# A Deep Learning Dataloader with Shared Data Preparation

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Parallelly executing multiple training jobs on overlapped datasets is a common practice in developing deep learning models. By default, each of the parallel jobs prepares (i.e., loads and preprocesses) the data independently, causing redundant consumption of I/O and computations. Although a centralized cache component can reduce the redundancies by reusing the data preparation work, each job's random data shuffling results in a low sampling locality causing heavy cache thrashing. Prior work tries to improve the sampling locality by enforcing all the training jobs loading the same dataset in the same order and pace. However, such a solution is only efficient under strong constraints: all jobs are trained on the same dataset with the same starting moment and training speed. In this paper, we propose a new data loading method for efficiently training parallel DNNs with much flexible constraints. Our method is still highly efficient when different training jobs use different but overlapped datasets and have different starting moments and training speeds. To achieve this, we propose a dependent sampling algorithm (DSA) and a domain-specific cache policy. Moreover, a novel tree data structure is designed to efficiently implement DSA. Based on the proposed technologies, we implemented a prototype, named JOADER, which can share data preparation work as long as the datasets are overlapped for different training jobs. We evaluate the proposed JOADER , showing a greater versatility and superiority of training speed improvement (up to 200% in ResNet18) without affecting the accuracy.

## 1 Introduction

The rapid development of deep learning frameworks and tools [3, 22] has enabled and facilitated researchers and practitioners in a multitude of disciplines (e.g., physics [28], chemistry [18], biology [12], and Earth science [23] in addition to computer science) to start developing their own DNN models solving various problems at hand. In such cases, they usually need to simultaneously run multiple training jobs on the same dataset or overlapped datasets[1], due to the purpose of model selection [15], hyper-parameter tuning [6], network architecture searching [25], etc.

Although widely used, the training of DNNs is usually time-consuming and tricky, which may affect the development efficiency, especially for general users from fields other than artificial intelligence. In literature, various approaches have been proposed to reduce the DNN training time such as data preparation optimization [4], communication overhead reduction [10, 9, 29], GPU memory optimization [8, 16, 26], and compiler-based operator optimization [7, 13, 17].

[1]This could happen when we train models on different datasets sharing a particular portion of data, e.g., running the full experiment on ImageNet and further tuning parameters on TinyImageNet [11]).

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Different from the above existing work, our focus is on the data preparation aspect, or more specifically, the data loading aspect, in the context of parallel DNN training. On the one hand, Mohan et al. [19] have observed that data preparation, which consists of data loading and the subsequent data preprocessing, occupies a significant portion of parallel DNN training time. This is due to the fact that multiple training jobs on the same data copy essentially create redundant data preparation efforts. Ideally, to reduce the load of hardware and boost the training efficiency, the prepared data should be cached in memory and serve for multiple jobs. However, the cache may thrash with a high probability as each job adopts a fully random shuffling strategy that independently shuffles the dataset in each epoch. On the other hand, it is until recently have some studies investigated the shuffling process itself [20, 19, 21]. These studies investigate to what extent the fully random shuffling is required and have observed that various shuffling strategies breaking the full randomness still yield competitive results. We name such randomness as *correlated randomness* as opposed to the full randomness. For example, prior work [19, 20] proposed to solve the cache thrashing issue by enforcing that all jobs iterate over the dataset in the same order and pace, and observed no accuracy loss. However, such a solution maximizes its performance under some strong constraints, i.e., all training jobs have to start simultaneously, use the same dataset, and have similar training speeds.

This paper proposes a new data loading method for training multiple parallel jobs. Specifically, we aim to relax the constraints of prior work in three aspects: 1) all jobs can start and end freely, 2) different datasets can be used as long as they are partially overlapped, and 3) jobs' training speeds may vary widely. For the above purposes, we first propose the *dependent sampling algorithm* (DSA) to schedule the sampling for each job while ensuring their correlated randomness. DSA is inspired by the correlated sampling theory [5] and it mainly involves three steps: 1) divide the dataset into two parts of intersection set and difference set, 2) correlate the selections between intersection and difference, and 3) share some operations for the intersection set to improve locality Furthermore, we design a domain-specific cache policy *RefCnt* that is tailored for parallel DNN training. It is aware of the fact that all jobs iterate over the dataset once in an epoch, and thus it can evict data that is least likely to be used in the near future. Finally, we design a novel data structure *dependent sampling tree* to efficiently implement the DSA by elaborately organizing the intersection sets and difference sets. Based on the above technologies, we implement a prototype system named JOADER and integrate it into PyTorch for evaluation. The evaluation results show that JOADER can boost training efficiency in more flexible cases. For example, it achieves up to 200% training speedup when models of different sizes are parallelly trained without affecting the accuracy.

The main contributions of this paper inlcude:

- A new dataloader for parallel DNN training on overlapped datasets.
- A sampling algorithm to increase the sampling locality with guaranteed randomness. The algorithm reaches the global optima when there are two training jobs.
- A domain-specific cache policy and a novel tree-based data structure for efficient implementation.
- Experimental evaluations demonstrating the performance improvements.

The rest of this paper is organized as follows. Section 2 introduces the background information. Section 3 describes the dependent sampling algorithm and our cache policy, followed by the description of the dependent sampling tree in Section 4. Section 5 presents the evaluation of the implemented dataloader JOADER. The paper is concluded in section 6.

# 2 Background

## 2.1 Problem Statement

In each epoch, a DNN training job sends data requests to the storage device and passes through the dataset in random order. For the brevity of the presentation, we assume that each data request involves only one data point/element, and we name the data access order over all the data elements as the *access path* for each epoch. Consider an example in Figure 1, where we assume two training

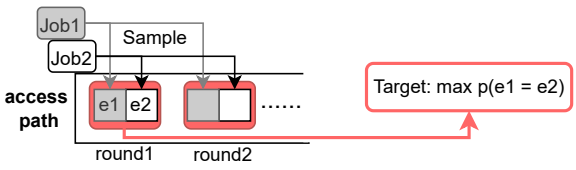



Figure 1: The access path of two training jobs. The target is to maximize the probability of sampling the same element in each round for all the jobs.

jobs are running, and the two sampled data elements ('e1' and 'e2') requested by the two jobs are packaged and prepared together in one *round*.

Cache can hold the prepared data to serve the future duplicated requests. To fully utilize the cache when multiple DNN models are being trained simultaneously in a server, one could generate a cache-friendly access path by encouraging all jobs to request the same data within a short period, i.e., improving the sampling locality in each round. Meanwhile, we still need to ensure the correlated randomness for each job. To this end, we formulate the problem as follows:

**Problem 1** *Assume there are $n$ training jobs on overlapped datasets, where the $i$-th job $J_i$ trains a model on dataset $D_i$ with cardinality $|D_i|$. In each round, $p(e_j^i) = \frac{1}{|D_i|}$ indicates the probability of data element $e_j$ being picked via sampling uniformly at random by $J_i$. For a given round, the goal is to maximize $p(e^1 = e^2 = ... = e^n)$, while maintaining $p(e^i) = \frac{1}{|D_i|}$.*

### 2.2 Related Work

**Dataloader for multiple training jobs.** Cerebro [20] is proposed to avoid redundant data preparation work for the task of model selection. It partitions the dataset across the servers into clusters and hopes the models iterate over data from one server to another instead of shuffling data. Although it can help avoid redundant data preparation work across servers, the work is still repeated on a single server. CoorDL [19] provides a more general solution for hyper-parameter optimization. In one turn, it distributes one batch to each job and caches the batch into the buffer for future usage. The batches are evicted until all jobs have consumed them in an epoch. However, CoorDL still has some limitations: 1) all jobs must be set up with the same batch size; 2) when the jobs have different training speeds, the faster jobs must wait for the slower ones; 3) CoorDL cannot handle situations where the jobs arrive at different moments or are trained on different datasets.

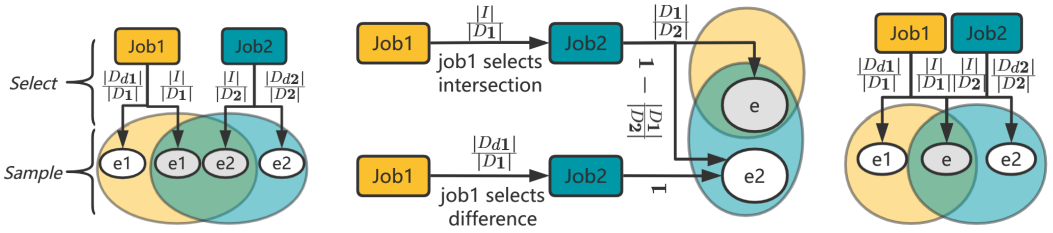
**Domain-specific caching.** Crafting a cache for a specific domain is not a new idea [24]. For example, Quiver [14] uses local solid-state drive (SSD) caches to eliminate the impact of slow reads from the remote storage. However, it is too expensive to prepare enough memory against cache thrashing for DNN training, especially considering the fact we need to cache not only the previous reading but also the results of data preprocessing. In this paper, we propose a well-designed sampling algorithm to improve the sampling locality with a small cache.

**Correlated sampling.** As a theoretical problem, correlated sampling [5] aims to minimize the probability of two sampling results being equal, given that the two players are sampling uniformly at random from two subsets of the same set. Some sampling strategies [27] give the optimal solution for two players. In this paper, we practically solve a related problem by designing an algorithm and a novel data structure for efficient implementation while enjoying their theoretical results.

## 3 Dependent Sampling Algorithm

### 3.1 Algorithm Design

To solve Problem 1, we propose a dependent sampling algorithm (DSA) to maximize the sampling locality while ensuring correlated randomness. In the following, we first present the algorithm in two-job case and then extend it to the n-job case.




(a) *Intersection calculating*: 1) calculate the intersection and difference; 2) let job select subset first and then sample in this subset randomly.

(b) *Dependent selecting*: Job2 makes decisions conditioned on the decisions of Job1, under the constraints that the probabilities of Job2 selecting difference set and intersection are uniformly distributed according to the total probability law.

(c) *Shared sampling*: if two jobs both select the intersection, then they share the sampling results.

Figure 2: Dependent sampling algorithm in the two-job case.

**Two-job Case.** The proposed *dependent sampling algorithm* (DSA) involves three steps: 1) intersection calculating, 2) dependent selecting, and 3) shared sampling. For a better illustration, we assume two jobs $J_1$ and $J_2$ are being trained on two overlapped datasets $D_1$ (the yellow circle) and $D_2$ (the green circle), respectively, as shown in Figure 2.

First, we divide the datasets into three subsets: *intersection* set $I = D_1 \cap D_2$ and two *difference* sets $D_{d1} = D_1 \setminus I, D_{d2} = D_2 \setminus I$. For the default *independent sampling algorithm* (ISA), which is currently used by PyTorch, it then processes the data sampling uniformly at random in the dataset for each job independently. As shown in Figure 2a, the ISA algorithm can be divided into two steps: *selecting* and *sampling* procedures. In the selecting procedure, each job selects between its difference and intersection sets for sampling. In the sampling procedure, each job randomly picks an element $e$ from the subset chosen in the previous step. This two-step process maintains the uniform distribution of sampling for each job.

Then, the proposed DSA takes a careful modification in the sampling procedure. As shown in Figure 2c, if the two jobs both select the intersection set, one shares the sampling result to another. We call this modification *shared sampling*. Based on shared sampling, the probability that two jobs selecting the same element is equal to that both jobs selecting the intersection set.

To further improve locality, we need to improve the probability of both two jobs selecting the intersection, which can be achieved according to the conditional probability, as shown in Figure 2b. The independent events in the selecting procedure can be transformed into the dependent ones, making the probability of the second job conditional. That is, $J_2$ makes decisions conditioned on the decision of $J_1$.

To maximize the sampling locality, we formulate the conditional probabilities of dependent selecting in case of $|D_1| < |D_2|$. For $J_1$, the dependent selecting procedure is the same as that in ISA: selecting $I$ with the probability of $\frac{|I|}{|D_1|}$, and $D_{d1}$ with the probability of $\frac{|D_{d1}|}{|D_1|}$. The selecting procedure of $J_2$ makes the decision according to $J_1$. If $J_1$ selects $D_{d1}$, then $J_2$ must select another difference set $D_{d2}$. If $J_1$ select $I$, then $J_2$ selects $I$ with the probability of $\frac{|D_1|}{|D_2|}$ and $D_{d2}$ with probability $\frac{|D_2|-|D_1|}{|D_2|}$. Figure 2b shows the above selecting procedure. For the case of $|D_2| < |D_1|$, we just need to exchange the order of $J_1$ and $J_2$ to apply the above rules.

According to above conditional probabilities, the probability of $p(e_j^1 = e_i^2)$ is $\frac{|I|}{\max(|D_2|,|D_1|)}$, which is also the theoretical upper bound as discussed in Theorem 1. The proof of this theorem can be found in Proof 1.

**Theorem 1** *Assume there are 2 jobs $J_1$, $J_2$ sampling uniformly at random from two datasets $D_1$, $D_2$ respectively, and the elements they sampled are $e_j^1$, $e_i^2$. The probability of these two elements being equal cannot be larger than $\frac{|I|}{\max(|D_2|,|D_1|)}$ where $I = D_1 \cap D_2$.*

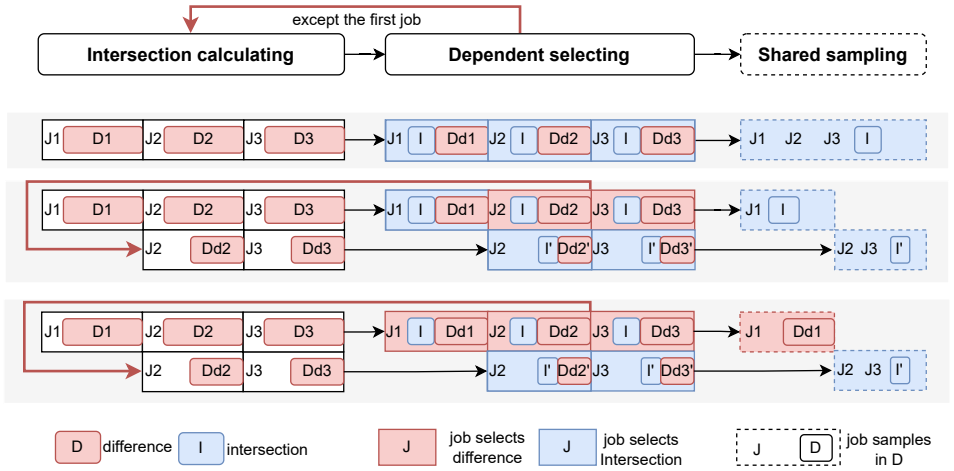



Figure 3: An illustration for the $n$-job case of DSA. We assume three jobs $\{J_1, J_2, J_3\}$ with datasets $\{D_1, D_2, D_3\}$, and there are three possible sampling cases: 1) all jobs select intersection, 2) some jobs select intersection and some select difference, and 3) all jobs select difference. Note that if there is one job selecting difference, all following jobs must select difference.

**N-job Case.** DSA of $n$-jobs is similar to two-jobs at the beginning. The difference happens when some jobs select difference sets. We need to go through the above procedure recursively for these jobs rather than sampling in the current difference set directly, as shown in Figure 3. The reason for recursive execution is that some data may only be shared by part of the jobs. In the extreme case, if all datasets are mutually disjoint, then the intersection in the first recursion is empty and the DSA is useless. Therefore, we need to further check whether there are shared data for part of the jobs.

The conditional probability for n-jobs that are sorted by the cardinalities of their datasets is formulated as follows. 1) The first job $J_1$ selects the intersection with the probability of $\frac{|I|}{|D_1|}$ while selecting the difference with the probability of $\frac{|D_{d1}|}{|D_1|}$. 2) If the job $J_{i-1}$ selects the difference set, then job $J_i$ must select the difference set. If job $J_{i-1}$ selects the intersection set, then job $J_i$ selects the intersection with the probability of $\frac{|D_{i-1}|}{|D_i|}$ while selecting the difference set with the probability of $\frac{|D_i|-|D_{i-1}|}{|D_i|}$. For the above algorithm, we can observe that if one job selects the difference, then all subsequent jobs will choose the difference, which is an important feature for the data structure in Section 4.

The pseudo code of DSA are shown in Appendix B for completeness. Here, we provide the theoretical result as follows, whose proofs can be found in Appendix D.2.

**Theorem 2** *Assume there are multiples jobs with the $i$-th job $J_i$ trained on dataset $D_i$. The probability of any element $e$ being picked for $J_i$ via DSA is $p(e) = \frac{1}{|D_i|}$.*

### 3.2 RefCnt Cache Policy

If we allow jobs to vary greatly in speed, then some data may be evicted before they are consumed by all jobs. Therefore, we need a cache policy that can take out the data that is least likely to be used (i.e., the data that least jobs will request) in the near future. Classical cache policies (e.g., LRU and LFU) are not suitable for deep learning scenarios, where each training job reads each data only once in an epoch. So the total number of request to each data equals to the number of jobs using the data for training. To this end, if we could record the count of data request, the number of request to this data in near future could be derived. The fewer jobs that access the data, the higher the eviction priority of that data.

Based on the above understanding, we design the caching policy *RefCnt*. We maintain a reference count for each data element. If a job will request the element in the future, we increase its reference

count; if a job reads the data, we decrease its reference count. When the cache is full, the data with the lowest reference count will be evicted.

# 4 Dependent Sampling Tree

The time complexity of DSA is $O(nm)$, where $n$ is the number of jobs and $m$ is the maximum cardinality of the datasets. The worst case happens if all jobs select the difference set in each recursion, which means we need $n$ recursions for $n$ jobs. Each recursion has the time complexity of $O(m)$.[2] To reduce the time complexity of sampling, we can maintain the intersections for DSA to avoid the intersection calculation in each recursion, which can make each recursion cost constant time. In the worst case of DSA, the complexity of DSA is reduced from $O(nm)$ to $O(n)$.

We propose the dependent sampling tree for organizing the intersections. Although the number of intersections is $2^n - 1$ for $n$ sets, only $n - 1$ intersections are valuable. In DSA, we only need to calculate the intersection of datasets for those jobs that select the difference, as shown in Figure 3. When the jobs are sorted by the cardinality of their datasets, the job selecting the difference will make all following jobs select the difference sets. Therefore, the possible combinations for jobs that select the difference must be the suffixes of the sorted jobs array. For example, if there are three sorted jobs $[J_1, J_2, J_3]$, the groups of jobs choosing the difference set can only be one of the following three sets, including $[J_1, J_2, J_3]$, $[J_2, J_3]$, and $[J_3]$. Thus, the sets to be calculated are $\{D_1 \cap D_2 \cap D_3, D_2 \cap D_3\}$.

## 4.1 Tree Definition

For $n$ datasets $\{D_1, D_2, ..., D_n\}$ in ascending order w.r.t. $|D_1| < |D_2| < ... < |D_n|$, we need to calculate $n - 1$ intersections $\{D_1 \cap ... \cap D_n, D_2 \cap ... \cap D_n, ..., D_{n-1} \cap D_n\}$. We build a dependent sampling tree, where each intersection is an internal vertex, and each difference set is a leaf. Meanwhile, all vertices in a path from the root to a leaf represent a complete dataset. The definition of dependent sampling tree is as follows

**Definition 1** *Dependent sampling tree is a binary tree that stores a collection of datasets $\{D_1, D_2, ..., D_n\}$, with $|D_1| < |D_2| < ... < |D_n|$. In the tree, the root is the intersection of all datasets $I = D_1 \cap D_2 ... \cap D_n$. The right child is the difference of the smallest dataset $D_1 \setminus I$. The left child is a new dependent sampling tree of $\{D_2 \setminus I, D_3 \setminus I, ..., D_n \setminus I\}$. When there is only one dataset, the tree has one vertex that contains the dataset.*

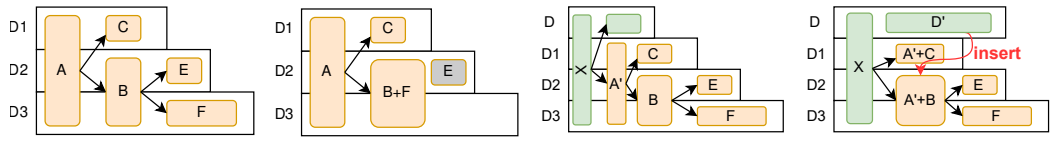
If there are $n$ datasets, the height of the dependent sampling tree is $n - 1$. Figure 4a shows an example with three sets $\{D_1, D_2, D_3\}$ with $|D_1| < |D_2| < |D_3|$. In this tree, vertex A contains $D_A = D_1 \cap D_2 \cap D_3$, and vertex B contains $D_B = (D_2 \setminus D_A) \cap (D_3 \setminus D_A)$. For the leaf vertices $C$, $E$, and $F$, they maintain the other differences.

## 4.2 Tree Operations

When executing the DSA algorithm, dependent sampling tree should maintain the datasets into intersection and difference sets dynamically. Furthermore, dependent sampling tree should also support the situations when a job finishes its execution or a new job starts. In this part, we discuss the operations provided by dependent sampling tree, which are *sampling*, *deletion*, and *insertion*.

**Sampling.** The intersections of the flow in Figure 3 can be represented by one or multiple vertexes of the rightest path in the dependent sampling tree. Thus, the process of DSA is to traverse the path from the root to the most-right leaf in the tree. When traversing to an internal vertex, the jobs will

[2] 1) The intersection calculation needs to traverse the dataset so its complexity is linear to the cardinality of the dataset. 2) In the worst case, only the first job needs to make the decision, and others follow it. Therefore, dependent selecting needs constant time. 3) Sampling only needs to pick a random number between 0 and the size of the dataset, which needs constant time.




(a) Dependent sampling tree of three datasets $\{D_1, D_2, D_3\}$ (b) Delete a dataset $D_2$ from the dependent sampling tree (c) Insert a smaller dataset $D$ ($|D| \leq |D_1|$) in dependent sampling tree (d) Insert a larger dataset $D$ ($|D| > |D_1|$) in dependent sampling tree

Figure 4: Illustration for the proposed dependent sampling tree.

decide whether to select this vertex according the conditional probability. For those jobs that do not select this vertex, they will make decisions in the right child tree.

Sometimes the intersection can be composed of multiple vertexes. For example, if only the job of $D_3$ selects difference at $A$ in Figure 4a, then it can sample both in $B$ and $F$. In these cases, the algorithm needs to first select a vertex that belongs to the intersection and then sample.

**Dataset Deletion.** When a job finishes training or is killed, we should delete its dataset from the tree. The *deletion* involves two steps: 1) deleting the corresponding leaf vertex, and 2) merging its parent vertex with its sibling vertex. For example, if we want to delete $D_2$ in Figure 4a, we need to delete the leaf vertex $E$ and merge vertex $B$ with $F$, as shown in Figure 4b. The time complexity is $O(1)$ since deleting and merging both cost constant time.

**Dataset Insertion.** The *insertion* operation depends on the cardinality of the new dataset $D$. When $D$ is smaller than the smallest set in the sampling tree, the procedure involves three steps: 1) calculating the intersection of it and the root of the tree, 2) setting the new intersection as the new root of the dependent sampling tree, and 3) setting the difference set of the old root as the right child and the difference of $D$ as the left child for the new root. For example, Figure 4c shows the new tree after inserting a smaller dataset $D$. The new root $X$ is the intersection of $D$ and the set in vertex $A$. The old tree becomes the right sub-tree, and the difference of the dataset $D$ becomes the left leaf.

When the new dataset $D$ is larger than the smallest dataset in the sampling tree, we need to insert it recursively: 1) calculating the intersection of $D$ and root $A$ and set it as new root $X$, 2) adding the difference of $A$ to its children, 3) deleting the old root $A$ and adding their children to the new root, 4) recursively executing step 1 on $X$'s right child with the difference set of $D$ until the difference set is the smallest in the current sub-tree. An example of this procedure is shown in Figure 4d. In the worst case, the time complexity of *insertion* is $O(n|D|)$ if the new dataset $D$ is the largest [3].

## 5 Evaluation

In this section, we evaluate JOADER on ImageNet with the family of ResNet models. We denote the default dataloader strategy in PyTorch as the 'Baseline' method, and further compare JOADER with the state-of-the-art method CoorDL [19]. Due to the space limit, we mainly show two experiments in the paper and more experimental results can be found in the appendix. First, we profile the server when there are multiple training jobs to show the bottlenecks and compare JOADER with CoorDL and Baseline in synchronous case. Second, we evaluate JOADER in asynchronous cases: 1) multiple models trained at different speeds, 2) multiple jobs arrived at different moments, and 3) multiple datasets only partially overlapped. For the results in the appendix, we summarize them as follows: 1) ablation studies show that both DSA and RefCnt improve the training efficiency , 2) the I/O speed is not the bottleneck for data loading , and 3) we train ResNet18 with PyTorch and JOADER separately to show JOADER does not affect the accuracy.

The evaluated models are the basic models with their default settings in torchvision [1], and trained on top of the PyTorch 1.6.0 DL framework. The experiments were conducted on a GPU server with

[3] In practice, the dataset is a bitmap set of data indices in the dataset, e.g., a unique integer id that identifies data. Therefore, the insert operation is very fast, which only takes a little time compared with training time.

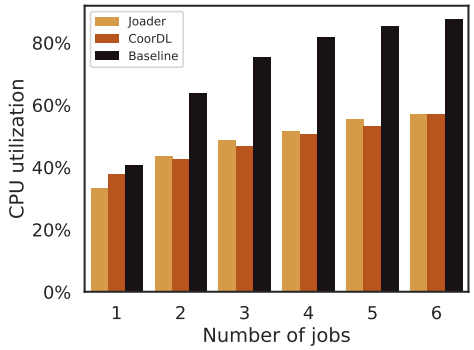



Figure 5: The CPU utilization when training multiple ResNet18 models simultaneously. CPU quickly becomes the bottleneck for the Baseline dataloader of PyTorch, while both CoorDL and JOADER can greatly reduce CPU utilization.

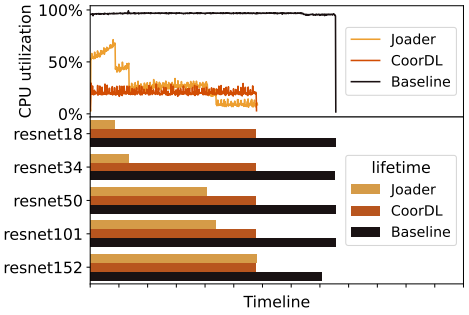
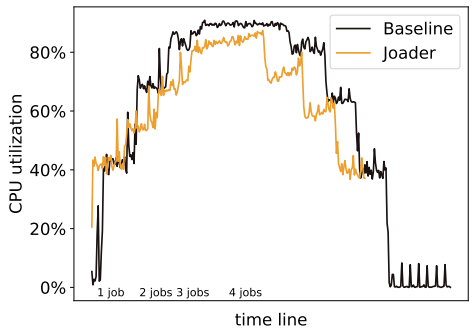
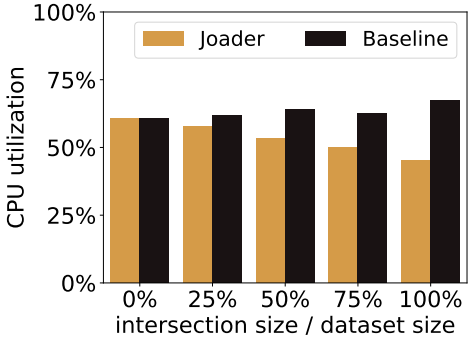



Figure 6: The CPU utilization and lifetime when training 5 models with different speeds. JOADER can reduce the CPU utilization and keep each job's own speed. In contrast, fast job must wait for the slow job for CoorDL (e.g., ResNet18 costs the same time with ResNet152).




(a) The CPU utilization and lifetime when four jobs start at different moments. JOADER is faster and causes less CPU load because it can reuse the data preparation work from previous jobs.




(b) The CPU utilization when datasets are partially overlapped. JOADER can reduce 10% CPU utilization when half of datasets are in the intersection set for 2 jobs.

Figure 7: Training jobs arrive at different moments and have different datasets

two Intel Xeon Gold 5118 CPUs @ 2.30GHz (24 physical cores and 48 threads), 500GB RAM, and 6 TITAN RTX GPUs. The server ran Ubuntu 18.04 with GNU/Linux kernel 4.15.0. The disk is Symbios Logic MegaRAID SAS-3 3316 of 1GB/s read speed.

### 5.1 Performance in Synchronous Cases

In this experiment, we start training multiple ResNet18 models with different hyper-parameters. **We try to make fair comparison by forcing the jobs training on the GPUs of the same version to get the theoretically same training speeds. All jobs train the same model on the same dataset and start at the same time.**

We start from training 1 model to training 6 models on ImageNet, and profile the server for different dataloaders. The experiments show that the CPU utilization tends to be the bottleneck, as shown in Figure 5.

The CoorDL and JOADER both can reduce the CPU utilization greatly by sharing the data preparation. Additionally, it shows that JOADER can achieve close performance to CoorDL when all training jobs start at the same time and have a similar speed. Note that CoorDL is designed for such synchronous cases while JOADER can also handle asynchronous cases as we will later show. The time cost of each

epoch is shown in Table 1. With 6 training jobs, JOADER saves $44.8\%$ training time with over 40% less CPU utilization, compared the default Baseline in PyTorch.

Table 1: The time cost of different dataloaders when training multiple ResNet18 models in an epoch.

| Number of jobs | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| CoorDL (min) | 32.82 | 31.73 | 32.73 | 34.76 | 35.28 | 37.98 |
| JOADER (min) | 32.70 | 32.49 | 33.72 | 36.32 | 37.44 | 37.55 |
| Baseline (min) | 31.74 | 35.01 | 40.04 | 52.11 | 57.67 | 67.85 |

### 5.2 Performance in Asynchronous Cases

Next, we consider three asynchronous cases when multiple models differ in training speeds, arriving moments, and datasets.

**Multiple models with different training speeds.** We first simultaneously train five models (ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152) with different training speeds, and show the lifetime and CPU utilization results in Figure 6. With the Baseline dataloader provided by PyTorch, each job preprocesses the data individually so that the CPU load is extremely heavy. CoorDL enforces multiple jobs requesting the data in lockstep to share the data preparation work, and it achieves less CPU load and higher training speed than Baseline. However, the faster jobs (smaller models) must wait for the slower jobs, making CPU idle during the training period. JOADER can both reduce the CPU load by sharing the preprocessing work and keep these jobs' own training speeds. For example, ResNet18 runs $4\times$ faster using JOADER compared with CoorDL. Furthermore, this experiment suggests that our system has less profitable for larger models because GPU tends to be the bottleneck for those models.

**Multiple models arriving at different moments.** Next, we consider the situation that multiple DNN training jobs start at different moments. Since CoorDL must wait for all the jobs to be ready for training, we only compare JOADER with Baseline in this experiment. Figure 7a shows the CPU utilization when four ResNet18 training jobs arrive at different moments. JOADER takes up less CPU load than the Baseline. This is due to the fact Baseline repeats the data preparation work for new jobs, whereas JOADER reuses the prepared data from previous jobs. As a result, JOADER saves 17.1% training time and 8.7% CPU utilization.

**Multiple models with partially overlapping datasets.** Finally, we show the CPU utilization when the two datasets only partially overlap in Figure 7b. The datasets are re-sampled from ImageNet. We use the ratio between intersection size and dataset size to measure the overlapping degree. The larger the intersection, the more data to share. We still compare with the Baseline method only since CoorDL requires all jobs having the same access path over datasets. From the figure, we can observe that the CPU utilization of the same dataset is only 60% of that of different datasets. Additionally, the CPU utilization always decrease as the intersection size increases, showing that JOADER is valid with some overlapping elements. The reason is that the DSA algorithm can get the same sampling results with the max probability and the cache policy evicts the useless data with higher priority.

## 6 Conclusion

Accelerating DNN training is a fundamental challenge in deep learning. This paper presents an approach to boosting training efficiency by caching data prep work and reusing them to parallel training jobs. To prevent cache thrashing, we propose DSA to improve sampling locality with ensured randomness. We also propose a novel tree-based data structure to efficiently implement the proposed algorithm. Furthermore, a domain-specifical cache policy is proposed for evicting data that is least used in the near future. At last, we evaluate the proposed approach and the results show significant improvement in asynchronous cases while achieving close performance in synchronous cases to CoorDL which is specially designed for such cases.

# References


[1] The package consists of popular datasets, model architectures, and common image transformations for computer vision. `https://pytorch.org/vision/stable/index.html`, 2021.

[2] Rust program language. `https://github.com/rust-lang/rust`, 2021.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. {FlashNeuron}:{SSD-Enabled} {Large-Batch} training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 387–401, 2021.

[5] Mohammad Bavarian, Badih Ghazi, Elad Haramaty, Pritish Kamath, Ronald L Rivest, and Madhu Sudan. Optimality of correlated sampling strategies. *Theory of Computing*, 16(1), 2020.

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[9] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 676–691, 2021.

[10] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *SysML 2019*, 2019.

[11] Guoxi Xu Jiayu Wu, Qixiang Zhang. Tiny imagenet challenge.

[12] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[13] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeong-In Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. *Advances in Neural Information Processing Systems*, 34, 2021.

[14] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 283–296, 2020.

[15] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[16] 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.

[17] Zhongyi Lin, Evangelos Georganas, and John D Owens. Towards flexible and compiler-friendly layer fusion for cnns on multicore cpus. In *European Conference on Parallel Processing*, pages 232–248. Springer, 2021.

[18] Adam C Mater and Michelle L Coote. Deep learning in chemistry. *Journal of chemical information and modeling*, 59(6):2545–2559, 2019.

[19] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proceedings of the VLDB Endowment*, 14(5):771–784, 2021.

[20] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment*, 13(12):2159–2173, 2020.

[21] Thao Truong Nguyen, François Trahay, Jens Domke, Aleksandr Drozd, Emil Vatai, Jianwei Liao, Mohamed Wahib, and Balazs Gerofi. Why globally re-shuffle? revisiting data shuffling in large scale deep learning. In *IEEE International Parallel & Distributed Processing Symposium*, 2022.

[22] 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.

[23] Markus Reichstein, Gustau Camps-Valls, Bjorn Stevens, Martin Jung, Joachim Denzler, Nuno Carvalhais, et al. Deep learning and process understanding for data-driven earth system science. *Nature*, 566(7743):195–204, 2019.

[24] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[25] Haotian Tang, Zhijian Liu, Shengyu Zhao, Yujun Lin, Ji Lin, Hanrui Wang, and Song Han. Searching efficient 3d architectures with sparse point-voxel convolution. In *European Conference on Computer Vision*, pages 685–702. Springer, 2020.

[26] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531, 2021.

[27] Yun-Fu Wu. Correlated sampling techniques used in monte carlo simulation for risk assessment. *International Journal of Pressure Vessels and Piping*, 85(9):662–669, 2008. Special Issue: Advances in Structural Integrity of Nuclear Components in Asian Power Plants.

[28] Lenka Zdeborová. Understanding deep learning is also a job for physicists. *Nature Physics*, 16(6):602–604, 2020.

[29] Zhenwei Zhang, Qiang Qi, Ruitao Shang, Li Chen, and Fei Xu. Prophet: Speeding up distributed dnn training with predictable communication scheduling. In *50th International Conference on Parallel Processing*, pages 1–11, 2021.

# Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See `https://anonymous.4open.science/r/Joader-6F4D`
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

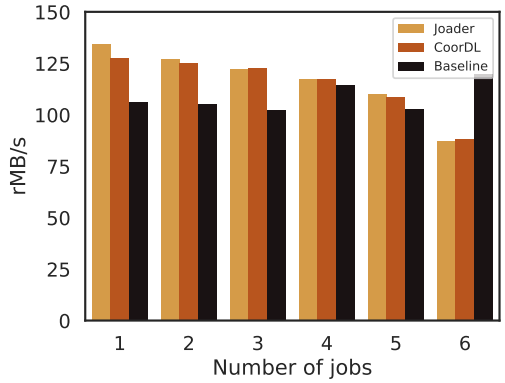



Figure 8: When training multiple ResNet18 models simultaneously, the I/O speed is shown above.

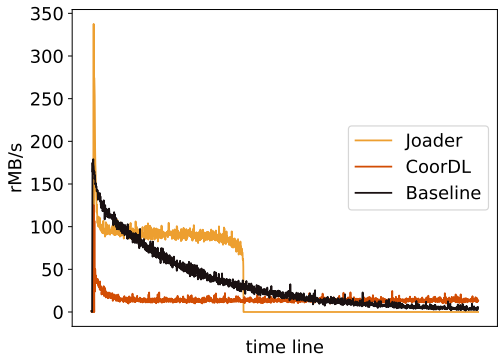



Figure 9: When training 5 models with different speeds at the same time, the I/O speeds are shown above.

# A Evaluation

In this section, we first present the I/O speed in the synchronous cases and asynchronous cases to show JOADER can reduce the redundant I/O work. Then, we evaluate the algorithm DSA and RefCnt separately for the ablation test. Finally, we show the loss and accuracy trace of ResNet18 in 40 epochs to demonstrate JOADER does not affect convergence speed. We denote the default dataloader strategy in PyTorch as the 'Baseline' method, and further compare JOADER with the state-of-the-art method CoorDL. **At last, we show the cost of the operation in DSA**.

## A.1 I/O speed

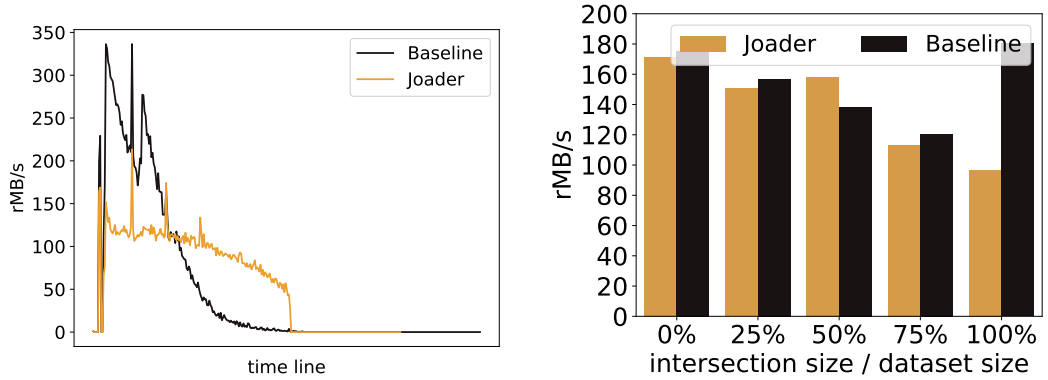
In this part, we show the I/O speed in the synchronous and asynchronous cases. Note that I/O is not the bottleneck in our server (Maximum I/O speed of the server is about 1GB/s). Figure 8 shows the I/O from training one job to training six jobs. The I/O speed of Baseline tends to increase while that of CoorDL and JOADER tends to decrease, because CoorDL and JOADER can reuse the data in memory. When we train 5 jobs (ResNet18, ResNet34, ResNet50, ResNet101, ResNet152) with different speed, the I/O speed is shown in Figure 9. Although the I/O speed of CoorDL is far less than JOADER, the fast job must wait for a slow job causing inefficiency. JOADER can make jobs run at their own speeds. Figure 10a show the I/O speed for four jobs that start at different moments. Due to the redundant I/O work, the baseline has a more considerable I/O speed. For JOADER, the more data is shared, the smaller the I/O speed, as shown in Figure 10b.

## A.2 Algorithm evaluation

In this section, we first compare ISA, the default sampling algorithm in PyTorch, with DSA to show the performance in two cases: 1) datasets overlap partially, and 2) datasets vary in size. Then we further compare the RefCnt with the generic cache policy in the above cases. We evaluate these algorithms in different cache sizes and the evaluation metric is the count of *cache misses*, indicating the number of elements not read from the cache. For $n$ datasets $D_1, D_2, ..., D_n$, the maximum count of cache misses is $|D_1| + |D_2| + ... + |D_n|$ that all the data is read from the storage, while the minimum is $|D_1 \cup D_2 \cup ... \cup D_n|$ that all the repeated data is hit in cache.

We construct the following cases for evaluation[4] in Table 2:

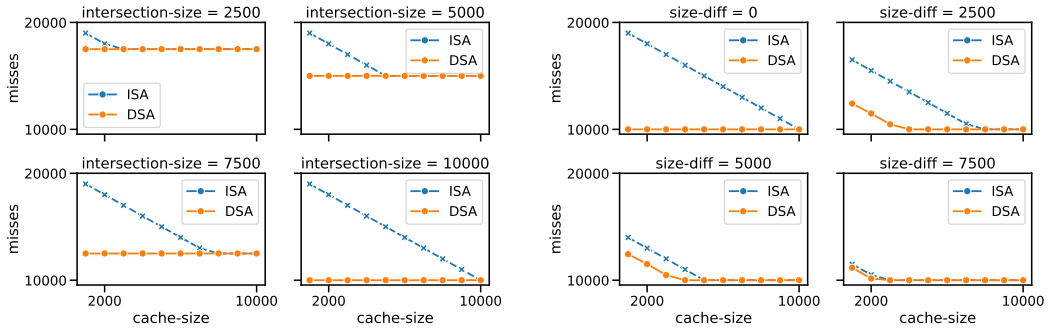
[4] 1) $D = [0, 1e4]$ means dataset $D$ constains all number from 0 to 10000. 2) $D = sample([0, 13333], 10000)$ means sample a subset $D$ with 10000 of size from $[0, 13333]$ uniformly at random




(a) When four jobs start at different moments, the I/O speeds are shown above.

(b) I/O speed are shown above when intersection sizes are different.

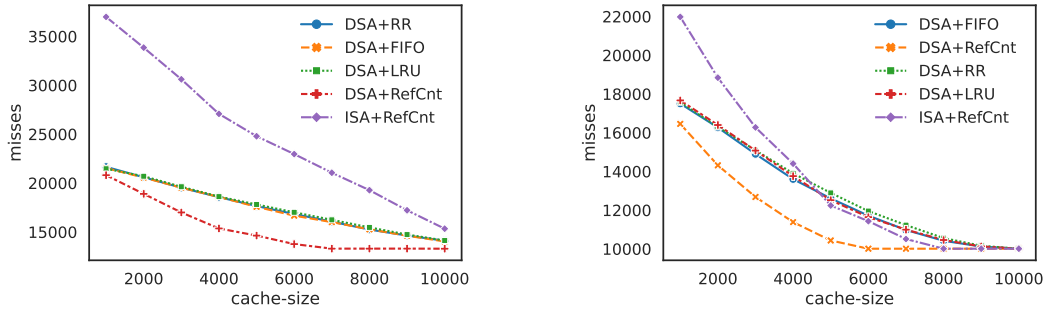
Figure 10: Training jobs arrive at different time and have different datasets




(a) Misses count of DSA and ISA for different intersections with a cache of size from 1 to 10000. DSA can always get the minimum misses.

(b) Misses count of DSA and ISA for different dataset sizes with a cache of size from 1 to 10000. DSA can reduce lots of misses.

Figure 11: Misses count of DSA and ISA in various scenes for 2 jobs




(a) Misses count for 4 jobs in an utterly random scene. RefCnt can reduce 10% misses for the same cache size.

(b) Misses count for 4 jobs with datasets of different sizes. RefCnt can reduce 10% misses for the same cache size.

Figure 12: The number of misses with different cache policies applied with dependent sampling algorithm.

Table 2: Configuration of the number of jobs and datasets

| Algorithm | Case | Num | way of construct dataset |
|---|---|---|---|
| Sampling Algorithm | overlapping partially | 2 | $D_1 = [0, 1e4]; D_2 = [0, 1e4]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [7500, 17500]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [5000, 15000]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [2500, 12500]$ |
| | | 4 | $D_1, D_2, D_3, D_4 = sample([0, 13333], 10000)$ |
| | varying in size | 2 | $D_1 = [0, 1e4]; D_2 = [0, 1e4]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [0, 7500]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [0, 5000]$ |
| | | 2 | $D_1 = [0, 1e4]; D_2 = [0, 2500]$ |
| | | 4 | $D_1 = [0, 1e4]; D_2 = [0, 7500], D_3 = [0, 5000], D_4 = [0, 2500]$ |
| Cache Policy | overlapping partially | 4 | $D_1, D_2, D_3, D_4 = sample([0, 13333], 10000)$ |
| | varying in size | 4 | $D_1 = [0, 1e4]; D_2 = [0, 7500], D_3 = [0, 5000], D_4 = [0, 2500]$ |

**Dependent sampling algorithm.** In this part, we show DSA can significantly reduce the cache misses in various of scenes. To simulate various scenes, we generate multiple datasets with different sizes and different intersections. We use DSA to access them in shuffling order from the storage with the cache of different sizes (e.g., from one-slot to holding all datasets) to evaluate the number of misses, compared with the experimental results with ISA (shuffling independently).

We start the the evaluation in two jobs training on $dataset_1$ and $dataset_2$, respectively. We set up eight different configurations for the two datasets, which can be divided into two types: the different sizes of intersections and different sizes of the datasets, as shown in Table 2.

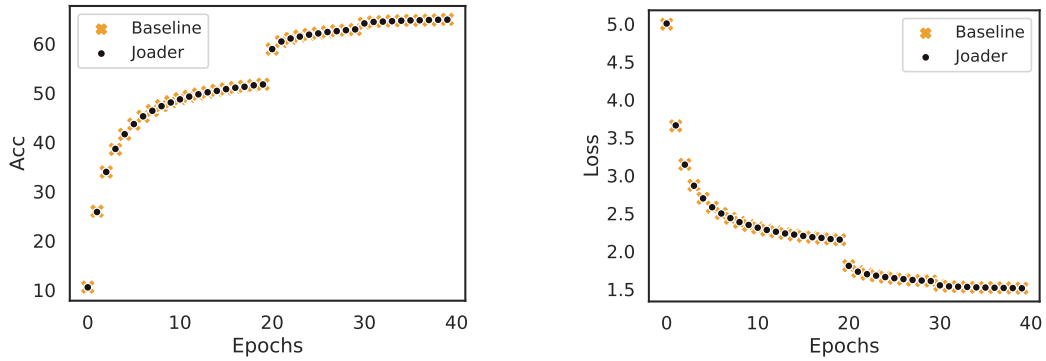
For different intersection sizes, Figure 11a plots the relations between cache size and the count of cache misses. For ISA, the misses count is 2x higher than the size of dataset in the case of one-slot cache, shows that all elements are missed and it decreases linearly as the cache size increase. Unless the cache can hold all the data of intersection, the misses count could be minimized. However, DSA can always get the minimum cache misses regardless of the cache size. The reason is that in the selecting procedure, all jobs must select the same subset when the datasets' sizes are equal, due to the conditional probability of 1. Thus, their sampling results would be the same, meaning that all jobs will access the same element simultaneously when the datasets with the same size.

Figure 11b plots the evaluation results on for the datasets with different sizes. For one-slot cache, DSA can reduce 50% cache misses compared with ISA when the dataset sizes differ by 25%. The cache held 30% dataset could get the best performance for DSA while ISA needs to cache 75% dataset to reach optimal. DSA needs to cache 25% data because the size difference brings some uncertainty in the selecting procedure.

Then, we evaluate the DSA algorithm on four DNN training jobs to show the outstanding performance of the DSA algorithm on multiple datasets. Multiple datasets bring more complexity to the intersection as there can be $2^n - 1$ intersection sets for n sets. To simulate an utterly random scene, we random sample 10,000 elements from 13,333 elements four times, making four different datasets for evaluation. Compared ISA in the random scene, DSA can reduce 50% misses with a one-slot cache (20000 misses out of 40000 misses), as shown in Figure 13a.

DSA also provides good performance for multiple jobs training on datasets with different sizes. Compared to ISA in Figure 13b, one-slot cache can reduce 30% cache misses (16000 misses out of 22000 misses) with DSA when four jobs are training on four datasets, with a 25% different size.

**RefCnt cache policy.** In this part, we compare cache policy RefCnt with the generic cache policies LRU, FIFO, and random replacement (RR) to show 10% misses reduction with the same cache size. We evaluate these policies in above two situations of four jobs: utterly random scene and different dataset sizes. Figure 12 shows the results, which can be summarized as two improvements: 1) RefCnt




(a) Accuracy trace

(b) Loss trace

Figure 13: Trace of training ResNet18 in 40 epochs

cache policy can reduce 10% misses in the same cache size compared with general cache policies, and 2) RefCnt cache policy can get the best performance with only caching the 60% dataset while the generic cache policies require holding the whole dataset.

### A.3 Correctness experiments

To demonstrate JOADER does not affect convergence speed w.r.t. accuracy and loss, we train ResNet18 on ImageNet-1K with JOADER and PyTorch[5]. Figure 13 shows that the loss descent trajectory and accuracy trajectory are almost the same.

### A.4 Cost of operation in sampling tree

**We have conducted an additional experiment by randomly inserting 128 datasets into our system and then evaluating the time cost of dataset operations. The size of each dataset is between 1,000,000 and 2,000,000 elements (note that ImageNet contains 1,400,000 images). The average cost of inserting each dataset is 0.57 seconds. And the time cost of the operations remains nearly constant and does not change as the number of data increases.**

**Deletion and sampling are also very efficient in our algorithm. Specifically, sampling takes an average of 0.000054 seconds for each element, and deletion takes an average of 0.00000105 seconds. We will add the corresponding description and experiment in our rebuttal revision.**

## B Dependent Sampling Algorithm

In this section, we show the pseudo-code of the DSA, which involves two functions: selecting and sampling. The selecting algorithm is shown in Algorithm 1, and the sampling algorithm is shown in Algorithm 2.

## C Sampling in Dependent Sampling Tree

The *sampling* operation involves two steps: 1) *Deciding:* choose a vertex for each job, and 2) *Sampling:* select an element in the corresponding vertex for each job randomly.

In *deciding*, we start the procedure from the root of the tree. In the beginning, all jobs are marked as *undecided*. In each vertex, all undecided jobs need to decide whether to select the current vertex, i.e., the intersection $I$ in the DSA algorithm. If the job chooses the current vertex, we will remove them from the undecided job set. Then, the remaining undecided jobs are put down to the child and make decisions recursively until the undecided job set is empty, like Algorithm 1.

[5]To activate DSA, we run 2 jobs at different moments

**Algorithm 1:** Selecting Procedure

**Data:** $n$ jobs $J_1, ..., J_n$ and $n$ datasets $D_1, ..., D_n$
**Result:** The subset that each job selects

$JOB \leftarrow [J_1, ..., J_n]$; // Array of all jobs
$DS \leftarrow [D_1, ..., D_n]$; // Array of all datasets sorted by their cardinality
$S \leftarrow \emptyset$; // Set of data that are not selected
**while** $JOB \neq \emptyset$ **do**
  $I \leftarrow \bigcap D_i - S, \forall D_i \in DS$ ;
  $c \leftarrow |I|$;
  **for** $D$,$J$ ***in*** $DS$, $JOB$ **do**
    // $\frac{c}{|D|-|S|}$ is the conditional probability $\frac{|D_{i-1}|}{|D_i|}$
    **if** ***rand***$(0, 1) \leq \frac{c}{|D|-|S|}$ **then**
      $J$ selects $I$;
      $DS \leftarrow DS - D$;
      $JOB \leftarrow JOB - J$;
      $c \leftarrow |D|$;
    **else**
      break;
  **if** *There is no job selecting* $I$ **then**
    $JOB[0]$ selects $D[0] - I$;
    $DS \leftarrow DS - D[0]$;
    $JOB \leftarrow JOB - JOB[0]$;
  $S \leftarrow S \cup I$ ;

**Algorithm 2:** Sampling Procedure

**Data:** $n$ jobs $J_1, ..., J_n$ and $k$ subsets $S_1, ..., S_k$
**Result:** The sampling results of each job

$JOB \leftarrow [\{J_i, ..., J_j\}, ...]$; // The job set where job selects the correspond subset
$SUBSET \leftarrow [S_1, ..., S_k]$; // The subset that the correspond job set selects
**for** $i$ ***in*** $0..k$ **do**
  $subset \leftarrow SUBSET[i]$;
  $job\_set \leftarrow JOB[i]$;
  Sample $e$ uniformly at random in $subset$;
  **for** $job$ ***in*** $job\_set$ **do**
    $job$ picks $e$;

However, in *deciding*, there may be an intersection that contains multiple vertices of the sampling tree. As shown in Figure 4c, if job $J_2$ with $D_2$ and job $J_3$ with $D_3$ do not choose the root $X$ in the first recursion, then they should make decision in the intersection $(D_2 - D_X) \cap (D_3 - D_X)$, which contains the datasets in vertices $A$ and $B$. In this case, we should combine these vertices into a big set first and start the deciding procedure once. If someone chooses the big set, it should choose a vertex from the big set randomly again.

After the deciding procedure, each job is assigned to a vertex randomly. Then, we need to select an element in the corresponding vertex randomly. Based on the *united sampling* discussed in Section 3, those jobs that are assigned to the same vertex should share the same sampling result.

Each data item in the dataset should be read exactly once for a job in an epoch. Intuitively, the data item should be deleted from the vertex to avoid being read by the job repeatedly. This process raises the issue that other jobs may need the deleted element that some have read. For example, suppose in the Figure 4a, job $J_1$ fetches the element $e_1$ from vertex $A$ while job $J_2$ and job $J_3$ fetch the element

Table 3: Notation in sampling algorithm

| | |
|---|---|
| $J_i$ | The i-th job |
| $D_i$ | Dataset that $J_i$ trains upon |
| $I$ | Intersection of all datasets |
| $D_{di}$ | $D_i - I_i$ |
| $\|D\|$ | Cardinality of set D |
| $J_i^I$ | Event that $J_i$ chooses the intersection |
| $J_i^D$ | Event that $J_i$ do not choose the intersection |
| $e_j^i$ | Element $e_j$ picked by $J_i$ in one round |
| $S$ | Set of elements that are not selected |

from other vertices. Thus, for $J_1$, we should remove $e_1$ from vertex $A$, but $J_2$ and $J_3$ still need the element $e_1$.

The compensation procedure is introduced in sampling to solve this issue, which adds the deleted element to the child vertex of the job that still needs it. To decide which vertex needs compensation, we construct a job set for each vertex that contains all the jobs that will sample in this vertex. For example, the job set of vertex $A$ is $\{J_1, J_2, J_3\}$ and the set of vertex $B$ is $\{J_2, J_3\}$ in Figure 4a. And after deleting some elements, we should construct a compensation set containing all the jobs that still need the deleted elements. If the compensation set includes the job set, the deleted elements are added to the corresponding vertex.

**Partially Sampling.** When some jobs are much faster than other jobs, we only need the sampling results for these fast jobs. The sampling procedures are the same as the above. However, the tree cannot be ordered after sampling because these datasets decrease fast and can be less than the above datasets in the tree soon. Therefore, we need to reorder the tree, just like the intersection procedure.

# D Proof of Algorithm

In this section, we present the derivation and the proof for the proposed algorithm. Proposition 1 shows the first principle for randomness. Table 3 presents some notations for the following proof.

**Proposition 1** *For any job $J_i$, the probability of choosing any element from dataset $D_i$ is $\frac{1}{|D_i|}$.*

## D.1 Derivation in Two-job Case

There are 2 jobs $J_1$ and $J_2$ which training upon datasets $D_1$ and $D_2$, while the intersection is $I = D_1 \cap D_1$ and two difference sets are $D_{d1} = D_1 - I, D_{d2} = D_2 - I$. Their cardinalities are $|D_1|$, $|D_2|$, $|I|$, $|D_{d1}|$, and $|D_{d2}|$. The event that $J_1$ selects the intersection is $J_1^I$, while the event that $J_1$ selects the difference set is $J_1^D$.

Due to the randomness, the constraints that we need to maintain are

$$\begin{cases} p(J_1^I) = \dfrac{|I|}{|D_1|}, p(J_1^D) = \dfrac{|D_{d1}|}{|D_1|} \\ p(J_2^I) = \dfrac{|I|}{|D_2|}, p(J_2^D) = \dfrac{|D_{d2}|}{|D_2|}. \end{cases} \tag{1}$$

Since $J_2$ makes the decision conditioned on $J_1$, we can the formulate the equations according to the law of total probability in below

$$\begin{cases} p(J_2^I) = p(J_2^I|J_1^I) * p(J_1^I) + p(J_2^I|J_1^D) * p(J_1^D) \\ p(J_2^D) = p(J_2^D|J_1^I) * p(J_1^I) + p(J_2^D|J_1^D) * p(J_1^D). \end{cases} \tag{2}$$

After *shared sampling*, our target is to maximize the probability of $J_1$ and $J_2$ both choose the intersection, which is positive correlated with $p(J_2^I|J_1^I)$. The $p(J_2^I|J_1^I)$ is

$$\begin{aligned} p(J_2^I|J_1^I) &= \frac{p(J_2^I) - p(J_2^I|J_1^D) * p(J_1^D)}{p(J_1^I)} \\ &\leq min(1, \frac{p(J_2^I)}{p(J_1^I)}) \\ &\leq min(1, \frac{|D_1|}{|D_2|}). \end{aligned} \tag{3}$$

Therefore, we can get the conditional probability in two cases

$$p(J_2^I|J_1^I) = \begin{cases} 1, & \text{if } |D_1| > |D_2| \\ \frac{|D_1|}{|D_2|}, & \text{if } |D_1| \leq |D_2|. \end{cases} \tag{4}$$

When $|D_1| \leq |D_2|$, the other conditional probabilities of $J_2$ conditioned on $J_1$ can be derived as follows

$$\begin{cases} p(J_2^I|J_1^D) = 0 \\ p(J_2^D|J_1^D) = 1 \\ p(J_2^D|J_1^I) = 1 - \frac{|D_1|}{|D_2|}. \end{cases} \tag{5}$$

In *dependent selecting* procedure, the probability of both choosing intersection set is $p(J_1^I) * p(J_2^I|J_1^I)$. With *data partition*, we can get

$$p(e_j^1 = e_i^2) = p(J_1^I) * p(J_2^I|J_1^I) \leq \frac{|I|}{\max(|D_2|, |D_1|)}, \tag{6}$$

while $e_j^1, e_i^2$ is the sampling results of $J_1, J_2$ in one round.

We then prove the the proposed dependent algorithm gives the optimal solution for the scenario of two-jobs.

**Proof 1** *Assume that the distribution of the event exists, and the probability of $p(e_j^1 = e_i^2)$ is unknown. When $e_j^1$ and $e_i^2$ are equal, they must belong to the intersection set $D_i$. Therefore, $p(e_j^1 = e_i^2)$ is equal to $p(e_j^1 = e_i^2, e_j^1 \in D_i, e_i^2 \in D_i)$. Due to the definition of joint probability, we can get*

$$p(e_j^1 = e_i^2) \leq p(e_j^1 \in D_i), \quad \textit{and} \quad p(e_j^1 = e_i^2) \leq p(e_i^2 \in D_i).$$

*Due to the constraint of randomness, $e_j^1$ and $e_i^2$ are both sampled under the uniform distribution, so $p(e_j^1 \in D_i) = \frac{|D_i|}{|D_1|}$ and $p(e_i^2 \in D_i) = \frac{|D_i|}{|D_2|}$. Then $p(e_j^1 = e_i^2) \leq \frac{|D_i|}{|D_1|}$ and $p(e_j^1 = e_i^2) \leq \frac{|D_i|}{|D_2|}$ both hold. Thus, for any distribution of the event, it holds that*

$$p(e_j^1 = e_i^2) \leq \frac{|D_i|}{\max(|D_2|, |D_1|)}.$$

### D.2 Derivation in N-job Case

Assume there are $n$ jobs $\{J_1, ..., J_n\}$ and $n$ datasets $\{D_1, ..., D_n\}$, while the intersection $I = \bigcap_{i=1}^n D_i$. For the $k$-th job where $k > 1$, the probability of the job $J_k$ chooses the intersection is

$$p(J_k^I) = p(J_k^I|J_1^I, J_2^I, ..., J_{k-1}^I) * p(J_1^I, J_2^I, ..., J_{k-1}^I) + x, \tag{7}$$

where x is the sum of probabilities of $J_k$ chooses intersection in other conditions. With the similar process in Proof 1, we can get the result for $k$-jobs, where the maximum probability of they all choose

intersection is also $p(J_1^I ... J_k^I) \leq \frac{|I|}{max(D_1,...,D_k)}$. Then, we can get

$$\begin{aligned} p(J_k^I | J_1^I, J_2^I, ..., J_{k-1}^I) &\leq min(\frac{p(J_k^I)}{p(J_1^I, J_2^I, ..., J_{k-1}^I)}, 1) \\ &\leq min(\frac{\frac{|I|}{|D_k|}}{\frac{|I|}{max(D_1,...,D_{k-1})}}, 1) \\ &\leq min(\frac{max(D_1, ..., D_{k-1})}{|D_k|}, 1). \end{aligned} \tag{8}$$

Assume that the datasets are sorted in ascending order w.r.t. their cardinalities, then the formula can be written as for the maximum probability

$$p(J_k^I | J_1^I, J_2^I, ..., J_{k-1}^I) = \frac{|D_{k-1}|}{|D_k|}. \tag{9}$$

The probability of $n$ jobs all choose intersection $I$ is

$$p(J_1^I, J_2^I, ..., J_n^I) = \frac{|I|}{|D_1|} * \frac{|D_1|}{|D_2|} * ... * \frac{|D_{n-1}|}{|D_n|} \leq \frac{|I|}{|D_n|}, \tag{10}$$

which is the theoretical maximum according to a similar Proof 1.

### D.3 Proof of Randomness

We prove the Proposition 1 for Algorithm 1. Although multiple jobs share the sampling results in the sampling procedure, the random sampling is not changed. Therefore, to prove Proposition 1, we only need to prove the subsets are randomly selected for each job. The Lemma is stated as follow

**Lemma 1** *In each loop of selecting procedure, the intersection $I$ is selected with the probability of $\frac{|I|}{|D_i|}$ for every job $J_i$.*

In the line 8-12 of Algorithm 1, the job $J_i$ chooses the intersection only if the previous job has chosen the intersection $I$, that the probability is

$$\begin{aligned} p(J_i^I) &= p(J_i^I | J_{i-1}^I, ...) * p(J_{i-1}^I, ...) \\ &= \frac{|I|}{|D_i| - |S|}. \end{aligned} \tag{11}$$

Therefore, the Lemma 1 is satisfied only if the probability of entering this loop is $\frac{|D_i|-|S|}{|D_i|}$, that is

**Lemma 2** *For each loop, the set of elements that are not selected is $S$, and the probability of entering this loop is $\frac{|D_i|-|S|}{|D_i|}$ for job $J_i$.*

**Proof 2** *We prove the Lemma 2 by induction on loop index $k$.*

***Base case.*** *Show Lemma 2 holds for the first loop.*

*The first loop ($k = 1$) must be entered, while $S$ is an empty set and $\frac{|D_i|-|S|}{|D_i|} = 1$. Thus Lemma 2 is satisfied.*

***Induction step.*** *Show that for any $k \geq 1$, if the $k$-th recursion holds Lemma 2, then $(k+1)$-th loop also holds.*

*The probability of entering the $k$-th loop is $\frac{|D_i|-|S|}{|D_i|}$, and if job $J_i$ does not select the intersection in*

*the $k$-th loop, then it entering the $(k+1)$-th loop. The probability of $J_i$ does not select intersection is*

$$
\begin{aligned}
p(J_i^D) &= p(J_k^D) + p(J_{k+1}^D) + ... + p(J_i^D) \\
&= p(J_k^D) + p(J_{k+1}^D|J_k^I) * p(J_k^I) + ... + p(J_i^D|J_{i-1}^I) * p(J_{i-1}^I) \\
&= p(J_k^D) + (1 - p(J_{k+1}^I|J_k^I)) * p(J_k^I) + ... + (1 - p(J_i^I|J_{i-1}^I) * p(J_{i-1}^I) \\
&= p(J_k^D) + p(J_k^I) - p(J_{k+1}^I) + p(J_{k+1}^I) - ... - p(J_i^I) \\
&= 1 - p(J_i^I) \\
&= 1 - \frac{|I|}{|D_k| - |S|} * \frac{|D_k| - |S|}{|D_{k+1} - |S|} * \frac{|D_{i-1}| - |S|}{|D_i| - |S|} \\
&= \frac{|D_i| - |S| - |I|}{|D_i| - |S|}
\end{aligned}
\tag{12}
$$

*where $J_k$ is the first job in job List $JOB$, then we can get the probability of entering the next loop is $\frac{|D_i|-|S|-|I|}{|D_i|}$ while new $S$ is $S \cup I$, which also satisfies the Lemma 2.*

***Conclusion.*** *Since both the base case and the inductive step have been proved as true, the Lemma 2 holds for every loop by mathematical induction.*

# E System Implementation

In this section, we describe the implementation of JOADER, which is a data loading management system for multiple DNN training jobs. The proposed dependent sampling algorithm applied with the dependent sampling tree is the core in JOADER. In details, we describe the system overview, three main components, and the adaption for distributed DNN training.

## E.1 Overview

**API.** JOADER API allows users to do two things: 1) creating and altering datasets, and 2) registering DNN training jobs. Before training, the user needs to create the dataset with a name in JOADER first. Then, to execute the training job on the dataset, the job should register itself to JOADER with the dataset's name. If multiple jobs are training on that dataset, JOADER will attach the dataset to the jobs for sharing reading and preprocessing. Each job in an epoch will be assigned to a unique id in JOADER, and JOADER will dispatch data according to the job id.

**Architecture.** JOADER includes a frontend and a backend. The backend is implemented in Rust [2] that accesses and processes data efficiently, while the frontend is implemented in Python to provide a user-friendly API. They communicate with each other in RPC. Figure 14 shows the architecture of JOADER, which consists of three components: Sampler, Loader, and Cache.

## E.2 Sampler and Loader

The Sampler is responsible for data sampling in JOADER. An instance of the Sampler consists of a sampling tree for a dataset. The datasets are organized in tables like the relational database. Each data tuple has a unique integer id (i.e., primary key) to identify itself, consisting of multiple elements, e.g., image, label, and the bounding boxes in ImageNet.

Each job registered to the sampler needs to specify the job's name, dataset's name, names of the needed columns in the table, and a predicate for filtering. Then, the sampler collects the ids of data tuples that meet the requirements and inserts the ids into the sampling tree corresponding to the dataset. Each sampling tree should pick the element uniformly at random in the DS algorithm. These elements (id of data tuple) will be transformed into the data requests according to the columns of each job needed. If the elements and the columns are the same, these data requests will be merged into one request.

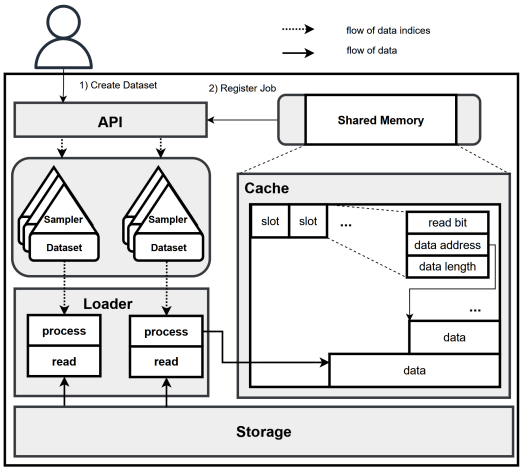



Figure 14: The architecture of JOADER.

The Loader is responsible for reading and processing data w.r.t. the data request from Sampler. In JOADER, each loader corresponds to one type of storage, e.g., POSIX file system, key-value database, or distributed storage. Loaders encapsulate different storages and provide a unified interface to Sampler to present good scalability and compatibility.

### E.3 Specific Cache Implementation

The Cache is used to store and share data between JOADER and the DNN training jobs, which is implemented by shared memory to avoid the cost of memory copying, serialization, and network. To manage data and reduce data race, the layout of cache is slotted, making multiple slots in the head of the cache, as shown in Figure 14. Each element in the cache is managed by a slot containing the start address, length, and read bit of the data. The read bit is used to determine whether the data has been consumed.

When Loader tries to load data, it needs to require slots and a contiguous block memory for storing this data. Then, the Loader dispatches the slot ids to training jobs that need the data. These jobs will access the data soon and set the read bit. When there is not enough memory, the reclamation program should be triggered. Data can be classified as consumed and unconsumed in the cache according to the read bit. Because those unconsumed data were sampled earlier than the data that Loader is trying to require memory, they will be accessed earlier than the new data and cannot be evicted.

We can only reclaim those consumed elements, which are prioritized in different levels. For example, there are two training jobs $J_1$ and $J-2$ all want to read four data $\{d_1, d_2, d_3, d_4\}$. In the first round of sampling, $d_1$ is consumed by both jobs. In the second round, job $J_1$ consumed $d_2$ while job $J_2$ consumed $d_3$. Then the memory reclamation program is triggered. Considering the constraint that each job should traverse the dataset once. In this time, data $d_1, d_2, d_3$ are in cache while data $d_1$ is no longer needed anymore while $d_2, d_3$ are still needed. Therefore, $d_1$ will be prioritized for eviction.

In JOADER, we reclaim data according to the number of reference of data, as shown in section 3.2.

### E.4

In our Joader, the time complexity of dataset operations, i.e., sampling, deletion, and insertion, are $O(1)$, $O(1)$, and $O(|D|)$, respectively, for each dataset. Moreover, the dependent sampling tree manages the **index** of each input rather than the input itself, and thus executing the operations is extremely fast. In our implementation, we use a bitmap to represent the dataset in our dependent sampling tree. The bitmap is very efficient for these operations. In practice, the operations only count up to a minor fraction of time consumption, which takes a tiny proportion of a total training epoch.

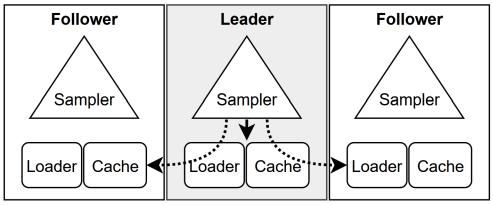



Figure 15: JOADER for distributed DNN training, where the leader JOADER is responsible for the sampling globally.

To further answer the question, we have conducted an additional experiment by randomly inserting 128 datasets into our system and then evaluating the time cost of dataset operations. The size of each dataset is between 1,000,000 and 2,000,000 elements (note that ImageNet contains 1,400,000 images). The average cost of inserting each dataset is 0.57 seconds. And the time cost of the operations remains nearly constant and does not change as the number of data increases.

Deletion and sampling are also very efficient in our algorithm. Specifically, sampling takes an average of 0.000054 seconds for each element, and deletion takes an average of 0.00000105 seconds.

### E.5 Towards Distributed Training

In model parallelism, the model is segmented into different parts that can run concurrently in different nodes. Only one node is responsible for data preparation. Therefore, JOADER need not change anything to apply to the distributed training in model parallelism.

In data parallelism, the dataset is divided into several partitions, where the number of partitions is equal to the total number of available nodes in the cluster. The model is replicated to the worker nodes. Each worker operates on its subset of the dataset to train the model locally. For each worker node, we need to set up a JOADER for sharing data preparation to multiple DNN training jobs in this node. However, the sampling work should be done globally to avoid redundant sampling.

To solve the above issue, we set multiple JOADER for multiple nodes, while only one leader JOADER can sample. The leader needs to dispatch the sampling results to the followers, as shown in Figure 15. During training, each sub-process of the distributed training job in data parallelism needs to register itself to the local JOADER in the same worker node with the same name and the assigned id. Meanwhile, the local JOADER should register a sub-sampler in leader JOADER. The local JOADER should keep fetching sampling results from the leader JOADER and load them to the local cache. By doing this, only leader JOADER is responsible for sampling and dispatching sampling results to followers. Notice that the cache is also distributed in different worker nodes, storing more data than a single cache.

The same data is sent to the same worker node when the leader dispatches sampling results. PyTorch uses the hash partition algorithm to dispatch data. For example, suppose a worker node array $nodes = [node_1, node_2, ..., node_n]$. The data with integer $id$ is sent to $nodes[id\%n]$. However, things will be more complicated when dealing with multiple training jobs. For example, two training jobs are on the worker node arrays $[node_1, node_2, ..., node_{n-1}]$ and $[node_2, node_3, ..., node_n]$, respectively. Due to the offset between the two arrays, the same data cannot be sent to the same worker node for the two jobs by one hash partition.

In JOADER, we solve it by hashing the data id twice. In the first hash, we try to dispatch the data to all the worker node in cluster. Therefore, the same data can be sent to the same node. However, some training jobs are not training on some worker nodes and the data can not be sent to these nodes for these jobs. Therefore, in the second hash, for the data dispatched wrongly, we will try to dispatch it to the nodes locally w.r.t. the configuration of each job.