Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training

Gyewon Lee, *Seoul National University and FriendliAI;* Irene Lee, *Georgia Institute of Technology;* Hyeonmin Ha, Kyunggeun Lee, and Hwarim Hyun, *Seoul National University;* Ahnjae Shin and Byung-Gon Chun, *Seoul National University and FriendliAI*

https://www.usenix.org/conference/atc21/presentation/lee

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training

Gyewon Lee[1,3] Irene Lee[2] Hyeonmin Ha[1] Kyunggeun Lee[1]
Hwarim Hyun[1] Ahnjae Shin[1,3] Byung-Gon Chun[1,3]*
Seoul National University[1] *Georgia Institute of Technology*[2] *FriendliAI*[3]

Abstract

Data augmentation is a widely adopted technique for improving the generalization of deep learning models. It provides additional diversity to the training samples by applying random transformations. Although it is useful, data augmentation often suffers from heavy CPU overhead, which can degrade the training speed. To solve this problem, we propose data refurbishing, a novel sample reuse mechanism that accelerates deep neural network training while preserving model generalization. Instead of considering data augmentation as a black-box operation, data refurbishing splits it into the partial and final augmentation. It reuses partially augmented samples to reduce CPU computation while further transforming them with the final augmentation to preserve the sample diversity obtained by data augmentation. We design and implement a new data loading system, Revamper, to realize data refurbishing. It maximizes the overlap between CPU and deep learning accelerators by keeping the CPU processing time of each training step constant. Our evaluation shows that Revamper can accelerate the training of computer vision models by 1.03×–2.04× while maintaining comparable accuracy.

1 Introduction

Deep learning (DL) is at the heart of modern AI-based applications, enabling various services such as computer vision [19, 20, 33, 34], automatic speech recognition [38], and natural language processing [17, 35]. DL models are trained by repeatedly adjusting the parameters of the models in order to minimize their loss with regard to training samples. The trained models must ensure generalization, the ability to appropriately process previously unseen input data.

Recently, *data augmentation* has been widely used in deep neural network (DNN) training to improve generalization of DL models including image classification [14, 15, 36, 39], object detection [15, 36], and automatic speech recognition [29]. By applying several transformations on training samples in a random manner [14, 15, 29], data augmentation provides additional samples to model training and thus helps improve model generalization. Because data augmentation is a stochastic process, every augmented sample is unique (Figure 1 (a)). Better model generalization from data augmentation, however, comes at the cost of expensive CPU operations. This CPU overhead often causes data augmentation to become a performance bottleneck of DNN training [4, 9, 13, 30].

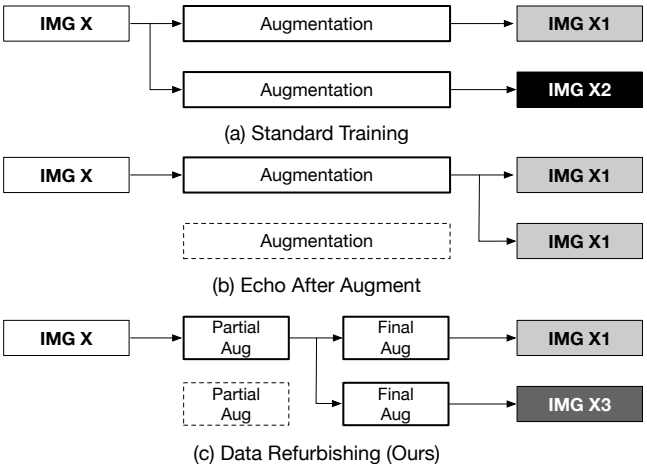

Figure 1: A high-level introduction of (a) standard training, (b) data echoing [9, 13] after augment, and (c) data refurbishing (ours). The dotted rectangles indicate computation saved by sample reuse.

To address the CPU overhead from data augmentation, recent works such as NVIDIA DALI [4] and TrainBox [30] utilize hardware accelerators such as GPUs and FPGAs for optimizing data augmentation. However, the stochastic nature of data augmentation makes it difficult to exploit accelerators that are optimized for parallel execution of homogeneous operations. Data echoing [9, 13], on the other hand, tries to reduce the amount of computation by splitting training pipelines into the upstream and downstream pipelines, and reusing previously prepared samples from the upstream pipeline in the

*Corresponding author.

downstream pipeline. However, it considers augmentation as a black-box operation and splits the DNN training pipeline to only before or after the augmentation. If the pipeline is split before data augmentation, the overhead from data augmentation remains unchanged. However, with the other option, the augmented samples are reused multiple times without further transformations as shown in Figure 1 (b). This decreases the number of unique samples generated from data augmentation—the sample diversity—to a great degree and degrades the accuracy of trained models.

To solve this problem, we propose data refurbishing, a novel sample reuse mechanism for fast DNN training data preparation. Data refurbishing splits the original data augmentation pipeline into the *partial augmentation* and *final augmentation* according to the given split policy, and reuses the intermediate results generated from the partial augmentation (Figure 1(c)). The *partially augmented samples* produced from the partial augmentation are cached, reused for a designated number of times, and renewed to preserve the diversity of augmented samples. The final augmentation applies the remaining portion of the full augmentation pipeline to the cached samples and produces fully augmented samples to be used for gradient computation.

Although reused, each partially augmented sample undergoes the final augmentation to produce a diverse set of augmented samples to be used for gradient computation. The number of unique samples, thus, remains almost the same, preserving the original sample diversity and model generalization. As such, data refurbishing is able to reduce CPU overhead from data augmentation with little generalization loss. Reducing computation overhead while maintaining enough sample diversity, this approach provides better trade-offs between training speed and model generalization than data echoing. We demonstrate these benefits both mathematically and empirically. Such characteristics make data refurbishing especially useful for exploration tasks, such as hyperparameter optimization [10, 27], which requires much DNN training with various configurations.

We design Revamper, a new caching and data loading system, to realize data refurbishing. Revamper shares similarity with systems that adopt intermediate data caching [18, 37], but it differs from such systems in that Revamper addresses new challenges that are specific to the context of DNN training. Because with data refurbishing a mixture of cached and non-cached samples is used for gradient computation, the CPU processing time may fluctuate depending on the number of cache misses in each step, which can deteriorate computation overlap between the CPU and DL accelerators. Revamper maintains a constant CPU computation overhead across epochs with the balanced eviction and within each epoch with the cache-aware shuffle, where an epoch denotes a complete pass on the entire dataset. The balanced eviction resolves the inter-epoch computation skew by evicting cached samples in a way that the number of cache misses is evenly distributed across epochs. To address the intra-epoch computation skew, the cache-aware shuffle utilizes information from the cache store to prepare training samples for each step to make the number of cache misses uniform over training steps.

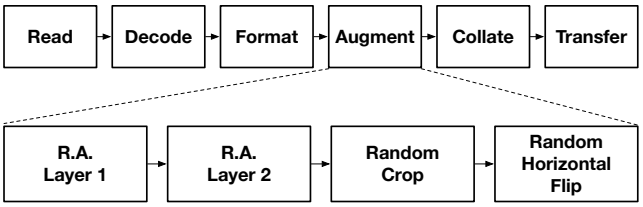

Figure 2: An illustration of a RandAugment [15] augmentation pipline in a typical data preparation pipeline. R.A. Layer stands for a RandAugment layer.

Revamper is implemented on PyTorch 1.6 [31], a widely used DL framework for DNN training. Its interface overrides the existing PyTorch dataloader so that users can utilize Revamper by giving a few additional parameters such as how to split the given augmentation pipeline and how many times to reuse each cached sample. Our evaluations on various computer vision models and datasets demonstrate that Revamper can reduce training time for DNN models by 1.03×–2.04× while maintaining comparable top-1 validation accuracy. Although we focused on evaluating Revamper on vision tasks where data augmentation is most widely used, it is notable that Revamper can also be applied to other domains such as speech [29] and language tasks [32], when augmentation pipelines can be split into multiple transformations.

2 Background

2.1 DNN Training

DNN training "trains" a DL model to perform a certain task by repeatedly adjusting the model weights with regard to the given set of training samples. Broadly speaking, training DNNs consists of two steps: data preparation and gradient computation. Until a certain termination condition is satisfied (e.g. target validation accuracy is met), the two steps are repeated for multiple epochs.

Data Preparation The data preparation step prepares training data to be fed to the DL model. Figure 2 describes a typical data preparation procedure, which is generally performed on CPU [8, 31]. The process starts with reading in the training data residing on a local or a remote storage in a random order in order to give randomness for each epoch. In general, each training data entry is a tuple of (x_i, y_i), where x_i represents the training sample at index i (e.g. an image, an audio clip, and a text snippet) and y_i the corresponding target of x_i (e.g. class and original text). The loaded training data are decoded and formatted into tensors, multi-dimensional arrays of numbers used in gradient computation.

Often, training data undergo random transformations called *data augmentation* (§ 2.2) in the next step. This optional step gives greater variation in the training set and helps train a more generalized DL model. The decoded and transformed data are collated into mini-batches before being sent to DL accelerators such as GPU and TPU. This mini-batching is necessary to perform stochastic gradient descent or its variants.

Gradient Computation The gradient computation step actually trains a DL model by adjusting the model parameters with regard to the gradients computed from the training data. This is done via forward computation and backward propagation. Forward computation calculates the loss, or the deviation of the produced result from the target value, for the given mini-batch of training samples. Backward propagation traverses the model in a reverse order and recursively computes the gradient of each layer with respect to the loss. The model parameters are then adjusted proportional to the gradients to minimize the loss.

It is important to note that the data preparation and the gradient computation steps can be overlapped, as they are typically executed on different hardware. Thus, ideally the processing time of data preparation can be hidden by that of gradient computation, and so the former has not been considered to have a significant impact on the speed of DNN training. However, recent development of specialized hardware accelerators [2, 3] has dramatically reduced the processing time required for the gradient computation step. Accordingly, the data preparation step is becoming the bottleneck of DNN training as pointed out in the recent works [9, 13, 30].

2.2 Data Augmentation

During the data preparation step, several random distortions, referred to as transformations, are applied to increase the effective number of training samples and thus to improve the generalization of DL models. These data transformation steps are collectively called data augmentation. Data augmentation is a common technique in many domains of DL, including computer vision [14, 15, 36, 39], automatic speech recognition [29] and natural language processing [12, 32].

A data augmentation pipeline is usually a sequence of multiple transformations. Here, each transformation is referred to as a layer. For example, RandAugment [15] consists of a sequence of RandAugment layers (Figure 2), each of which randomly applies one of the 14 distortions (e.g., shear, rotate, and solarize) to each sample. AutoAugment [14] searches a set of effective transformation sequences before training, and applies a sequence randomly selected from the set in every training step. As an example of an extreme use of data augmentation, Karas et al. [23] deployed at most 18 consecutive transformation layers when training generative adversarial network (GAN) models with limited data.

Multi-layered augmentations can also be seen in other domains. For example, SpecAugment [29], an augmentation method for automatic speech recognition tasks, can be decomposed into three transformation layers (time warp, frequency masking, and time masking). Such transformations are known to be computationally expensive, which is why popular speech recognition frameworks such as DeepSpeech provides an option that caches and reuses previously augmented samples [7]. Another example is CoSDA-ML [32] in natural language processing, which translates N random tokens into different languages. CoSDA-ML can be decomposed into N consecutive random transformation layers, each of which selects a token and translates it to a token in a randomly chosen language.

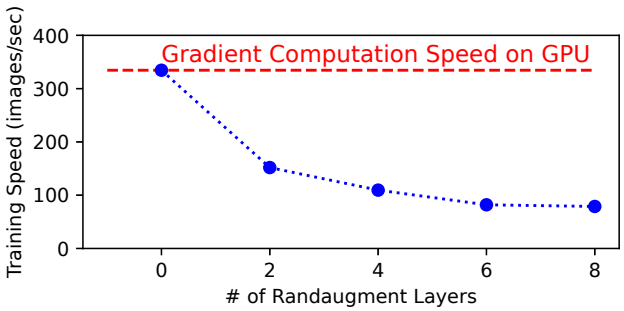

Figure 3: ResNet-50 training speed on ImageNet varying the number of RandAugment layers. The horizontal line indicates the gradient computation speed on GPU.

3 Motivation

3.1 Overhead of Data Augmentation

Data augmentation improves model generality, but it is often a bottleneck in DNN training due to its heavy CPU overhead from the multiple layers of transformations. To analyze the overhead of data augmentation, we measure the training throughput of ResNet-50 [19] model, a widely-used DL model for image classification, using the ImageNet [16] dataset with an example data preparation pipeline. As for data augmentation, we apply varying number of RandAugment [15] layers along with the random crop and random horizontal flip transformations. The number of RandAugment layers (N) is adjusted from zero to eight to investigate the CPU overhead from various loads of data augmentation. When N is zero, only the random crop and random horizontal flip are applied to training samples. We employ one NVIDIA V100 GPU and four physical CPU cores for the training, which is similar to an Amazon Web Service (AWS) p3.2xlarge instance, a standard cloud virtual machine used for DNN training.

Figure 3 plots the measured throughput of data preparation pipelines with different number of data augmentation layers. When only the random crop and flip are applied ($N = 0$), the

throughput of data preparation exceed that of gradient computation on GPU, making the data preparation step completely overlap with GPU operations. On the other hand, when the number of RandAugment layers is set to 2, which is known to produce the highest validation accuracy when training ResNet-50 on the ImageNet dataset [15], the DNN training process is bottlenecked by the data preparation. This problem becomes more severe as the number of data augmentation increases and the CPU overhead from data augmentation becomes heavier. From the result, we observe that the CPU overhead from data augmentation can be too large to be fully overlapped with the gradient computation, and thus data augmentation can be the main bottleneck in DNN training process.

3.2 Limitations of Existing Approaches

As the data preparation step is becoming the bottleneck for DNN training, there has been effort to reduce this overhead. However, due to the stochastic nature of data augmentation, such effort has failed to efficiently reduce the CPU overhead introduced by data augmentation pipelines.

Utilizing Hardware Accelerators Recent works such as NVIDIA DALI [4] and TrainBox [30] leverage hardware accelerators like GPUs and FPGAs for data augmentation. Unlike the gradient computation, which applies identical and deterministic computations to each training sample, data augmentation applies stochastic operations to each sample in a random fashion. Hence, it is difficult for data augmentation to efficiently utilize such hardware accelerators, which are optimized for massive parallel execution of homogeneous operations [9]. In addition, because such accelerators are frequently used for gradient computation, this approach may make the data preparation and gradient computation not overlapped.

Data Echoing Data echoing [9, 13] splits DNN training pipelines into the upstream and downstream pipelines, and reuses previously produced samples from the upstream pipeline in the downstream. For example, if we split the pipeline in Figure 2 between `Format` and `Augment` operations, the upstream pipeline would be `Read-Decode-Format` and the downstream pipeline would be `Augment-Collate-Transfer`. This approach is useful when the deterministic part of the data preparation pipeline, such as I/O from a remote storage, is the bottleneck. However, data echoing becomes less effective when stochastic data augmentation is the slowest part. With data echoing, the entire data augmentation pipeline is considered as a black-box operation, and so the samples are reused either before or after the augmentation process. If a sample is reused before the data augmentation, the reused sample needs to be re-augmented, and thus the overhead from data augmentation remains the same. Or, when fully augmented samples are simply reused for gradient computation, the number of unique augmented samples significantly decreases. As a result, data echoing fails to reduce the CPU overhead from data augmentation without severely harming the generalization of trained models. We further demonstrate this limitation in § 7.

Our observation suggests that it is necessary to devise a mechanism to reduce the computation overhead of CPU-heavy data augmentation techniques, while preserving generalization of the model obtained by data augmentation.

4 Data Refurbishing

We propose **data refurbishing**, a simple and effective sample reuse mechanism for input pipelines of DNN training that alleviates CPU computation for data augmentation while preserving the generalization of trained models. Data Refurbishing caches and reuses partially augmented samples generated from the *partial augmentation*, which consists of the first few transformations in the full augmentation pipeline. The rest of the augmentation pipeline—the *final augmentation*—is applied to the partially augmented samples from the cache in order to produce fully augmented samples. Reusing partially augmented samples reduce CPU computation while further transforming them with the final augmentation maintains the sample diversity obtained by data augmentation.

Data refurbishing introduces two additional configurations, the *reuse factor* and the *split policy*. The reuse factor represents how many times to reuse each cached sample, and the split policy determines how to split the full augmentation pipeline into the partial and final augmentations. Note that configuring Revamper is simple since its configuration space is small. The reuse factor is an integer that is typically smaller than five, and the number of split strategies, which is identical to the number of augmentation layers, does not exceed twenty even in extreme cases [23]. Applying data augmentation without reusing data—the *standard data augmentation*—and data echoing are both special cases of data refurbishing, as will be explained later in this section.

Data refurbishing can reduce the CPU computation required for data augmentation with minimal loss of the generalization of the trained model, given that the final augmentation provides enough sample diversity. In the rest of this section, we mathematically explain how data refurbishing preserves the sample diversity produced from the standard data augmentation.

Problem Formulation Let $\mathscr{X}$ and $\mathscr{X}'$ denote the sample space before and after augmentation, respectively. An augmentation A can then be represented as a finite set of functions such that $A \coloneqq \{f_1, f_2, \ldots, f_{|A|}\}$ for $\forall_i\, f_i : \mathscr{X} \to \mathscr{X}'$. Note that, in the standard data augmentation, we randomly choose $f_i \in A$ and produce an augmented sample $f_i(x)$ for a given input sample $x \in \mathscr{X}$ in every epoch. Then, the partial augmentation A_P and the final augmentation A_F of A can also be represented as some augmentations that satisfy $\{f_F \circ f_P | f_P \in A_P, f_F \in A_F\} = A$. In the rest of this section, we make the following assumptions to simplify our analysis.

Assumption 1. ***Discrete Uniform Distribution***

For an augmentation A, the probability of choosing $f \in A$ is uniform.

$$\forall_{f \in A}\, P(f) = \frac{1}{|A|}$$

Assumption 2. ***Balanced Eviction***

All the cached samples are reused exactly r times before being evicted from the cache.

Assumption 3. ***Uniqueness of Composed Augmentation***

Any composition of partial and final augmentation functions always produces a unique fully augmented sample.

$$\forall_{f_P, g_P \in A_P}\ \forall_{f_F, g_F \in A_F}\ \forall_{x \in \mathscr{X}}$$
$$((f_P \neq g_P)\ or\ (f_F \neq g_F)) \rightarrow (f_F \circ f_P)(x) \neq (g_F \circ g_P)(x)$$

Now let $A(x)$ denote the set of all possible augmented samples produced by an augmentation A given an input sample x. Then, Assumption 3 implies that $A(x) = \{f_1(x), f_2(x), \ldots, f_{|A|}(x)\}$ and $|A_P| \times |A_F| = |A| = |A(x)|$.

Under the above formulation, applying data refurbishing for k epochs with reuse factor r to an augmentation A for an input sample $x \in \mathscr{X}$ can be represented as a sampling process such that the samples are taken r times from $A_F(y)$ for every y sampled $\frac{k}{r}$ times from $A_P(x)$, respectively. Given k, data refurbishing can have any $1 \leq r \leq k$ by the definition. Note that data echoing and the standard data augmentation are both the special cases of data refurbishing, since data echoing is equivalent to data refurbishing with $((A_F = \{I\})$ and $(r > 1))$, and the standard data augmentation is equivalent to data refurbishing with $((A_F = A)$ or $(r = 1))$, where I denotes the identity function.

Therefore, the following theorem holds:

Theorem 1. ***Expectation of Unique Samples***

$$\mathbb{E}(U) = |A| \left(1 - \left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)^r\right)^{\frac{k}{r}}\right)$$

where U denotes the number of unique samples produced by applying data refurbishing to A given a single input sample.

The proof is given in the supplemental material.

In Theorem 1, $\mathbb{E}(U)$ is maximized when the standard data augmentation ($r = 1$ or $|A_F| = |A|$) is applied, and minimized when data echoing ($r > 1$ and $|A_F| = 1$) is applied. The expected number of unique samples of data refurbishing lies between the two.

Figure 4 visualizes the impact of r and $\frac{log|A_F|}{log|A|}$ to $\frac{\mathbb{E}(U)}{\mathbb{E}(U^*)}$ where $\mathbb{E}(U^*)$ denotes the expected number of unique samples of the standard data augmentation. In this figure, we assumed the same data augmentation pipeline used in our evaluation (§ 7), which consists of two RandAugment layers followed by a random crop and a random horizontal flip layers. Intuitively, $log|A|$ means the number of transformations comprising the data augmentation pipeline, since $|A|$ grows exponentially as the number of transformations increases. Therefore, the y-axis means the ratio of the number of transformations of final augmentation to that of full augmentation pipeline, assuming each transformation can produce the same number of unique outputs from a given input.

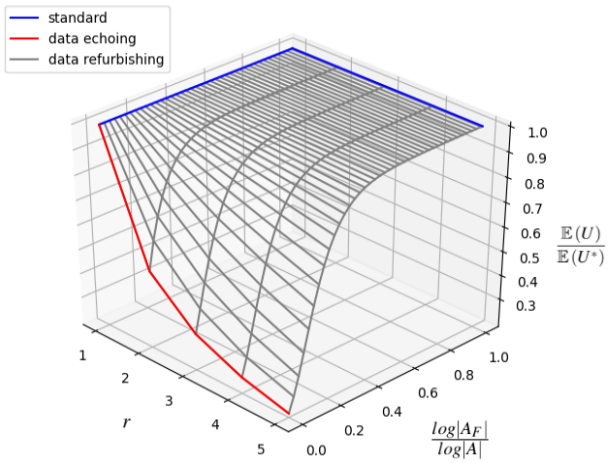

Figure 4: The normalized expected number of unique samples. The x-axis represents reuse factor r, y-axis $\frac{\log|A_F|}{\log|A|}$, and z-axis the normalized expected number of unique samples with respect to that of the standard data augmentation, $\mathbb{E}(U^*)$. Intuitively, the y-axis means the ratio of the number of transformations of final augmentation to that of full augmentation pipeline.

As the figure shows, data refurbishing is robust to the growth of reuse factor r given that $\frac{log|A_F|}{log|A|}$ is greater than 0.4. However, when $\frac{log|A_F|}{log|A|}$ decreases below 0.4, the expected number of unique samples decreases sharply as r grows. This suggests that we can save computation without significant loss of the model generalization as long as the final augmentation provides sufficient sample diversity.

Based on the above analysis, the goal of a good split policy is to find a split where the final augmentation provides sufficient sample diversity above some threshold with the minimal amount of computation. To do so, it is most desirable for the final augmentation to consist of the transformations that provide high sample diversity with little computation. In our evaluation setup in § 7, for example, one RandAugment layer is computationally heavy but can produce only 14 possible augmented samples from an input sample; on the other hand, random crop ($padding = 3$) along with random horizontal flip can produce a total of 98 augmented samples from an input sample with fewer CPU cycles.

However, if this property does not hold (i.e., the last few layers do not provide sufficient diversity or are computationally heavy), achieving both fast training speed and high accuracy

with data refurbishing might be difficult. In this case, one can consider reordering transformations inside the augmentation pipeline given that the transformations are interchangeable or such reordering puts negligible effect on augmented samples.

5 Revamper Design

We design Revamper, a new data loading system that efficiently implements data refurbishing. It incorporates data refurbishing to existing data preparation procedures of DL frameworks such as PyTorch [31] and TensorFlow [8] by replacing existing data loading systems such as PyTorch dataloader and tf.data [28].

In traditional data loading systems, all the samples in each epoch and step undergo the same data-preparation pipeline. However, with data refurbishing, a mixture of cached and non-cached samples are used to prepare fully augmented samples for the gradient computation. Because cached samples only need the final augmentation to be further applied whereas non-cached samples require both partial and final augmentation, the amount of computation needed for each step and epoch may fluctuate with the number of cache misses. This then causes fluctuation in the CPU processing time. However, the gradient computation time on DL accelerators is consistent throughout the training process, both with and without data refurbishing. For this reason, the CPU processing time may not be effectively overlapped with the DL accelerator processing time when data refurbishing is implemented in a naïve fashion.

Revamper overcomes this challenge by keeping the number of cache misses constant both across epochs and within each epoch, which effectively makes the CPU processing time for each mini-batch consistent throughout the training. First, the **balanced eviction** strategy evenly distributes the number of cache misses across epochs while ensuring that every cached sample is used for gradient computation for the same number of times. Within an epoch, the **cache-aware shuffle** leverages the cache information to choose training samples for mini-batches in order to keep the CPU computation time constant for each step.

We explain broader contexts where Revamper can be used. Revamper is applicable to both local (i.e., only one DL accelerator is used) and distributed (i.e., multiple DL accelerator or machines are used) training environments, because independent Revamper processes are created for each DL accelerator or machine. However, it assumes that training data is accessible from the local disk of each machine, which requires the size of training data that are assigned to each machine to be smaller than the capacity of its local disks. Hence, Revamper currently does not consider network overhead from fetching training data from a shared cloud storage. For such environments, one can consider using distributed caching systems for DNN training [25] along with Revamper.

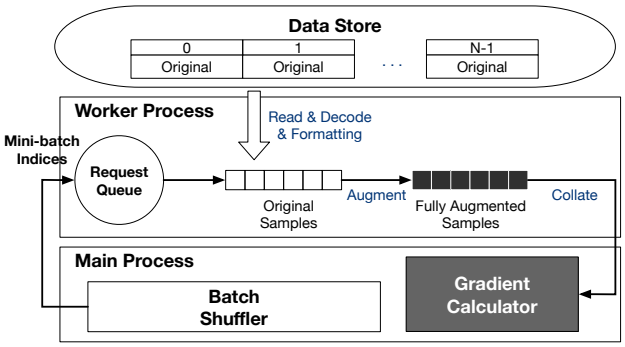

Figure 5: The architecture of a traditional data loading system (PyTorch dataloader).

5.1 Revamper Overview

Before going into Revamper, we briefly explain the existing data loading systems with the architecture of PyTorch dataloader as an example. Figure 5 explains the architecture of PyTorch dataloader, which consists of one main process and one or more worker processes. Each training sample is typically represented with a sequential integer spanning from 0 to $N-1$, where N denotes the total number of samples. The order of training samples within each epoch is decided by the batch shuffler, which randomly chooses the sample indices for mini-batches. The mini-batch indices are transferred to a worker process. After receiving them, the worker process reads, decodes, and formats the corresponding training samples from the data store. The read samples are then augmented following the user-given augmentation pipeline, making fully augmented samples. The augmented samples are then collated to make a batched sample and transferred to DL accelerators for gradient computation.

The architecture of Revamper (Figure 6) differs from those of traditional data loading systems, mainly in that (1) it has a cache store that stores partially augmented samples in memory or on disk and (2) its main process maintains a separate shuffler that selects the indices to be evicted from the cache. In addition, Revamper adopts the balanced eviction and cache-aware shuffle to stabilize the data preparation time of each mini-batch. Even with such modifications, the epoch boundaries are still intact, meaning that all the original training samples are used exactly once within each epoch of DNN training.

Data Preparation Procedure Figure 6 illustrates the end-to-end data preparation procedure of Revamper in detail. (1) Before starting each training epoch, the evict shuffler selects the samples that need to be evicted from the cache store, following the balanced eviction strategy. By doing so, Revamper balances the number of non-cached samples across epochs while ensuring that each cached sample is reused for the same number of times. (2) The cache store then invalidates the selected samples. (3) After the eviction, the main process

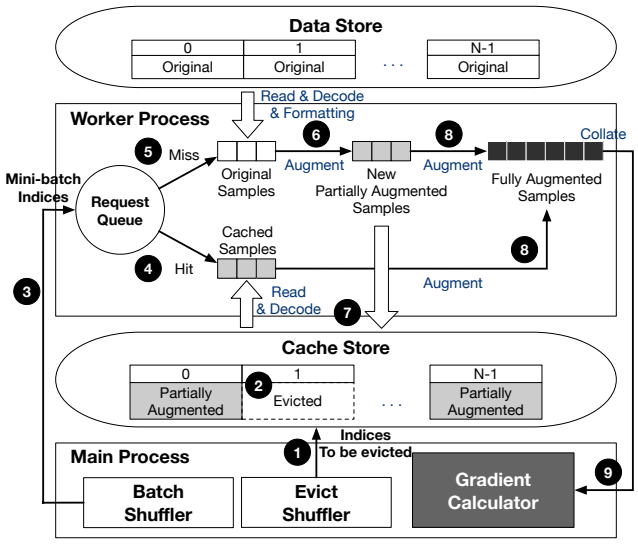

Figure 6: The architecture of Revamper and its end-to-end data preparation procedures.

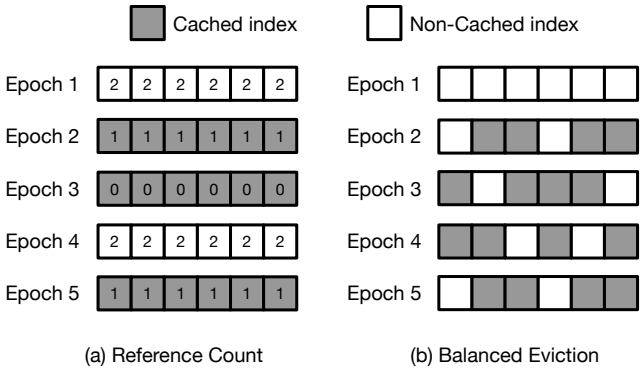

Figure 7: An example distribution of cache misses with (a) reference count algorithm and (b) the balanced eviction.

allocates mini-batch indices to each worker sampled from the batch shuffler. When sampling mini-batch indices, the batch shuffler adopts the cache-aware shuffle in order to make the CPU processing time of each mini-batch stable. (4) Each worker process fetches mini-batch indices from the request queue and reads the corresponding cached samples from the cache store. (5) For the missed samples, the worker process reads the original training samples from the data store and (6) applies the partial augmentation. (7) The worker process then stores the new partially augmented samples in the cache store in order to reuse them in the future epochs. (8) Once all the partially augmented samples for the requested indices are ready, the worker process applies the final augmentation to them. (9) Finally, the fully augmented samples are collated and transferred for the gradient computation.

Cache Store The cache store provides an interface similar to that of key-value stores. It supports `get(I)`, `put(I, S)`, and `remove(I)` methods, where `I` denotes an index and `S` denotes a partially augmented sample to be cached. Partially augmented samples are either stored in memory or on disk according to the user-given `store_disk`. If the `store_disk` is turned off, partially augmented samples are stored in an in-memory hash map that maps indices and the corresponding cached samples. Hash maps are appropriate for DNN training, because it provides $O(1)$ data access by sacrificing performance of range access (i.e., reading all the indices from k_1 to k_2), which is not necessary during DNN training.

Storing partially augmented data on disk is useful when training models with a large dataset whose size exceeds a hundred of GBs [16]. Revamper performs disk I/O in one of the following two ways depending on the size of partially augmented samples. If the size of each sample is below threshold (16KB by default), Revamper batches multiple I/O requests to reduce system call overhead. To batch write requests, the cached samples are firstly written to in-memory write buffers and flushed to shared log files when the buffers are full. Revamper also batches multiple sample reads within a mini-batch by packing multiple read requests into a single system call using the `AIO` library of Linux [26]. The cache store periodically clears up invalidated data through background compaction [5], which makes new log files that contain only the valid samples. If the size of each sample is large enough, on the other hand, system call overhead becomes negligible. In this case, Revamper stores each sample in a separate encoded file in order to avoid compaction overhead.

5.2 Balanced Eviction

The balanced eviction is a cache eviction policy that maintains an even spread of computation overhead across the training epochs as well as ensures that each sample is reused for the same number of times. Preparing data from the non-cached samples requires more computation than doing so from the cached samples, because the former needs both the partial and final augmentations to be applied. The computation overhead of each epoch may thus vary depending on the number of cache misses in the epoch when a naïve design of data refurbishing such as the reference count algorithm is used. The reference count algorithm maintains the remaining reference for each cached data and evicts the cached data when a sample's corresponding reference becomes zero. Although this algorithm ensures that each sample is reused for the same number of times, this approach results in uneven distribution of computation overhead across epochs. As shown in Figure 7 (a), some epochs need to prepare a large number of non-cached samples while others do not, because the remaining references of all the indices decrease at the same rate and become zero at the same time. Such uneven distribution of non-cached samples increases the blocking time between CPU and DL accelerators. Because computation required for

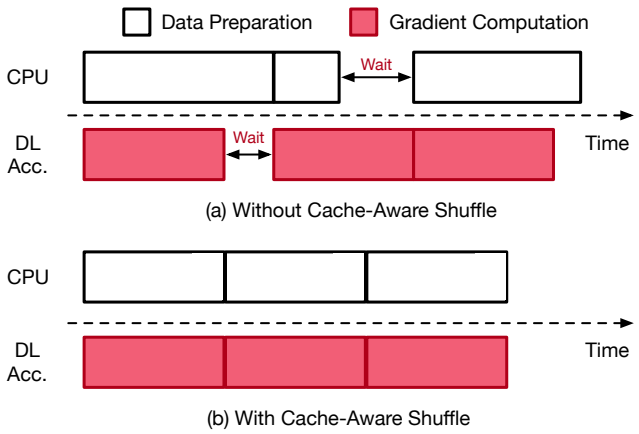

Figure 8: An example illustration of CPU and DL accelerator utilization with and without the cache-aware shuffle. The bidirectional arrow blocking time caused by uneven mini-batch processing time. Each block represents the computation time for corresponding batches.

data augmentation is skewed to a small number of epochs, DL accelerators may wait CPU in such epochs and vice versa in the other epochs.

To solve this problem, we propose the balanced eviction. At the start of each training epoch, the evict shuffler samples $\frac{N}{r}$ indices to be evicted, where N denotes the number of training samples and r denotes the reuse factor. In addition, the shuffler samples the indices without replacement and repeats the same sampling order until the end of training process. By adopting this strategy, the balanced eviction evenly distributes the computation overhead across epochs after the first epoch, by evicting the same number of partially augmented samples in each epoch. It also ensures that except for the the first r epochs, each cached sample is reused exactly r times. Figure 7 (b) illustrates an example of the data loading procedure with $N = 6$ and $r = 3$. After the first epoch, two ($N/r = 2$) samples are evicted from the cache and replaced with new partially augmented samples. All the cached samples are also evicted every three ($r = 3$) epochs, because the evict shuffler always selects indices in a same order.

5.3 Cache-Aware Shuffle

While the balanced eviction effectively addresses the inter-epoch computation skew, the intra-epoch computation skew may still slow down the training speed. Figure 8 (a) illustrates a worst-case example that can happen when the batch shuffler does not adopt the cache-aware shuffle. The non-cached indices are skewed to the first and the third mini-batch, whereas the second batch only contains cached indices. Because the processing time of the data preparation fluctuates while that of the gradient computation remains the same, this results in unnecessary blocking between CPU and DL accelerators.

The cache-aware shuffle solves this problem by leveraging cache information when deciding the indices of the samples for each mini-batch. Because cached samples are evicted only before starting each training epoch, Revamper knows the indices of the evicted samples by the beginning of each epoch. By utilizing this knowledge, the cache-aware shuffle prepares mini-batches in a way that each mini-batch has the same ratio of cached to non-cached samples. Figure 8 (b) shows that the cache-aware shuffle makes the processing time for all mini-batches stable and helps avoid blocking time between CPU and DL accelerators. We ensure the randomness of the mini-batch indices by randomly sampling from both non-cached indices and cached indices. This does not adversely affect the validation accuracy of trained models as shown in § 7.4, because the training order has little impact on the model accuracy as long as it is random [25].

6 Implementation

We implement Revamper with 2000+ lines of code based on PyTorch 1.6 [31] with Python 3.7. Revamper overrides the existing PyTorch dataloader with almost identical interface except for additional parameters such as the reuse factor, the final and partial augmentation, and whether to store cached samples to disk or not. To use Revamper, users only need to override the existing `torch.utils.data` package with our code.

Due to the global interpreter lock (GIL) of Python [11], Revamper workers are executed on separate processes, which makes it hard to share cached samples among the workers if the samples are stored in memory. As a workaround, Revamper puts cached samples inside the main process and transfers them to necessary workers. Such inter-process tensor transfer may cause frequent shared memory allocation, because PyTorch's `multiprocessing.queue` puts tensors inside the shared memory region when they are transferred between processes, To avoid this problem, Revamper preallocates buffers in the shared memory region and reuses those buffers for inter-process communications.

7 Evaluation

Environment We perform our evaluation on a dedicated training server equipped with 2×Intel Xeon E5-2695v4 CPU (18 cores, 2.10GHz, 45MB Cache), 256GB DRAM, a NVIDIA V100 GPU, and a Samsung 970 Pro 1TB NVMe SSD. We adjust the ratio of the number of CPU cores to the number of GPUs by setting different numbers of CPU cores using a docker container [6] and fixing the number of GPUs to one. By default, we set the CPU-GPU ratio to four, which effectively emulates the ratio in AWS P3 instances [1], but we also evaluate Revamper on different CPU-GPU ratios. Our evaluation is done on PyTorch 1.6. We replace PyTorch

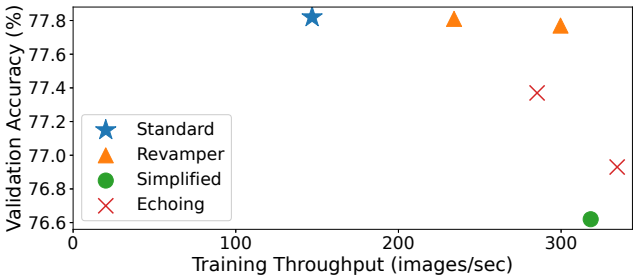

Figure 9: Training throughput and model validation accuracy of ResNet50 trained on ImageNet with diverse settings using RandAugment. Different points of the same setting represent the results under different reuse factors (2 or 3).

dataloader with Revamper for data preparation.

DNN Models and Datasets We evaluate Revamper on several DNN models for image recognition and on two datasets, ImageNet [16] and CIFAR-10 [24], which represent large and small datasets respectively. We train ResNET-50 [19] on ImageNet. On CIFAR-10, we train VGG-16 [33], ResNET-18 [19], MobileNet [20], and EfficientNet-B0 [34].

Baselines We evaluate data refurbishing implemented in Revamper against the following baselines.

- **Standard**: The standard setting represents the canonical DNN training with full augmentation without any reuse mechanism. The accuracy of the model trained under this setting serves as the target accuracy for the other data reusing mechanisms.
- **Data Echoing**: We evaluate data echoing [9, 13] with `echo-after-augment` strategy, in which each fully augmented sample is reused r times, where r denotes the user-given reuse factor. We do not evaluate the other two strategies, `echo-before-augment` and `echo-after-batch`, since they are less relevant and/or not a good baseline. When the training data resides in local SSDs, data echoing with `echo-before-augment` strategy is almost identical to the standard setting with additional encoding and disk write. `echo-after-batch` is reported to result in a lower accuracy with little training throughput improvement [13]. To make a fair comparison, we keep the size of the cache store for data echoing equal to that of Revamper.
- **Simplified**: In this setting DNN models are trained with no reuse mechanism but with fewer transformation layers compared to those of the standard setting. This approach is a baseline optimization for reducing the computation overhead of data augmentation by simply removing one or more transformations.

We use the identical model hyperparameters (e.g., the number of training epochs, learning rate, batch size, and optimizer) for each setting.

We do not evaluate a baseline without augmentation, since random crop and random flip are considered as the norm for training computer vision models. Such baseline only results in a lower accuracy with little improvement on training throughput compared to the simplified setting, as training models without augmentation has been reported to result in a significantly lower accuracy [30], and the simplified setting already makes training throughput bounded by GPUs.

Augmentation and Split Policy We apply RandAugment [15] and AutoAugment [14], the two state-of-the-art data augmentation techniques, accompanied with the random crop and random horizontal flip. After § 7.2, we use RandAugment for the data augmentation methodology.

In most of the experiments except for § 7.2, we use a single split policy: RandAugment or AutoAugment layers for the partial augmentation, and the rest of augmentation (i.e., random crop and random horizontal flip) for final augmentation.

7.1 Comparison with Baselines

ImageNet Training with RandAugment We first evaluate the training throughput and the top-1 accuracy of ResNet-50 trained on ImageNet dataset with RandAugment using Revamper and the other baselines. We follow the model hyperparameters (or configurations) from [15]. We set the number of RandAugment layers (N) to 2 and the distortion magnitude (M) to 9. Then we have a total of four transformation layers, including random crop and flip layers. We also follow the original paper when configuring other hyperparameters, such as the learning rate per batch size, the optimizer and its configurations, and the number of total epochs. We evaluate Revamper and data echoing with two different reuse factors, 2 and 3. For the simplified setting, we use a simpler data augmentation pipeline consisting of a random crop and a random horizontal flip layers. We execute three runs for each point and report the averaged results.

As shown in Figure 9 (a), when training ResNet-50, Revamper achieves top-1 accuracy comparable to the accuracy of 77.82% under the standard setting with better training throughput. With the reuse factor of 2, 77.81% accuracy is achieved with 1.59× training speed-up. With the reuse factor increased to 3, the training throughput improves by 2.04× while maintaining a comparable accuracy of 77.77%. On the other hand, data echoing fails to achieve comparable validation accuracy, having only 77.37% and 76.93% top-1 accuracy for the reuse factor of 2 and 3, respectively. The result demonstrates that Revamper is beneficial over data echoing in that Revamper can maintain comparable accuracy to that of the standard setting, whereas data echoing cannot avoid accuracy degradation even with the smallest reuse factor 2. For example, Revamper with the reuse factor 3 provides 0.84% validation accuracy improvement compared to data echoing with the same reuse factor.

The result also shows that Revamper provides better trade-

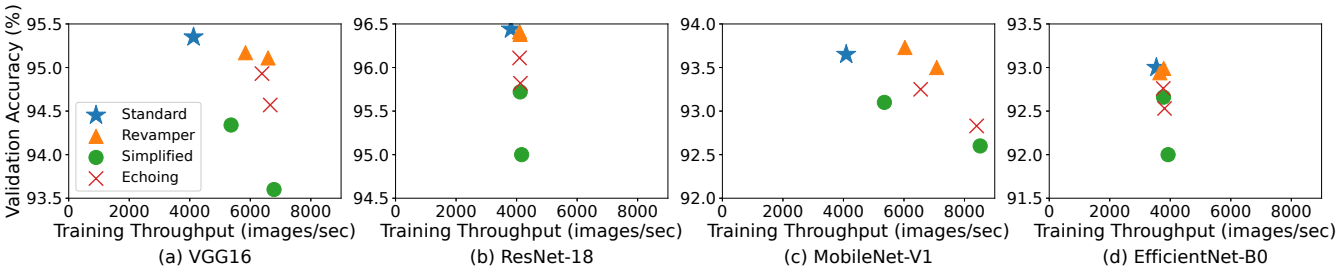

Figure 10: Training throughput and top-1 validation accuracy of DNN models trained on CIFAR-10 using RandAugment. Different points of the same setting represent measurements under different reuse factors (2 or 3) for Revamper and data echoing and under different numbers of removed transformation layers (1 or 2) for the simplified setting.

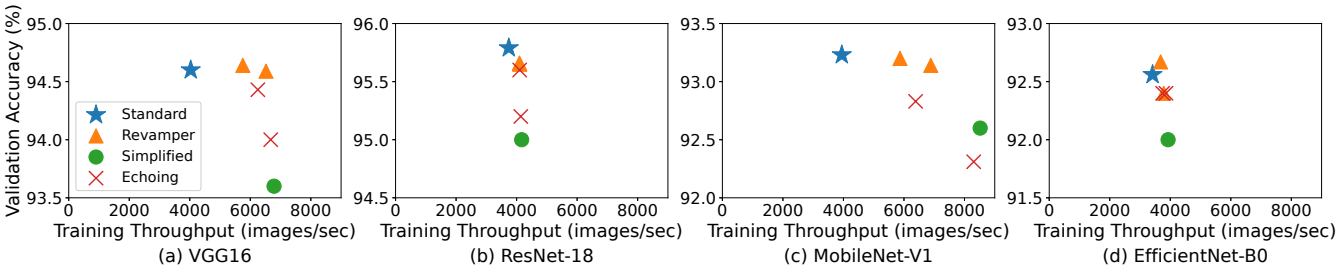

Figure 11: Training throughput and top-1 validation accuracy of DNN models trained on CIFAR-10 using AutoAugment. Different points of the same setting represent the results under different reuse factors (2 or 3).

off points between training throughput and accuracy than data echoing. For example in ResNet-50, compared to data echoing with the reuse factor 2, Revamper with the reuse factor 3 provides faster training throughput (299.66$images/sec$ vs. 285.25$images/sec$) and better validation accuracy (77.77% vs. 77.37%).

The simplified setting achieves the worst validation accuracy with the training throughput similar to that of Revamper with the reuse factor 3, demonstrating that naïvely removing transformations from the pipeline does not provide a good trade-off between training throughput and accuracy.

CIFAR-10 Training with RandAugment Next, we present the performance of our evaluation of training models on a small dataset. We set the number of RandAugment layers (N) to 2 and the distortion magnitude (M) to 10. Same as ImageNet training, we evaluate data refurbishing and data echoing with two reuse factors, 2 and 3. For the simplified setting, we evaluate two different augmentations: one with random crop and random horizontal flip and the other with an additional RandAugment layer. We execute three runs for each point and report the averaged results.

The evaluation results are summarized in Figure 10. For VGG-16 (Figure 10 (a)) and MobileNet-V1 (Figure 10 (c)), Revamper achieves 1.42×–1.73× speed-up while maintaining validation accuracy comparable to the standard setting. However, for ResNet-18 (Figure 10 (b)) and EfficientNet-B0 (Figure 10 (d)), Revamper does not show significant training throughput improvement, exhibiting only 1.03×–1.08× speed-up. This is because these models require more GPU computation time for the gradient computation, and so the training process is bottlenecked by the GPUs rather than the CPUs. Such results suggest that Revamper is beneficial only when DNN training tasks are CPU-bound, but we predict that more training tasks will benefit from Revamper in near future considering rapid performance improvement of DL accelerators [2, 3].

Although data echoing and the simplified setting show an improved training throughput, their validation accuracy deviates much from that of the standard setting. Figure 10 also shows that at many points Revamper demonstrates both higher accuracy and faster training throughput compared to those of data echoing and the simplified setting. Likewise, Revamper provides better trade-offs between training throughput and accuracy than the other baselines.

CIFAR-10 Training with AutoAugment We then evaluate DNN models trained with AutoAugment [14] on CIFAR-10. We use the same configuration we used in the RandAugment evaluation except for the augmentation method. For the simplified setting, we evaluate a data augmentation pipeline with random crop and random horizontal flip, since we cannot manually adjust the number of layers once the policy is found by AutoAugment. We execute three runs for each point and report the averaged results.

Figure 11 demonstrates the results. Compared to the stan-

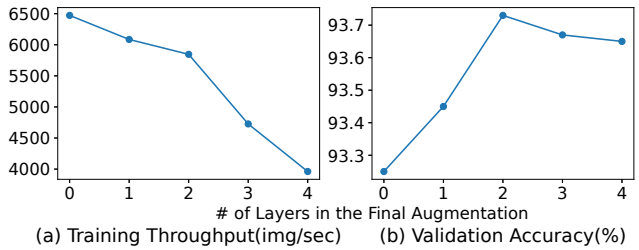

Figure 12: The training throughput and the top-1 validation accuracy for different split policies (MobileNet-V1 on CIFAR-10).

dard setting, Revamper shows 1.08×–1.75× speed-up while achieving comparable top-1 accuracy within ±0.16% range. Similar to the RandAugment evaluation, models that require less GPU computation (VGG16, MobileNet) has shown greater training throughput gain compared to the models (ResNet-18, EfficientNet-B0) that need heavy GPU computation. Revamper again has better trade-off points between training throughput and accuracy compared to data echoing and the simplified setting.

7.2 Augmentation Split Policy

We evaluate and analyze the trade-offs between training throughput and accuracy for different split policies. The final augmentation contains the last one to three transformation layers of the RandAugment pipeline from Figure 2. Figure 12 summarizes how the number of final augmentation layers affects training throughput and accuracy. Training throughput decreases as the number of the transformations in the final augmentation increases due to heavier CPU overhead. On the other hand, the top-1 accuracy peaks when the final augmentation consists of two transformations and does not increase with additional transformations. This is because the final augmentation with two transformations provides enough sample diversity. As we describe in § 4, once enough sample diversity has been achieved, further providing sample diversity does not significantly improve the model generalization but only degrades the training throughput.

7.3 CPU-GPU Ratio

We evaluate the training throughput change of training MobileNet-V1 on the CIFAR-10 dataset and ResNet50 on the ImageNet dataset with CPU-GPU ratios varying from two to six. As summarized in Figure 13, the training throughput of Revamper scales well upon the increasing number of CPUs, as long as it is not bottlenecked by DL accelerators. Also, the performance gain from Revamper is maximized in training environments with fewer CPUs.

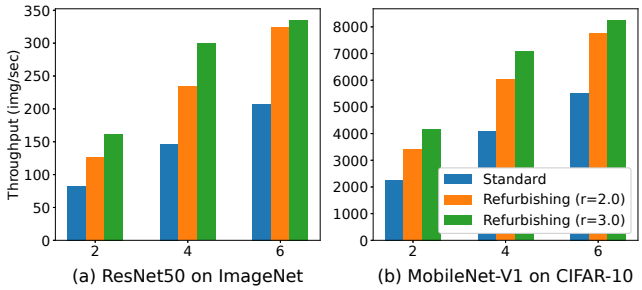

Figure 13: The training throughput of ResNet50 on ImageNet and MobileNet-V1 on CIFAR-10 for varying CPU-GPU ratios.

	Balanced + CAS	Naïve	Standard
VGG-16	5839.08	4746.91	4125.07
ResNet18	4098.36	3884.40	3813.30

Table 1: The comparison of the throughput (*images*/*sec*) of data refurbishing with and without Revamper's key features, balanced eviction and cache-aware-shuffle (CAS).

7.4 Revamper Key Design Features

Revamper provides distinctive features—the balanced eviction and cache-aware shuffle—in order to efficiently support data refurbishing. We evaluate how these features further improve the DNN training throughput compared to the naïve approach with the reference count algorithm and the random shuffle. As Table 1 shows, the naïve approach provides 1.15× faster training throughput for VGG-16 training on CIFAR-10 than the standard data loading system, but with the balanced eviction and cache-aware shuffle, the speed-up gain can be as much as 1.42× compared to the standard one. For ResNet18, however, there is no evident additional speed-up gain with the balanced eviction and the cache-aware shuffle. Since the heavy GPU computation needed for ResNet18 training causes the gradient computation to be the main bottleneck, data refurbishing itself has no significant improvement in the training throughput. In summary, the balanced eviction and the cache-aware shuffle of Revamper contributes much to training throughput improvement whenever the DL accelerator is not the main bottleneck. Cache-aware shuffle, although it adjusts the sample order when preparing each mini-batch, does not adversely affect the accuracy of the model, as evidenced in Figure 14. As such, the balanced eviction strategy and cache-aware shuffle help Revamper support data refurbishing efficiently without negatively affecting the model generalization.

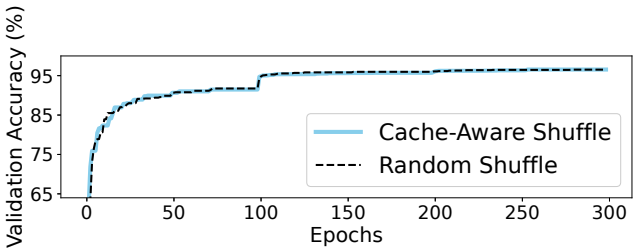

Figure 14: The change in validation accuracy over training epochs of ResNet18 trained on CIFAR-10 with different shuffle strategies.

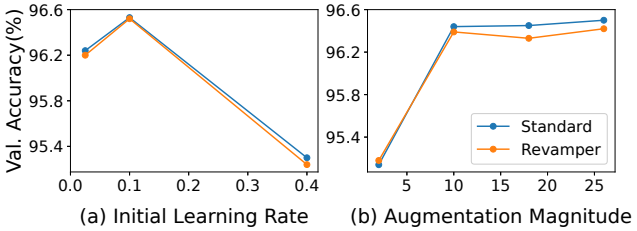

Figure 15: The top-1 validation accuracy of ResNet18 trained with different hyperparameter configurations.

7.5 Robustness to Hyperparameter Change

In this evaluation, we show that Revamper preserves the model accuracy of the standard setting under various hyperparameters. We have varied two hyperparameters–the initial learning rate and the distortion magnitude of RandAugment. As shown in Figure 15, the accuracy of the model trained with Revamper is well aligned with the one trained with the standard setting. This indicates that Revamper can achieve the validation accuracy comparable to that of the standard setting on various hyperparameter configurations.

8 Related Work

We discussed the limitations of existing approaches that accelerate data augmentation in § 3.2. In this section, we introduce other works that are closely related to our system.

Accelerating Data Preparation Quiver [25] proposes a caching system between local and remote storage shared by multiple DNN training jobs, in order to optimize slow data read from remote storage with limited cache. To achieve this goal, it leverages internal information of DL frameworks to optimize cache loading and eviction. Quiver and Revamper differ in that Quiver focuses on sharing cached training data among multiple tasks while ensuring randomness of training order, whereas Revamper focuses on reusing partially augmented data while keeping sample diversity obtained from data augmentation. OneAccess [21] proposes to use a shared data preparation pipeline to train multiple DNN models with the same dataset. Revamper, on the other hand, focuses on optimizing data augmentation within a single DNN training task. SMOL [22] dynamically adjusts the fidelity of input data to avoid data preparation bottleneck at inference time. Unlike SMOL, Revamper focuses on fast DNN training with data augmentation rather than DNN inference.

Intermediate Data Caching Many data processing systems such as Spark [37] and Nectar [18] adopts intermediate data caching, which caches and reuses frequently used intermediate results in order to reduce computation overhead. Similarly, Revamper reuses intermediate results in the augmentation pipeline but instead handles stochastic data. In the context of DL pipelines, it is necessary to consider new aspects that have not yet been considered in previous work, such as maintaining the diversity of augmented samples and maximizing computation overlap between CPU and DL accelerators. In this work, we propose complete system design and implementation that address these new challenges.

9 Conclusion

In this paper, we present data refurbishing, a novel sample reuse mechanism that accelerates DL training while maintaining model generalization. We realize this idea by designing and implementing Revamper, a new caching and data loading system that solves system-side challenges from caching partially augmented data. Revamper has shown 1.03×–2.04× speed-up in DNN training while maintaining comparable accuracy. We hope that this work will encourage further research to rethink well-studied topics like caching in systems in the new context of deep learning.

Acknowledgments

We thank our shepherd Jonathan Mace and the anonymous reviewers for their insightful feedback. We also thank Youngseok Yang, Taegeon Um, Soojeong Kim, Taebum Kim, Yunmo Koo, Jaewon Chung, and the other members of the Software Platform Lab at Seoul National University for their comments on the draft. This work was supported by the MSIT(Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program (2020-0-01649) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation), the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test), and Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Amazon EC2 P3 - Ideal for Machine Learning and HPC - AWS. https://aws.amazon.com/ec2/instance-types/p3/.

[2] Cloud Tensor Processing Units (TPU). https://cloud.google.com/tpu/docs/tpus.

[3] NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/.

[4] NVIDIA DALI. https://github.com/NVIDIA/DALI.

[5] RocksDB Compaction. https://github.com/facebook/rocksdb/wiki/Compaction.

[6] Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints/.

[7] Training Your Own Model—DeepSpeech 0.9.3 documentation. https://deepspeech.readthedocs.io/en/r0.9/TRAINING.html#augmentation.

[8] 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 *USENIX OSDI*, pages 265–283, 2016.

[9] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic optimization with laggard data pipelines. *NeurIPS*, 33, 2020.

[10] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *ACM SIGKDD*, pages 2623–2631, 2019.

[11] David Beazley. Understanding the python gil. In *PyCON Python Conference*, 2010.

[12] Jiaao Chen, Zichao Yang, and Diyi Yang. Mixtext: Linguistically-informed interpolation of hidden space for semi-supervised text classification. *arXiv preprint arXiv:2004.12239*, 2020.

[13] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2020.

[14] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *IEEE CVPR*, pages 113–123, 2019.

[15] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *NeurIPS*, 2020.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE CVPR*, pages 248–255. Ieee, 2009.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *USENIX OSDI*, volume 10, pages 1–8, 2010.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE CVPR*, pages 770–778, 2016.

[20] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[21] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *USENIX HotCloud*, 2019.

[22] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *VLDB*, 14(2):87–100, 2020.

[23] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. *NeurIPS*, 33, 2020.

[24] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[25] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *USENIX FAST*, pages 283–296, 2020.

[26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *ACM SOSP*, pages 447–461, 2019.

[27] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *MLS*, volume 1, 2020.

[28] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[29] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. Specaugment: A simple data augmentation method for automatic speech recognition. *Interspeech*, pages 2613–2617, 2019.

[30] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: An extreme-scale neural network training server architecture by systematically balancing operations. In *IEEE/ACM MICRO*, pages 825–838. IEEE, 2020.

[31] 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. In *NeurIPS*, pages 8026–8037, 2019.

[32] Libo Qin, Minheng Ni, Yue Zhang, and Wanxiang Che. Cosda-ml: Multi-lingual code-switching data augmentation for zero-shot cross-lingual nlp. In *ICJAI*, pages 3853–3860, 2020.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[34] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

[36] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *IEEE ICCV*, pages 6023–6032, 2019.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, pages 15–28, 2012.

[38] Albert Zeyer, Kazuki Irie, Ralf Schlüter, and Hermann Ney. Improved training of end-to-end attention models for speech recognition. *Interspeech*, pages 7–11, 2018.

[39] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *ICLR*, 2018.