

Preparation Meets Opportunity: Enhancing Data Preprocessing for ML Training With Seneca

Omkar Desai
Syracuse University

Janki Bhimani
Florida International University

Ziyang Jiao
Syracuse University

Shuyi Pei
Samsung Semiconductor
Bryan S. Kim
Syracuse University

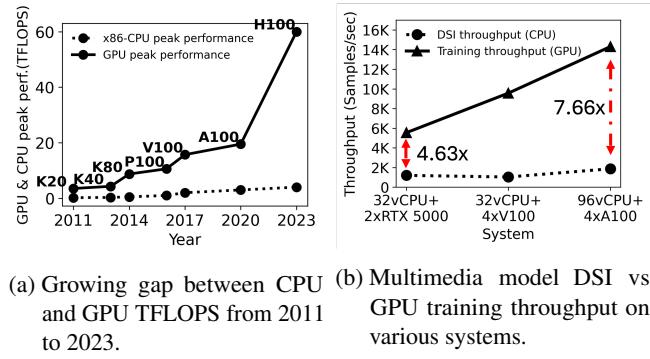
Abstract

Input data preprocessing is a common bottleneck when concurrently training multimedia machine learning (ML) models in modern systems. To alleviate these bottlenecks and reduce the training time for concurrent jobs, we present **Seneca**, a data loading system that optimizes cache partitioning and data sampling for the data storage and ingestion (DSI) pipeline. The design of Seneca contains two key techniques. First, Seneca uses a performance model for the data pipeline to optimally partition the cache for three different forms of data (encoded, decoded, and augmented). Second, Seneca opportunistically serves cached data over uncached ones during random batch sampling so that concurrent jobs benefit from each other. We implement Seneca by modifying PyTorch and demonstrate its effectiveness by comparing it against several state-of-the-art caching systems for DNN training. Seneca reduces the makespan by 45.23% compared to PyTorch and increases data processing throughput by up to 3.45 \times compared to the next best dataloader.

1 Introduction

Input data preprocessing is an essential step in all machine learning (ML) training jobs. During this, the data storage and ingestion (DSI) pipeline fetches samples from storage, decodes them into tensors, transforms and augments them as required, and loads them into the GPU for training. These tasks are resource-intensive, requiring high I/O bandwidth and compute parallelism. The throughput of the DSI pipeline has a significant impact on training performance as GPUs depend on data from the DSI pipeline to train models.

With ML jobs training concurrently to maximize GPU utilization, the DSI pipeline has become a bottleneck, especially for image, video, audio, and recommendation models [3, 5, 21, 35, 42, 61]. Figure 1 illustrates hardware trends affecting ML training. Figure 1a shows the widening gap between popular NVIDIA GPUs used for training and CPU performance. Figure 1b illustrates the impact of the CPU-GPU



(a) Growing gap between CPU and GPU peak performance in TFLOPS from 2011–2023 [7, 8, 11, 19, 44–50] (Figure 1a) resulting in DSI pipeline bottlenecks for multimedia model training (Figure 1b).

(b) Multimedia model DSI vs GPU training throughput on various systems.

Figure 1: The growing gap between CPU and GPU peak performance in TFLOPS from 2011–2023 [7, 8, 11, 19, 44–50] (Figure 1a) resulting in DSI pipeline bottlenecks for multimedia model training (Figure 1b).

performance gap on multimedia model training by comparing the DSI throughput without training and the training throughput without DSI for SwinT [37], a transformer model on three real systems. The dotted line shows the upper bound of DSI throughput, while the solid line reflects the upper bound of training throughput, revealing that DSI is the bottleneck. The disparity between DSI and training throughput increases from 4.63 \times on the RTX 5000 server to 7.66 \times on the A100 server, indicating that the DSI pipeline bottleneck is exacerbating. The combination of a relatively slow CPU limits preprocessing performance, and the limited host DRAM forces more frequent data retrieval from slow storage, leading to training stalls and an overall performance degradation [30, 57, 58].

While several prior works propose caching as a solution for improving the DSI pipeline throughput [20, 21, 26, 31, 33, 41], we identify two unique challenges that limit its effectiveness. First, multiple intermediate forms of data exist in the pipeline, and which form to cache is not straightforward and remains unexplored. Data can exist in one of three forms: encoded for high storage density, decoded into tensors, and augmented randomly for training. Determining which form to cache is subtle because encoded data requires more CPU-driven pre-

processing [21, 57], while data in later stages (e.g., augmented) take up more space (up to $15 \times$ [24]). The optimal form to cache depends on training job parameters, cache capacity, and the performance of the slowest DSI component, factors not considered by previous works.

Second, current cache designs and data sampling for the DSI pipeline perform poorly when multiple jobs train models concurrently over the same dataset. More specifically, the inherent nature of random sampling makes poor use of cache as data are sampled agnostic of what is available in the cache. For example, SHADE [26] implements an importance-based sampling but its approach is incompatible with concurrent training, and Quiver [31] presents a substitution-based sampling compatible with concurrent training but suffers from high bandwidth contention due to over-sampling.

We address these challenges by designing the following:

1. **Model-Driven Partitioning (MDP).** We build a high-level performance model for the DSI pipeline based on the insight that we can estimate the cache hit rate due to random sampling. This model is then used to predict the data pipeline throughput, constrained by the slowest component, and allows Seneca to partition the cache optimally for different data forms.
2. **Opportunistic Data Sampling (ODS).** Based on the insight that a training job may benefit from the other’s data sampling activities that share the same dataset, we design an opportunistic data sampler that serves cached data over uncached ones. This is possible because the sampling sequence does not need to adhere to the predetermined pseudo-random sequence as long as it appears random and each training data is consumed once in an epoch.

Preparation meets opportunity in Seneca by combining the design principles of MDP and ODS: integrating cache partitioning and data sampling to efficiently alleviate preprocessing bottlenecks in ML training. We implement Seneca by modifying the PyTorch [3] dataloader and using Redis [4] to cache data. Seneca improves DSI pipeline throughput by up to $3.45 \times$ compared to the next best dataloader and reduces makespan by 45.23% over PyTorch, without compromising training accuracy.

The contributions of this work are as follows:

- We design a performance model for the DSI pipeline in distributed ML training and develop a model-driven partitioning (MDP) scheme that uses the performance model to optimize cache partition sizes and improve training throughput. MDP is applicable to all types of ML training jobs but benefits preprocessing heavy jobs the most.
- We develop an opportunistic data sampling (ODS) scheme that improves the cache hit rate and multi-job training throughput by replacing uncached data with cached ones while maintaining a pseudo-random order. The effectiveness of ODS increases with more concurrently running jobs.

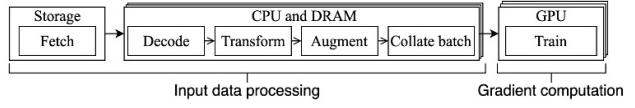


Figure 2: The ML training process: (1) fetches data from storage, (2) decodes, transforms, and forms a minibatch, and (3) loads the minibatch into the GPU for training. This repeats until the target accuracy is reached. Encoded data is small, but decoded and augmented data are much larger.

- We demonstrate the effectiveness of Seneca by evaluating it against five state-of-the-art solutions across seven models (3.4–633.4 million parameters) and three datasets (142GB–1.4TB) on five hardware configurations.
- The artifacts of Seneca are open source and available at <https://github.com/swiftomkar/seneca-fast26-pytorch>

2 Background

ML models are trained over multiple rounds called epochs. During an epoch, the entire dataset is processed exactly once. Each epoch is divided into multiple iterations, where each iteration processes a random, non-overlapping subset of data known as a minibatch. Training continues until further epochs no longer improve the model’s accuracy.

Figure 2 outlines the two main components of the ML training process: (1) input data preprocessing and (2) gradient computation. In preprocessing, data is fetched from storage, decoded, transformed, augmented, and collated into mini-batches before being loaded onto the GPU for gradient computation. While decoding and collation are static preprocessing operations, transformations and augmentations are randomly applied to improve model generality. Table 1 describes the data preprocessing steps for various ML model types and their resource demands. We note that multimedia and high-dimensional data have resource-intensive DSI pipelines due to larger sample sizes. During gradient computation, the model updates its weights and biases based on the training data.

ML model training is resource-intensive and time-consuming, so large GPU clusters are often used to speed up training through techniques like data parallelism [55], model parallelism [14], and pipelined parallelism [23]. Our work focuses on optimizing the data preprocessing bottleneck commonly seen in multimedia ML (image, audio, and video) and deep learning recommendation models (DLRM) that use data parallelism [40, 61]. In data parallelism, the dataset is divided and processed in parallel across multiple GPUs, each running a copy of the same model. After every batch, the models are synchronized across GPUs using centralized methods (e.g., parameter servers [34]) or decentralized methods (e.g., all-

Table 1: DSI pipeline for various ML model types. All types of training data undergo preprocessing before training. For all model types, decoding converts a dataset file into a tensor, while transformation and augmentation steps apply operations on the decoded tensor before collating samples into a batch.

Model type	Preprocessing steps				Resource demand
	Decode	Transform	Augment	Collate	
Image	Image file → Tensor	Resize Normalize	Random crop Random flip	[images:labels]	High
Audio	Audio file → Tensor	Fourier transform Padding	Time stretch Time masking	[audio:labels]	High
Text	Text file → Tensor	Padding Truncation	Shuffling Masking	[text:labels]	Low
Recommendation	Tabular → Tensor	Padding Truncation	Shuffling Masking	[feature vector: user action]	High

reduce [12] or ring-reduce [56]).

The following are the three main steps in the DSI pipeline.

1. **Fetching data from storage.** The first step in ML data preprocessing is reading data from the slow, remote storage service. DSI pipelines manage datasets from a few gigabytes to several petabytes in low-cost storage. To avoid the memorization of specific patterns, data samples must be fetched in random order, making caching difficult [26, 32].
2. **Transforming data.** Before training, each data sample must undergo several processing steps (illustrated in Table 1) which are typically applied on the fly [21]. These transformations performed by the DSI pipeline can be computationally intensive and generally run on the CPU to support user-defined transformations [42]. Common transformations necessary for all training jobs include decompressing, decoding, and collating samples. Additionally, every job also has specific static transformations (e.g., tokenization for text, quantization for audio) and random augmentations (e.g., image rotation, audio stretching) to improve model accuracy and generality [6, 42]. Random augmentations are especially critical to training generalizable models for image classification, object detection, and speech recognition.
3. **Loading data to the GPU.** The final step is loading data to GPUs for gradient computation. To prevent stalls, the DSI pipeline must provide data at or above the GPU’s data ingestion rate, which depends on the model size, training algorithm’s arithmetic intensity, and the GPU’s FLOPS.

3 Existing approaches

We categorize works related to Seneca into four categories: system modeling, preprocessing optimization, hardware acceleration, and cache optimizations.

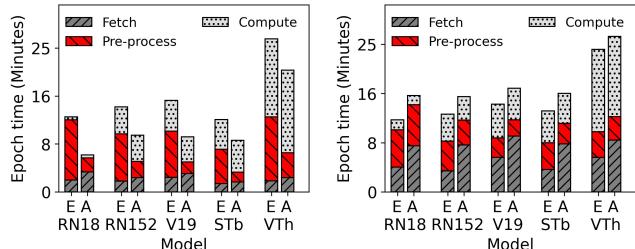
System modeling: Lobster [36] models the I/O and DSI pipeline of a distributed training system to balance I/O and preprocessing threads. NoPFS [17] builds a performance

model for a distributed storage system given a clairvoyant knowledge of data sampling, and uses this to prefetch data from slower storage to cache. Our work is inspired by high-level system modeling and builds a mathematical model of the entire DSI pipeline. Our model predicts the throughput for different forms of data in the cache and can be used to partition cache space and minimize preprocessing bottlenecks.

Preprocessing optimization: Several prior works have identified preprocessing of training data as a major bottleneck. Cedar [60], FastFlow [57], GoldMiner [59], Cachew [20], and Pecan [21] address this issue by optimizing and offloading preprocessing tasks to remote CPUs based on profiled system metrics. Our work cost-effectively optimizes the DSI pipeline, without relying on remote CPUs. PRESTO [24] and Revammer [33] explore caching partially or fully processed data to minimize repeated preprocessing computations on the same data, but don’t consider their data inflation in conjunction with resource constraints presented by the training hardware. Our work uses this insight and allocates an optimal ratio of the available cache to each form.

Hardware acceleration: DALI [43], TrainBox [52], and FusionFlow [27] uses hardware accelerators (GPUs and FPGAs) for preprocessing. This makes it challenging to apply user-defined transforms while also making poor use of expensive accelerators that are optimized for SIMD operations that differ from the stochastic nature of random augmentations. While our implementation is based on using CPUs for preprocessing, the core concepts of our work can be extended to data loaders that leverage hardware accelerators.

Cache optimization: Due to I/O bottlenecks and large dataset sizes in DNN training, many works focus on reducing data fetch time through caching. MINIO [41] uses a shared cache with a no-eviction policy to avoid thrashing, but its cache hit rate is limited by the cache-to-dataset size ratio. SHADE [26] and iCache [13] use importance sampling [25], but since sample importance varies across jobs, sharing a cache for concurrent jobs is difficult. Quiver [31] allows sample substitution from the cache but suffers from high oversampling overhead. Our approach introduces a cache-aware opportunistic sam-



(a) Epoch completion times with 450GB cache (b) Epoch completion times with 250GB cache

Figure 3: Fetch, preprocess, and compute times when data is cached in encoded ('E') or augmented ('A') form for ResNet-18 (RN18), ResNet-152 (RN152), VGG-19 (V19), SwinT big (STb), and ViT huge (VTh) with different cache sizes.

pler that prioritizes cached samples over uncached ones while maintaining the randomness and uniqueness needed for high model accuracy.

4 Challenges for the DSI pipeline

Based on the required data preprocessing and random data sampling of DNN model training, We describe below the two key challenges in effectively using limited cache space to alleviate the DSI pipeline bottleneck.

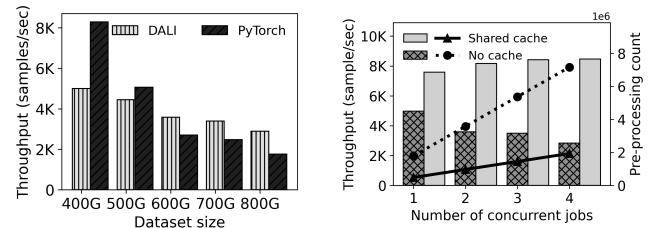
4.1 Determining which data form to cache

Training data can be cached in different forms (encoded, decoded, or augmented), but choosing the right form involves a space-time trade-off, making it challenging to determine the optimal choice. Table 2 illustrates the trade-off in caching encoded, decoded, and augmented data across three different metrics. In terms of data density, the encoded data is the highest (smallest in size), thus for a given capacity, more encoded data can be cached. Training readiness represents how preprocessed the data is: for this metric, the final augmented data is the most training-ready. Cache worthiness qualifies how useful it is to cache the data. While encoded and decoded data can be reused across epochs, repeatedly using the same randomly augmented data risks overfitting due to insufficient randomness in the augmentations.

We illustrate the subtlety of caching in the DSI pipeline by measuring the performance of five models when caching

Table 2: Data forms and their trade-offs.

	Data density	Training readiness	Cache worthiness
Encoded	High	Low	High
Decoded	Low	Medium	High
Augmented	Low	High	Low



(a) DSI throughput with respect to varying dataset size (in GB) for popular open-source dataloaders.

(b) DSI throughput wrt. number of jobs with and without caching. Lines: preprocessing count; Bars: DSI throughput.

Figure 4: Drawback of OS page-cache for random access patterns (Figure 4a) and the impact of sharing preprocessed data with concurrently training jobs (Figure 4b).

either encoded or augmented data at two different cache capacities (Figure 3). We add Redis [4] to PyTorch [3] for caching either encoded or augmented data of the OpenImages dataset [28] on a CloudLab [18] system with $4 \times$ A100 GPUs, 2×24 core AMD 7413 CPUs, 512 GB DRAM, 200 Gbps Mellanox ConnectX-6 NIC, and a remote storage service (NFS) for datasets. The measurements show the time spent fetching data, preprocessing on the CPU, and computing on the GPU. Caching augmented data ('A') reduces preprocessing time since only data from storage needs processing, whereas caching encoded data ('E') requires preprocessing all data. However, caching augmented data ('A') increases fetch time due to the larger tensor size, which limits the number of samples stored in cache.

Interestingly, the trade-off between caching strategies depends on cache size. With 450GB (Figure 3a), caching preprocessed data significantly reduces preprocessing time by 69.91%, while fetch time only increases by 34.85% on average. However, with 250GB (Figure 3b), the benefit of caching preprocessed data is minimal: preprocessing time decreases by just 11.36%, while fetch time rises by 87.2% on average.

While Figure 3 shows the performance impact of caching encoded vs. augmented data based on cache size, other hardware factors also play a role. A faster CPU reduces preprocessing time, and faster storage shortens data fetch time. While we only compare caching encoded or augmented data, a range of options exists, such as caching decoded data or splitting the cache between different forms. Deciding how to best allocate a limited cache for the DSI pipeline is a complex, non-linear problem influenced by multiple variables.

4.2 Caching for random sample accesses

The inherent nature of random sampling makes it difficult to implement effective caching policies and we observe that multiple training jobs that share the same dataset do not synergize. In Figure 4, we demonstrate the effect of DRAM size and

number of concurrent jobs by measuring the DSI throughput when training ResNet-50 [22] for image classification on the CloudLab system described in §4.1.

Figure 4a compares two open-source dataloaders, PyTorch [3] and DALI [43], both reliant on the system-wide page cache. The results show that the page cache’s LRU-based algorithm performs poorly with random access patterns. As the dataset size grows, more data is fetched from slow remote storage, degrading DSI pipeline performance. Increasing the dataset size from 400GB to 600GB reduces DSI throughput by 28.41% for DALI and 67.34% for PyTorch. While PyTorch outperforms DALI when the dataset fits in page cache, DALI’s more efficient cache usage gives it an advantage as dataset size increases.

Figure 4b illustrates inefficiencies in concurrent training, where each job redundantly preprocesses images independently. Training four concurrent PyTorch jobs without caching leads to 7.16 million total preprocessing operations for 1.7 million OpenImages [28] samples (dotted line). As a result, increasing jobs from one to four reduces total DSI throughput by 46.80% (black bars), while GPU utilization remains below 80%. To demonstrate the benefits of shared caching, we add a 350GB Redis cache with PyTorch to store and share preprocessed data. With a portion of the dataset in cache, preprocessing operations drop $3.7\times$ (solid line), and aggregate training throughput improves by 11.81% (gray bars). While sharing preprocessed data reduces redundant preprocessing, performance gains are marginal and jobs fail to benefit from higher concurrency. A more effective sampling policy that optimizes cache utilization could further enhance performance.

5 Design and Implementation of Seneca

We now describe the two key components of our work: (1) Model-Driven Partitioning (MDP)—a cache partitioning scheme that uses a high-level performance model of the DSI pipeline to determine the optimal ratio of cache to be allocated to each form, and (2) Opportunistic Data Sampling (ODS)—a data sampler that prioritizes serving cached data over uncached ones to improve cache hit rate and multi-job DSI throughput.

5.1 Model-driven partitioning (MDP)

We present a high-level performance model that estimates the overall performance of the DSI pipeline for an ML training job. Our formulation is based on data parallel training in a typical setup with GPU training nodes along with remote caching and storage services as shown in Figure 5. Each GPU processes a different batch in parallel and model parameters are synchronized after every batch. Our mathematical model estimates DSI throughput given (1) parameters for the hardware components of the training nodes (such as an abstract CPU, GPU, memory, and network performance), (2) performance

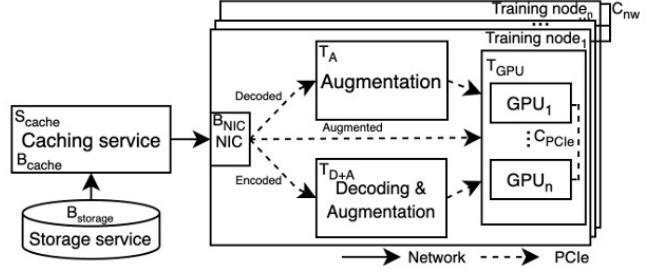


Figure 5: The DSI pipeline model.

of the remote cache and storage services (such as cache and storage bandwidth) (3) parameters for the training job (such as the dataset size, ML model size, and the average sample size), (4) gradient communication overhead, and (5) the proportions of memory allocated to the three data forms (i.e., encoded, decoded, and augmented).

Table 3 summarizes the model parameters and their de-

Table 3: Summary of parameters used in the model.

Notation	Description
T_{GPU}	Per node GPU ingestion throughputs (sample/s)
T_{D+A}, T_A	Per node CPU throughputs for decoding and augmenting data (sample/s)
B_{PCIe}	Per node PCIe bandwidth (B/s)
B_{cache}	Maximum remote cache bandwidth (B/s)
$B_{storage}$	Maximum remote storage bandwidth (B/s)
B_{NIC}	Per node network bandwidth (B/s)
S_{cache}	Size of remote cache (Bytes)
S_{data}	Size of a encoded data sample (Bytes)
N_E, N_D, N_A	Number of cached samples in encoded, decoded, and augmented forms
$N_{storage}$	Number of samples in storage
N_{total}	Number of samples in the dataset
M	Size inflation factor for preprocessed data
C_{PCIe}	Intra-node gradient communication overhead (Bytes)
C_{nw}	Inter-node gradient communication overhead (Bytes)
DSI_E, DSI_D, DSI_A	DSI performance for accessing encoded, decoded, and augmented caches
DSI_S	DSI performance for fetching data from storage
$DSI_{overall}$	Overall DSI performance
x_E, x_D, x_A	Proportions of memory allocated to the three data forms
n	Number of training nodes

scriptions. The training node parameters (T_{GPU} , T_A , T_{D+A} , B_{NIC} , B_{PCIe}) reflect the performance of hardware components in a single node. To obtain the maximum performance of a homogeneous n -node training cluster, we multiply these values by the number of nodes (n) in the cluster. Although the performance of cache and storage service (B_{cache} , $B_{storage}$) may be internally constrained by various components (e.g., storage device, DRAM bandwidth), we abstract them to the maximum achievable bandwidth from a training node.

Our model accounts for gradient communication overhead that occurs over PCIe and network after every batch to synchronize gradients across all GPUs as it may overlap with preprocessing tasks. We calculate the PCIe and network overheads separately using the number of nodes and GPUs per node. The network and PCIe overhead, C_{nw} and C_{PCIe} for a batch is given by $\frac{2 \times (n-1)}{n} \times \beta N$ [56], where n is the number of GPUs per node in the case of C_{nw} and the number of nodes in case of C_{PCIe} and βN is the model size in megabytes. For NVLink-connected GPUs [51], gradient communication incurs no overhead since NVLink, a dedicated interconnect, is used when available. Specifically, for intra-node NVLink, $C_{PCIe} = 0$, and for inter-node NVLink, both C_{PCIe} and C_{nw} are set to 0. While we estimate the overhead of ring-reduce which is used in this case, C_{nw} and C_{PCIe} can represent overheads for other gradient communication algorithms as well.

Given these parameters, we approach the problem of modeling the DSI pipeline by considering four different cases of data accesses: (1) when the data needed by the DNN training job is already augmented and in cache § 5.1.1; (2) when the data is decoded and in cache § 5.1.2; (3) when the data is encoded and in cache § 5.1.3; and (4) when the data is in storage § 5.1.4. Our high-level model formulates the performance for each case independently, and unifies them into one model by considering the probabilities of all cases § 5.1.5.

5.1.1 Augmented data in memory

We first model the performance of accessing augmented data stored in memory (DSI_A). For ease of calculation, we use throughput (samples per second) for all the units for the system parameters. Because GPU performance is nominally expressed in floating point operations per second (FLOPS), we measure the data ingestion rate for the GPU in terms of samples per second. The components involved in this scenario are cache, GPUs, and the interconnects (NIC and PCIe) between them. The limiting factor for DSI_A in an n -node training cluster could be the cache bandwidth (B_{cache}) or the aggregate performance of one of the hardware components in the cluster, which includes GPU ingestion rate ($n \times T_{GPU}$), network bandwidth ($n \times B_{NIC}$), and PCIe bandwidth ($n \times B_{PCIe}$), as shown in Equation 1. PCIe and network bandwidths scale down with both, augmented data size and respective gradient communication overheads (C_{nw} and C_{PCIe}), while cache bandwidth scales down by the size of augmented data ($M \times S_{data}$).

$$DSI_A = \min \left(\frac{B_{cache}}{M \times S_{data}}, \frac{n \times B_{NIC}}{(M \times S_{data}) + C_{nw}}, \frac{n \times B_{PCIe}}{(M \times S_{data}) + C_{PCIe}}, n \times T_{GPU} \right) \quad (1)$$

To compute the probability of accessing augmented data, we model the number of samples that are in the augmented form (N_A). Given x_A as the portion of memory allocated for caching augmented data (where $0 \leq x_A \leq 1$), the number of augmented samples in memory is $x_A \times S_{mem}$ divided by the size of a tensor, $M \times S_{data}$. However, N_A should also account for the case when the dataset is small enough to fit entirely in memory, as shown in Equation 2.

$$N_A = \min \left(N_{total}, \frac{x_A \times S_{mem}}{M \times S_{data}} \right) \quad (2)$$

5.1.2 Decoded data in memory

Next, we model the performance of accessing decoded data from memory (DSI_D). This scenario involves training node CPUs for applying random augmentations, along with GPUs, remote cache, and interconnects (Network and PCIe) between them. Similar to the GPU, we represent CPU throughput in samples per second for random augmentations, even though its nominal metric is FLOPS. The limiting factor for DSI_D is either the remote cache bandwidth (B_{cache}) or the aggregate performance of one of the hardware components in the cluster, which includes CPU augmentation throughput ($n \times T_A$), GPU ingestion rate ($n \times T_{GPU}$), and interconnect throughput ($n \times B_{NIC}$, $n \times B_{PCIe}$), as shown in Equation 3.

$$DSI_D = \min \left(\frac{B_{cache}}{M \times S_{data}}, \frac{n \times B_{NIC}}{(M \times S_{data}) + C_{nw}}, n \times T_A, \frac{n \times B_{PCIe}}{(M \times S_{data}) + C_{PCIe}}, n \times T_{GPU} \right) \quad (3)$$

To compute the probability of accessing decoded data from memory, we model the number of samples that are in the decoded form (N_D). Given x_D as the portion of memory allocated for decoded data, the number of decoded samples in memory is $x_D \times S_{mem}$ divided by the size of a tensor, $M \times S_{data}$. However, N_D should also account for the case when the dataset is small enough so that both N_D and N_A fit entirely in memory, as shown in Equation 4.

$$N_D = \min \left(N_{total} - N_A, \frac{x_D \times S_{mem}}{M \times S_{data}} \right) \quad (4)$$

5.1.3 Encoded data in memory

Now, we model the performance of accessing encoded data stored in memory (DSI_E). this involves not only the cache,

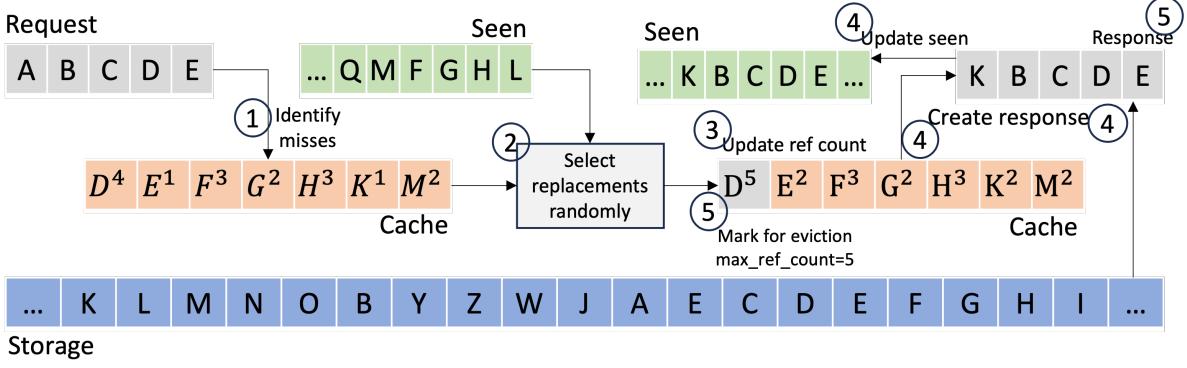


Figure 6: An example operation of ODS. ODS maintains two metadata structures: (1) per-job seen bit vector to track whether a data sample has been used during that epoch, and (2) per-dataset status and reference count for each data sample. These are used to opportunistically replace requested samples that miss in the cache with those that hit while ensuring randomness.

network, PCIe, and GPU, but also the CPU for decoding and augmenting the data samples (T_{D+A}). This resulting DSI_E is shown in Equation 5.

$$DSI_E = \min \left(\frac{B_{cache}}{S_{data}}, \frac{n \times B_{NIC}}{S_{data} + C_{nw}}, n \times T_{D+A}, \frac{n \times B_{PCIe}}{(M \times S_{data}) + C_{PCIe}}, n \times T_{GPU} \right) \quad (5)$$

For the number of encoded data samples in memory (N_E), we consider two scenarios: (1) when the dataset is small enough so that all N_E , N_D , and N_A can be fully cached, and (2) when the dataset is large. For the latter case, we compute the portion of memory allocated for encoded data, $x_E \times S_{mem}$, and divide it by the size of the encoded data, S_{data} . Both scenarios are considered in Equation 6.

$$N_E = \min \left(N_{total} - (N_A + N_D), \frac{x_E \times S_{mem}}{S_{data}} \right) \quad (6)$$

5.1.4 Data in storage

Finally, we model the performance of accessing storage, DSI_S , shown in Equation 7. This is similar to DSI_E in Equation 5, but also includes the storage throughput as the potential limiting factor for performance.

$$DSI_S = \min \left(DSI_E, \frac{B_{storage}}{S_{data}} \right) \quad (7)$$

The number of samples only in storage is simply the dataset that does not fit in memory, as shown in Equation 8

$$N_{storage} = N_{total} - N_A - N_D - N_E \quad (8)$$

5.1.5 Overall DSI performance model

We combine Equations 1–8 and express the overall DSI throughput ($DSI_{overall}$) as shown in Equation 9.

$$DSI_{overall} = \frac{N_A}{N_{total}} \times DSI_A + \frac{N_D}{N_{total}} \times DSI_D + \frac{N_E}{N_{total}} \times DSI_E + \frac{N_{storage}}{N_{total}} \times DSI_S \quad (9)$$

We use Equation 9 to estimate DSI throughput given different proportions of memory partitioned for caching encoded, decoded, and augmented data (x_E, x_D, x_A). We show the correctness of this model in the later validation section, § 6.

5.2 Opportunistic data sampling (ODS)

Given the optimal cache partitioning for improving the DSI throughput, ODS aims to improve the cache hit rate when multiple training jobs share the same dataset. The key idea behind ODS is to opportunistically supply data samples already present in the cache. However, it must still ensure that (1) a training job only sees the same data sample once in any epoch, (2) the same augmented samples are never reused across epochs, and (3) the order of supplied data is random.

We illustrate the operation of ODS through Figure 6. We represent each unique data sample with alphabets, and maintain (1) a per-job seen bit vector to track whether or not a sample has been seen during its epoch, and (2) a per-dataset status and reference count for each of its data samples. Although Seneca has three tiers of caches (augmented, decoded, and encoded), we illustrate with only the augmented cache for simplicity.

- When a batch request arrives, ODS identifies the misses (samples not in the cache) based on status.
- Using the seen bit vector for the requested job, ODS opportunistically replaces misses with hits (samples in the cache) that have not been seen. Hits that have been seen by the job do not replace misses.

3. The reference counts for samples that hit in the cache are incremented.
4. Response to the batch request is sent with replacements, and the seen bit vector is updated.
5. If any of the reference counts reach the defined threshold for eviction, a background thread removes the data samples and replaces them with different random samples from storage with their reference counts reset.
6. The seen bit vector is reset at the end of its epoch.

Using the seen bit vector, ODS guarantees that each job uses a data sample once during each epoch. With the eviction threshold set to the number of jobs, the reference count and seen bit vector together ensure that augmented data will not be used across epochs. Although the eviction of an augmented sample from the cache is deterministic, when it will be supplied to a job is nevertheless random as it depends on the composition of the requested random samples. Furthermore, which data samples populate the augmented cache after an eviction is based on a pseudo-random number generator, warranting the random sampling behavior of ODS.

The overhead of using ODS is negligible. First, the required metadata for ODS to maintain is only in the megabyte range. We use 1 bit per data sample for the per-job seen bit vector, and 1B per data sample for encoding the data status (augmented, decoded, encoded, or storage) and the reference count together. As an example, training 8 concurrent jobs on the ImageNet-1K dataset [15] (1.3M samples) yields 2.6MB of ODS metadata. Furthermore, the compute overhead of ODS is limited to only four additional metadata operations, all of which are constant time and in the nanoseconds range. Specifically, these are lookup and update operations on the in-memory ODS metadata.

5.3 Implementation

We implement Seneca by modifying PyTorch (v1.12.0) [3] dataloaders, with approximately 4200 lines of code changes. These changes include both MDP (§ 5.1) and ODS (§ 5.2). Seneca can be used as a drop-in replacement for the default PyTorch dataloader. For caching, Seneca uses Redis [4], an open-source in-memory key-value store, but other data stores can be used instead.

Table 4: Hardware configurations for the GPU servers used.

	In-house server	AWS p3.8xlarge	Azure NC96ads_v4
GPU config	2×RTX5000	4×V100	4×A100
GPU Mem (GB)	32	64	320
CPU config	AMD Ryzen 9 3950X	Intel Xeon E5-2686 v4	AMD EPYC 7V13
DRAM cap. (GB)	115	244	880
Network BW. (Gbit/s)	10	10	80
NFS BW. (MB/s)	500	256	250

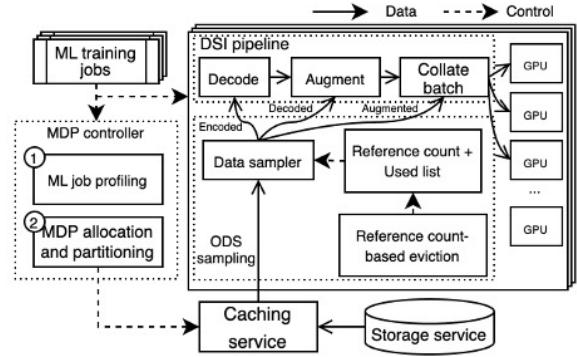


Figure 7: The overall architecture of Seneca. At initialization, MDP partitions the cache according to the system and dataset parameters. During runtime, ODS maximizes the cache hit rate by replacing samples that miss in the cache with hits.

Table 5: Performance model values for DSI model validation.

Notation	In-house server	AWS p3.8xlarge	Azure NC96ads_v4
T_{GPU}	4550 sample/s	9989 sample/s	14301 samples/s
T_{D+A}	2132 sample/s	3432 sample/s	9783 samples/s
T_A	4050 sample/s	6520 sample/s	12930 samples/s
B_{NIC}	10 Gb/s	10 Gb/s	80 Gb/s
B_{PCIE}	32 GB/s	32 GB/s	64 GB/s
B_{cache}	10 Gb/s	10 Gb/s	30 Gb/s
$B_{storage}$	500 MB/s	256 MB/s	250 MB/s
S_{cache}	64 GB	64 GB	64 GB
S_{data}	114 KB	114 KB	114 KB
M	5.12×	5.12×	5.12×

Figure 7 shows the overall architecture of Seneca. At the start of training, MDP determines cache partition sizes and allocates them in the caching service. We use a brute-force approach to find the optimal cache split by calculating DSI throughput for all combinations at 1% granularity. This approach is used for simplicity since the optimal cache split is typically calculated once per dataset and incurs negligible overhead (<1s). ODS replaces cache misses with hits in runtime and includes modules to track job progress and sample status ensuring randomization and uniqueness of data.

6 DSI Model Validation

We validate our DSI pipeline performance model by comparing its output with real measurements across four hardware platforms (1×in-house server, 2×in-house servers, 1×AWS server, and 1×Azure server) using several fixed cache partition sizes. The purpose of this validation is to assess the correlation between the model and actual performance, not to demonstrate that MDP achieves optimal results. Table 4 shows the known hardware specs for the server types used, while Table 5 lists profiled values like GPU and CPU throughput for our model. For each server type, we measure GPU

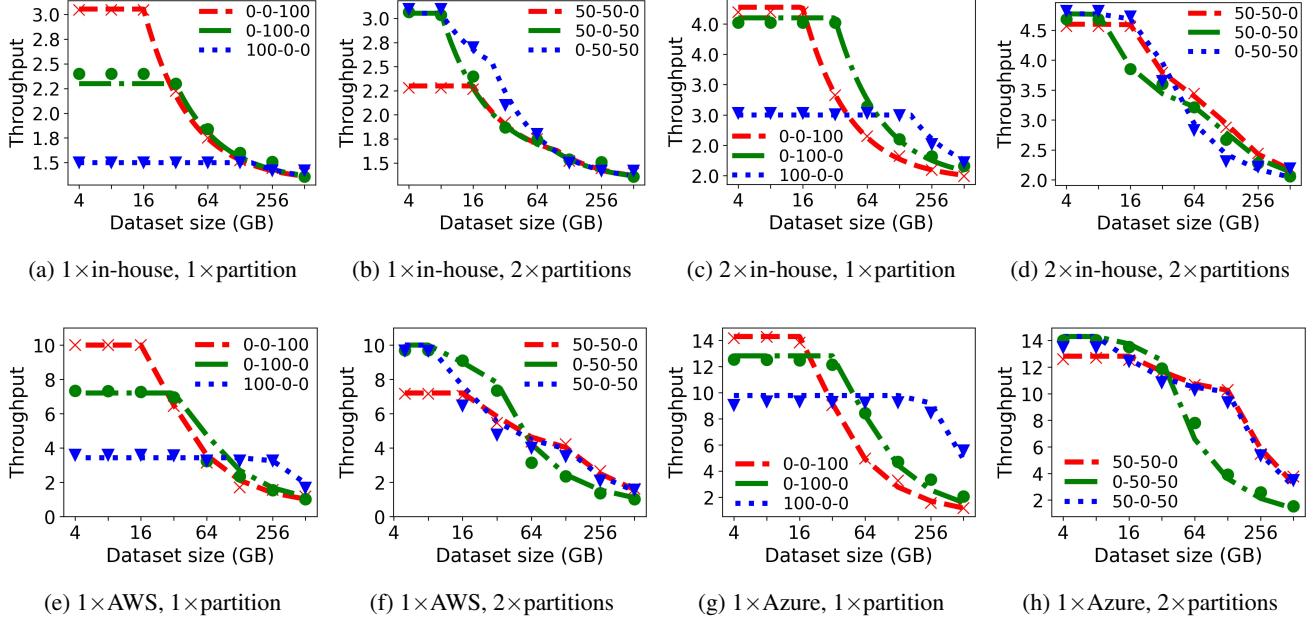


Figure 8: Validation of the DSI pipeline performance model against measured throughput in samples/sec (Y-axis) while varying the dataset size in GB (X-axis). The lines indicate modeled performance for each cache partition while the markings of the same color indicate measured performance. X-Y-Z denote a split of X% encoded cache, Y% decoded cache, and Z% augmented cache. The correlation between modeled and measured values is at least 0.90.

and CPU throughput using DS-Analyzer [39] and remote storage bandwidth using `fio` [1]. We double the performance values of the in-house server for validating our model on distributed training across 2 in-house servers. Overall, the Azure server generally outperforms the in-house and AWS servers, except in remote storage speed. We use the ImageNet-1K dataset [15], and replicate samples to generate a large dataset that reaches up to 512GB. We also limit the size of the caching service to 64GB on all systems to test scenarios where the dataset exceeds memory capacity.

Figure 8 compares modeled performance (lines) with measured performance (markings) for six different cache partitions across four configurations. There are countless partition possibilities, so we focus on simple configurations: three with a single cache and three with two equal-sized caches. Modeled and measured results are color-matched and show a strong correlation overall. The Pearson correlation coefficient for all 24 combinations is at least 0.90. The lowest correlations are 0.91 for the 50/50 encoded-augmented cache on the AWS server (blue line in Figure 8f) and 0.92 for the 100% encoded cache on the in-house server (blue line in Figure 8a). For distributed training on two in-house servers (Figures 8c and 8d), the bottleneck shifts to remote cache bandwidth (B_{cache}) due to limited network capacity, a constraint accurately predicted by our model.

As shown in the model, the relationship between the DSI pipeline throughput and the cache configuration is nontrivial. The DSI pipeline throughput depends on the performance of

training node hardware (GPU and CPU), cache and storage services, as well as the dataset and model parameters (size and sample size). For example, when the dataset is small, then it is advantageous to have preprocessed data in the cache: there is no reason not to as it avoids both preprocessing and I/O stalls as shown by the red lines in Figures 8a, 8c, 8e, and 8g. However, for larger datasets, the larger size of preprocessed data can lead to performance degradation due to increased cache misses. If only one type of cache can be used, using an encoded cache is better with large datasets, as illustrated by the blue lines and markings in Figures 8a, 8c, 8e, and 8g. With two types of caches, the best configuration is no longer a clear-cut answer, although, at very large dataset sizes, their throughputs become similar, shown in Figures 8b, 8d, 8f, and 8h.

7 Evaluation

We demonstrate the effectiveness of Seneca using several DNN and transformer models with the largest possible batch size up to 1024 on three datasets across five hardware configurations, incorporating single node and distributed training on a variety of GPUs: (1) 1x and 2x in-house servers with 115 GB remote cache over 10 Gbps, (2) AWS p3.8xlarge [8] VM with 400 GB remote cache over 10 Gbps on an AWS x2iedn.4xlarge [9] VM, and (3) 1x and 2x Azure NC96ads_v4 [11] VMs with 400 GB remote cache over 30 Gbps on an Azure E64-16s v6 [10] VM. Datasets are stored

Table 6: Characteristics of popular open source datasets used for training image classification models. Using the profiled performance model values, MDP determines the best split for the two servers: (X-Y-Z) indicate a split of X% encoded cache, Y% decoded cache, and Z% augmented cache. MDP splits in bold are presented in the evaluation.

Dataset	Images	Classes	Avg. Image size	Footprint	MDP				
					1×in-house server	2×In-House servers	AWS p3.8xlarge	1×Azure NC96ads_v4	2×Azure NC96ads_v4
ImageNet-1K [15]	1.3M	1000	114.62KB	142 GB	58-42-0	40-59-1	0-81-19	0-48-52	0-53-47
OpenImages V7 [28]	1.9M	600	315.84KB	517 GB	62-37-1	58-41-1	52-48-0	5-95-0	6-93-1
ImageNet-22K [2]	14M	22000	91.39KB	1400 GB	100-0-0	100-0-0	100-0-0	100-0-0	100-0-0

Table 7: Key features of evaluated dataloaders.

	Reduces CPU overhead	Improves cache hit rate	Supports multiple jobs
PyTorch [3]	✗	✗	✗
DALI [43]	✓	✗	✗
SHADE [26]	✗	✓	✗
MINIO [41]	✓	✗	✓
Quiver [31]	✗	✓	✓
MDP	✓	✗	✓
Seneca	✓	✓	✓

on a remote storage service, which in our case is an NFS server with 10-12 Gbps network bandwidth. We use image classification models due to their substantial preprocessing overhead, but the core concepts of our work can be applied to all preprocessing-heavy machine learning training. The characteristics of the datasets are summarized in Table 6 and represent a wide range of dataset sizes and sample sizes. How the cache should be partitioned depends on the system and dataset parameters, and Table 6 includes the MDP-determined partitions (encoded-decoded-augmented). Due to space constraints, we present results only for unique MDP partitions.

We evaluate two of our designs, MDP-only and Seneca (which includes both MDP and ODS), against five dataloaders. We provide a brief description for each of the compared dataloaders and summarize their key features in Table 7.

PyTorch is a popular open-sourced dataloader [3, 53].

DALI is an open-sourced optimized library that pipelines and offloads preprocessing tasks to the GPU [43].

SHADE caches and preferentially samples data with higher importance [26]. We use the unmodified version of SHADE that is publicly available.

MINIO is originally built on top of DALI and does not evict samples once in the cache [40]. We implement this same policy on top of PyTorch for evaluation.

Quiver samples 10× more data and forms a batch with those that return the fastest [31]. Quiver is not open-sourced, and we faithfully implement Quiver on top of PyTorch.

We answer the following questions in the evaluation.

- Do models converge faster with Seneca? (§ 7.1)

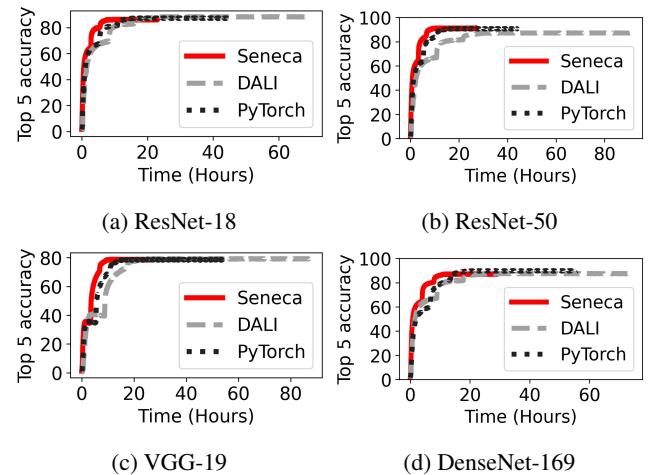


Figure 9: Top-5 accuracy of training four ML models on the Imagenet-1K dataset using the Azure VM. Seneca completes 250 epochs significantly faster without compromising accuracy.

- Does Seneca scale with hardware resources? (§ 7.2)
- Does Seneca scale with the number of training jobs? (§ 7.3)
- How does Seneca perform on different models and datasets? (§ 7.4)

7.1 End-to-end training performance

We demonstrate how four popular DNN model architectures converge by 250 epochs using Seneca on the Azure server, as illustrated in Figure 9. For this evaluation, we choose two GPU-intensive (VGG-19 and DenseNet-169) and two less GPU-intensive models (ResNet-18 and ResNet-50) training on ImageNet-1K which is a dataset commonly used to evaluate image model accuracies [22, 29, 54]. We choose PyTorch and DALI for comparison as training to convergence takes a considerable amount of time and the two open-sourced dataloaders are widely used and stable. We show the top-5 accuracy with respect to the time spent training. First, we observe that Seneca enables DNN models to achieve convergence accuracies significantly faster than PyTorch and DALI. Second, models trained with Seneca follow the same trend in

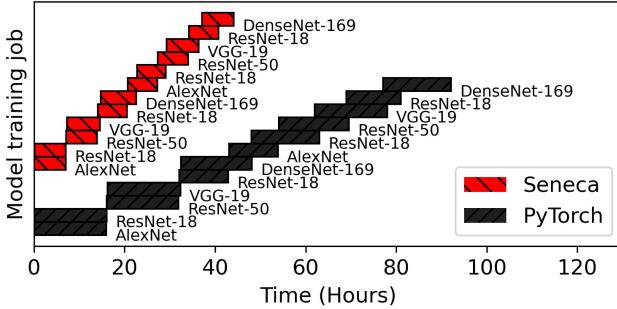


Figure 10: The model training time for 12 image classification jobs (50 epochs each) for 5 different models on the AWS server [8]. For the multi-job training, Seneca reduces the training time by 45.23% compared to PyTorch.

accuracy as those trained with PyTorch and DALI, with an error of less than 2.83% in the final training accuracy. This demonstrates that our optimizations (MDP and ODS) do not impact on training accuracy.

Next, we investigate time to convergence by measuring the end-to-end model training time for the DNN models in Figure 9. Compared to the PyTorch dataloader, Seneca enables these models to complete the 250 epochs and achieve accuracies of 86.1% for ResNet-18, 90.82% for ResNet-50, 78.78% for VGG-19, and 89.05% for DenseNet-169, with respective speed improvements of 48.51%, 38.09%, 49.16%, and 47.83%. When compared against DALI, the improvements are 66.88%, 70.00%, 68.53%, and 60.70% respectively. For single-job training, Seneca outperforms the other two dataloaders, thanks to MDP which enables caching samples in decoded and augmented formats.

We also simulate a real-world training environment on the AWS server using a scheduler to launch jobs arriving at random times. We limit the number of concurrent jobs to two. Figure 10 shows the training progress for 12 jobs (a mix of large and small models) on the ImageNet-1K dataset sharing the same DSI pipeline, each trained for 50 epochs. We make three key observations: (1) Seneca reduces the total training time to 45.23% of PyTorch. (2) Unlike PyTorch, where each job has an independent data pipeline, Seneca’s caching system allows data sharing, reducing redundant fetch and preprocessing operations. (3) the last job (DenseNet-169) completed in 6.97 hours compared to the sixth job (DenseNet-169) which completed in 7.84 hours since it ran alone during the second half, fully utilizing the DSI resources.

7.2 Hardware sensitivity

We evaluate Seneca’s performance across different training nodes and configurations to demonstrate its scalability on larger and distributed systems using the OpenImages dataset. We evaluate across the in-house, AWS, and Azure servers

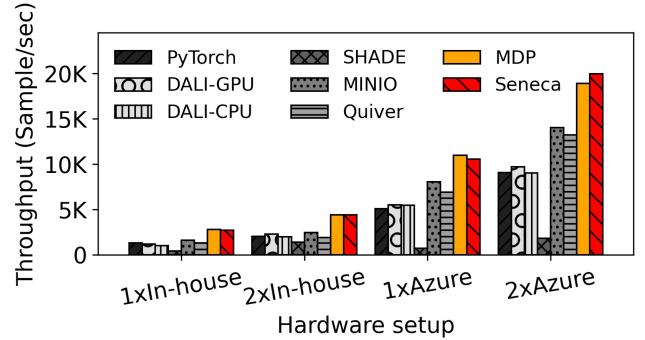


Figure 11: Throughput for single job distributed training on one and two in-house and Azure servers. On two Azure nodes, Seneca is 1.89 \times faster than one.

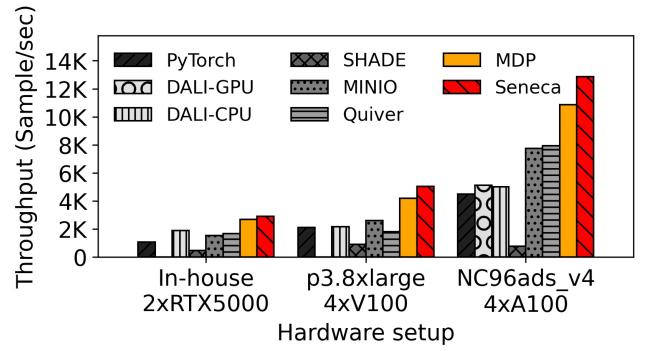


Figure 12: Throughput for training 2 jobs concurrently on different hardware platforms. Seneca performs well across a wide range of system configurations.

to cover a wide range of training systems. We test on the in-house and Azure servers to evaluate multi-node distributed training and on the in-house, AWS, and Azure servers to evaluate concurrent training across three distinct training systems.

We evaluate Seneca’s performance on distributed systems by measuring single-job training throughput on two in-house and two Azure servers in distributed data parallel mode with remote caching (Figure 11). Network bandwidth and gradient communication are key considerations in this setup. On 2 \times in-house servers, Seneca is 1.6 \times faster than MINIO and 1.62 \times faster than Seneca on a single in-house server. While gradient communication overhead is negligible due to ring-reduce [56], the 10 Gbps network bottlenecks throughput, limiting scaling to 1.62 \times instead of the expected $\approx 2\times$. On an Azure server with 80 Gbps, this bottleneck is eliminated, allowing Seneca to scale by 1.89 \times going from one to two nodes. On 2 \times Azure nodes, Seneca outperforms MINIO, the next best by 42.39%.

Next, we evaluate Seneca’s scalability across various ML training systems. Figure 12 shows throughput for two jobs training concurrently on the in-house, AWS, and Azure servers. While the AWS and Azure servers have slower remote storage bandwidths, they are faster overall, with more pow-

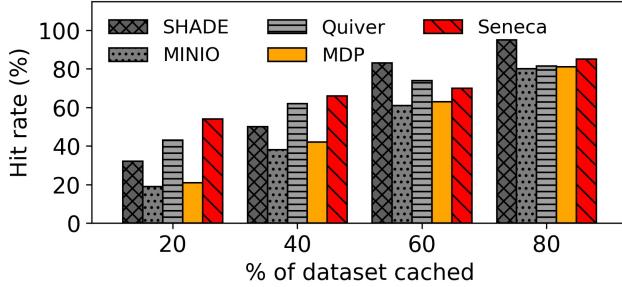


Figure 13: Cache hit rate when concurrently training three models. Seneca improves cache efficiency, achieving 54% hit rate with only 20% of the dataset cached.

Table 8: CPU and GPU utilization for four concurrently training jobs on the in-house server using various dataloaders.

	PyTorch	DALI-CPU	MINIO	Quiver	MDP	Seneca
CPU	88%	88%	91%	96%	43%	54%
GPU	72%	76%	79%	80%	98%	98%

erful GPUs, which are better suited for ML training [38] as shown in Table 5. We make three observations: (1) Seneca’s throughput increases by 4.44× when moving from the in-house server ($2 \times$ RTX5000) to the Azure server ($4 \times$ A100). (2) Seneca outperforms the next best dataloader by 1.61× on Azure (vs. Quiver), 1.93× on AWS (vs. MINIO), and 1.52× on the in-house server (vs. DALI-CPU). (3) DALI-GPU fails for two or more concurrent jobs on the in-house and AWS servers due to insufficient GPU memory.

Finally, we evaluate Seneca’s cache hit rate with varying cache sizes to evaluate its performance with larger datasets and limited cache capacity. Figure 13 presents the cache hit rate, calculated as the total cache hits across all partitions divided by the number of samples in the dataset, during concurrent training of AlexNet, ResNet-50, and MobileNetV2 on ImageNet-1K. Seneca achieves a 54% hit rate with just 20% of the dataset cached, 11% higher than Quiver, the next best. Seneca improves the hit rate by replacing uncached samples with cached ones not seen in the current epoch. Even with 40% of the dataset cached, Seneca maintains the highest hit rate of 66%. With 60% and 80% cached, SHADE’s hit rate surpasses Seneca but throughput is still the slowest due to it’s single threaded design. MINIO and MDP show hit rates roughly equal to the percentage of cached data since they do not implement specialized policies to improve the hit rate.

7.3 Load sensitivity

We evaluate Seneca’s impact on concurrent model training by measuring DSI pipeline throughput on the Azure server with 400 GB remote cache while varying the number of concurrent jobs (Figure 14). We use the OpenImages dataset as it is

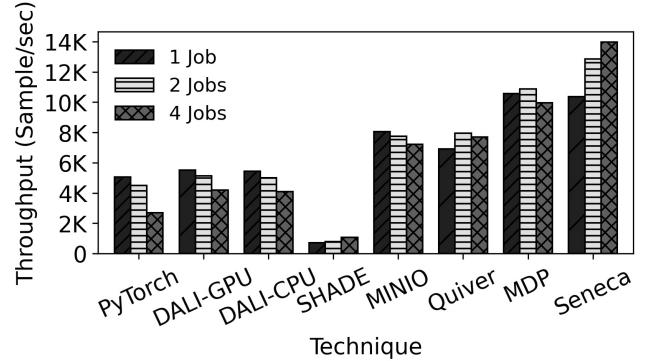


Figure 14: Aggregate DSI throughput for up to four concurrent models on the Azure server with $4 \times$ A100 GPUs. Seneca boosts DSI pipeline throughput by 1.81× compared to Quiver for four jobs.

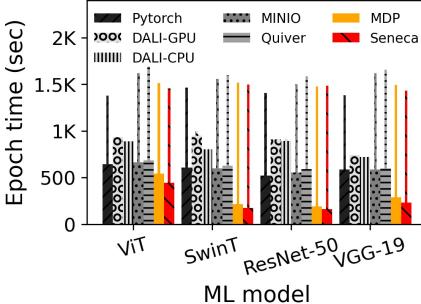
larger than the size of the remote cache and thus represents a common scenario. We also show the GPU and CPU utilization for four concurrently training jobs in Table 8 to show that Seneca saturates GPU resources.

We make five observations: (1) Our optimizations, MDP and Seneca outperform all other dataloaders by at least 28.97% (single job on MINIO vs single job on Seneca). (2) For four concurrently training jobs, Seneca is the best performing and outperforms Quiver, the next best performing dataloader by 1.81×. (3) For four or more concurrent jobs on a single Azure server, the performance of Seneca is bounded by the GPU at 98% utilization (Table 8), limiting speedup with added concurrency. (4) PyTorch, DALI, MINIO, and Quiver are limited by I/O and CPU preprocessing and do not scale well with concurrent jobs. (5) Seneca outperforms SHADE by 13.18× due to SHADE’s single-threaded design.

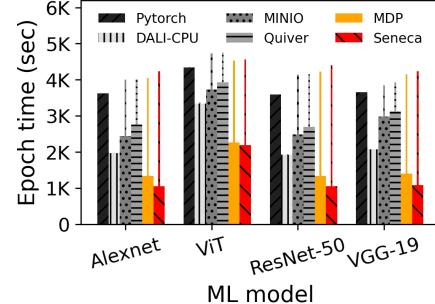
7.4 ML model and dataset sensitivity

Finally, we show Seneca’s generality across different ML models, datasets, and hardware setups. For each dataloader, we train two models concurrently and measure the first epoch completion time (ECT) and the average ECT for subsequent epochs. The first epoch time represents execution with cold page and user-level caches, while subsequent epoch times reflect execution with warmed-up caches. We exclude SHADE due to its single-threaded design. For this evaluation, we use two transformer models—ViT [16], and SwinT [37] and three DNN models—VGG-19, ResNet-50, and AlexNet to cover a wide range of model architectures.

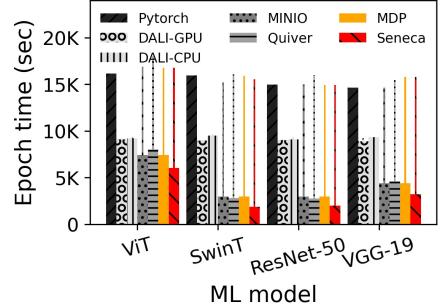
We make three observations for training with ImageNet-1K on one Azure server with 400 GB remote cache (Figure 15a): (1) Seneca’s stable ECT for the largest vision transformer (ViT-h) is 31.36% lower than the next best dataloader, PyTorch while for ResNet-50, it is 3.45× faster than MINIO. (2) When the dataset size (142 GB) is significantly smaller



(a) Imagenet-1K on 1×Azure



(b) OpenImages on 1×AWS



(c) ImageNet-22K on 1×Azure

Figure 15: Stable epoch completion time (bars) and first-epoch time (lines) for concurrent training of popular DNN models (two jobs) across different datasets, servers, and dataloaders. The first epoch reflects training when the cache is not warmed up, while subsequent epochs represent training with the cache in use. Lower is better.

than DRAM size (880 GB), the entire dataset is in page-cache and PyTorch’s stable ECT is faster than DALI by at least 31.36% (VGG-19). (3) MINIO and Quiver’s encoded data cache cannot mitigate redundant decoding and augmentation on the CPU, reducing the gains from their optimizations.

Next, to understand the impact of larger sample sizes ($2.75 \times$ larger than ImageNet-1K) on training, we examine the ECTs for models shown in Figure 15b with the OpenImages dataset on the AWS server with 400 GB remote cache. We make four observations: (1) On the AWS system, DSI pipeline is a bottleneck due to limited I/O bandwidth and CPU threads. On this system, training VGG-19 with Seneca is $1.91 \times$ faster than with DALI-CPU. (2) By caching more data samples in the decoded form to mitigate the higher DSI overheads, Seneca’s stable ECT is lower by 87.22% for AlexNet, 87.15% for ViT, 85.53% for ResNet-50, and 47.85% for VGG-19 compared to the DALI-CPU, the next best dataloader. (3) Caching dataloaders (MINIO, and Quiver) reduce fetch stalls by caching encoded data, reducing stable ECT by up to 39.85% (ResNet-50 on MINIO). (4) On the V100 GPUs of the AWS server, DALI-GPU fails for concurrent training jobs due to limited GPU memory.

For ImageNet-22K (1.4 TB) on the Azure server with 400 GB remote cache (Figure 15c), we make four observations: (1) PyTorch, DALI-CPU, and DALI-GPU which depend on the page-cache perform poorly for large datasets. (2) Due to the limited cache size, MDP allocates 100% of the available memory for caching raw data, thus performing similar to MINIO. (3) Thanks to ODS, Seneca’s ECT is 29.35% lower on average when compared to the next best dataloader across all evaluated models. (4) Seneca reduces stable ECT for SwinT by $8.37 \times$ by mitigating I/O and preprocessing bottlenecks.

8 Conclusion

We present Seneca, a cache partitioning and data sampling system for optimizing concurrent DNN training on distributed

systems. Seneca consists of two techniques: model-driven partitioning that finds the best way to partition the cache to improve DSI throughput and opportunistic data sampling that preferentially selects cached data samples to maximize the cache hit rate. In doing so, Seneca reduces model training time and multi-job makespan by 45.23% and improves the DSI pipeline throughput by up to $3.45 \times$ on concurrent jobs.

Acknowledgments

We would like to thank our shepherd, Ali Anwar, and the anonymous reviewers for their constructive feedback. We also thank Syracuse University Information Technology Services (ITS) for providing computing resources and support. This research was supported in part by the National Science Foundation under award numbers CNS-2008453, CSR-2323100, CAREER-2338457, CSR-2402328, and CAREER-2443515, and by Samsung through the Alternative Sustainable and Intelligent Computing Industry-University Cooperative Research Center (NSF #1822165). This work is also supported through the Global Research Support Program in the Digital Field supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant RS-2024-00428758, funded by the Ministry of Science and ICT (MSIT), South Korea. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporting agencies.

References

- [1] Flexible I/O tester. <https://github.com/axboe/fio>.
- [2] ImageNet-22K. <https://image-net.org/download-images.php>.
- [3] PyTorch. <https://pytorch.org>.

- [4] Redis. <https://redis.io>.
- [5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Daniel Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from <https://www.tensorflow.org/>.
- [6] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jirí Simsa, and Chandramohan A. Thekkath. tf.data service: A case for disaggregating ML input data processing. In *ACM Symposium on Cloud Computing (SoCC)*, pages 358–375, 2023. <https://doi.org/10.1145/3620678.3624666>.
- [7] Amazon Web Services (AWS). Amazon EC2 P2 instances. powerful, scalable GPU instances for high-performance computing. <https://aws.amazon.com/ec2/instance-types/p2/>, 2023.
- [8] Amazon Web Services (AWS). Amazon EC2 P3 instances. accelerate machine learning and high performance computing applications with powerful GPUs. <https://aws.amazon.com/ec2/instance-types/p3/>, 2023.
- [9] Amazon Web Services (AWS). AWS EC2 X1 instance types. <https://aws.amazon.com/ec2/instance-types/x2i/>, 2024.
- [10] Azure. 'E' family memory optimized VM size series. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/memory-optimized/e-family?tabs=epsv6%2Cceasv6%2Cev5%2Cedv5%2Ceasv5%2Cepsv5>, 2025.
- [11] Azure. NC_A100_v4 sizes series. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nca100v4-series?tabs=sizebasic>, 2025.
- [12] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 626–635. IEEE, 2020. <https://ieeexplore.ieee.org/document/9155446>.
- [13] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Xian-He Sun, and Gang Chen. iCache: An importance-sampling-informed cache for accelerating I/O-bound DNN model training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 220–232. IEEE, 2023. <https://ieeexplore.ieee.org/document/10070964>.
- [14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012. <https://dl.acm.org/doi/10.5555/2999134.2999271>.
- [15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009. <https://doi.org/10.1109/CVPR.2009.5206848>.
- [16] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. <https://arxiv.org/pdf/2010.11929/1000>.
- [17] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning I/O. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 92:1–92–15, 2021. <https://doi.org/10.1145/3458817.3476181>.
- [18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019. <https://www.usenix.org/conference/atc19/presentation/duplyakin>.
- [19] GCP. GCP GPU platforms. https://cloud.google.com/compute/docs/gpus#nvidia_p100_gpus, 2024.
- [20] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *USENIX Annual Technical Conference (ATC)*, pages 689–706, 2022. <https://www.usenix.org/conference/atc22/presentation/graur>.
- [21] Dan Graur, Oto Mraz, Muyu Li, Sepehr Pourghannad, Chandramohan A Thekkath, and Ana Klimovic. Pecan: Cost-efficient ml data preprocessing with automatic transformation ordering and hybrid placement. In *USENIX Annual Technical Conference (ATC)*,

- pages 649–665, 2024. <https://www.usenix.org/conference/atc24/presentation/graur>.
- [22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. <https://doi.org/10.1109/CVPR.2016.90>.
- [23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019. https://proceedings.neurips.cc/paper_files/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html.
- [24] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1825–1839, 2022. <https://dl.acm.org/doi/abs/10.1145/3514221.3517848>.
- [25] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *PMLR International Conference on Machine Learning (ICML)*, pages 2525–2534, 2018. <http://proceedings.mlr.press/v80/katharopoulos18a.html>.
- [26] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. SHADE: Enable fundamental cacheability for distributed deep learning training. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 135–152, 2023. <https://www.usenix.org/conference/fast23/presentation/khan>.
- [27] Taeyoon Kim, ChanHo Park, Mansur Mukimbekov, Hee-lim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon. Fusionflow: Accelerating data preprocessing for machine learning with CPU-GPU cooperation. *Proceedings of the VLDB Endowment*, 17(4):863–876, 2023. <https://doi.org/10.14778/3636218.3636238>.
- [28] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Siyamalan Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Vicente Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. Openimages dataset. <https://storage.googleapis.com/openimages/web/index.html>, 2017.
- [29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Conference on Neural Information Processing Systems (NIPS)*, pages 1106–1114, 2012. <https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>.
- [30] Michael Kuchnik, Ana Klimovic, Jiri Simska, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems*, 4:33–51, 2022. https://proceedings.mlsys.org/paper_files/paper/2022/hash/d0e90e9a9310570dfa643aa3b2da6e89-Abstract.html.
- [31] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *Conference on File and Storage Technologies (FAST)*, pages 283–296, 2020. <https://www.usenix.org/conference/fast20/presentation/kumar>.
- [32] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.
- [33] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *USENIX Annual Technical Conference (ATC)*, pages 537–550, 2021. <https://www.usenix.org/conference/atc21/presentation/lee>.
- [34] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014. https://papers.nips.cc/paper_files/paper/2014/hash/935ad074f32d1e8f085a143449894cdc-Abstract.html.
- [35] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175, 2021. <https://www.usenix.org/conference/atc21/presentation/lim>.
- [36] Jie Liu, Bogdan Nicolae, and Dong Li. Lobster: Load balance-aware I/O for distributed DNN training. In *ACM International Conference on Parallel Processing (ICPP)*,

- pages 1–11, 2022. <https://dl.acm.org/doi/abs/10.1145/3545008.3545090>.
- [37] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021. <https://ieeexplore.ieee.org/document/9710580>.
- [38] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018. <https://ieeexplore.ieee.org/document/8425458>.
- [39] Jayashree Mohan. MSR-fiddle/DS-Analyzer. <https://github.com/msr-fiddle/DS-Analyzer/tree/main>, 2021.
- [40] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on multi-tenant clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 579–596, 2022. <https://www.usenix.org/conference/osdi22/presentation/mohan>.
- [41] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 14(5):771–784, 2021. <http://www.vldb.org/pvldb/vol14/p771-mohan.pdf>.
- [42] Derek Gordon Murray, Jiri Simska, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proceedings of Very Large Data Bases Endowment (PVLDB)*, 14(12):2945–2958, 2021. <http://www.vldb.org/pvldb/vol14/p2945-klimovic.pdf>.
- [43] NVIDIA. NVIDIA Data Loading Library. <https://developer.nvidia.com/dali>, 2023.
- [44] NVIDIA. NVIDIA A100. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf>, 2024.
- [45] NVIDIA. NVIDIA H100. <https://www.nvidia.com/en-us/data-center/h100/>, 2024.
- [46] NVIDIA. NVIDIA P100. <https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf>, 2024.
- [47] NVIDIA. NVIDIA TESLA K20. https://www.nvidia.com/content/PDF/kepler/NV_DS_TeslaK_Family_May_2012_LR.pdf, 2024.
- [48] NVIDIA. NVIDIA TESLA K40. <https://www.nvidia.com/content/PDF/kepler/nvidia-tesla-k40.pdf>, 2024.
- [49] NVIDIA. NVIDIA TESLA K80. <https://www.nvidia.com/en-gb/data-center/tesla-k80/>, 2024.
- [50] NVIDIA. NVIDIA V100. <https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf>, 2024.
- [51] NVIDIA. NVLink and NVLink Switch. <https://www.nvidia.com/en-us/data-center/nvlink/>, 2024.
- [52] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: an extreme-scale neural network training server architecture by systematically balancing operations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 825–838. IEEE, 2020. <https://ieeexplore.ieee.org/document/9251955>.
- [53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>.
- [54] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Selam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018. <https://doi.org/10.48550/arXiv.1812.00669>.
- [55] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009. <https://dl.acm.org/doi/10.1145/1553374.1553486>.

- [56] Zhenheng Tang, Shaohuai Shi, Wei Wang, Bo Li, and Xiaowen Chu. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020. <https://doi.org/10.48550/arXiv.2003.06307>.
- [57] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fast-Flow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proceedings of the Very Large Data Bases Endowment (VLDB)*, 16(5):1086–1099, 2023. <https://www.vldb.org/pvldb/vol16/p1086-um.pdf>.
- [58] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 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)*, pages 945–960, 2022. <https://www.usenix.org/conference/nsdi22/presentation/weng>.
- [59] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, et al. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023. <https://doi.org/10.1145/3589773>.
- [60] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. cedar: Optimized and unified machine learning input data pipelines. *Proc. VLDB Endow.*, 2025. <https://doi.org/10.14778/3705829.3705861>.
- [61] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Huawei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 1042–1057, 2022. <https://doi.org/10.1145/3470496.3533044>.

A Artifact Appendix

Abstract

This paper proposes Seneca, a system that alleviates input data preprocessing bottlenecks and reduces training time for concurrent jobs. It introduces two key techniques: (1) a performance model for the data pipeline that optimally partitions the cache among three types of data, and (2) a cache-aware sampling strategy where Seneca prioritizes serving cached

data over uncached data during random batch sampling. This artifact includes the code and the steps to train a model using Seneca and comparing it against popular baselines. The experiments have been performed on an in-house server as well as VMs from AWS and Azure.

Scope

The provided code and scripts support the reproduction and evaluation of the following:

- Training performance of image models using Seneca under varying cache allocations.
- Impact of Seneca on final model accuracy.
- Comparative performance of popular open-source dataloaders, including PyTorch and NVIDIA DALI.
- Performance evaluation of academic systems such as SHADE, MINIO, and Quiver.
- Support for training and evaluating multiple models and model types using Seneca.

Contents

A.0.1 PyTorch and Torchvision

The implementation accompanying this work is developed using PyTorch, a widely adopted open-source library for machine learning model training and inference, and TorchVision, a companion library that provides efficient image and video data loading and preprocessing utilities.

While the artifact is built on top of PyTorch and TorchVision, the underlying concepts and design are generalizable and can be applied to a wide range of machine learning libraries which have compute-intensive data preprocessing pipelines.

A.0.2 Redis

This work uses Redis as the in-memory datastore for caching data samples in various stages of processing—such as encoded, decoded, or pre-processed forms. Redis was chosen for its simplicity, low-latency performance, and broad community support. However, the design is not tied to Redis; any high-performance in-memory key-value store can be used as a drop-in replacement.

The maximum memory allocated for caching can be configured via a flag passed to the training script or by directly modifying the redis configuration file. If the cache is deployed on the same node as the training job, care must be taken to provision sufficient memory to accommodate both the cache and the training process. Alternatively, the cache can be hosted on a separate node or distributed across a cluster of caching nodes, depending on workload size and memory constraints.

A.0.3 Dataloader comparisons

Seneca has been evaluated against several dataloaders, including:

- PyTorch’s native dataloader
- NVIDIA DALI, both with and without GPU offload
- Dataloaders from prior academic and industry works, such as SHADE, MINIO, and Quiver

All baseline implementations are integrated on top of a common PyTorch version to ensure consistency. Each dataloader can be selected and evaluated using appropriate flags in the provided training script, enabling reproducible and fair comparisons across different data loading backends.

A.0.4 Docker container

To simplify setup, we provide a pre-built open-source Docker container with all the necessary tools, dependencies, and libraries required to run Seneca. This container is based on the NVIDIA PyTorch base image from the NGC catalog (<https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch>), ensuring compatibility with GPU-accelerated workloads.

You can access the container via the link provided in § A.0.4. For step-by-step setup instructions, please refer to the detailed documentation available in our GitHub repository: https://github.com/swiftomkar/seneca-fast26-pytorch/blob/master/AEC_readme.md

Hosting

The artifact is available on github repositories:

- **PyTorch:** <https://github.com/swiftomkar/seneca-fast26-pytorch>
- **TorchVision:** <https://github.com/swiftomkar/seneca-fast26-torchvision>
- **Seneca Docker:** https://hub.docker.com/repository/docker/omkarbdesai/seneca_cudall.7_cudnn8.5/general

Requirements

To run Seneca, you will need an NVIDIA GPU with CUDA support. The system has been tested on NVIDIA Quadro RTX 5000, V100, and A100 GPUs. Please ensure the machine has sufficient DRAM for local caching in addition to the memory required by the training process. Alternatively, you may configure a remote caching node.

Our evaluations were conducted on machines with a minimum of 115 GB of DRAM and a 16-core CPU. If you opt for a remote caching setup, ensure that the system has sufficient network bandwidth to maintain optimal performance.