigid design also prevents CoorDL from being deployed as a live service with training processes arriving at different moments. Thirdly, CoorDL requires the data loading and pre-processing pipeline to be implemented in DALI, requiring substantial extra engineering if the pipeline implementation is not already using DALI. Finally, the existing CoorDL codebase [34], and the DS-Analyzer project around it, is written in Python 3.6, which has been deprecated since 2021 by PyTorch and reached Python end-of-life in December that year.

Joader [66] is a standalone shared data loading solution that supports sharing over multiple datasets. A server is configured in which all datasets have been registered. Training clients, also known as jobs, then communicate with this server using RPC. Joader reduces CPU utilization by having one server that does the data loading and pre-processing for multiple jobs, even if those jobs require datasets that are not identical but just overlap. In the scenario where models train on the same base dataset, this allows models to train at different speeds and still share part of their data loading. Joader achieves this flexibility across different datasets through a technique called dependent sampling. However, this sampling also comes with an important drawback; it requires intersection calculations to run at every iteration, which adds a high CPU cost. Furthermore, Joader only has a proof-of-concept implementation [67], which is written in Rust making it very difficult to adapt existing deep learning training codebases to. Datasets have to be converted to the proprietary format that Joader expects and only image data with specific parameterization is supported. There is no support for a variety of image pre-processing operations other than the pipeline that is hardcoded in Rust. Finally, data reaches the training jobs as NumPy matrices which require tensor conversion and host-to-device transfer, and batching during training is not supported, which are all detrimental to data loading and training performance.

Next, we delve deeper into TensorSocket.

3 TensorSocket

This section presents TensorSocket[1], our shared data loader that capitalizes on the redundancy among similar but separate data loaders of collocated training processes. We propose a solution to inefficient hardware utilization and resource wastage by minimizing redundant work and hardware resource consumption while ensuring that downstream training processes are not impacted. By detaching the data loading pipeline from each training process, we can merge several training processes into a single data loading pipeline. This single data loader can expose the training data for use in each collocated training process.

3.1 Overview

Figure 2b illustrates how our shared data loader works. The figure shows an example of three collocated training processes, where, in yellow, the tasks of the *producer* are shown, and in blue, the training process with the *consumer* is shown.

At its core, our system is composed of a *producer* and a number of *consumers*. The *producer* holds a single data loading process along with some bookkeeping, while the *consumers* iterate on data sent by the producer. This producer-consumer workflow can be swapped in place of DL training framework-specific `DataLoader` objects, such as the PyTorch DataLoader.

[1]https://github.com/itu-rad/tensorsocket

```python
# train.py (without TensorSocket)
data_loader = DataLoader(dataset)
for batch in data_loader:  # Loop over dataset
    ...  # Model training iteration
```

(a) Conventional training script without TensorSocket.

```python
# producer.py
data_loader = DataLoader(dataset)
producer = TensorProducer(data_loader)
for _ in producer:  # Loop over dataset, send data
    pass
producer.join()  # Clean-up
```

(b) TensorSocket producer script.

```python
# consumer.py (or train.py)
consumer = TensorConsumer()
for batch in consumer:  # Receive & loop over data
    ...  # Model training iteration
```

(c) TensorSocket consumer script.

Fig. 3. Example TensorSocket implementation requiring minimal code changes training for a single epoch. The top listing depicts a standard model training script. The data loader is split from the main training process, creating a producer process and a consumer process.

Given that the producer is the owner of the data loading pipeline as well as responsible for data generation, it would be regressive to copy each batch of data into every collocated training process. Instead, once the producer has prepared a batch of data, the consumers are all given the location of the data batch to use in their respective processes. Every consumer has a queue that holds up to a few of these locations. This introduces some flexibility to prevent training hiccups (e.g., a training process falling behind during a batch) from interfering with the other training processes.

3.2 Design & Implementation

TensorSocket is a library built around PyTorch as it is the most widely used deep learning framework available. A crucial limitation common in other data sharing solutions [35, 66] discussed in Section 2 is that the solution itself implements the complete data loader. This limits the adoption as it requires the user to adapt to the specific codebase, in addition to the library version dependencies, of that solution. We prevent these shortcomings by setting TensorSocket up as a wrapper around data loaders implemented in PyTorch instead of a separate data loader itself. This ensures out-of-the-box compatibility with any PyTorch training script, e.g. ones that use PyTorch or Hugging Face data loaders. While this means that the current implementation is PyTorch specific, implementations of similar wrappers for other frameworks such as TensorFlow won't be prohibitive as these frameworks generally follow the same principles for data loading.

Figure 3 shows an example usage of TensorSocket. Using TensorSocket involves copying the data loading logic of a training script to a separate producer script and adding a consumer

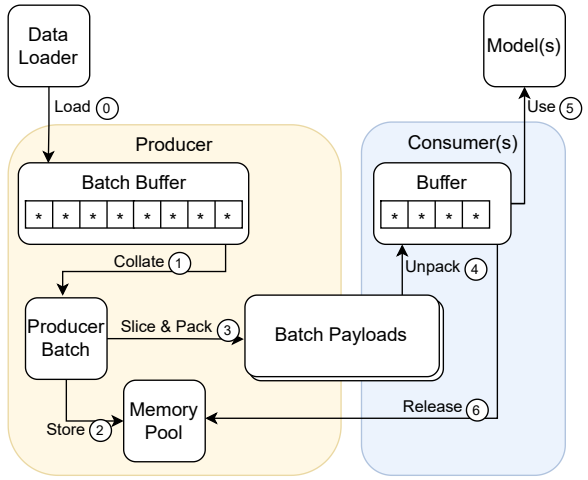


Fig. 4. Overview of TensorSocket's internals. Arrows denote PyTorch tensor and pointer operations.

to the training script.[2]The PyTorch *data loader* is isolated to a different process and wrapped in a *TensorProducer*, which is then iterated over similar to a conventional data loader. The training process receives the data automatically by iterating over a *TensorConsumer*.

Figure 4 illustrates TensorSocket's key components and operations, which we detail next.

3.2.1 Producer. TensorSocket splits data loading from training. The data loading producer becomes a server that can dynamically process and serve incoming consumer clients. A TensorSocket producer instance is initialized with a data loader object. It is exposed as an iterator that itself iterates over the nested data loader it is initialized with. The producer repeatedly requests ⓪ the contained data loader to fetch the data from disk. It will pause iterating over the data loader whenever the consumers currently do not need extra data, notified by communication between the producer and consumers. This can also be the case when there are no consumers present as in this case there is no need for any data loading.

3.2.2 Consumer. Abstracting away the data loader into the producer allows the consumer to be lightweight. As TensorSocket's producer and consumer directly replace the data loading batch iterator in the training script, the consumer similarly takes the form of an iterable object that fetches new data ⑤ whenever available. If there is no data available, the consumer halts and waits. This design makes for a minimal one-line swap in training script code.

3.2.3 Communication. The communication between the producer and consumers is done using ZeroMQ sockets. ZeroMQ is an open-source library that allows for sharing atomic messages with low latency. In TensorSocket's case, we use ZeroMQ sockets for communication between the producer and consumers using a PUB/SUB pattern [70]. This is a multicast pattern that is flexible and scalable, allowing one producer to connect to multiple consumers.

The data is shared over these sockets by the producer. Once a consumer has fetched a readied batch of data for use in training, it notifies the producer by sending an acknowledgment message back to it, allowing for the producer to continuously keep the consumers fed new data. When multiple consumers are training simultaneously, the producer will wait for an acknowledgment

[2]The full example can be found in our code repository at https://github.com/itu-rad/data-sharing.

from all consumers before releasing a piece of data. This ensures that all consumers have iterated on a batch before it is deleted.

Whenever data is shared with a consumer, the producer will store ② a reference to that data. Once a consumer has finished a batch and moves on to the next, it will notify ⑥ the producer. The producer will release the associated memory when all consumers are finished with it, allowing the producer to ready the next batch.

Depending on the dataset and models, consumers may take a long time to go through their training data batches. In order to be continuously aware of consumers, producers send and receive heartbeat messages from their consumers over a different socket. The producer will detach from consumers that it has not received a heartbeat from in a while.

3.2.4 Data sharing. Data sharing is at the heart of TensorSocket. If the data sharing implementation is not efficient, it will become a bottleneck that would outweigh the benefits of sharing. There are two ways data sharing can be done.

Some solutions share the data bytes directly with the training processes via inter-process communication [66]. This surfaces some concerns regarding sharing efficiency and data duplication. Increasing the size of the training data directly increases the size of the network messages in such a solution, potentially leading to slowdowns. Furthermore, while the data loading itself is unified, the resulting data is then duplicated for every client, spiking memory consumption and data movement costs.

In TensorSocket's case, we share small packets containing pointers to the data instead of the data itself. Following our earlier design philosophy, we heavily borrow from PyTorch's existing data management. PyTorch introduces Tensor objects which are data matrices, similar to NumPy matrices, that contain all data that PyTorch runs on. While PyTorch is most commonly known as a Python library, much of the internals such as Tensors are defined in C++. We can use this to our advantage by extracting the data pointer and other necessary information. This pointer is then shared by the producer to the consumers, which in turn use this data to reconstruct the Python tensor object without any data duplication.

PyTorch as a library is heavily optimized for running with multiple threads as data workers and on multiple GPUs and machines. The tensor implementation contains methods for dealing with concurrency and distribution, including tensor rebuilding. By using this somewhat hidden PyTorch functionality we can pack ③ and unpack ④ batch payloads of tensors.

Tensors in PyTorch hold data that can be on the host system (i.e., CPU), but can also be put on the GPU. Transferring data to the GPU is a costly operation. By inheriting PyTorch tensor methods, TensorSocket can reconstruct tensors on both the CPU and GPU. This means that the producer can put the data on the GPU once, after which all consumers collocated on that GPU can access it. Furthermore, we can rely on PyTorch's tensor management for our shared data. Tensors are kept in memory as long as any of the producers or consumers hold a reference to it.

Finally, using the capabilities of frameworks such as PyTorch gives us access to fast GPU-to-GPU communication methods like NVLink while sharing the tensors. This allows TensorSocket to efficiently share data even if the models train on different GPUs. Data is loaded on one of the GPUs after which it is directly shared to the other GPUs with direct NVLink interconnects.

3.2.5 Synchronization. Considering that consumers may train different models and training is stochastic, we should expect that consumers do not process a batch in identical time. This led us to introduce a batch buffer on the consumer side. Instead of actively requesting the next batch on iteration, consumers can hold up to N batches (i.e., pointers to the tensors of batches) in their buffer. This allows for the producer to actively pre-fetch data, and for the consumers to drift at most N batches apart. Both the buffering and the pre-fetching hide the latency of various parts of

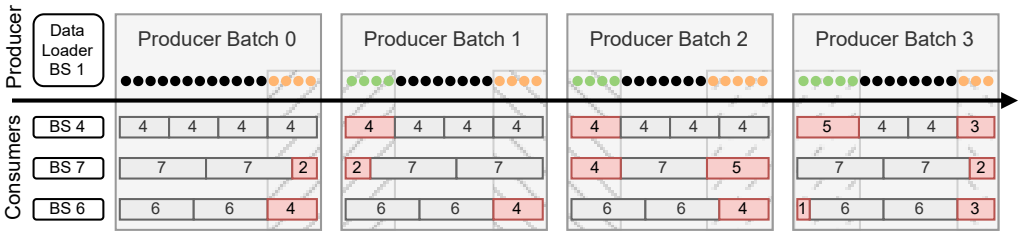


Fig. 5. TensorSocket with flexible batch sizing training consumers with different batch sizes. The batch sizes have been divided by their common multiple for clarity. The producer collates 16 data loader batches into a single producer batch and then splits the producer batch into appropriate batches per consumer. These consumers request batches of 4, 7, and 6 respectively. These batch sizes do not all divide 16 cleanly, which results in a fraction of the data having to be repeated. The repeated amount of data points can vary between batches; batch 1 and 2 repeat four while batch 3 repeats five.

the data loading pipeline. When designing this queue, we experimentally found that a buffer as small as two batches is enough to provide maximum training throughput while training similar tasks. Increasing the buffer size can be beneficial when training processes fluctuate more widely in their speed. It should be noted that increasing the buffer size does increase the GPU memory requirement of the system as more batches need to be kept on the GPU simultaneously.

While the buffering scheme described above relaxes the conditions for data sharing among the consumers, TensorSocket by design targets scenarios where the consumers train on the same dataset at or around the same time. Therefore, the consumers have to be balanced in terms of their training speed even if they do not process data in identical time. Whenever a process trains too fast, the consumer iterator will automatically halt as there is no data available. This frees up resources for other consumers to make up for the difference. In this case, the GPU can be time-shared among the consumers. Modern GPUs enable this through services such as NVIDIA multi-streams or Multi-Process Service (MPS) [50]. Especially, MPS allows efficient time and spatial sharing of the GPU. The GPU sharing and inclusion of the consumer buffer allow to balance the load of the consumers automatically, resulting in higher training throughput and GPU utilization. As a result, TensorSocket is able to train models efficiently, even when models diverge in terms of complexity. GPU resources are allocated in such a way that the consumers go through the data at the same rate.

Since the TensorSocket producer acts as a server producing data for the consumers, we also need to account for consumers that connect at different times. Once an epoch has already started, any new consumers fall behind and have to wait for the next epoch to start training. We introduce a leniency measure called rubberbanding to provide a window for consumers to join training. If a consumer joins before 2% of the dataset has been iterated on in an epoch, the producer will halt all other consumers to let that consumer synchronize. The percentage of the dataset that serves as the cutoff point is configurable. We found that rubberbanding is an effective method for allowing users to spawn multiple consumers without fear of them not joining fast enough.

3.2.6 Flexible Batch Sizing. By default, TensorSocket restricts consumers to training on the same batches at the same time, requiring all training processes to utilize the same batch size. We introduce an alternative mode of operation to relax these constraints. Under *flexible batch sizing*, consumers tell the producer what batch size the producer should supply them. The producer supplies batches of different sizes to their respective consumers using the slicing properties of PyTorch tensors.

Enabling flexible batch sizing is done by setting batch size arguments when initializing the producer and the consumers, lines 2 in Figures 3b and 3c.

TensorSocket directly share the pointers to the batch tensors under default operation. With flexible batches, the producer and consumer batches are no longer the same. Instead, the producer readies *producer batches*, which are larger than the batch sizes of the consumer. The producer batch is allocated as a continuous block of memory on the GPU. Pointers to appropriate sliced batches are sent to the consumers. This way, the consumers can train on varying batch sizes, while still going through the data at the same rate.

Figure 5 illustrates flexible batch sizing in practice. In this example, the producer serves consumers with requested batch sizes of 4, 7, and 6. The producer collates ① the data it receives from the data loader into producer batch sizes of 16. These batches are then sliced ③ appropriately for every consumer.

When the consumer batch sizes do not cleanly divide the producer batch size, a share of data has to be repeated between producer batches. The amount of repeated data for a producer batch can be up to (not including) the largest consumer batch size $\max\{b_c\} - 1$, as otherwise more consumer batches would have fit in that producer batch. The maximum repeated share divides this by the size of the producer batch $\frac{\max\{b_c\}-1}{b_p}$. As a large producer batch minimizes repetition, we recommend having it at least twice as large as the largest consumer batch, making this share never exceed 50%.

We argue that data repetitions are an acceptable tradeoff for flexibility as TensorSocket reduces data movements considerably w/o flexible batching enabled. Repetitions only occur if the consumer batch sizes do not cleanly divide the producer batch size, with Figure 5 depicting a challenging scenario. In practice, we expect the batch sizes requested by consumers to be in powers of two.

3.2.7 Batch Order. TensorSocket's unified data loading results in consumers training on the data points in the same sequence. For some tasks, such as hyperparameter tuning or model architecture search, it may be beneficial to include more variation by loosening this order requirement.

With producer batches, we introduce two ways to add variety in how the consumers go through the data. Firstly, we can utilize the flexible batch sizing mode and add offsets to the consumers, causing their batches to be carved from the producer batch with different breakpoints. This results in batches that are not identical between consumers. Secondly, we can shuffle the order of batches within a producer batch. If a consumer splits a producer batch in, e.g., 4 batches, it can then go through these in random order. This results in data being visited in different order between consumers. If desired, both techniques can be combined. The amount of variation in visiting then becomes a product of producer batch size, at the cost of GPU memory. Similar to flexible batch sizes, these two techniques can be activated when initializing the producer.

3.3 Use Case Scenarios

We go over a few use case scenarios that would benefit from TensorSocket, and how our implementation makes them possible.

3.3.1 Centralized Always-Available Loading. When exploring a dataset, it is invaluable for users to seamlessly start and stop training jobs. TensorSocket allows for a high degree of flexibility and stability by abstracting away the data loading from the training job itself. Once TensorSocket is running on a server, consumers can come and go as they please. The consumers ping the producer with heartbeats such that the producer knows how many training processes are active at a given time. Consumers may either join training at any point in an epoch or wait until a new epoch starts, depending on the configuration. In the latter case, the producer buffers the first few batches at the start of a new epoch to provide a short window in which new consumers are accepted. In such a

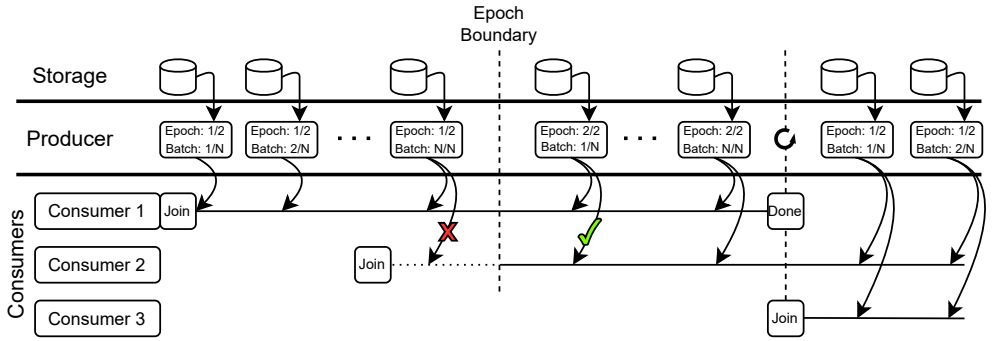


Fig. 6. Illustration of how TensorSocket allows the producer to run continuously and supply new consumers with batches.

case the producer will halt the other consumers in order for the new consumer to quickly iterate over the buffer.

Figure 6 shows an example of how new jobs, represented as consumers, are handled by our shared data loading system. In the example, consumer 2 joins in too late during the first epoch, having to wait until the second epoch starts. Consumer 3 joins in at an epoch boundary and immediately starts consuming data batches.

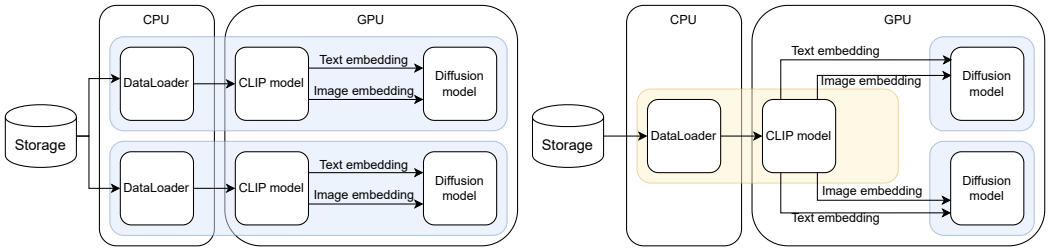
3.3.2 Native Inter- and Intra-GPU Sharing. One way of increasing the hardware utilization for DL training is training multiple models at the same time, i.e., *workload collocation* [10, 38]. Workload collocation can improve training throughput [13, 50, 54, 63] when the hardware resource needs of the individual training processes are not large enough to utilize all the available CPU and GPU resources [7, 27] or are bottlenecked by their input pipelines [8]. This often reduces the aggregate runtime when more than one model has to be trained, even though the training time per model usually goes up due to not having exclusive access to the GPU.

TensorSocket allows for sharing data on a single GPU between any number of consumers, boosting efficiency while collocating models. This additionally reduces redundant memory consumption, as the memory requirement for training processes is lowered due to not needing a data loader for each separate training process.

Our implementation also supports collocation across multiple GPUs. Data batches are seamlessly moved between GPUs when the data is needed on a different device. Thus, TensorSocket is able to leverage GPU-GPU interconnects with lower latency and higher bandwidth than CPU-GPU interconnects, as also mentioned in Section 3.2.4. We evaluate this scenario in Section 4.2.

3.3.3 Sharing for Mixed Workloads. Mixed workloads that train models at different speeds, for instance when model complexity differs significantly, can be difficult to optimize from a data loading perspective. TensorSocket supports mixed workloads by allocating more hardware resources to heavier training processes than lighter processes. We bound the models to be within a certain amount of batches from each other (as described in Section 3.2.5). The result is that slower models are sped up and lighter models are slowed down so that all models traverse the epoch in the same amount of time. We evaluate this scenario in Section 4.5.

3.3.4 Sharing Generative Tasks Online. In some cases, it is beneficial to move more tasks to the *producer*. For example, the training of some generative models requires pre-computed data representations in the form of embeddings for training the diffusion prior. These embeddings are usually



(a) Conventional deep learning input pipeline. (b) Shared input pipeline with TensorSocket.

Fig. 7. Sharing example for generative DALL-E image generation models.

generated before training [48], *offline*, but can be generated on the fly, *online*, via a model inference task on the GPU. Online generation offers flexibility for new data and circumvents having to use extra disk space to store the embeddings generated a priori but is more taxing on the hardware resources while training. TensorSocket can move the data loading operations and the embedding generation task to its *producer* as it is essentially part of the data loading pipeline. This minimizes the computational footprint on not just the CPU but also the GPU when sharing. Figure 7 illustrates this scenario for the DALL-E model with and without sharing, and Section 4.4 evaluates it.

4 Results

We now quantify the expected benefits of the data and work sharing enabled by TensorSocket. Our evaluation aims to answer the following questions:

- What is the impact of TensorSocket on training efficiency?
- What are the cost savings TensorSocket can provide?
- How do the benefits of TensorSocket vary across different hardware setups and machine learning pipelines?
- How does TensorSocket compare to state-of-the-art data sharing solutions for model training?

To answer these questions we evaluate TensorSocket in a variety of scenarios inspired by the use cases listed in Section 3.3.

Application	Model	Dataset
Image Classification	RegNetX 002 RegNetX 004 ResNet18 MobileNetV3-Small 0.75 MobileNetV3-Large 1.00	ImageNet
Audio Classification	CLMR	LibriSpeech
Image Generation	DALL-E 2 (Diffusion Prior)	CC3M
LLM Fine-Tuning	Qwen2.5 0.5B	Alpaca

Table 1. Evaluated models and datasets.

4.1 Experimental Setup

Use cases. We seek to demonstrate the value of TensorSocket on a range of workloads that benefit from different degrees of shared data loading. We therefore evaluate DL models from the domains of computer vision, audio classification, image generation, and natural language processing. We investigate a wide range of popular computer vision models from TIMM [65], use CLMR as our audio classification workload [53], source the image generation model from a well-known and tested PyTorch implementation of DALL-E 2 [48, 62], and evaluate Qwen2.5 [68], a popular LLM from Alibaba, using TorchTune [57]. The datasets chosen for our evaluation are ImageNet-1K [11], LibriSpeech [43], Conceptual Captions (CC3M) [52], and Alpaca [56], respectively. Table 1 lists the evaluated models and the corresponding datasets.

Hardware setup. We evaluate the scenarios on multiple hardware configurations. Table 2 details the cloud instances and on-prem servers used in our evaluations. The cloud configurations allow for testing the CPU utilization benefits of TensorSocket by varying the amount of vCPUs while keeping the GPU count the same. The A100 server features multiple GPUs allowing us to evaluate data sharing when each GPU trains a separate model. Finally, the H100 server's GPU is large enough to collocate multiple DALL-E 2 training tasks. As a result, the variety of the hardware setups allow us to evaluate the impact of TensorSocket on different environments, use case scenarios, and collocation options.

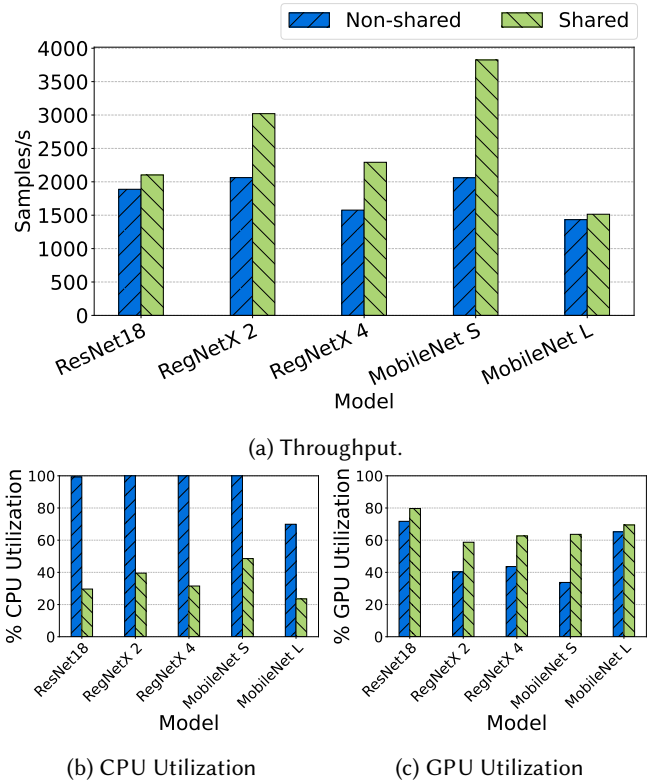
Modern GPUs support different primitives for workload collocation on a single GPU [49]. In this work, we utilize NVIDIA Multi-Process Service (MPS) [42], unless stated otherwise, since it is shown to allow flexible collocation while exhibiting high performance [50]. Processes executed under MPS share both GPU memory and the streaming multiprocessors (SMs). The MPS daemon automatically handles the sharing of the SMs across the collocated processes.

Metrics. We train the same models on the same dataset without changing the learning process and thus without impacting accuracy. Instead, we focus on the training speed and hardware utilization as the performance metrics. We quantify the training speed via *samples/s*, the amount of training samples processed by the training loop per second. *CPU Utilization* is measured via `top` [25]. We measure *GPU Memory Usage* with SMI [39] and *GPU Utilization* with `dcgm`'s [40] SM Activity, which is shown to illustrate a finer-grained view on GPU utilization compared to other readings from the `dcgm` tool [69]. Finally, we report average data movement for disk *(I/O)* via `iostat` [20] and for *PCIe* and *NVLink* via `dcgm`.

Training runs. All experiments are run with Python 3 and PyTorch 2, and the latest versions of the respective model repositories as of writing, using the radT platform [49]. We ran everything twice to validate the results and stick to the default model parameter settings as specified by the model repositories. We set the total number of data-loading workers across the collocated workloads

Instance	(v)CPUs	GPU	VRAM	Cost
H100 Server	24	H100	80 GB	-
A100 Server	128 (*48)	4x A100	4x 40 GB	-
AWS g5.2xlarge	8	A10G	24 GB	$1.212
AWS g5.4xlarge	16	A10G	24 GB	$1.624
AWS g5.8xlarge	32	A10G	24 GB	$2.448

Table 2. On-prem servers and cloud instances used in evaluation. Costs are on demand per hour costs for corresponding cloud instances [2]. The A100 server is limited to a max of 48 cores to mimic Azure offerings with A100 GPUs (Figure 1), which provide a 12:1 vCPU to GPU ratio.



(a) Throughput.

(b) CPU Utilization

(c) GPU Utilization

Fig. 8. Image classification training on the A100 server with 4-way collocation, where each GPU has one instance of the same model training, w/o sharing via TensorSocket.

to the number of available CPU cores (or to 48 for the A100 server as explained in Table 2). These workers are split equally among the training processes in the experiments with conventional data loading (no sharing). Finally, we enable GPU pre-fetching for our baselines as TensorSocket by default prefetches data to the GPU.

4.2 Image Classification

Impact on throughput and hardware utilization. We first evaluate TensorSocket's impact on the training efficiency over the most basic collocation scenario, where the same model is trained on a separate GPU available on the server (e.g., an hyper-parameter tuning scenario). We train a variety of image classification models, as listed in Table 1, on ImageNet. ResNet18, RegNetx 4, and MobileNet L are more demanding models to train, while RegNetX 2 and MobileNet S are smaller. Among our hardware setups (Table 2), the A100 server is the only one with multiple GPUs available. With 12 CPU-cores per GPU, this scenario additionally showcases TensorSocket's benefits when the CPU-to-GPU ratio is too low to fully utilize the whole system, which is common among lower-price cloud offerings (Figure 1).

Figure 8 reports the per-model training throughput and hardware utilization. In the case of no shared data loader, the training script runs separately on each GPU. When using TensorSocket, we direct the producer to GPU 0 and launch a consumer on each of the four GPUs. The producer and the consumers can communicate via NVLink, which is available on this server across the GPUs.

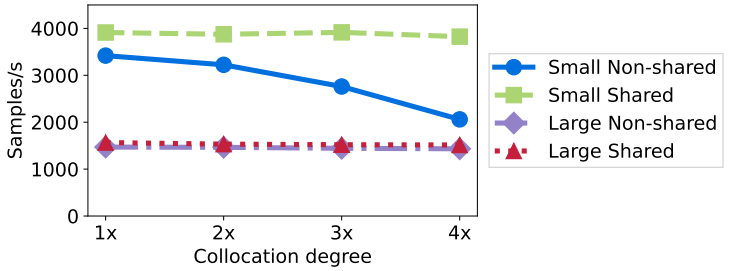


Fig. 9. Per-model training throughput of MobileNet Small and Large with increasing degree of collocation on the A100 server. Each collocated model is trained on a separate GPU.

TensorSocket increases the training throughput across all workloads. In MobileNet S's case the throughput almost doubles, whereas for models such as ResNet18 and MobileNet L the increase ranges from 5% to 10%. The degree of improvement correlates with the computational complexity of the models.

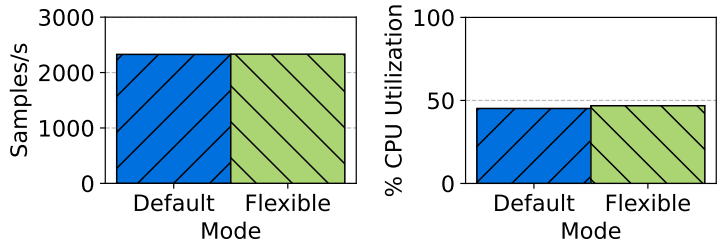
For models such as ResNet18, RegNetX 2, RegNetX 4, and MobileNet S, Figure 8 reveals that under traditional data loading the CPU is fully utilized while the GPUs are not. This underlies that the CPU becomes the bottleneck causing underutilization of the GPU resources. Sharing via TensorSocket resolves this bottleneck by reducing the stress on the CPUs while achieving a higher GPU utilization in addition to the throughput benefits.

On the other hand, for models that are not CPU-bound such as MobileNet L, TensorSocket provides marginal benefits on the throughput and GPU utilization. However, it frees up 70% of CPU resources. The savings in CPU resources can allow for collocating additional workloads on the CPU-side in an on-prem setting and cutting the costs in a cloud setting.

Data movement. Table 3 reports the disk I/O, PCIe, and NVLink traffic in addition to GPU memory usage for training four MobileNet L models from Figure 8. As TensorSocket's producer loads the data once and broadcasts it to all the consumers after preparing it, it is able to greatly reduce the disk I/O and PCIe bandwidth utilization compared to the baseline, replacing it with NVLink communication instead. There is a small increase in VRAM usage on the producer GPU. This is due to TensorSocket 's default configuration holding some extra data ready on the GPU for e.g. buffering.

	GPU	I/O	PCIe	NVLink	VRAM
Baseline	0	613 MB/s	270 MB/s	-	8.5 GB
	1		267 MB/s	-	8.5 GB
	2		268 MB/s	-	8.5 GB
	3		267 MB/s	-	8.5 GB
Shared	0 (Both)	161 MB/s	286 MB/s	-	9.8 GB
	1 (Cons)		23 MB/s	267 MB/s	8.5 GB
	2 (Cons)		24 MB/s	269 MB/s	8.4 GB
	3 (Cons)		23 MB/s	268 MB/s	8.4 GB

Table 3. Disk I/O, CPU-to-GPU PCIe, and GPU-to-GPU NVLink traffic and GPU memory usage for four MobileNet L models training on separate A100 GPUs.



(a) Throughput (b) CPU Utilization

Fig. 10. Training three MobileNet Small models on the H100 server with a batch size of 128 (default) versus batch sizes of 128, 192, and 224 (flexible), following the proportions of the example distribution featured in Figure 5.

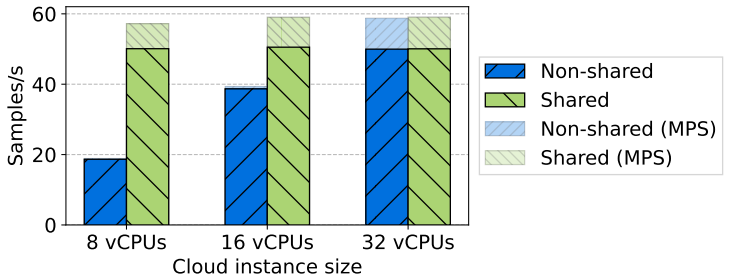


Fig. 11. Samples/s per collocated training on AWS G5 Instances for CLMR - 4-way collocation on the same GPU using MPS and multi-streams across different vCPU counts w/o data sharing via TensorSocket.

Degree of collocation. We also provide a sensitivity analysis with respect to varying levels of collocation in Figure 9. For this we use both MobileNets as they are the models that exhibit the most and least benefit from TensorSocket. TensorSocket yields a throughput increase for both the small and large MobileNet in all configurations. On the other hand, increasing the amount of models trained simultaneously has little effect on the large model as the CPU is not the limiting factor (Figure 8). Conversely, scaling up collocation for the small MobileNet relies on TensorSocket to maintain high throughput.

Flexible batching. Finally, we evaluate the effect of flexible batches on training speed and CPU utilization. Figure 10 compares training three models with the same batch size to training three models with differing batch sizes. Flexible batching sustains training throughput while only incurring minimal CPU overhead to orchestrate the different batches required by the consumers.

4.3 Audio Classification

We train the CLMR audio classification models in a 4-way collocated fashion on different AWS instances in order to showcase the impact of data and work sharing on host-side resources. The results are shown in Figure 11.

For this setup, in addition to MPS-based sharing, we evaluate running collocated processes as a separate GPU stream, which provides more restricted sharing, but may sometimes be the only option in a shared hardware setup such as the cloud. The blurred parts of the bars, therefore,

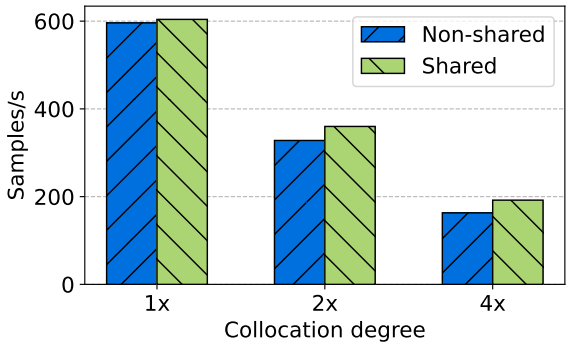


Fig. 12. Samples/s per collocated online training of DALL-E on the H100 server with 1-, 2- and 4-way collocation and w/o sharing via TensorSocket.

highlight the additional throughput benefits of MPS-based sharing over multi-streams. Regardless, TensorSocket is compatible with any form of GPU sharing primitive.

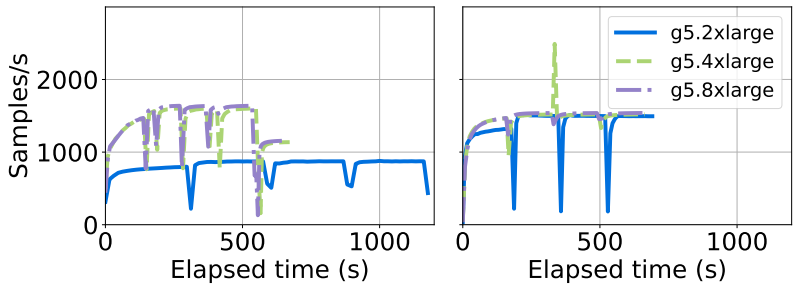
From Figure 11, on the machine with the highest number of vCPUs, the workload with and without sharing achieve the same throughput. This indicates that the number of vCPUs necessary to sustain the GPU throughput for this workload is met. Without TensorSocket, however, the smallest instance size of 8 vCPUs performs drastically worse than the largest with 32 vCPUs. TensorSocket effectively reduces the amount vCPU requirement by 75%. The result is that under TensorSocket all three sizes of cloud instances achieve high training throughput. Based on the costs reported in Table 2, this leads to cloud cost savings of about 50%.

4.4 Image Generation

As mentioned in Section 3.3, TensorSocket can be used to share not just tasks on the CPU but also on the GPU. We analyze such a pipeline using the DALL-E 2 image generation (diffusion) workload. When training DALL-E 2, data passed to its training process must pass through a CLIP model (Section 3.3.4 and Figure 7). CLIP translates the input data into a representation that can be used by the trained model. The CLIP model can be seen as a model with frozen weights when training the diffusion model. This essentially boils down to running inference tasks on the GPU as part of the data preparation process for DALL-E training.

With TensorSocket, we can move the CLIP model inference to the producer of the shared data loader. In this scenario, we aim to showcase how TensorSocket can also reduce redundancy in the computational footprint on GPUs. By only needing a single CLIP model, we can collocate multiple DALL-E 2 diffusion models without running multiple instances of CLIP inference. This workload is carried out on the H100 Server machine, as it is capable of supporting 4-way collocation of diffusion model training.

Figure 12 shows the impact. Since the H100 server has enough CPUs to feed the one GPU (Table 2), this evaluation scenario is not CPU-bound. Nevertheless, we observe a speedup over non-shared operation under collocation. Under 2- and 4-way collocation TensorSocket is 10% to 15% faster in aggregate throughput than without when running online training. The throughput per individual training process gets reduced as expected, since in this setup the GPU is highly utilized even without collocation due to the demanding model. This shows that TensorSocket can enable data and work sharing not only on CPUs but also on GPUs.



(a) Non-shared (b) Shared

Fig. 13. Runtime and aggregate training throughput of mixed workloads (RegNetX 2 and RegNetX 4) on AWS G5 Instances with and without data sharing via TensorSocket.

4.5 Model Selection

In addition to the evaluations that revolved around specific domains, we show how our shared data loader supports mixed workloads, which is useful for model selection. As mentioned in Section 3.3.3, the design of our shared data loader supports training of a mixed set of models through reallocating GPU resources between training processes. We evaluate model selection by collocating two different model training processes on different AWS cloud instances. As the training speed of the models differs, we report on the aggregate training throughput. Figure 13 shows the results for a mixed workload consisting of a collocated RegNetX 2 and RegNetX 4. The runs on the left use conventional non-shared data loading whereas those on the right use TensorSocket. For the g5.8xlarge and g5.4xlarge AWS instances, the CPU does not constitute a bottleneck, and we therefore do not see substantial throughput gains by sharing. However, we are able to closely approximate the throughput of these larger instances with the smaller g5.2xlarge instance when sharing. In contrast, the workload throttles heavily on the small instance when not using shared data loading. For the g5.2xlarge instance, sharing is therefore not only strictly necessary for running this workload efficiently but also able to deliver almost the same throughput at half the instance cost, as seen in Table 2.

4.6 Language Model Fine-tuning

Given the prevalence of Large Language Models (LLMs) today, the final use case we evaluate TensorSocket on is an LLM fine-tuning task. Unlike prior use cases, LLM training is generally bound on GPU compute and memory instead of data loading and CPU utilization. For our LLM fine-tuning task, we use the A100 server to train two Qwen2.5 models using TorchTune on the Alpaca dataset, which means TensorSocket wraps around a Hugging Face data loader instead of a standard PyTorch one. We take the default fine-tuning recipe of TorchTune and set the batch size to 8. The non-sharing scenario, *baseline*, runs one model per A100 GPU. The TensorSocket run has the producer on a separate A100 GPU, with the two models training on GPU 1 and 2. This way we can observe the data traffic (PCIe and NVLink) and GPU memory use separately for the producer and the consumers.

The results are shown in Table 4. Due to LLMs being GPU-bound, TensorSocket only slightly increases the training speed for this use case. The LLMs being trained require a low amount of PCIe communication (48 MB/s on average) even for the baseline. TensorSocket's producer reveals that only 341 KB/s is required to run the data loader. The NVLink utilization, which is used by TensorSocket to directly communicate training data between GPUs, is only ~150KB/s. This data

loading is insignificant compared to the rest of the PCIe traffic. Finally, the GPU memory utilization for the models training on GPUs 1 and 2 is the same for the baseline and the shared case, indicating that there is no GPU memory overhead for the TensorSocket consumers. There is a small 1.5 GB memory requirement for running the producer on GPU 0. This is a combination of CUDA overhead and TensorSocket with the default configuration of buffering data, similar to results in Table 3.

4.7 Comparison to other sharing techniques

After having assessed the value of shared data loading via TensorSocket in deep learning training, we compare TensorSocket to state-of-the-art methods, CoorDL [35] and Joader [66], that achieve data loading speedups via sharing. [3]

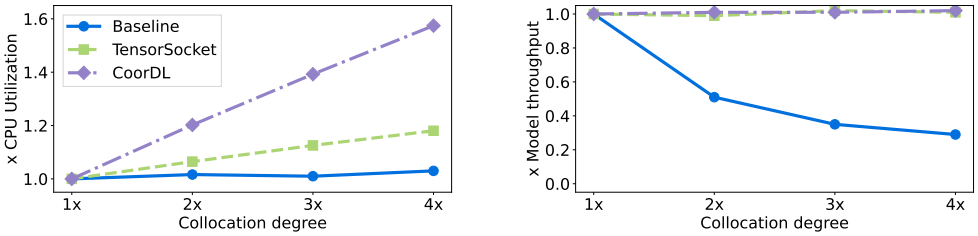
Comparing to CoorDL surfaces a couple of challenges due to the age of the library [34]. CoorDL has been designed as a plugin for NVIDIA DALI and is written for Python 3.6. This version of Python, however, is deprecated by PyTorch since 2021, which means modern versions of the framework are not compatible with it. TensorSocket is incompatible with PyTorch 1 as the deep learning framework made sweeping changes with the introduction of PyTorch 2. This complicates establishing a fair comparison between CoorDL and TensorSocket. Nevertheless, we run CoorDL using the evaluation script provided by the authors of the original work [35] to run it as efficiently as possible. We adjust the parameters for TensorSocket accordingly. This means that automatic mixed precision is disabled, the batch size is set to 512, and there are 4 data loading workers. We also choose ResNet18 as the evaluated model following CoorDL's evaluation. Finally, while reporting the results, we normalize the per-model training throughput and hardware utilization values by dividing them by the values achieved by single model training (no-collocation). This normalization is to further eliminate the impact of any unfair differences between the diverging libraries of the corresponding codebases.

Figure 14 reports the result of the comparison. These experiments utilize the A100 machine with 4 GPUs. Each instance of the ResNet18 model is being trained on ImageNet and on a separate GPU (as in Section 4.2). Figure 14a notes the scaling of CPU utilization as collocation increases. TensorSocket only marginally increases CPU load under higher degrees of collocation, while CoorDL requires more CPU resources to keep up. The CPU utilization of our baseline, not using either CoorDL or TensorSocket, is close to constant. This can be explained by Figure 14b, which shows the throughput scaling as the degree of collocation increases. Both CoorDL and TensorSocketthave no issue keeping the per-model training throughput the same despite the higher load. The baseline, however, heavily throttles, losing almost 75% of the performance under 4x load. This throttling can explain the low CPU utilization, as the data loading workers can not keep up with the training loops and thus the models idle. In general, while both TensorSocket and CoorDL can

[3]While we point out the many difficulties in establishing a fair comparison across all the codebases in the rest of this section, we are grateful for the authors of both CoorDL [35] and Joader [66] for providing an open-source implementation.

	GPU	Tokens	PCIe	NVLink	VRAM
Baseline	1	7.5k/s	48 MB/s	-	7.3 GB
	2	7.4k/s	48 MB/s	-	7.3 GB
Shared	0 (Prod)	-	.3 MB/s	-	1.5 GB
	1 (Cons)	7.5k/s	48 MB/s	152 KB/s	7.3 GB
	2 (Cons)	7.6k/s	48 MB/s	153 KB/s	7.3 GB

Table 4. Training speed, PCIe and NVLink traffic, and GPU memory usage for Qwen2.5 0.5B LLM training using TorchTune on separate A100 GPUs.



(a) Normalized CPU utilization.

(b) Normalized per-model training throughput.

Fig. 14. CPU utilization and throughput scaling of baseline (no-sharing), CoorDL, and TensorSocket on the A100 system under different levels of collocation. The scaling is compared to training a single model with that technique without collocation. Each collocated ResNet18 model runs on a separate GPU. TensorSocket is able to achieve the maximum throughput without the deployment restrictions and higher CPU requirement of CoorDL.

provide maximum throughput, TensorSocket does so with considerably fewer CPU resources while also being more flexible and less intrusive in usage.

Joader has a proof-of-concept implementation in Rust that is compatible with the latest version of PyTorch [67]. Specifically, the current implementation of Joader does not require PyTorch at all; rather, it uses Numpy to store data instead. While this does improve the compatibility of the implementation, it raises serious performance issues that hamper its effectiveness in real-world scenarios. Specifically, 1) Joader's image pre-processing and dataset (ImageNet) support is fully hardcoded, 2) images are delivered as NumPy matrices instead of Tensors and 3) Joader does not have support for mini-batches. Therefore, we take a number of concessions in order to support a comparison to Joader that is as fair as possible. Firstly, for (1), we investigate Joader's pre-processing pipeline and configure our TIMM training script to use the same transformations that are hardcoded for Joader. Note that it is not possible to use the exact same transformation code in both, as Joader is written in Rust and pre-processing pipelines available in online model sources like TIMM are typically defined in Python. Then, we address (2) and (3) by having the training script in Joader's case ask for enough data to fill up the batch after which it can construct the tensor and send it to the GPU. This tensor construction from many NumPy matrices, however, is very expensive and will cripple the performance. As for the sake of this performance comparison we are not interested in the loss of the model, we opt to only construct a tensor out of the first batch of data. Subsequent iterations then wait for the new data to come in, only to train on the same, first batch again. This minimizes the overhead of batching for Joader.

Figure 15 shows the results of comparing Joader this way to TensorSocket on the H100 system. We maximize the performance of all techniques by using MPS for sharing the resources of the H100 GPU. As the baseline, we train 1 to 8 collocated MobileNetV3's without any sharing enabled. Furthermore, we restrict the amount of data loading workers to 8. This means that, under no sharing, every model only has 1 worker when training 8 models simultaneously. Collocation amounts that do not divide 8, such as 5, have the workers divided unevenly in order to sum to 8. Keep in mind that the training scripts themselves are allowed to use further CPU resources. As expected, TensorSocket performs well even when running with high amounts of collocation, dropping no performance up to and including 6-way collocation. Only with 7- and 8-way collocation is there a drop in performance. This comes in stark contrast to non-shared training, where throughput goes down quickly. In fact, the data loading is such a bottleneck for non-shared training here that the summed throughput of collocation never exceeds that of non-collocated training. Joader

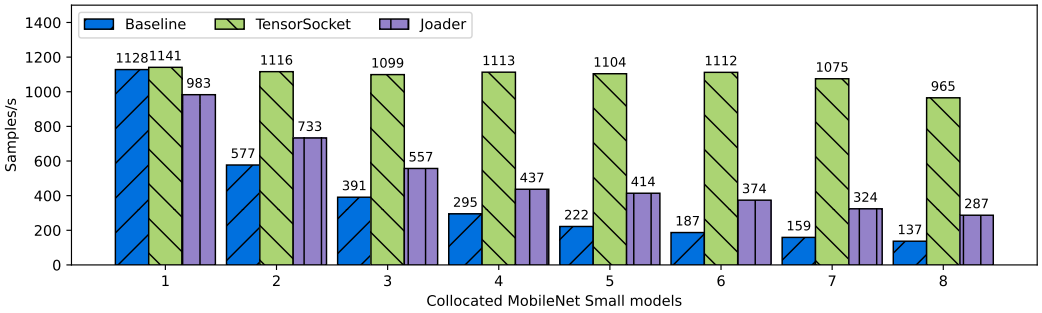


Fig. 15. Comparison of model training throughput with varying degrees of collocation under constrained CPU resources on the H100 system between baseline (no data sharing), Joader [66], and TensorSocket. Each collocated model training is MobileNetV3-Small trained on ImageNet. TensorSocket is able to provide data to many collocated models with little impact on throughput, even under constrained CPU circumstances.

comfortably outperforms non-shared training but is far behind the efficiency of TensorSocket. This is likely caused by the extra overhead introduced by Joader's data sampling algorithm (as described in Section 2). While Joader's algorithm provides flexibility for training with different speeds and datasets, this sacrifices efficiency in terms of CPU resource use, which in turn impacts the efficiency of training itself.

5 TensorSocket Going Forward

Having evaluated TensorSocket's impact on training efficiency, cost savings, and flexibility, this section highlights the key results while discussing further applicability of TensorSocket.

Target Workloads. TensorSocket alleviates a range of computational and resource-dependent bottlenecks. In general, small scale DL training workloads that are prone to exhibit input-bound pipelines or a high degree of CPU utilization can benefit greatly from shared data loading. If used on workloads in the cloud with similar characteristics as those presented in our work, which are prevalent in practice [60], TensorSocket may reduce costs up to a substantial 50% reduction as shown in Section 4.

Achieving these benefits does require some conditions to be met. Training jobs have to be collocated to make use of a shared data loader and are required to train with similar speed on the same dataset. Loosening up these requirements may allow shared data loading to become an attractive option for more workloads, though, that may come at the cost of reduced benefits.

Generalizability. We analyzed a complex data loading pipeline that includes generating intermediate representations of our data through an auxiliary model that is not being trained in Section 4.4. The ability to support unusual steps like these is a testament to the generalizability of TensorSocket. In addition, we can further dissect the data loading pipeline for finer-grained sharing. This allows for transformations and augmentations that are specific to each training process while only doing costly work, such as image decoding, once. For other techniques that use GPUs for data pre-processing [26, 41, 59], this means that TensorSocket can be deployed together with them to support GPU-offloading of transformation and augmentation operations while keeping redundancy and computational footprint low.

TensorSocket is implemented around PyTorch 2 and has no other large dependencies. We leave as much of the implementation as possible, such as the data structures, over to PyTorch to minimize the complexity of our codebase. This makes TensorSocket easily maintainable for future PyTorch versions and extendable. Other frameworks, such as TensorFlow, can use TensorSocket now

by using PyTorch as a data sharing intermediate. A native implementation in other frameworks would require just the re-implementation of the wrapper that allows for tensor deconstruction and reconstruction, *TensorPayload*, estimated ~59 lines of code.

6 Related Work

Historically, there has been a plethora of work on benchmarking and optimizing the computational efficiency of the core of model training and serving. However, in the recent years, data loading and pre-processing have been gaining more attention [6, 31], as with ever-increasing model sizes and training throughput requirements the cost of data loading bottlenecks is growing rapidly [72].

Murray et al. [36] and Mohan et al. [35] emphasize the impact of the data loading pipelines on training efficiency. The former presents the tf.data framework to ease the tuning for the computational efficiency of data pre-processing. The latter proposes CoorDL, a data loading library, and MinIO, a software cache with the goal of reducing cache thrashing. These works, among others [4, 24], also advocate for sharing for DL data loading tasks, but more at the cluster-level rather than the finer-grained view of TensorSocket.

To further optimize data loading flows, Plumber [28] extends tf.data by automatically detecting and resolving misconfigured pipelines, Lotus and Presto provides data pipeline optimization insights [5, 21], while Cedar [71] and Pecan [17] orchestrate optimizations to the input data pipeline. Joader, proposed by Xu et al. [66], provides an alternative to CoorDL that offers extra flexibility when sharing data for training tasks, allowing for shared training on multiple datasets at the same time. It manages this via a novel data sampling solution that optimizes sharing at the cost of repeated dataset intersection calculations. Such calculations can be costly, as Section 4.7 demonstrated.

Behme et al. [8] explore lossy image compression's role in mitigating data loading bottlenecks. They demonstrate that moderately compressed data maintains accuracy comparable to benchmarks while saving 30% storage and how this technique complements the software cache of MinIO [35].

TensorSocket also borrows ideas from works on data and work sharing in databases such as StagedDB [18], QPipe [19], and SharedDB [14], and works that analyze the trade-offs of sharing [23, 47]. These works aim at sharing work that is common across concurrent database queries. In contrast, TensorSocket applies similar sharing ideas to DL data preparation.

7 Conclusion

In this paper, we presented TensorSocket, a novel data loading mechanism that enables data and work sharing in data pre-processing pipelines of deep learning training. The key insight behind TensorSocket is that tuning efforts to achieve the best model architecture and parameters require training several models on the same data. This results in shared tasks across these training processes. We demonstrated that TensorSocket can double training throughput while substantially reducing the number of CPU cores to achieve that throughput. Furthermore, it is easy to adopt in existing training pipelines, enables certain training scenarios on restricted hardware resource setups, and can halve cloud setup costs as a result. Finally, TensorSocket is compatible with existing techniques that aim at increasing the computational efficiency of the data pre-processing tasks.

Acknowledgements

This work is funded by the Independent Research Fund Denmark's (Danmarks Frie Forskningsfond; DFF) Sapere Aude and Inge Leh-mann programs under grant agreement number 0171-00061B and 0171-00062B, respectively. We also thank DSAR lab members at IT University of Copenhagen for their support, and the reviewers of SIGMOD for their constructive feedback.

References

[1] Amazon. 2023. https://aws.amazon.com/ec2/. Accessed: 2023-11-30.

[2] Amazon. 2023. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2024-01-22.

[3] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems. arXiv:2007.03051.

[4] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A Thekkath. 2023. tf.data service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, USA, 358–375.

[5] Rajveer Bachkaniwala, Harshith Lanka, Kexin Rong, and Ada Gavrilovska. 2024. Lotus: Characterization of Machine Learning Preprocessing Pipelines via Framework and Hardware Profiling. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Vancouver, BC, Canada, 30–43. https://doi.org/10.1109/IISWC63097.2024.00013

[6] Oana Balmau. 2022. Characterizing I/O in Machine Learning with MLPerf Storage. *SIGMOD Rec.* 51, 3 (2022), 47–48. https://doi.org/10.1145/3572751.3572765

[7] Sebastian Baunsgaard, Sebastian Benjamin Wrede, and Pınar Tözün. 2020. Training for Speech Recognition on Coprocessors. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). Association for Computing Machinery, Japan, 1–10.

[8] Lennart Behme, Saravanan Thirumuruganathan, Alireza Rezaei Mahdiraji, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. The Art of Losing to Win: Using Lossy Image Compression to Improve Data Loading in Deep Learning Pipelines. In *Proceedings of the 39th IEEE International Conference on Data Engineering*. IEEE, Anaheim, California, 936–949.

[9] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, USA, 113–123.

[10] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS*. Association for Computing Machinery, USA, 127–144.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, IEEE, USA, 248–255.

[12] Jesse Dodge, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A. Smith, Nicole DeCario, and Will Buchanan. 2022. Measuring the Carbon Intensity of AI in Cloud Instances. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency* (Seoul, Republic of Korea) *(FAccT '22)*. Association for Computing Machinery, New York, NY, USA, 1877–1894. https://doi.org/10.1145/3531146.3533234

[13] Connor Espenshade, Rachel Peng, Eumin Hong, Max Calman, Yue Zhu, Pritish Parida, Eun Kyung Lee, and Martha A. Kim. 2024. Characterizing Training Performance and Energy for Foundation Models and Image Classifiers on Multi-Instance GPUs. In *Proceedings of the 4th Workshop on Machine Learning and Systems* (Athens, Greece) *(EuroMLSys '24)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/3642970.3655830

[14] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.* 5, 6 (feb 2012), 526–537. https://doi.org/10.14778/2168651.2168654

[15] Google. 2023. https://cloud.google.com/compute?hl=en. Accessed: 2023-11-30.

[16] 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 *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 689–706. https://www.usenix.org/conference/atc22/presentation/graur

[17] Dan Graur, Oto Mraz, Muyu Li, Mohammad Sepehr Pourghannad, Chandramohan A. Thekkath, and Ana Klimovic. 2024. Pecan: Cost-Efficient ML Data Preprocessing with Automatic Transformation Ordering and Hybrid Placement. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, USA, 649–665. https://www.usenix.org/conference/atc24/presentation/graur

[18] Stavros Harizopoulos and Anastassia Ailamaki. 2005. StagedDB: Designing Database Servers for Modern Hardware. *IEEE Data Eng. Bull.* 28, 2 (2005), 11–16. http://sites.computer.org/debull/A05june/stavros.ps

[19] 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* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/1066157.1066201

[20] iostat. 2024. https://linux.die.net/man/1/iostat

[21] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. 2022. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1825–1839. https://doi.org/10.1145/3514221.3517848

[22] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, USA, 947–960.

[23] Ryan Johnson, Nikos Hardavellas, Ippokratis Pandis, Naju Mancheril, Stavros Harizopoulos, Kivanc Sabirli, Anastassia Ailamaki, and Babak Falsafi. 2007. To Share or Not To Share?. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, Austria, 351–362. http://www.vldb.org/conf/2007/papers/research/p351-johnson.pdf

[24] 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)*. USENIX Association, Renton, WA, 12. https://www.usenix.org/conference/hotcloud19/presentation/kakaraparthy

[25] kernel.org 2021. *top(1) — Linux manual page*. kernel.org. https://man7.org/linux/man-pages/man1/top.1.html.

[26] Taeyoon Kim, ChanHo Park, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeo. 2024. FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation. *Proc. VLDB Endow.* 17, 4 (2024), 863–876. https://doi.org/10.14778/3636218.3636238

[27] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB* 12, 11 (2019), 1399–1412.

[28] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems* 4 (2022), 33–51. https://proceedings.mlsys.org/paper_files/paper/2022/hash/d0e90e9a9310570dfa643aa3b2da6e89-Abstract.html

[29] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. MLSys, USA, 230–246.

[30] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*. Springer, Germany, 19–34.

[31] LMPerf. 2023. https://mlcommons.org/benchmarks/storage/.

[32] Fabio Maschi and Gustavo Alonso. 2023. The Difficult Balance Between Modern Hardware and Conventional CPUs. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/3592980.3595314

[33] Microsoft Azure. 2023. https://azure.microsoft.com/en-us/products/virtual-machines. Accessed: 2023-11-30.

[34] Microsoft Research. 2024. https://github.com/msr-fiddle/CoorDL/tree/master. Accessed: 2024-07-10.

[35] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.* 14, 5 (jan 2021), 771–784. https://doi.org/10.14778/3446095.3446100

[36] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. Tf.Data: A Machine Learning Data Processing Framework. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2945–2958. https://doi.org/10.14778/3476311.3476374

[37] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2159–2173. https://doi.org/10.14778/3407790.3407816

[38] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In *ICPP*. Association for Computing Machinery, Japan, 15:1–15:10.

[39] NVIDIA. 2012. https://developer.nvidia.com/nvidia-system-management-interface

[40] NVIDIA. 2022. *Data Center GPU Manager Documentation*. Technical Report. NVIDIA. https://docs.nvidia.com/datacenter/dcgm/latest/dcgm-user-guide/.

[41] NVIDIA. 2023. https://github.com/NVIDIA/DALI. Accessed: 2023-05-19.

[42] NVIDIA. 2023. https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2023-05-27.

[43] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEE, USA, 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[44] 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., Canada, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[45] David A. Patterson, Joseph Gonzalez, Urs Hölzle, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. *Computer* 55, 7 (2022), 18–28. https://doi.org/10.1109/MC.2022.3148714

[46] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research* 20, 1 (jan 2019), 1934–1965.

[47] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing data and work across concurrent analytical queries. *Proc. VLDB Endow.* 6, 9 (jul 2013), 637–648. https://doi.org/10.14778/2536360.2536364

[48] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. arXiv:2204.06125 [cs.CV]

[49] Ties Robroek, Aaron Duane, Ehsan Yousefzadeh-Asl-Miandoab, and Pinar Tozun. 2023. Data Management and Visualization for Benchmarking Deep Learning Training Systems. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*. Association for Computing Machinery, USA, 1–5.

[50] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pinar Tözün. 2024. An Analysis of Collocation on GPUs for Deep Learning Training. In *Proceedings of the 4th Workshop on Machine Learning and Systems, EuroMLSys 2024, Athens, Greece, 22 April 2024*. ACM, Greece, 81–90. https://doi.org/10.1145/3642970.3655827

[51] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* 63, 12 (nov 2020), 54–63. https://doi.org/10.1145/3381831

[52] Piyush Sharma, Nan Ding, Sebastian Goodman, and Radu Soricut. 2018. Conceptual Captions: A Cleaned, Hypernymed, Image Alt-text Dataset For Automatic Image Captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, Melbourne, Australia, 2556–2565. https://doi.org/10.18653/v1/P18-1238

[53] Janne Spijkervet and John Ashley Burgoyne. 2021. , 673-681 pages. https://doi.org/10.5281/zenodo.5624573

[54] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, Greece, 1075–1092. https://doi.org/10.1145/3627703.3629578

[55] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Italy, 3645–3650. https://doi.org/10.18653/V1/P19-1355

[56] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. https://github.com/tatsu-lab/stanford_alpaca.

[57] torchtune maintainers and contributors. 2024. *torchtune: PyTorch's finetuning library*. PyTorch. https//github.com/pytorch/torchtune

[58] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. 2014. ADDICT: advanced instruction chasing for transactions. *Proc. VLDB Endow.* 7, 14 (oct 2014), 1893–1904. https://doi.org/10.14778/2733085.2733095

[59] 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. *Proc. VLDB Endow.* 16, 5 (jan 2023), 1086–1099. https://doi.org/10.14778/3579075.3579083

[60] Gaël Varoquaux, Sasha Luccioni, and Meredith Whittaker. 2025. Hype, Sustainability, and the Price of the Bigger-is-Better Paradigm in AI. In *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency, FAccT 2025, Athens, Greece, June 23-26, 2025*. ACM, Athens, Canada, 61–75. https://doi.org/10.1145/3715275.3732006

[61] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, China, 1865–1878. https://doi.org/10.1145/3448016.3457286

[62] Phil Wang. 2022. https://github.com/lucidrains/DALLE2-pytorch.

[63] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. *Proceedings of Machine Learning and Systems* 3 (2021), 599–623.

[64] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, USA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[65] Ross Wightman. 2019. https://github.com/rwightman/pytorch-image-models. https://doi.org/10.5281/zenodo.4414861

[66] Jingwei Xu, Guochang Wang, Yuan Yao, Zenan Li, Chun Cao, and Hanghang Tong. 2022. A Deep Learning Dataloader with Shared Data Preparation. *Advances in Neural Information Processing Systems* 35 (2022), 17146–17156.

[67] Jingwei Xu, Guochang Wang, Yuan Yao, Zenan Li, Chun Cao, and Hanghang Tong. 2024. https://github.com/XieJiann/Joader. Accessed: 2024-07-10.

[68] Qwen: An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[69] Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tözün. 2023. Profiling and Monitoring Deep Learning Training Tasks. In *Proceedings of the 3rd Workshop on Machine Learning and Systems, EuroMLSys 2023, Rome, Italy, 8 May 2023*, Eiko Yoneki and Luigi Nardi (Eds.). ACM, Italy, 18–25. https://doi.org/10.1145/3578356.3592589

[70] ZeroMQ. 2023. https://zeromq.org/. Accessed: 2023-12-11.

[71] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. 2024. https://doi.org/10.48550/arXiv.2401.08895 arXiv:2401.08895 [cs].

[72] 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 *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1042–1057. https://doi.org/10.1145/3470496.3533044

Received January 2025; revised April 2025; accepted May 2025