SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training

Redwan Ibne Seraj Khan
Virginia Tech

Ahmad Hossein Yazdani
Virginia Tech

Yuqi Fu
University of Virginia

Arnab K. Paul
BITS Pilani, Goa

Bo Ji
Virginia Tech

Xun Jian
Virginia Tech

Yue Cheng
University of Virginia

Ali R. Butt
Virginia Tech

Abstract

Deep learning training (DLT) applications exhibit unique I/O workload behaviors that pose new challenges for storage system design. DLT is I/O intensive since data samples need to be fetched continuously from a remote storage. Accelerators such as GPUs have been extensively used to support these applications. As accelerators become more powerful and more data-hungry, the I/O performance lags behind. This creates a crucial performance bottleneck, especially in distributed DLT. At the same time, the exponentially growing dataset sizes make it impossible to store these datasets entirely in memory. While today's DLT frameworks typically use a random sampling policy that treat all samples uniformly equally, recent findings indicate that not all samples are equally important and different data samples contribute differently towards improving the accuracy of a model. This observation creates an opportunity for DLT I/O optimizations by exploiting the data locality enabled by importance sampling.

To this end, we design and implement SHADE, a new DLT-aware caching system that detects fine-grained importance variations at per-sample level and leverages the variance to make informed caching decisions for a distributed DLT job. SHADE adopts a novel, rank-based approach, which captures the relative importance of data samples across different mini-batches. SHADE then dynamically updates the importance scores of all samples during training. With these techniques, SHADE manages to significantly improve the cache hit ratio of the DLT job, and thus, improves the job's training performance. Evaluation with representative computer vision (CV) models shows that SHADE, with a small cache, improves the cache hit ratio by up to 4.5× compared to the LRU caching policy.

1 Introduction

Deep learning (DL) approaches are increasingly being employed to solve crucial complex problems. The use of DL has become common in disparate domains such as health sciences [28, 29, 43, 64], environmental sciences [41, 47], bio-technical systems [48], high-energy scientific experiments [16], finance [25,33,35,39], smart cities [12,19], industrial production [13, 79], autonomous vehicles, and IoT systems [45, 58, 72]. Moreover, DL has given rise to a huge market that is expected to reach 12.12 billion dollars by 2025 [4]. To meet the demands of unprecedented scale and performance, DL researchers and practitioners are developing distributed DL, which employs distributed computing and storage resources to support DL. While promising, the approach poses numerous challenges in handling massive workloads while keeping the usage cost in check.

DLT is extremely compute-intensive and data-intensive [24], and the resource demands vary at different phases of the process [22, 42]. A key challenge is efficiently matching the DL application needs with available system resources. A common practice is to scale up/out a DL training job using multiple compute accelerators such as GPUs, FPGAs, or custom ASICs; that is, by using data parallelism [71] with each accelerator, e.g., GPU, holding a replica of the model and processing a subset of the training data in parallel.

A large body of research has focused on optimizing the efficiency of computing [53, 57], scheduling [37, 82], and data communication [74, 77, 81] for DL jobs. This is because data-parallel DL training is both compute-intensive—typically requiring multiple GPUs to train in parallel—and communication-intensive [17, 59, 75]—newly calculated model gradients are transferred or broadcast to all the involved GPUs for iterative model updates. However, as state-of-the-art research [14, 56] demonstrates, the efficiency of data storage and retrieval can also significantly impact the end-to-end performance of DL training.

To better understand the impact of data storage configuration on distributed DL training efficiency, we perform an experiment to study the performance difference when distributed DL jobs are run using a local or a remote storage medium. Figure 1 shows that remote storage mediums can significantly impact the training time ($\sim 2.5\times$) compared to faster storage mediums, i.e., RAM, even though all the rest

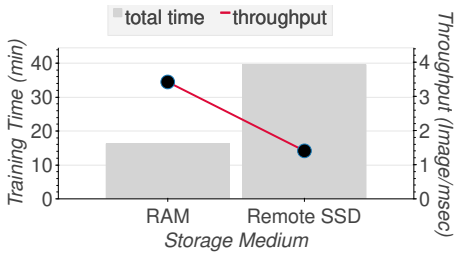


Figure 1: Training throughput and time comparison of a single job using 2 nodes and 8 GPUs with ResNet-18. Remote storage comprises of SSDs on a BeeGFS server.

of the training configurations were kept the same. This result is in line with recent studies [20, 70], which show that I/O can take up from 85-90% of the total training time. As high-performing accelerators can consume the training data samples faster, efficient I/O can significantly reduce the training time.

However, it remains challenging to improve the I/O efficiency for distributed DLT as the I/O workloads of a DLT job exhibit unique patterns: (1) full-object, sequential, read-only accesses at per-object level; (2) dominant, small, random I/Os spread across the whole training sample dataset [88]; and (3) highly concurrent I/Os [70]. Today's high-performance and distributed storage systems, such as parallel file systems (PFS) [21, 40], network file systems (NFS) [63], and cloud object stores [2, 5] are inefficient at supporting dsitributed DL applications. This is especially true given the excessive metadata overhead for small-I/O-intensive accesses [85].

For efficient I/O, faster storage mediums like RAM are needed, but compared to increasingly large training datasets that can range from terabytes [36, 67] to petabytes [10], these resources are often too small, even on large supercomputers like Piz Daint (64GB RAM/node) [3] and Fugaku (32GB RAM/node) [8]. Moreover, because of the high cost of GPUs, DL jobs are mainly run by renting GPU Spot VMs [1, 9, 15, 49, 56, 78] that are 6-8× cheaper than dedicated VMs. As these VMs are preemptive, meaning these can be terminated at any time depending on available resources, DL training has to be resumed from a checkpoint on a different VM leading to the loss of local SSD state. As a result, instead of local SSDs, large datasets are put in persistent cloud storage, and training is conducted on VMs that access the cloud storage remotely.

Worse, conventional wisdom holds that the I/O workload of a DL training job is not *cache-friendly* due to the aforementioned I/O randomness and lack of data locality [85]. This property renders existing caching policies (such as LRU and LFU) ineffective, as there is no recency or frequency pattern to exploit. Recent work such as Mercury [84], CoorDL [65], Quiver [56], and Hoard [69] try to solve this I/O problem by employing caching techniques. Unfortunately, none of them provides fundamental solutions that enable the ability to cache (i.e., *cacheability*) a DLT job's working set. The main reason is that these works consider that each sample will only be accessed once in every epoch (one iteration over the dataset). However, as has been shown by prior work [50], some samples are more important than others in DL training. Hence, if we can design effective mechanisms and policies to exploit this importance variance, we can fundamentally improve the cacheability for DL training.

In this paper, we show that we can deliver better cacheability by designing a new dataset sampling algorithm inspired by importance-sampling [61] and an effective caching policy atop that. DL models are trained on a dataset in batches (multiple equal partitions of the entire dataset). Our sampling algorithm combines the intra-batch importance of individual data samples with inter-batch importance to detect the most important samples for placing in the in-memory pooled cache. We develop a novel technique of *rank-based importance* that ranks the training samples within a batch based on their contributions to increasing the overall accuracy of the model. Rank-based importance further helps increase the probability of identifying (predicting) the most important samples in later epochs. Using this technique, we further design a *priority-based sampling strategy* that ensures multiple accesses to the important samples within an epoch to train more on hard-to-learn samples to increase the accuracy improvement rate. As a result, our caching solution keeps the most important samples in the cache and avoids random evictions, which in turn improves the cache hit ratio and training throughput.

Specifically, this paper makes the following contributions.

- We introduce a novel, rank-based importance calculation approach to precisely identify the relative importance of data samples for DLT jobs.
- We design a priority-based sampling policy to exploit the data locality of samples.
- We present the design and implementation of SHADE, a new DLT-aware caching system that incorporates rank-based importance scores and the priority-based sampling policy to improve the I/O efficiency for DLT jobs.
- We incorporate and evaluate SHADE in the widely-used DLT framework PyTorch and compare SHADE against a series of baseline and advanced caching and sampling methods. Our results show that SHADE: improves the read hit ratio by up to 4.5× given the same cache size, increases the training throughput by up to 2.7×, and reaches accuracy convergence by up to 3.3× faster compared to a baseline LRU caching policy.

SHADE is open source and publicly available at:
https://github.com/R-I-S-Khan/SHADE.

2 Background

2.1 Distributed Deep Learning Training

There are mainly three types of distributed DL training techniques: *data-parallel training* [71], *model-parallel train-*

ing [30], and *pipeline parallelism* [44] that combines data-parallel and model-parallel training. While this paper focuses mainly on data-parallel training based on Stochastic Gradient Descent (SGD), our approach is applicable to other training methods as well.

A Deep Neural Network (DNN) model consists of multiple layers of computation units whose output is the input for subsequent units. DNN model training consists of a forward propagation method, which sequentially moves information related to the input data through all model layers and generates a prediction. For example, in an image recognition application, image pixels information is moved through the layers for predicting image contents. To generate the prediction, DL defines a cost/loss function with respect to the forward propagation output and ground truth labels. The DL process aims to minimize the cost function through a process of increasing or decreasing the weights of the outputs of the intermediary layers of the model so that it can improve its prediction. This step is known as backward propagation, which adjusts the parameters of the DL model starting from the outermost layer back up to the input layer through a technique known as gradient descent optimization. Gradient descent adjusts the parameters in the opposite direction of the gradient. SGD is a stochastic approximation of gradient descent optimization; instead of calculating the gradient from the entire data set, SGD randomly selects a subset of training data samples from the entire dataset to reduce computation cost.

In a typical data-parallel, SGD-based training, the whole training dataset is partitioned and processed in parallel by multiple GPU devices. Each GPU has a replica of the same DNN model, which is iteratively synchronized with other GPUs using centralized communication techniques, e.g., parameter server [59]) or decentralized communication techniques, e.g., all-reduce [17].

2.2 I/O Characteristics of Data-Parallel DL Training

DL training applications feature unique characteristics that differentiate them from conventional data-intensive applications such as big data analytics [31, 83] and web applications [6, 7]. A DL training job typically runs multiple epochs, with each epoch consuming the entire training dataset once in a random permutation order. Each epoch is further divided into multiple batches. At the beginning of processing a batch, each GPU process loads a randomly-sampled bulk (i.e., a mini-batch) of training data whose size is configurable. These behaviors lead to highly-concurrent, read-only, repetitive, and totally random I/O accesses. Therefore, a common belief is that such I/O patterns are not cache-friendly to traditional caching policies that exploit recency and/or frequency-based data locality, such as the widely used LRU, LFU, and ARC [62].

2.3 DL Training with Importance Sampling

Traditionally, SGD-based DL training is oblivious to the "importance" of training samples and simply applies random sampling or shuffling to generate a random permutation order at the end of each training epoch, thereby treating all training samples equally. Recently, researchers found that in SGD-based DL training, a specific set of training samples tend to generate little-to-no impact on the model quality and, therefore, can be ignored [50, 61]. This process of finding the set of training samples that are more important than others, i.e., contribute the greatest towards the loss function, is known as importance sampling. That is, a few samples would lead to a higher loss between hidden layer output and target label in backward propagation after a few epochs.

Therefore, by prioritizing training using samples with relatively higher importance, i.e., the ability to contribute towards building model accuracy, a DL training job can achieve improvement in both training time and test errors [46, 50].

In SGD, gradient $g(x)$ is estimated by sampling from a uniform distribution p where x is a data sample from a minibatch. Importance sampling estimates $g(x)$ using a new data distribution q (such that $q(x) > 0$ whenever $p(x) > 0$) to speed up the process. That is,

$$\mathbb{E}_{p(x)}[g(x)] = \mathbb{E}_{q(x)}[\frac{p(x)}{q(x)}g(x)] \quad (1)$$

It has been proved [11] that the variance of gradient is minimized when Eq. 2 is maintained, i.e., to ensure gradient variance reduction, optimal data distribution $q^*(x)$ should be proportional to sample's gradient norm $|g(x)|$.

$$q^*(x) \propto p(x)|g(x)| \quad (2)$$

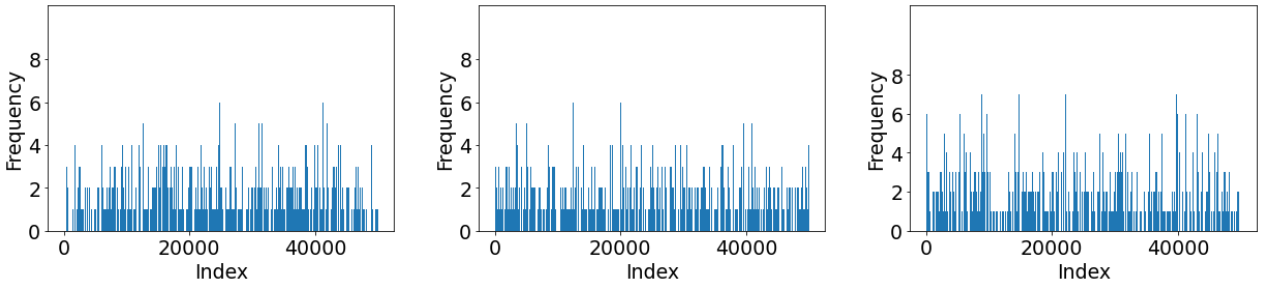
In practice, the feed-forward loss is often used to measure the importance of each data sample as an alternative of gradient.

3 Motivation

3.1 Exploiting Importance Sampling

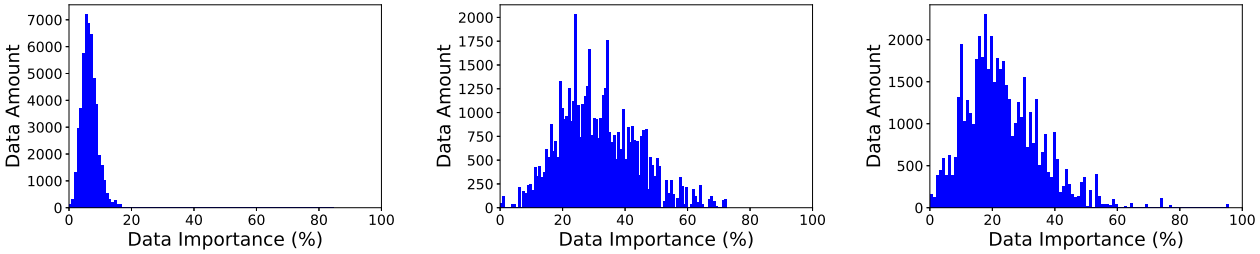
As discussed earlier, the shuffling-based sampling method passes over the entire training dataset in each epoch, making DL training not cache-friendly and failing to make efficient use of faster storage mediums such as main memory or SSDs. However, as observed in Figure 1, there exists ample opportunity to make efficient use of faster storage devices to enhance the performance of DL training. In this regard, importance sampling treats training samples differently and introduces inherent data locality that can be exploited by a caching system to make better use of faster storage mediums.

To better understand the implication of importance sampling on DL training and dataset caching, we analyze importance sampling based training over benchmarking datasets.



(a) Sample access pattern in epoch 1. (b) Sample access pattern in epoch 20. (c) Sample access pattern in epoch 89.

Figure 2: Frequency of samples accessed across different epochs in default single process importance sampling (CIFAR-10).



(a) Epoch 1. (b) Epoch 20. (c) Epoch 89.

Figure 3: Distribution of data importance as the number of epochs increases in single process default importance sampling on the CIFAR-10 dataset. Data importance is the ability of a sample to contribute towards improving the accuracy of the model.

Sample Access Pattern. We first analyze the sample access pattern of importance sampling. We use the CIFAR-10 dataset [54] to train a ResNet-18 DNN model using a loss-based importance sampling algorithm [50] for a single GPU. As shown in Figure 2, 26.5%, 26.6%, and 26.2% of the samples are accessed more than once in epochs 1, 20, and 89, respectively. More importantly, 9.6% of the samples are accessed 3 times or more in epoch 89, indicating a good data locality within a training epoch.

Data Importance Distribution. We further analyze the importance scores of training samples. Unlike standard random sampling, which treats each sample equally, not all samples contribute equally to model training. As shown in Figure 3(a)-(c), in epoch 1, the importance scores of all samples are clustered towards the least-important end of the spectrum; whereas during epochs 20 and 89, more samples become more important. In particular, in epoch 20, around 49.91% of samples have a normalized importance score greater than 30%. This observation further implies that the importance information could be exploited by a priority caching policy to optimize the I/O efficiency of DL training.

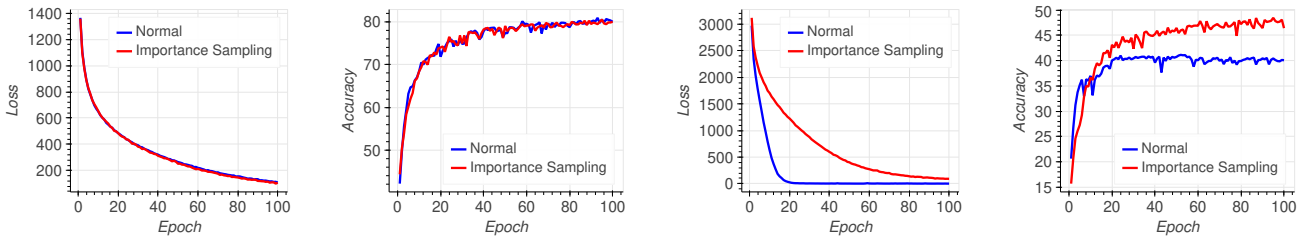
Impact of Importance Sampling on Training. Next, we analyze the impact of importance sampling on training quality. In this test, we use the CIFAR-10 and CIFAR-100 datasets and train a ResNet-18 model using standard random sampling and a loss-based importance sampling method. As shown in Figure 4(a)-(b), we verify that importance sampling incurs negligible impact on the model accuracy for the CIFAR-10 dataset. The CIFAR-100 dataset is much harder to predict than CIFAR-10 due to the larger number of classes present in the dataset. Figure 4(c) shows that importance sampling does not have a drastic loss degradation implying that it has a good learning rate, which further contributes to its improved accuracy. Figure 4(d) shows that importance sampling can achieve better accuracy in under 20 epochs than the accuracy achieved by normal baseline random sampling in 100 epochs. This is because random sampling just shuffles the dataset indices, which does not contribute much towards quickly learning fine-grained details of the dataset.

4 SHADE Design

Our study in §2 sheds light on the potential to enable fundamental data locality for DL training workloads and motivates a new caching system co-designed with the DL training framework. This section presents the challenges and design principles of SHADE, followed by the design detail.

4.1 Challenges

Our goal is to achieve a caching system that can exploit importance sampling to improve the cache efficiency for DLT's I/O workload. One may think that a priority-based caching policy that always prioritizes the most important samples could effectively improve the read hit ratio. However, as shown in Figure 5, a naive priority-based caching policy achieves the same low read hit ratio as standard LRU and LFU. Ideally,



(a) Loss vs. Epoch (CIFAR-10). (b) Accuracy vs. Epoch (CIFAR-10). (c) Loss vs. Epoch (CIFAR-100). (d) Accuracy vs. Epoch (CIFAR-100).

Figure 4: Comparison of loss and accuracy convergence of ResNet-18 model using single process default importance sampling against baseline training on the CIFAR-10 and CIFAR-100 datasets.

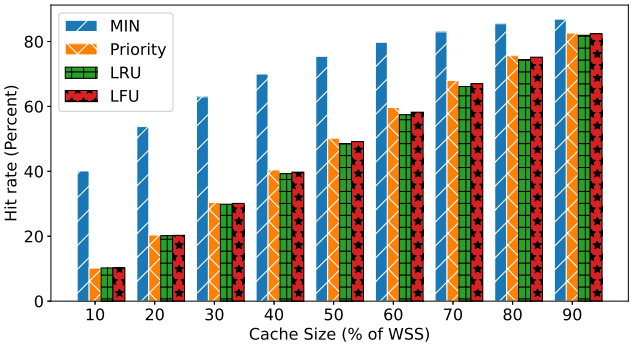


Figure 5: Comparison of different caching policies during ResNet-18 model training over the CIFAR-10 dataset. Working Set Size (WSS) denotes the percentage of cached dataset.

Belady's MIN cache replacement policy [18] achieves an optimal read hit ratio assuming perfect future knowledge: Belady's MIN replaces the item that will be accessed furthest in the future (Figure 5). In the context of DLT, an ideal priority caching policy would accurately capture the priorities (i.e., importance scores) of training samples, resembling the optimal behavior of the offline MIN. To make it even better, the policy could take advantage of the importance information to prefetch important samples into the cache. This way, the new policy can potentially outperform Belady's MIN when incorporating prefetching [87].

The key insight of this paper is that DLT treats different training samples differently and that the priorities of I/O accesses are inherently predictable, therefore exposing interesting exploitation opportunities to fundamentally improve the I/O efficiency. However, it also poses non-trivial challenges to effectively translate the potential exploitation opportunities to the I/O efficiency improvement.

First, default importance sampling (importance sampling considered in prior works) assigns per-minibatch scores, which are too coarse-grained and inaccurate. That is, all samples of a single minibatch are, by default, assigned the same importance scores. This creates ambiguity, which leads to inaccurate estimation of the per-sample importance and, thus, loss of cache efficiency. Ideally, we would want an importance score that precisely tells us the relative importance that each sample carries within a minibatch.

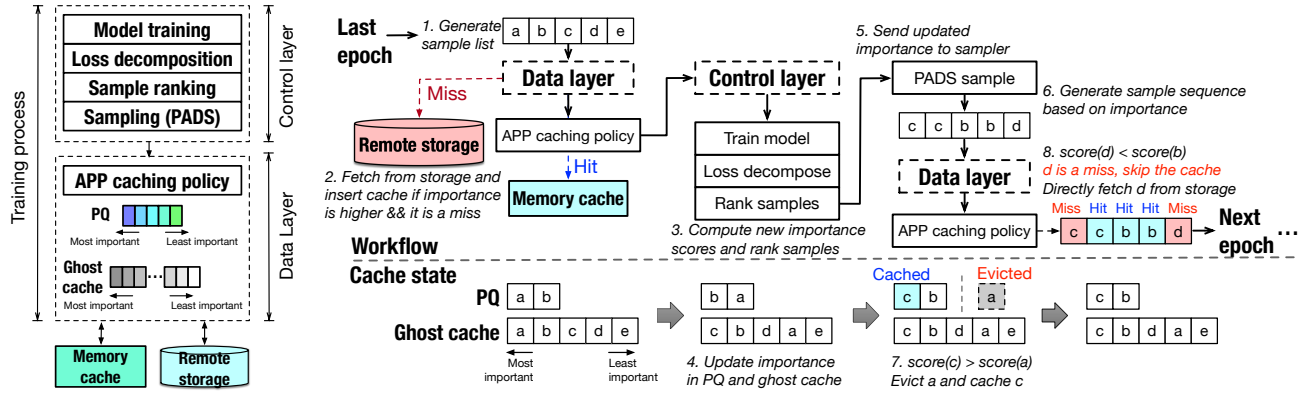
Second, even if important samples are identified properly, aggressively feeding the DL model with repetitive samples might make training model biased. Thus, it is necessary to ensure that the accuracy is not compromised while trying to increase the hit rate of samples.

Third, importance scores are constantly changing and may get stale quickly. The same sample in a later minibatch may contribute differently toward the model than it did in an earlier minibatch. Thus, capturing the most up-to-date importance score information is imperative to make informed caching decisions.

In the next section, we discuss how we use four novel techniques to address each of these challenges.

4.2 SHADE Overview

SHADE consists of two main components—the control layer and the data layer. The control layer provides the data layer with the list of samples needed for training. For the first iteration, the data layer fetches samples from a remote storage and populates the cache with the samples that are to be accessed first. During training, the control layer finds the importance (loss decomposition + ranking) associated with the samples and the priority queue (PQ) and ghost cache tracking the importance of samples in the data layer are updated. Based on the newer importance, a *sampler* in the control layer prepares a samples list with associated repetitions information. When the data layer receives the list of samples, it checks whether it is beneficial to cache a newer item instead of evicting a cached prior sample. Let's suppose the sample being accessed has higher importance than the min_sample (sample having lowest importance in the current cache). In this case, the min_sample is evicted, and the current sample is cached using our new Adaptive Priority-aware Prediction (APP) cache policy. This process is repeated throughout the entire DL training. As SHADE keeps the most important samples in the distributed cache and repeatedly uses these hard-to-learn samples for training, it can ensure improved rate of accuracy and a good cache hit ratio. Figure 6 shows the architecture of SHADE along with the components and interactions therein.



(a) SHADE's logical architecture. (b) SHADE's workflow.

Figure 6: SHADE architecture overview. (b) In illustration of how SHADE's components interact in a single epoch.

4.2.1 Control Layer

The SHADE control layer performs two main functions. (1) It calculates the importance scores associated with data samples, and (2) it samples the data for different training processes. The importance scores are then transferred to the data layer in real time for making prefetching and caching decisions.

The SHADE control layer features three techniques to find accurate, fine-grained importance scores for each data sample. The first technique finds out the importance of samples in per-sample granularity (i.e., fine-grained). The second technique uses fine-grained importance and ranks the samples to make them suitable for priority-based caching. Finally, the third technique uses rank-based importance to build a list of important samples with repetitions to be used for training that will increase the read hit ratio and maintain a good learning curve.

Loss Decomposition. In default importance sampling, the forward training loss is calculated for minibatches, and this forward loss is then assigned as the importance score for all the samples in the minibatch [61]. As a result, the default importance sampling method calculates the ability of a minibatch to contribute towards improving the overall accuracy of the model instead of the data samples themselves. However, as expected, not all the samples of a minibatch contribute equally to the accuracy improvement of the model. Therefore, we need the sample-level loss information, i.e., the loss of individual samples of a minibatch, to calculate the importance score of each data sample.

To address the first challenge concerning the coarse-grained importance scores at the minibatch granularity, SHADE uses both the sample-level and minibatch-level cross entropy loss information to decompose the coarse-grained importance scores into per-sample scores. The cross entropy for each sample denotes the uncertainty with which the model could predict the class label for a sample. Measuring the uncertainty helps SHADE to detect the importance of a sample.

Assume a minibatch has S data samples, and T represents the number of class labels. This constructs an output layer for the DNN model with a matrix that has a dimension of $S \times T$. Each row of the matrix encodes the raw likelihood or logits of a sample for each of the T-class labels. To capture the contribution of each data sample, SHADE decomposes the loss function and calculates the loss corresponding to each sample in the minibatch. SHADE decomposes the loss function using two steps. In the first step, SHADE calculates the categorical-cross entropy for each sample in the minibatch. The categorical-cross entropy for each sample, E_{sample}, is defined in Eq. 3:

$$E_{sample} = -\sum_{i=1}^{T} T_i \log S_i, \tag{3}$$

where T_i represents the hot-encoded truth label for a given sample under class i, and S_i denotes the softmax probability for a sample in a minibatch for the class i.

S_i is calculated using Eq. 4:

$$S_i = \frac{e^{r_i}}{\sum_{j=1}^{T} e^{r_j}}, \tag{4}$$

where r_i denotes the raw likelihood of a sample for class i, and the denominator is a normalization term. SHADE uses a softmax normalization over a standard normalization method for two reasons. (i) This method can effectively identify small and large variations in raw logit values and thus assign the importance scores accordingly. (ii) The raw logit values can be negative, so taking exponents ensures that we always end up with a positive value. SHADE uses the per-sample-based entropy loss for finding and feeding the model with the most important samples.

The second step of the loss decomposition involves calculating the minibatch importance necessary for adjusting the model weights. As softmax is continuously differentiable, it is possible to calculate the derivative of the cost function with respect to every weight of a DNN model. SHADE uses all the per-sample-based entropy losses found from the first stage (Eq. 3) for calculating a mean entropy loss according to Eq. 5:

$$E_{batch} = \frac{\sum_{k=1}^{S} E_{sample_k}}{S} \quad (5)$$

A higher entropy for a sample means that the model generates multiple predictions for a single sample out of the several T different possibilities, i.e., the model faces more difficulties in generating a single accurate prediction for that sample. Correspondingly, a lower entropy for a single sample signifies that the model can generate a single prediction for it with high enough accuracy. Thus, a lower entropy value for a sample means that the sample is not highly important in increasing the accuracy of the model in later epochs, and a higher entropy value signifies the opposite. The reason is straightforward: samples that the model has already learned cannot help much in increasing the accuracy of the model in later epochs, and only by learning the harder samples can the DLT job improve the accuracy. Ideally, an entropy value of zero means that the difference between the predicted and ground-truth label is an absolute zero and that the sample is accurately learned. In practice, however, the entropy cannot reach zero as there are no useful models that have 100% accuracy.

Our goal is to prioritize samples that have higher entropy during model training so that the model can learn these hard-to-learn samples better. Consequently, the loss decomposition method enables SHADE to capture hard-to-learn samples from a minibatch without extra transformation of the raw data.

Rank-based Importance Score. Even though the per-sample entropy score provides a simple tool for quantifying the importance of different samples, it does not tell how much different samples contribute to the accuracy of the model when sampled together in a single minibatch. The relative rankings allow SHADE to prioritize the most important set of samples from each minibatch and, thus, the entire training dataset. To identify the *relative* contribution of a sample in a minibatch, we derive a log-based ranking method shown in Eq. 6:

$$rank_i = \log(\sum_{k=1,k\neq i}^{B} I(l_i > l_k) + b_0) \quad (6)$$

The rank-based importance score for the i^{th} sample in a minibatch with B samples is denoted by $rank_i$. l_i and l_k denote the entropy loss for the i^{th} and k^{th} sample, respectively. b_0 is a bias term used for fixing the range of ranks in the log scale. I is an identity function that returns 1 when the condition $l_i > l_k$ is true, and 0 otherwise. For each k item in a batch B, this condition helps to place each sample in the proper rank in a minibatch. A sample having a higher loss gets a higher rank.

Consider the following example. Assume two minibatches, $B1$ and $B2$, contain samples <4, 5, 6> and <7, 8, 9>, respectively. Assume the samples in $B1$ have entropy scores of <0.3, 0.5, 0.4> and samples in $B2$ have <0.6, 1.2, 0.8>, respectively. These entropy values are raw values, which will be problematic when comparing the sample importance across minibatches. For example, a priority-queue-based cache would rank sample 5 from $B1$ in the lower half globally when sorting all samples from both the two minibatches, even though sample 5 is the most important one in $B1$. Sorting these samples by the entropy scores would give us a relative rank with respect to each of the two minibatches $B1$ and $B2$, meaning sample 5 and 8 are the most important in $B1$ and $B2$, respectively.

To get the accurate changes in importance score, i.e., whether the importance score is increasing, staying unchanged, or decreasing, SHADE uses the log scale. In this case, the relative importance scores remain in the same range and can be used for priority differentiation in a priority queue data structure. Relative scores are desirable for three reasons.

First, our method guarantees that the per-sample-based importance scores from different minibatches and epochs are in the same range in order to precisely differentiate their priorities in the cache. Different samples from different minibatches would share the same rank in our method if they contributed the same proportion in improving the accuracy of the model when grouped in their corresponding minibatches. Whereas in the default importance sampling method, all samples from the same minibatch are assigned the same importance, resulting in at most one minibatch that will have the highest importance score, r_{max}. This is erroneous as all samples in the same minibatch do not contribute equally to improving the accuracy. When the number of minibatches is more than the minibatch size, SHADE is guaranteed to capture more important samples than the default method, and this, in turn, helps in improving the read hit ratio of the cache. Specifically, if the DLT job is configured to train on N samples with a minibatch size of B, where N is significantly larger than B (which is the common case in DLT), then SHADE can effectively identify (N/B) important samples in one epoch. In contrast, the default method can only capture B important samples.

Second, although models are constantly being updated during the training, training against harder-to-learn samples may help mitigate the high volatility of the accuracy rate, therefore leading to smooth model training.

Third, the relative ranks make it easy to predict the data importance online: the top x% important data in a minibatch is guaranteed to be within the set of the top x% of the whole dataset according to our defined importance score. Based on this property, SHADE effectively offers an implicit prefetching mechanism, which will be described in Algorithm 1.

Priority-based Adaptive Data Sampling (*PADS*). At the end of an epoch, when SHADE has calculated the (rank-based) importance of samples, the SHADE sampler sends the data layer the list of sample indices that should be used for training.

However, instead of naive random shuffling, the SHADE sampler first prioritizes the samples that contributed the most to the accuracy of the model by constructing a multinomial probability distribution of the data samples, as there are many possible outcomes/selections of the dataset. Based on the generated distribution, the sampler builds a list of important samples for shuffling and sharding across different DLT processes. SHADE seamlessly combines prefetching and caching:

based on the updated importance scores and the list of samples provided by the sampler, SHADE data layer prefetches the most important samples to the distributed in-memory cache. The sampler provides the data layer with a list of repetitive samples, which helps the data layer automatically prefetch important samples in real-time. The design strikes a balance between the cache efficiency (read hit ratio) and model accuracy. SHADE keeps track of the loss convergence of the model in real-time to decide the number of repetitive sample accesses in order to boost the hit rate without sacrificing the accuracy of the model. To do so, on noticing a steep decline in the loss convergence curve or a stagnant accuracy curve, SHADE intentionally shuffles the important samples to avoid training against a small subset of the most important samples. This way, the system is able to mitigate aggressive importance sampling, which minimizes training biases.

SHADE's *PADS* policy plays a crucial role in increasing the hit rate of a limited-sized cache and, in certain cases, can even outperform offline MIN. Consider the following example. Assume we have ten samples <1, 2, 3, 4, 5, 6, 7, 8, 9, 10> during training. The samples have no repetitions, as random sampling does not consider the importance of the samples. Assume each training process trains on five samples, and the cache holds two samples. Assume the sampler provides samples [1, 3, 5, 6, 8] randomly for training. Then, offline Belady's MIN will put any of the two samples that will be used by the training process. Assume it puts [1, 3] in the cache. In this case, the hit rate will be only 40% <hits 1, 3>.

In the case for SHADE, *PADS* would create a samples list with repetitions based on importance. Assume the samples are [4, 7, 3, 5, 3, 5, 2, 2, 1, 10] and samples 3, 5, and 2 are the most important. Then *PADS* would provide the training process with the samples <3, 3, 5, 5, 2> from the list of samples so that <3, 5> can be cached. The hit rate is now 80% <hits 3 3 5 5>. Even if we consider repetitions in the case of Belady's MIN, it is bounded by the access pattern of the sample list provided by the random sampler. For example, if the sampler provides <4, 5, 5, 1, 10> to the training process, on knowing the future, Belady's MIN can, at most, cache 5 (as it is needed twice) along with another sample. Assume it caches <5, 10>, so the hit rate will be only 60% <hits 5, 5, 10>.

SHADE's sampling method is fully decentralized and does not require a centralized server to coordinate. Decentralized sampling means that each training process derives the importance scores of the samples independently based on its own local model training.

4.2.2 Data Layer

The SHADE data layer provides mechanisms and policies for cache eviction and prefetching.

A challenge regarding sample caching is that the importance scores are constantly changing, even within an epoch, as one data sample can be accessed in multiple minibatches. In order to address this challenge, we design a new cache policy called *Adaptive Priority-Aware Prediction (APP)*, a dynamic policy that updates the importance score of a data sample as soon as the importance score changes.

SHADE's data layer consists of two components: *an indexer* and *a pooled in-memory cache* that spans multiple key-value storage (KVS) servers. The index uses two heap-based priority queues (PQ and ghost cache) for tracking the samples along with their associated rank and access frequency for each training process.

The data layer introduces index numbering for each individual data sample. Index numbering enables the control layer to assign importance score at the sample granularity. Once the control layer calculates and assigns importance scores for the data samples, the indexer inserts the data sample index numbers, but not the actual sample data, into the PQ (priority queue for the current state of the cache). During data loading, the cache will use the importance scores provided by the indexer to make informed prefetching and eviction decisions. The data layer also performs serialization and deserialization when inserting and fetching image samples to and from the cache. The APP caching policy is shown in Algorithm 1.

The PQ and the ghost cache are sorted by the importance scores. PQ keeps track of the metadata state of currently cached samples in the cache, while the ghost cache tracks all the metadata state of the samples that have ever been cached (including those that have been evicted). The ghost cache entries do not store the actual data samples, but rather store a mapping between the data sample ID (the sample index) and the metadata tuple record of <ir, af>, where ir is the importance score and af is the access frequency.

During training, a cached sample might lose its importance if it is well-learned; that is, it might lose its priority in the cache and gets evicted. At the same time, another sample that has been evicted previously might turn out to be important and therefore gets inserted into the cache. The ghost cache helps decide whether a previously-evicted sample could be brought back into the cache.

When the cache is full, and the data sample to be processed is a miss, SHADE checks ghost cache for the previous importance score of the data sample (Algorithm 1 line 12): if this data sample had been previously evicted out of the cache, it should be included in the ghost cache. If the most recent importance score of this data sample is greater than or equal to that of the cached sample that has the smallest importance score, the currently-cached least-important sample (min_sample) is evicted from the data cache as well as from PQ. After the eviction, the data sample that is to be processed (and was previously evicted) is inserted in the cache (line 18).

In summary, by comparing the current importance scores of data samples already in the cache and that of the most recent importance score of the current data sample being processed, SHADE's data layer predicts and maximizes the likelihood of a sample being reused in the cache in the future.

Algorithm 1: Adaptive Priority-aware Prediction (APP) Cache.

```
Input and Initialization:
PQ: Priority queue for currently-cached samples, ghost_cache: Priority queue for all previously-trained samples
for epoch in total_epochs do
    for s in sample_dataset do
        v = score(s) # Calculate importance score based on Eq 3 and Eq 6
        ghost_cache.set(s,v) # Insert/update in ghost_cache
        if cache_hit then
            cache.get(s)
        else if cache_miss and cache_not_full then
            cache.insert(s) and PQ.set(s,v)
        else
            if cache_miss and ghost_cache.exist(s) then
                x = ghost_cache.get(s).score # Get the most recent score of data sample
                min_sample,min_score = PQ.min() # Find the sample with the minimal score in the cache
                # Check if the sample to be processed is more important than the least important one stored in the cache
                if x ≥ min_score then
                    cache.evict(PQ.pop(min_sample)) # Evict the least important sample from cache
                    cache.insert(s) and PQ.set(s,v) # Insert this sample into the cache and PQ
                else
                    read_from_storage(s) # The data is less important than any samples currently cached, skip caching
            else
                read_from_storage(s) # Evicting a known sample for an unknown one may not be beneficial, skip caching
```

5 Implementation

SHADE is implemented in PyTorch 1.7. PyTorch has three main components: `Dataset`, `Sampler`, and `DataLoader`. Dataset class provides the image dataset access points and exposes a *__get_item__* method that fetches a sample along with its target label for a given index. `Sampler` provides subsets of samples of the dataset to the training processes in random permutations. `DataLoader` uses the information provided by the sampler to load the samples in minibatches with the help of worker processes. In SHADE, we implement a new class by inheriting the PyTorch `Dataset` class. The `ShadeDataset` class has functionalities to combine the samples and their corresponding class labels so that `Dataloader` can fetch data samples easily from the remote storage.

We extend the `DistributedSampler` class to prepare the `ShadeSampler` class that has the main logic of the SHADE's *PADS* policy. It has APIs for communicating with the training processes to receive the calculated per-sample entropy value for each minibatch. At the end of each epoch, `ShadeSampler` forwards the important samples to the training processes.

We introduce the logic for the data layer by overriding the *__get_item__* and *__len__* method in the `ShadeDataset` class. The *__len__* method returns the total length of the `ShadeDataset`, and the *__get_item__* method exposes the indices associated with the data samples that enables the client layer to find importance in per-sample granularity. In addition, the *__get_item__* method is connected to the in-memory pooled cache to make decisions on caching and eviction based on the heap-based PQs of the data layer using the *APP* cache policy. For the in-memory pooled cache, we use Redis [7].

We have implemented an analysis framework atop the setup to facilitate experimentation and statistics collection. The framework takes as input the configurations of the experiment, which include paths of the training dataset, master's address:port, number of training nodes, number of GPU devices, number of epochs to run the test, the batch size, and the DNN model to be trained. The framework then sets up the environment accordingly. It collects GPU-related statistics using nvidia-smi [66] and I/O-related statistics using sar [73]. SHADE's implementation is system agnostic – DL practitioners do not need to write new code to use SHADE in their systems. SHADE does not use any extra system-level resources compared to a normal DL training with local/global caching.

6 Evaluation

6.1 Experimental Setup

Our study covers distributed training with multiple GPUs and a remote storage deployed on Chameleon Cloud [51]. Several recent works [26, 27, 52] have used Chameleon Cloud for conducting high-performance experiments, making it a repre-

sentative testing platform. Our method is compared against baseline distributed training in PyTorch—one of the most popular frameworks for deep learning [68]. Although caching policies are not publicly available in PyTorch, we have built an LRU caching policy on top of PyTorch to ensure a thorough evaluation of our proposed storage caching policy. In addition, we have evaluated SHADE against importance sampling with six different caching policies to perform a robust ablation study. For our analysis, an HDD-based NFS Server [63] is used as remote storage. For training, we have used eight NVIDIA P100 GPUs (PCIe with 16 GB memory) spread across four nodes. All the GPU nodes and storage nodes are connected via a 10 Gbps interconnect.

Our experiments primarily use the ImageNet-1K dataset [32], which contains ∼1.2 million images with a total size of 138 GB spanning 1,000 object classes. We also use CIFAR-10 [54]. We conduct our study on four representative computer vision (CV) distributed DL models, namely, Alexnet [55], ResNet-18, ResNet-50 [38], and VggNet [76]. The setup we use to evaluate our system is representative of CV distributed DL training, which has been used to evaluate prior research works [20, 56, 69, 84]. In the following, the reported percentages for cache size indicate the portion of the dataset that had been cached.

6.2 Cache Hit Ratio

In this set of experiments, we evaluate the performance of our APP caching policy against several other policies. This will help explain the extent to which SHADE is able to make better utilization of the limited cache space. The `baseline` uses PyTorch's default random sampling and LRU caching policy for eviction. We also implement and evaluate the offline Belady's MIN policy. In addition, we evaluate six SHADE policy variants based on importance sampling:

1. Priority-based LFU policy (`sh_pqlfu`), which evicts the samples with the least importance score based on a hybrid priority that combines the batch-forward loss (i.e., coarse-grained score) and sample access frequency. If the forward loss is the same, then eviction decision is made based on the access frequency of samples.
2. Priority-based policy (`sh_pq`), which uses the batch-based, goarse-grained forward loss as the importance score.
3. LRU (`sh_lru`), which uses the coarse-grained forward loss but evicts samples based on the recency of the items and not the importance scores.
4. LFU (`sh_lfu`), which uses the coarse-grained forward loss but evicts the least-frequently-used sample.
5. Random (`sh_rand`), which performs random evictions.
6. APP (`sh_app`), which makes eviction decisions using our APP policy but does not use loss decomposition, rank-based importance score, and *PADS* sampling.

Figure 7 shows the the average read hit ratios when training the three DL models (Alexnet, ResNet-50, VggNet) over the CIFAR-10 dataset under the studied policies with different cache sizes.

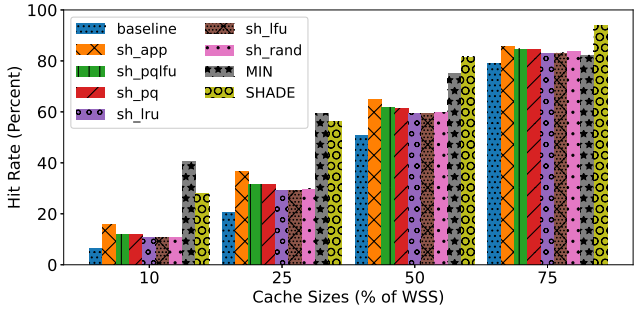


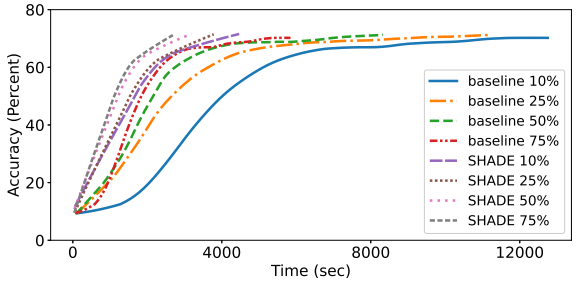
Figure 7: Comparison of the read hit ratio of various caching policies and cache sizes. The `sh_` prefix denotes a baseline version of `SHADE` that uses the coarse-grained importance. `SHADE` denotes our contribution, SHADE, with all techniques enabled. `WSS` denotes working set size.

We observe that the margin by which SHADE performs better than policy 1–6 increases as the cache size becomes smaller. When only 10% of the WSS is cached, SHADE, with all techniques incorporated, shows a 4.5× higher read hit rate than the baseline LRU. SHADE without the importance derivation techniques (`sh_app`) can achieve 2.67× and SHADE without the APP cache policy (all the SHADE `sh_` baselines except `sh_app`) can achieve around 1.94× higher hit rates than the baseline. The reason for the improved hit rate is that our techniques are able to predict which samples the training processes would need in the future for better accuracy, and hence it approaches the hit rates of the optimal MIN and even outperforms MIN in some cases (WSS 50% and 75%). This is because MIN's knowledge about samples' future access pattern relies on the sampler. However, SHADE manipulates the sampler to create the desired future sample access pattern, which will benefit the DLT job the most in terms of both training duration and accuracy. This sample access pattern comprising multiple hard-to-learn samples enables precise I/O prediction and maximizes the likelihood of a sample being reused in the cache in the future. By ensuring a higher hit rate with limited available cache space, SHADE holds effectively more data in the limited cache space, therefore achieving higher DLT efficiency.

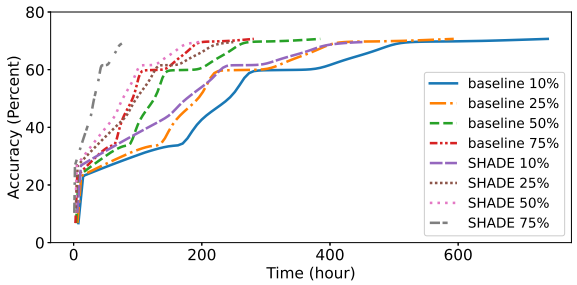
6.3 Accuracy vs. Time

In our next tests, we evaluate how model accuracy changes over time for SHADE when compared to the baseline at different WSS. This shows how quickly SHADE is able to train a model and increase the accuracy using a very small cache even when the baseline has the advantage of using more cache space than SHADE.

Figure 8 shows that SHADE has a better accuracy improvement rate compared to baseline policy. For example, SHADE can achieve up to 3× faster accuracy convergence compared

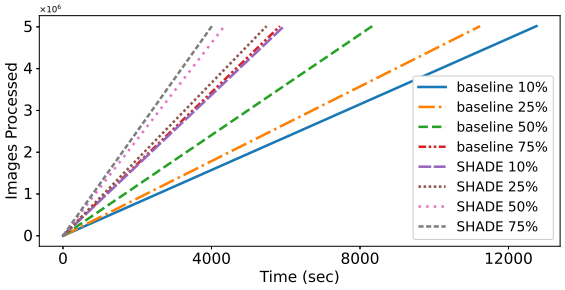


(a) Alexnet

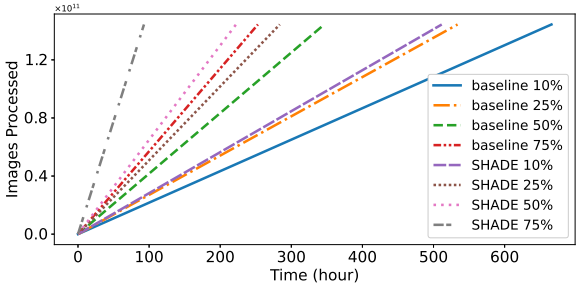


(b) ResNet-50

Figure 8: Accuracy improvement rate of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).



(a) Alexnet



(b) ResNet-50

Figure 9: Throughput of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).

to baseline storing 10% of the dataset in the cache when being trained on the Alexnet model. Fine-grained relative importance of samples helps SHADE detect the most important, i.e., hard-to-learn samples, train more on them and thus improve the accuracy quickly to reach convergence. Again when being trained on the ResNet-50 model, SHADE continuously maintains a better accuracy improvement rate compared to baseline at similar WSS. SHADE can reach accuracy convergence 3.3× faster compared to baseline at 75% WSS. The accuracy improvement rate of the baseline with a larger cache is not always better in Figure 8(a) because the baseline uses random sampling. Random sampling places equal emphasis on all of the samples and hence cannot improve the accuracy quickly by training more on the hard-to-learn samples. The improvement in accuracy vs. time curve for the baseline comes only from caching more data. Hence, our results show that even if a larger portion of the dataset is cached, naively caching data items without proper techniques to exploit data locality can not guarantee improved performance.

6.4 Throughput

In our next experiment, we evaluate the throughput of SHADE, which help demonstrate the superiority of SHADE in processing data while storing a limited portion of the dataset. Figure 9(a) shows that SHADE while caching just 10% of the dataset has around 2.3× better throughput compared to baseline policy caching 10% of the dataset. The baseline matches the throughput performance of SHADE only when the baseline is caching 7.5× more data compared to SHADE. In the experiment with ResNet-50, shown in Figure 9(b), we observe that SHADE at 75% WSS has 2.7× higher throughput compared to baseline at similar WSS. Even at lower WSS, SHADE can achieve higher throughput compared to baseline at higher WSS. For example, SHADE at 50% WSS has a 1.14× higher throughput than baseline at 75% WSS. The improvement in the ability to process more images is due to the ability of SHADE to exploit data locality with APP cache policy. Although baseline at 75% WSS has a slightly higher throughput compared to SHADE at 25% WSS, it is unable to get a better accuracy improvement rate seen in Figure 8(b). This is because SHADE can exploit data locality and has techniques to train on important samples which ensures a better accuracy improvement rate.

6.5 Minibatch Load Time

In our next test, we evaluate the performance gain observed in minibatch load time. Consistency in minibatch load time is important so that all the training processes can remain coordinated. It also shows the effectiveness that a caching policy has in exploiting data locality. Figure 10 shows the average minibatch load time of the GPUs during the course of training with the vertical lines showing the standard deviation of the load time within a single epoch.

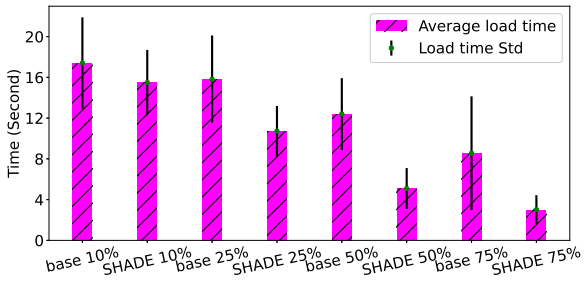


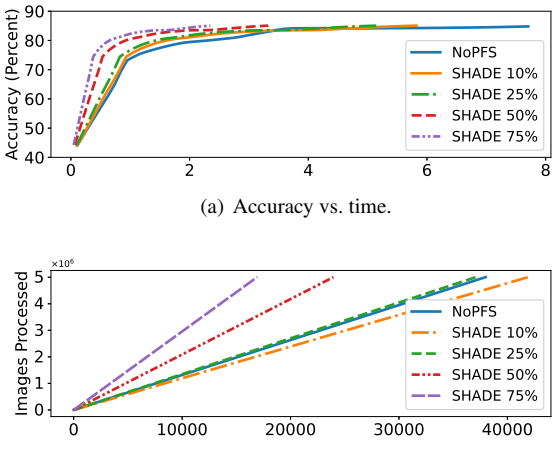
Figure 10: GPUs' minibatch load time when training ResNet-50. Percentages denote the amount of cached dataset.

As we can see in the figure, SHADE can achieve a lower mean load time compared to baseline at similar and higher WSS. The baseline at 50% and 75% WSS has 2.5× and 1.7× higher minibatch load time compared to SHADE at 50% WSS. Moreover, SHADE can maintain a small standard deviation in minibatch load time. Ideally, we would expect the standard deviation in minibatch load times to be low if larger portions of the dataset get cached because of using a higher portion of the fast RAM storage. However, it is not the case for baseline, even if it caches larger portions of the dataset in the cache. Average minibatch load time is highly variant for baseline caching 50% and 75% of the dataset. The baseline at 75% WSS has a 3.9× higher standard deviation in minibatch load time compared to SHADE at 75%.

6.6 End-to-End System Comparison

In our last set of experiments, we compare the performance of SHADE against NoPFS [34], a state-of-the-art storage system for improving the I/Os of DLT workloads. NoPFS exploits the seeds that generate the random access pattern when performing SGD-based DLT to predict when and where a training sample will be accessed. Similar to our baseline, NoPFS uses random sampling of indices. The difference lies in that NoPFS does not consider importance and uses *Clairvoyance* (i.e., seeds that generate random access patterns) to approximate "future distances" of Belady's MIN [18]. SHADE considers fine-grained importance of samples and uses *PADS* policy to prioritize samples for training.

For fair comparison, we keep the experimental setup and training parameters the same for both SHADE and NoPFS while training on the CIFAR-10 dataset. Figure 11 shows that NoPFS incurs a 4.5× and 2.4× increase in training time to reach accuracy convergence compared to SHADE at 75% and 50% WSS, respectively. At the same time, SHADE has 2.2× and 1.6× better throughput compared to NoPFS when working at 75% and 50% WSS, respectively. SHADE can still attain accuracy convergence faster even at 10% WSS. SHADE performs better than NoPFS as it adopts a prefetching system that aims to approximate Belady's MIN; it is bounded by the sample access pattern provided by the sampling policy and



(a) Accuracy vs. time.

(b) Throughput.

Figure 11: Comparison of SHADE and NoPFS [34]. Percentage denotes the percentage of cached dataset.

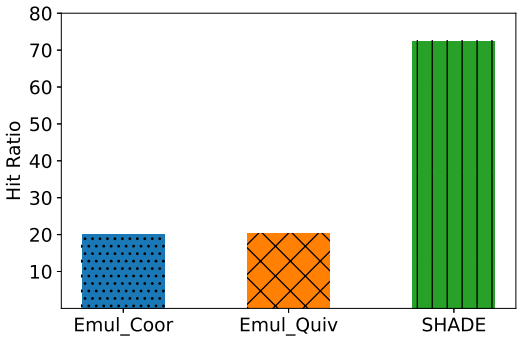


Figure 12: Comparison of the read hit ratio of different caching policies at 20% WSS of CIFAR-10.

hence prioritizes all samples equally. As a result, it takes more time to reach accuracy convergence compared to SHADE, which trains more on hard-to-learn samples to increase the accuracy improvement rate faster.

We further compare the cache hit ratios of SHADE with state-of-the-art DLT caching policies. we configure a small cache space of 20% of the WSS over the CIFAR-10 dataset using the ResNet-18 model against emulated caching policies including CoorDL [65] and Quiver [56]. To understand the impact of these techniques in importance-aware training, we use a loss-based importance sampling technique [50] inspired by Mercury [84]. For emulating CoorDL and Quiver, we create our own implementations of the core techniques of CoorDL and Quiver, which we name as `Emul_Coor` and `Emul_Quiv`, respectively. Both `Emul_Coor` and `Emul_Quiv` use a KVS as a cache similar to SHADE. `Emul_Coor` ensures that no items are ever evicted from the cache once these are inserted in the cache. In the case of `Emul_Quiv`, we implement the substitutability technique, which replaces a missed sample with a sample already in the cache to avoid memory thrashing.

Figure 12 shows that both `Emul_Coor` and `Emul_Quiv` can only extend their utilization up to the size of the cache

(∼ 20%) because these caching policies are not importance-aware and therefore cannot exploit the data locality of the important samples perfectly in importance-aware training. Both of these policies populate the cache using random samples and hence are unable to get a good hit rate by exploiting the repetitions among samples that occur throughout training. On the other hand, as SHADE can manipulate the sampling process (*PADS* policy) and keep repeated samples in the cache, it can achieve a higher hit ratio (72.5%) and thus outperforms both `Emul_Quiv` and `Emul_Coor` by 3.6×.

7 Related Work

Several recent works have explored the use of importance sampling for optimizing the system efficiency of DL workloads [23, 84]. iCACHE [23] is an importance-sampling-informed DLT cache. Although this approach uses a form of fine-grained importance similar to SHADE, it does not have a rank-based relative score scheme and SHADE's *PADS* sampling approach, due to which it may suffer from a lower cache hit ratio than SHADE. Moreover, in case of a cache miss, iCACHE uses substitutability, which may impact the training accuracy convergence.

Mercury [84] improves DL training efficiency by exploiting the important samples. Mercury is not an I/O cache, and therefore, unlike SHADE, it does not handle data replacement and eviction.

CoorDL [65] analyzes the data retrieval process in PyTorch and proposes a MinIO cache, which populates the cache with a random set of data items from the first epoch, and keeps these items in the cache during the training with no item being ever evicted. However, as shown in Figure 12, simply caching random samples does not provide the expected performance gain.

A body of work is focused on optimizing the I/O components of DL applications. NoPFS [20] adopts a prefetching approach that uses hardware level configurations to take caching decisions based on a sample access pattern obtained from trying to approximate Belady's MIN. However, in common online training like hyperparameter tuning experiments [60] with different random seeds, such sample access patterns change constantly and hence are not readily available. We address this constant change in sample access pattern through our dynamic cache management policy without depending on hardware configurations for boosting our performance.

Hoard [69], Quiver [56], and FanStore [86] explore the idea of adding a global caching layer to the GPU cluster for improving the training performance of DL workloads. DeepIO [88] proposes an entropy-aware mechanism for determining next minibatches but it does not offer any cache eviction policies and suffers from lack of dataset randomization. DIESEL [80] is a comprehensive storage solution that supports key-value-based metadata service, task-level caching, and chunk-based shuffling. However, these works do not focus on how to enable fundamental data locality for DLT jobs. SHADE, on the other hand, exploits importance sampling to enable data locality for DLT jobs.

8 Conclusion

The I/O pipeline is a major bottleneck in distributed DLT when data is read from a remote storage. To address this bottleneck, ad hoc solutions such as using faster local storage devices (e.g., SSDs) had been employed. However, those ad hoc solutions cannot fundamentally address the I/O efficiency of DLT workloads. Although caching is possible for DLT, naively caching redundant samples does not provide any benefits. SHADE realizes a DLT-aware caching policy, which takes advantage of the fine-grained importance scores of data samples in order to enable a high level of data locality, and therefore, fundamental cacheability for DLT jobs. Our evaluation demonstrates that SHADE improves the read hit ratio of a small memory cache (of only 10% of the WSS of the dataset) by up to 4.5× compared to traditional, non-DLT-aware caching policies, thus significantly improving the DLT performance.

9 Acknowledgments

We thank our shepherd, Raju Rangaswami, and our anonymous reviewers for their detailed feedback and valuable suggestions. This work is sponsored in part by the NSF under the grants: CSR-2106634, CCF-1919113/1919075, CNS-2045680, OAC-2004751, and OAC-2106446. Results presented in this paper were obtained using the Chameleon testbed supported by the NSF.

References

[1] Amazon EC2 Spot Instances. Run Fault Tolerant workloads for up to 90% off. https://aws.amazon.com/ec2/spot/.

[2] Amazon S3. https://aws.amazon.com/pm/serv-s3/.

[3] CSCS. 2021. Piz Daint. https://www.cscs.ch/computers/piz-daint/.

[4] Global Conversational AI Market Report 2021. https://tinyurl.com/Global-AI-Market-Report-2021.

[5] Google Cloud Storage. https://cloud.google.com/storage.

[6] Memcached. https://memcached.org/.

[7] Redis. https://redis.io/.

[8] RIKEN Center for Computational Science. 2021. About Fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/.

[9] Use Spot VMs with Batch. https://learn.microsoft.com/en-us/azure/batch/batch-spot-vms.

[10] Franklin Abodo, Robert Rittmuller, Brian Sumner, and Andrew Berthaume. Detecting work zones in shrp 2 nds videos using deep learning based computer vision. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 679–686. IEEE, 2018.

[11] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[12] Muhammad Aqib, Rashid Mehmood, Aiiad Albeshri, and Ahmed Alzahrani. Disaster management in smart cities by forecasting traffic plan using deep learning and gpus. In *International Conference on Smart Cities, Infrastructure, Technologies and Applications*, pages 139–154. Springer, 2017.

[13] Jorge F Arinez, Qing Chang, Robert X Gao, Chengying Xu, and Jianjing Zhang. Artificial intelligence in advanced manufacturing: Current status and future outlook. *Journal of Manufacturing Science and Engineering*, 142(11), 2020.

[14] 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.

[15] Stephen Balaban. Deep learning and face recognition: the state of the art. *Biometric and surveillance technology for human and activity identification XII*, 9457:68–75, 2015.

[16] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[17] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 626–635. IEEE, 2020.

[18] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[19] Sweta Bhattacharya, Siva Rama Krishnan Somayaji, Thippa Reddy Gadekallu, Mamoun Alazab, and Praveen Kumar Reddy Maddikunta. A review on deep learning for future smart cities. *Internet Technology Letters*, page e187, 2020.

[20] Roman Böhringer, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. *arXiv preprint arXiv:2101.08734*, 2021.

[21] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[22] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347, 2014.

[23] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Sun Xian-He, and Gang Chen. icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training. In *2023 IEEE International Symposium on High-Performance Computer Architecture*, 2023.

[24] Xue-Wen Chen and Xiaotong Lin. Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525, 2014.

[25] Eunsuk Chong, Chulwoo Han, and Frank C Park. Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies. *Expert Systems with Applications*, 83:187–205, 2017.

[26] Joon Yee Chuah. Machine learning gpu power measurement on chameleon cloud. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 181–181, 2017.

[27] Joaquin Chung, Zhengchun Liu, Rajkumar Kettimuthu, and Ian Foster. Elastic data transfer infrastructure (dti) on the chameleon cloud. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

[28] Angel Cruz-Roa, Hannah Gilmore, Ajay Basavanhally, Michael Feldman, Shridar Ganesan, Natalie NC Shih, John Tomaszewski, Fabio A González, and Anant Madabhushi. Accurate and reproducible invasive breast cancer detection in whole-slide images: A deep learning approach for quantifying tumor extent. *Scientific reports*, 7:46450, 2017.

[29] Padideh Danaee, Reza Ghaeini, and David A Hendrix. A deep learning approach for cancer detection and relevant gene identification. In *PACIFIC SYMPOSIUM ON BIOCOMPUTING 2017*, pages 219–229. World Scientific, 2017.

[30] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

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

[33] Xiao Ding, Yue Zhang, Ting Liu, and Junwen Duan. Deep learning for event-driven stock prediction. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[34] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[35] J Fombellida, I Martín-Rubio, S Torres-Alegre, and D Andina. Tackling business intelligence with bioinspired deep learning. *Neural Computing and Applications*, pages 1–8, 2018.

[36] Google. YouTube-8M Segments Dataset. https://research.google.com/youtube8m/.

[37] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[39] James B Heaton, Nick G Polson, and Jan Hendrik Witte. Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1):3–12, 2017.

[40] Jan Heichler. An introduction to beegfs, 2014.

[41] M Shamim Hossain and Ghulam Muhammad. Environment classification for urban big data using deep learning. *IEEE Communications Magazine*, 56(11):44–50, 2018.

[42] Jeremy Howard and Sylvain Gugger. Fastai: a layered api for deep learning. *Information*, 11(2):108, 2020.

[43] Zilong Hu, Jinshan Tang, Ziming Wang, Kai Zhang, Ling Zhang, and Qingling Sun. Deep learning for image-based cancer detection and diagnosis- a survey. *Pattern Recognition*, 83:134–149, 2018.

[44] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[45] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[46] Tyler B Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. *Advances in Neural Information Processing Systems*, 31:7265–7275, 2018.

[47] Brigitte Juanals and Jean-Luc Minel. Categorizing air quality information flow on twitter using deep learning tools. In *International Conference on Computational Collective Intelligence*, pages 109–118. Springer, 2018.

[48] Vanessa Isabell Jurtz, Alexander Rosenberg Johansen, Morten Nielsen, Jose Juan Almagro Armenteros, Henrik Nielsen, Casper Kaae Sønderby, Ole Winther, and Søren Kaae Sønderby. An introduction to deep learning on biological sequence data: examples and solutions. *Bioinformatics*, 33(22):3685–3690, 2017.

[49] Jaret M Karnuta, Michael P Murphy, Bryan C Luu, Michael J Ryan, Heather S Haeberle, Nicholas M Brown, Richard Iorio, Antonia F Chen, and Prem N Ramkumar. Artificial intelligence for automated implant identification in total hip arthroplasty: A multicenter external validation study exceeding two million plain radiographs. *The Journal of Arthroplasty*, 2022.

[50] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2525–2534. PMLR, 10–15 Jul 2018.

[51] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[52] Kate Keahey, Pierre Riteau, Dan Stanzione, Tim Cockerill, Joe Mambretti, Paul Rad, and Paul Ruth. Chameleon: a scalable production testbed for computer science research. In *Contemporary High Performance Computing*, pages 123–148. CRC Press, 2019.

[53] Mohammad Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2611–2620. PMLR, 10–15 Jul 2018.

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

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[56] 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.

[57] Quoc V Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *ICML*, 2011.

[58] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[59] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[60] Lizhi Liao, Heng Li, Weiyi Shang, and Lei Ma. An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–40, 2022.

[61] Ilya Loshchilov and Frank Hutter. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

[62] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[63] Sun Microsystems. Rfc1094: Nfs: Network file system protocol specification, 1989.

[64] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246, 2018.

[65] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020.

[66] NVIDIA Corporation. nvidia-smi - NVIDIA System Management Interface program. http://manpages.ubuntu.com/manpages/precise/man1/alt-nvidia-current-smi.1.html.

[67] Yosuke Oyama, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, and Brian Van Essen. The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1641–1652, 2020.

[68] 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 *Advances in neural information processing systems*, pages 8026–8037, 2019.

[69] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018.

[70] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. Scalable deep learning via i/o analysis and optimization. *ACM Transactions on Parallel Computing (TOPC)*, 6(2):1–34, 2019.

[71] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[72] Viktor Rausch, Andreas Hansen, Eugen Solowjow, Chang Liu, Edwin Kreuzer, and J Karl Hedrick. Learning a deep neural net policy for end-to-end control of autonomous vehicles. In *2017 American Control Conference (ACC)*, pages 4914–4919. IEEE, 2017.

[73] Sebastien Godard. sar(1) - Linux Man Page, 2021. https://linux.die.net/man/1/sar.

[74] Shaohuai Shi, Qiang Wang, Xiaowen Chu, and Bo Li. A dag model of synchronous stochastic gradient descent in distributed deep learning. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 425–432. IEEE, 2018.

[75] Shaohuai Shi, Qiang Wang, Xiaowen Chu, Bo Li, Yang Qin, Ruihao Liu, and Xinxiao Zhao. Communication-efficient distributed deep learning with merged gradient sparsification on gpus. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 406–415. IEEE, 2020.

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

[77] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[78] Arnold Tunick. On benchmarking multiple gpu computing resources for faster training of deep neural networks. Technical report, CCDC Army Research Laboratory Adelphi United States, 2020.

[79] Jinjiang Wang, Yulin Ma, Laibin Zhang, Robert X Gao, and Dazhong Wu. Deep learning for smart manufacturing: Methods and applications. *Journal of Manufacturing Systems*, 48:144–156, 2018.

[80] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[81] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878*, 2017.

[82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[83] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[84] Xiao Zeng, Ming Yan, and Mi Zhang. Mercury: Efficient on-device distributed dnn training via stochastic importance sampling. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 29–41, 2021.

[85] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, John Cazes, and Niall Gaffney. Fanstore: Enabling efficient and scalable i/o for distributed deep learning, 2018.

[86] Zhao Zhang, Lei Huang, J Gregory Pauloski, and Ian T Foster. Efficient i/o for neural network training with compressed data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 409–418. IEEE, 2020.

[87] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 284–294, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156. IEEE, 2018.