# FastMatch: Enhancing Data Pipeline Efficiency for Accelerated Distributed Training

Jianchang Su, Masoud Rahimi Jafari, Yifan Zhang, and Wei Zhang*
*University of Connecticut*, CT, USA
{jianchang.su, masoud.rahimi_jafari, yifan.4.zhang, wei.13.zhang}@uconn.edu *Corresponding author

***Abstract*—Training a deep learning model typically involves two interdependent stages: preprocessing input data on the CPU and training the model on the GPU. This process begins with the CPU performing resource-intensive I/O operations to load raw data, which it then processes and supplies to the GPU. This creates a typical provider-consumer relationship between the CPU and GPU. However, the inefficiency of I/O and CPU operations, in contrast to the high capabilities of GPUs, often turns preprocessing into a significant bottleneck. The imbalance between CPU and GPU resources leads to inefficiency, as the underutilized GPU cannot fully exploit its computational potential.**

**To address this issue, we propose FastMatch, a system designed to selectively and precisely cache key data, thereby reducing the reliance on extensive I/O operations, and adjusting the CPU's performance to better align with the GPU's capabilities swiftly. FastMatch dynamically identifies important data and prompts the caching system to store it in fast DRAM. This cached data significantly enhances I/O performance while maintaining comparable accuracy levels. Additionally, FastMatch monitors and rectifies mismatches in CPU and GPU resources by tuning the number of preprocessing processes. This helps distribute preprocessing workloads more evenly, ensuring optimal utilization of both CPU and GPU resources. We implemented a prototype of FastMatch and successfully integrated it with TorchData. Our evaluations showed up to $2.2\times$ reduction in training time compared to vanilla PyTorch Dataloader.**

***Index Terms*—Preprocessing Caching, Resource Matching, Importance Sampling**

## I. Introduction

Recently, Distributed Training (DT) has become essential for training large-scale neural network models [30]. DT addresses challenges in academia, such as drug discovery, smart city infrastructure, and large language model (LLM) training [19], [4], and in industry, including automated IC design and autonomous vehicle systems [16], [12]. This training method demands significant computational power and large datasets, divided into data preprocessing and model training phases. The ML training input pipeline, crucial for preprocessing, involves extraction, transformation, and loading (ETL) processes. The training phase focuses on iterative computation and model parameter updates [26]. While GPUs and TPUs have expedited training, preprocessing is managed by CPUs using data pipelining [32].

Despite minimizing latency, data pipelining remains a bottleneck in DT workflows [20]. Enhancing input workflow efficiency can significantly reduce training time. However, optimizing data handling in DT presents challenges, particularly in I/O-intensive workloads. Fetching data samples from storage leads to I/O bottlenecks, and traditional caching algorithms like LRU and LFU are suboptimal. Aligning computational and memory demands with system resources adds complexity, as dynamic preprocessing tasks vary in resource needs. Efficiently balancing resource use to prevent wastage and underutilization is critical but challenging [20]. Deep learning frameworks like PyTorch [23], TensorFlow [2], and MxNet [5] require specifying preprocessing core numbers.

Frameworks such as Cachew [8], SHADE [15], `tf.data` [29], and DALI [22] have been adopted for their resource efficiency, offering solutions for DT. These frameworks distribute data processing among remote CPU workers and reuse cached data transformations but lack efficient caching policies for increasing dataset volumes. Without sophisticated caching policies, performance and resource efficiency suffer despite scalability.

To address these challenges, we propose FastMatch, a unified framework for DT input data preprocessing. FastMatch decouples preprocessing into an independent service, streamlining resource management during ETL operations. It includes a centralized controller, dispatcher, and distributed input data workers integrated with the storage cluster and cache databases. Built on the `TorchData` framework, FastMatch extends its capabilities to include caching and autoscaling. Our key contributions are:

- We optimize data cache utilization while maintaining precision by leveraging an algorithm that uses data gradients as importance scores to generate repeatable **"harder"** samples during training.
- We design a scaling strategy that dynamically allocates CPU cores for preprocessing each training epoch, based on the status of preprocessing and training machines, to maintain peak performance and minimize resource inefficiency. FastMatch optimizes resource allocation and utilization by determining the appropriate number of CPU cores for preprocessing.
- We implement FastMatch into `TorchData` for comprehensive evaluation and benchmarking. Results show FastMatch accelerates model convergence by 4-20$\times$, increases cache hit ratio by 20%, improves preprocessing throughput by $\sim$40%, and reduces CPU core-cost-time by up to $\sim$40%.

## II. Background and Motivation

### A. Overview of Deep Learning Preprocessing

Deep learning training is fundamentally based on forward and backward propagation processes, critical for iterative learning and adjustment of network parameters. With the growing complexity of neural network models and data volume, there's a need for advanced distributed training frameworks. Frameworks such as PyTorch [27], TensorFlow [2], and MXNet [5] not only have exhibited remarkable performance, but also inherently facilitate distributed training, promoting an environment conducive to improving deep learning methodologies, especially in terms of efficiency and scalability. However, recent research indicates a shift in the bottleneck from computational aspects to data preprocessing, primarily due to the I/O demands during the ETL pipeline. Consequently, several frameworks, including `tf.data`[29], PyTorch `DataLoader` [27], NVIDIA DALI [22], CoorDL [20], and SHADE [15], have been developed, focusing on efficient input data processing. In particular, Cachew introduces the concept of "Machine Learning Input Data Processing as a Service", a novel paradigm that separates data processing and introduces autocaching and autoscaling features into the preprocessing service. SHADE, on the other hand, presents a priority-based caching mechanism, selectively caching crucial data during each mini-batch to mitigate I/O bottlenecks.

### B. Motivation

Existing studies have not yet provided a comprehensive solution to the challenges posed by cache-unfriendly issues in preprocessing and resource mismatches during training. Here, we discuss two key challenges that highlight the necessity for FastMatch.

**Challenge 1: Inefficient cache utilization due to data shuffling.** Data shuffling in machine learning training leads to unpredictable memory access patterns, complicating cache optimization. This randomness makes it difficult to determine which data to cache, necessitating adaptable caching strategies.

Importance sampling is a potential solution that relies on consistent data patterns, particularly gradient behavior across training epochs [17], [14]. Figure 1 shows a uniform gradient pattern across multiple epochs, suggesting predictability in the data gradients and the feasibility of importance sampling for caching.

Consistent patterns enable precise weighting of samples, prioritizing the most informative ones. This improves sampling efficiency and estimate accuracy. The consistency in data gradient patterns provides a foundation for optimizing importance sampling methodologies, leading to more effective and reliable outcomes in data-centric research.

**Challenge 2: Resource capacity discrepancy and data skew during training.** To investigate the mismatch in resource provisioning between data preprocessing stages and subsequent model training phases, we conducted a micro-benchmark of ResNet-101 training on a single A100-80G GPU. This aimed to illuminate CPU, GPU, and memory usage patterns across various stages of model training and identify potential inefficiencies in resource allocation.

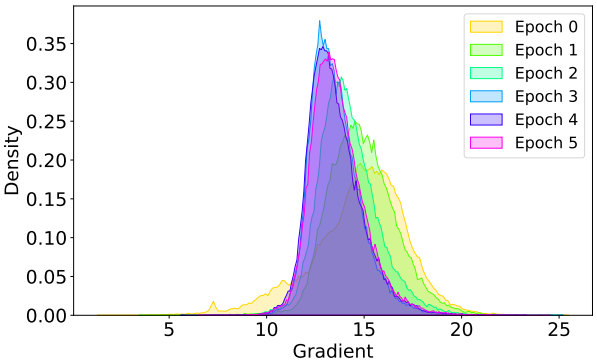



Fig. 1: Uniform data gradient patterns observed across epochs.

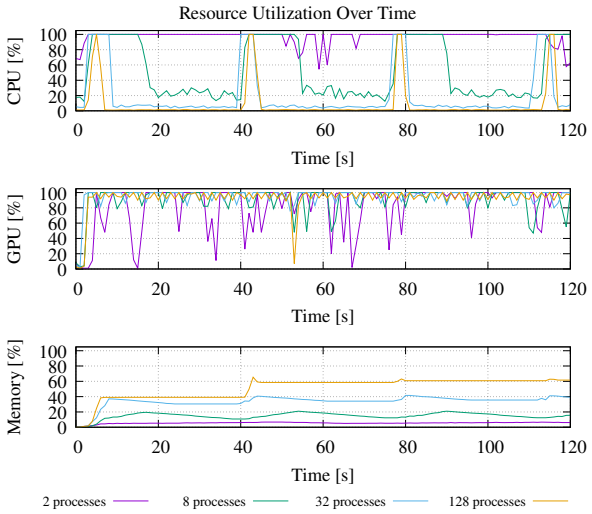



Fig. 2: Resource utilization across various process counts.

Figure 2 presents subfigures detailing computational resource utilization for different numbers of preprocessing processes during ResNet-101 training. These subfigures delineate the temporal dynamics of CPU and GPU usage, along with memory consumption, providing a granular view of resource leverage throughout the training process. The work distribution between the CPU and GPU can be viewed as a classic producer-consumer problem, where imbalanced resource allocation leads to system inefficiency. The CPU acts as the producer, preprocessing data and feeding it to the GPU, while the GPU acts as the consumer, training the model using the preprocessed data. An imbalance in the rate at which the CPU produces preprocessed data and the rate at which the GPU consumes it can result in underutilization of resources and suboptimal performance.

**Implications.** The challenges of inefficient cache utilization and resource capacity discrepancy in deep learning preprocessing and training underscore the need for innovative solutions. FastMatch addresses these issues by optimizing cache

utilization and resource allocation. FastMatch uses importance sampling to prioritize caching the most informative samples, reducing I/O bottlenecks. It also employs a scaling algorithm to balance the load between preprocessing and training tasks, optimizing resource usage. By decoupling preprocessing tasks and placing them closer to data storage, FastMatch reduces network I/O demands and improves training efficiency.

## III. System Design

To address the aforementioned challenges, we established the following design objectives and goals for FastMatch:

**G1: Improve the selection process for training data.** Employ a gradient-based method to dynamically evaluate data importance, selecting cache candidates that enhance learning efficiency and model performance.

**G2: Increase the efficiency and effectiveness of the data caching mechanism.** Ensure data diversity is maintained while minimizing the computational resources required for data processing, aiming to retain or surpass original accuracies.

**G3: Improve overall resource allocation efficiency.** Develop a system that adapts to dynamic workloads and responds to real-time CPU and GPU resource availability, reducing resource wastage.

To achieve **G1**, we introduce an importance-based caching mechanism. This method prioritizes caching important data during training, focusing on data that present the greatest challenge for retention and access. For **G2**, we develop a combination of a dispatcher and a novel data pipe operator. These components work within an in-memory caching system to optimize data distribution and caching, reducing latency in data retrieval. To address resource imbalances in deep learning training for **G3**, we propose an autoscaling controller for resource utilization. This controller periodically monitors resource utilization metrics and adjusts the distribution of preprocessing and training workloads, enhancing the efficiency of computing cluster resources and aligning computational demands with available resources.

### A. System Architecture

The architecture of FastMatch includes three main components as shown in Figure 3: **Trainer**, **Controller**, and **Preprocessing Service**. The training process starts with the Trainer requesting the dataset from the Controller. The Controller's Dispatcher checks the cache status and forwards the request to the Preprocessing Service. The Preprocessing Service processes the data based on a User-Defined-Function (UDF) and sends it to the Trainer. Meanwhile, the Autoscaler monitors resource usage and adjusts CPU allocation in the Preprocessing Service to optimize performance and efficiency.

### B. Workflow

**Requesting Submission.** The Trainer sends a request with the `TorchData` pipeline and initial number of preprocessing processes. We developed `RemoteDataloader`, based on `Dataloader2`, to serialize `DataPipe` containing UDFs and send it to the Controller via PyZMQ.

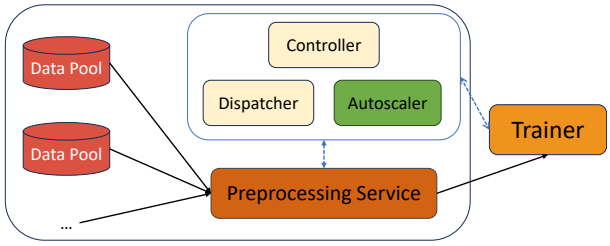



Fig. 3: FastMatch architecture overview.

The Controller deserializes the `DataPipe`, extracts the file list, and passes it to the dispatcher. The dispatcher checks the cache state and allocates preprocessing tasks among available workers. After the second training epoch, the Trainer calculates importance scores and sends them to the Controller, which uses importance sampling to prioritize significant data for preprocessing.

**Data Dispatching.** The Dispatcher manages data caching and preprocessing. It checks if files are cached, marks them as pending if not, or retrieves them if cached. The dispatcher distributes data to available workers in the Preprocessing Service. Cached data avoids disk loading, while new data undergoes normal processing and potential cache replacement.

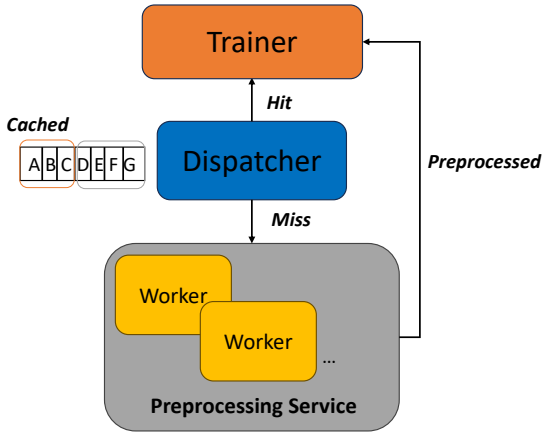



Fig. 4: FastMatch dispatcher workflow.

**Autoscaling Controller.** The Autoscaler optimizes resource allocation in the Preprocessing Service, balancing CPU and GPU resources. It monitors resource usage and adjusts allocations to maintain performance. An algorithm based on heuristic search determines the optimal number of preprocessing processes to minimize data output discrepancy.

**Preprocessing Service.** The Preprocessing Service is located near storage to reduce bottlenecks. It consists of multiple workers aligned with GPUs. The dispatcher assigns file lists based on workforce size and transmits data to trainers via PyZMQ TCP protocols. The process count is adjusted each epoch to enhance preprocessing efficiency.

**RemoteLoader API.** FastMatch's training module uses PyZMQ for communication between Trainer, Controller, and Preprocessing Service. The Trainer sends a `TorchData` datapipe to the Controller and receives processed data. After

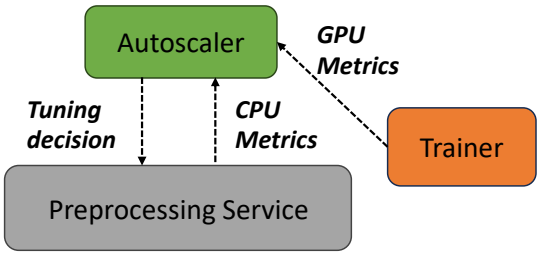



Fig. 5: FastMatch autoscaling workflow.

each epoch, it evaluates data importance and sends the analysis to the Controller. APIs are provided as shown in Listing 1.

```
with train_loader.record_importance(data_keys):
    output = model(data)
train_loader.update_gradient_info(data_keys, data.grad)
```

Listing 1: FastMatch `RemoteLoader` API.

## IV. Policy Design

### A. Caching Policy

**Importance-based Caching.** FastMatch uses importance sampling to manage intermediate data caching before random operations, reducing I/O overhead while maintaining data diversity and model accuracy.

**Intermediate Data Caching Node.** The caching node is placed before random operations to maintain data diversity. Cached samples are processed further to generate diverse training samples.

**Data Importance Score.** The data importance score is based on normalized gradients:

$$g_{i,t}^{norm} = \frac{g_{i,t} - M_t}{IQR_t}$$

where $M_t$ is the median gradient and $IQR_t$ is the interquartile range.

**Gradient-based Importance Sampling** FastMatch uses the algorithm in Algorithm 1 to sample important data. Data with higher gradients are prioritized for caching.

**Algorithm 1** Gradient-based Importance Sampling

**Input:** Dataset $D = \{(k_i, v_i)\}_{i=1}^{n}$
**Output:** Sampled dataset $S$

$K, \boldsymbol{V} \leftarrow$ Unzip $D$
$\mu \leftarrow \max(\boldsymbol{V})$
$\sigma \leftarrow \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}$
$P_i \leftarrow \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(v_i-\mu)^2}{2\sigma^2}}$ for $i = 1, 2, \ldots, n$
Normalize $P_i$: $P_i \leftarrow \frac{P_i}{\sum_{j=1}^{n} P_j}$
$P_i \leftarrow \text{Softmax}(P_i)$ for $i = 1, 2, \ldots, n$
Generate samples $S$ from $K$ with probabilities $P_i$
**return** $S$

**Caching and Reuse.** FastMatch caches intermediate data samples and checks the cache for new batches. Cache hits retrieve data from cache; cache misses process original samples and store them based on importance. Least important samples are evicted when the cache is full.

### B. Resources Matching

FastMatch optimizes resource allocation by balancing preprocessing and training demands, adapting to dynamic workloads and resource availability.

**Problem Formulation.** We use an Integer Linear Programming (ILP) model to balance CPU and GPU resource allocation:

$$\begin{aligned} \min \quad & \sum_{i \in \alpha} |p_i - q_i| \\ \text{s.t.} \quad & p_i = \sum_{j \in J} a_{ij}\beta_j, \ \forall i \in I \\ & q_i = \sum_{k \in K} b_{ik}\gamma_k, \ \forall i \in I \\ & \sum_{j \in J} \beta_j \leq C, \\ & \beta_j \in \mathbb{Z}^+, \ \forall j \in J \\ & \gamma_k \in \mathbb{Z}^+, \ \forall k \in K \end{aligned} \tag{1}$$

Here, $p_i$ and $q_i$ represent CPU and GPU resource production and consumption rates, respectively. $a_{ij}$ and $b_{ik}$ are allocation coefficients, where $j \in J$ represents the set of CPU resources and $k \in K$ represents the set of GPU resources. $C$ is the total resource capacity. The objective is to minimize the absolute difference between the production and consumption rates across all tasks $i \in I$, subject to the resource capacity constraints.

**Heuristic Approach.** Due to the complexity of aforementioned ILP problem, FastMatch uses a heuristic optimization algorithm for resource matching for real solution. The algorithm iteratively adjusts CPU allocations to minimize imbalance, as described in Algorithm 2.

**Algorithm 2** Resource Matching Algorithm

**Input:** $\alpha$: set of available resources; $\beta_i$: amount of CPU resource $i$ allocated for preprocessing; $\gamma_k$: amount of GPU resource $k$ required for training;
**Output:** Balance between preprocessing and training

Initialize resource allocation arrays $\beta, \gamma$
$\tau \leftarrow$ threshold for balance tolerance
**while** system not steady **do**
    $P_{CPU} \leftarrow$ CalculateCPUProductionRates($\beta$)
    $Q_{GPU} \leftarrow$ CalculateGPUConsumptionRates($\gamma$)
    $Imbalance \leftarrow |P_{CPU} - Q_{GPU}|$
    **if** $Imbalance > \tau$ **then**
        $\delta \leftarrow$ DetermineAdjustment($P_{CPU}, Q_{GPU}$)
        **if** $P_{CPU} > Q_{GPU}$ **then**
            $\beta \leftarrow$ DecreaseCPUAllocation($\beta, \delta$)
        **else**
            $\beta \leftarrow$ IncreaseCPUAllocation($\beta, \delta$)

FastMatch uses a feedback loop to monitor preprocessing and training performance, making real-time adjustments to

resource allocation based on CPU and GPU utilization, data production, and system throughput. This ensures the system remains optimized for performance and resource efficiency.

## V. Implementation

FastMatch is developed atop `TorchData` in ∼8K LOC, leveraging its data pipeline capabilities to establish a remote dataloader that facilitates the bidirectional flow of data. Communication between the training node and preprocessing services is orchestrated via PyZMQ, utilizing the PUSH/PULL mode for efficient data polling from each worker. For controller interactions, the REQ model is adopted to update information, including data importance scores and adjustments in resource allocation.

Resource usage is monitored using gAdvisor for CPU metrics and DCGM-Exporter for GPU activity within the training environment. FastMatch then applies a predefined policy to address any resource disparities after each training cycle, aiming to optimize resource utilization and enhance overall system performance.

## VI. Evaluation

We evaluated FastMatch using a setup with a storage cluster and a training node, equipped with four NVIDIA A100-80G GPUs. The storage was integrated into a Kubernetes cluster using CEPHFS [3] as the network filesystem. The trainer node connected to the storage cluster through a 10Gbps network link. To benchmark FastMatch, we used various datasets and models, focusing on vision-related tasks. We tested with ImageNet-1K [7], as well as other datasets like Caltech256 [9] and CIFAR-10 [1].

### A. Caching Experiments

We evaluated our caching system, FastMatch, against traditional and modern caching policies like the Least Recently Used (LRU) algorithm and the SHADE cache policy [15]. Our benchmarks showed that FastMatch outperformed in terms of throughput, cache hit rate, and reduced epoch time, using a fixed worker configuration. To ensure a fair comparison with LRU, which does not support data duplication, we pre-randomized and duplicated the dataset to maintain equivalent data volumes.

**Cache Hit Rate.** Through rigorous testing shown in Figure 6(a) and Figure 6(b) with both the ResNet-50 and Vision Transformer (ViT) models on the ImageNet-1K dataset, FastMatch significantly outperformed LRU and SHADE in cache hit rates across various cache sizes. Notably, for a cache size of 50, FastMatch achieved cache hit rates of 96% for ResNet-50 and 97% for ViT, marking over 60% improvement over LRU and around 7% over SHADE for ResNet-50. This pattern of superiority is consistent across different cache sizes and model architectures, showcasing FastMatch's adaptability and efficiency.

**Throughput.** In terms of throughput, FastMatch consistently surpassed both LRU and SHADE, underlining the system's enhanced efficiency as shown in Figure 7(a) and Figure 7(b). For the ResNet-50 model with a cache size of 50%, FastMatch achieved a throughput of 4439 images per second, which is approximately 8.75% higher than SHADE's throughput of 4082 images per second. Similarly, for the ViT model at the same cache size, FastMatch recorded a throughput of 4501 images per second, outperforming SHADE by about 9.24%. These improvements, though subtle, are crucial in high-stakes computational environments where they translate into significant time savings and operational efficiency, especially when scaled across extensive datasets and complex model architectures. The flat LRU throughput in Figure 7(b), despite increasing hit rates, is due to data processing overhead, communication bottlenecks, and computational limits that constrain throughput improvements.

The detailed performance evaluation of FastMatch reveals its significant impact on computational efficiency, enhancing data retrieval speeds, and thereby accelerating model training phases. By achieving superior cache hit rates, higher throughput, and reduced epoch times, FastMatch demonstrates its algorithmic sophistication and operational effectiveness.

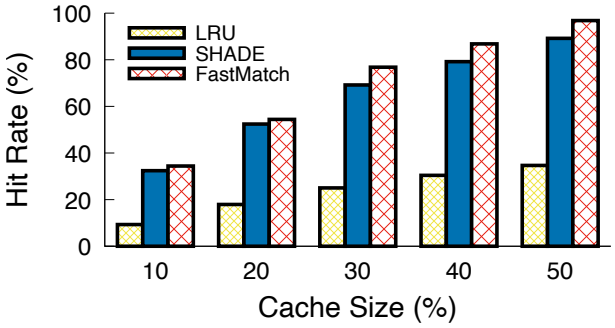
**Accuracy Analysis.** Our comprehensive evaluation of FastMatch, across several vision task datasets including CIFAR-10 with ResNet-50 and ImageNet-1K with ViT, demonstrates the system's robustness without compromising model accuracy. Importantly, FastMatch not only maintains accuracy levels but, in some cases, enhances them compared to baseline models.

Analyzing the performance on small dataset CIFAR-10, as depicted in Figure 8(a), FastMatch with ResNet-50 shows a gradual improvement in accuracy over time, reaching up to 82% accuracy at around 2000 seconds, closely aligning with the baseline's progression yet achieving this at a notably faster rate. This trend is even more pronounced in the Caltech256 dataset, where FastMatch achieves a similar accuracy improvement in significantly less time, as shown in Figure 8(b).

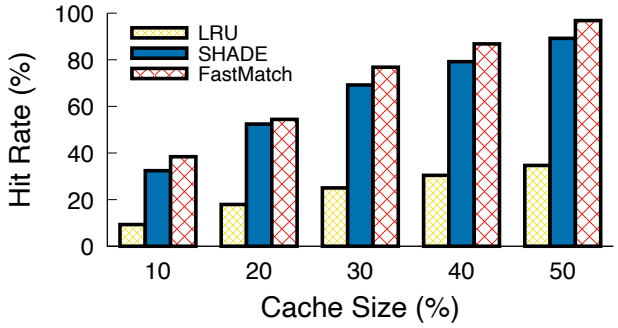
For the more complex ImageNet-1K dataset with ViT models, the baseline and FastMatch trajectories, illustrated in Figure 8(c), further reinforce the effectiveness of our approach. FastMatch not only keeps pace with the baseline's accuracy but also exhibits an upward trend, reaching 54.068% accuracy more quickly than the baseline model. This suggests that FastMatch's importance sampling mechanism contributes to more efficient learning, enhancing model performance without extending training time.

Crucially, our importance sampling strategy does not detract from model accuracy; instead, it even increases it. This indicates that FastMatch's caching mechanism, grounded in importance sampling, is not merely a means to manage cache more effectively but also a catalyst for faster model convergence. By prioritizing the processing of more **"informative"** data points, FastMatch accelerates the learning process, enabling models to achieve high accuracy in less time compared to traditional caching methods.

This analysis underscores the sophistication, efficacy, and robustness of FastMatch's caching system. By integrating enhanced throughput, superior cache hit rates, and an impor-

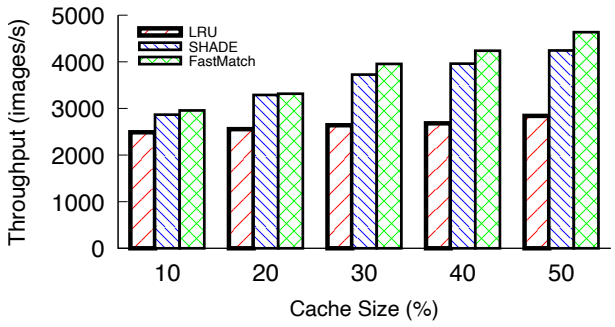



(a) ViT Cache hit rate

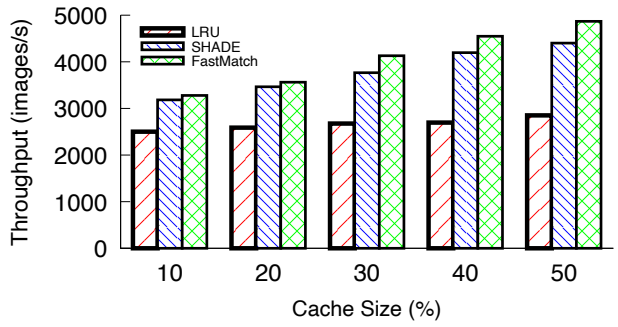



(b) ResNet-50 Cache hit rate

Fig. 6: Cache hit rate comparison for ViT and ResNet-50.

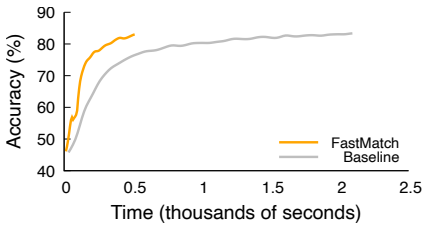



(a) ViT cache throughput

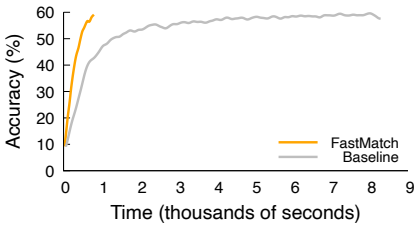



(b) ResNet-50 cache throughput

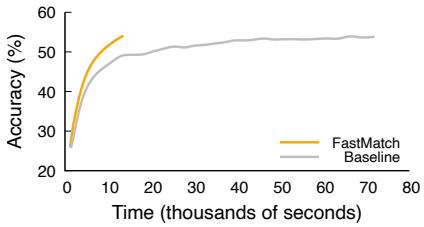
Fig. 7: Cache throughput comparison for ViT and ResNet-50.




(a) CIFAR-10 Accuracy with ResNet-50




(b) Caltech256 Accuracy with ResNet-50




(c) ImageNet-1K Accuracy with ViT

Fig. 8: Accuracy comparisons across different datasets and models.

tance sampling mechanism that occasionally improves model accuracy, FastMatch establishes a new benchmark in caching technologies. It confirms that strategic data handling can significantly contribute to the acceleration of model training phases, thereby reinforcing the utility of FastMatch in the landscape of advanced computational models and frameworks.

### *B. AutoScaling Experiments*

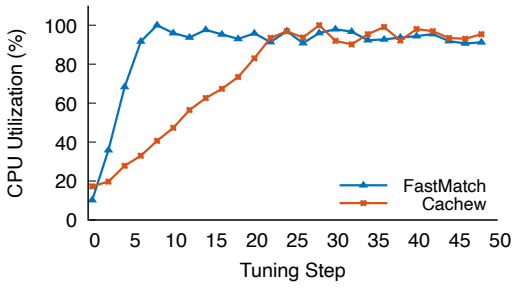
Autoscaling is crucial in complex computational environments for balancing resource utilization and optimizing operational costs. Our experimental setup, detailed in the above, leverages autoscaling to ensure optimal resource distribution between computational tasks and available hardware.

**Resource Allocation Efficiency.** To evaluate the efficiency of autoscaling policies, we compared FastMatch's evolutionary algorithm-based approach with Cachew's incremental scaling policy [8]. Our goal was to assess how rapidly and effectively each policy matches computational resources to the workload demands of each epoch; we adapted Cachew autoscaling at a single epoch level.

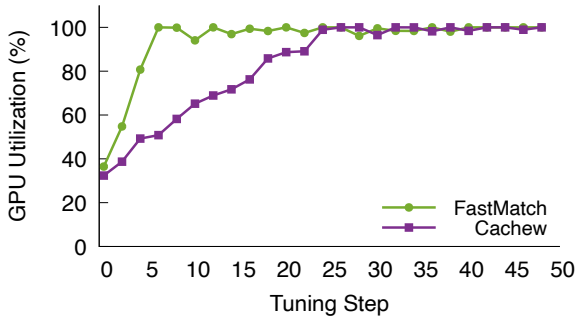
As Figure 9 shows the dynamic resource allocation capabilities of FastMatch versus the steady resource increment approach of Cachew. Unlike Cachew's linear resource increase, FastMatch's evolutionary algorithm adapts the allocation of the number of processes and GPU resources to the needs of each training epoch. This adaptability ensures that resources are not just increased, but are aligned with the computational requirements, enhancing efficiency, and reducing unnecessary expenditure.

The FastMatch algorithm exhibited by FastMatch significantly outperforms Cachew in terms of resource allocation speed and adaptability. By closely aligning resource distribu-

tion with the varying demands of computational tasks, FastMatch not only optimizes resource usage, but also contributes to cost efficiency.




(a) CPU utilization comparison.




(b) GPU utilization comparison.

Fig. 9: Comparative analysis of CPU and GPU utilizations between FastMatch and Cachew across tuning steps.

In summary, our experiments underline the superior adaptability and cost-efficiency of the autoscaling method implemented in FastMatch. This approach not only ensures optimal utilization of computational resources but also enhances the overall system performance by aligning resource distribution with real-time computational demands.

### C. Scalability Evaluation

We evaluated FastMatch's scalability by examining its performance across varying Requests Per Second (RPS) and worker counts, specifically from 1 to 64 workers. The aim was to assess how efficiently FastMatch scales in high-demand and multiple-tenant scenarios.

Figure 10 displays the relationship between RPS and dispatch latency, highlighting FastMatch's capability to maintain low latency even as request rates increase. This demonstrates the effectiveness of the system's task management under increasing loads.

Figure 11 shows the throughput scalability as the worker count increases to 64. The figure shows near-linear throughput growth, with a slight plateau as worker numbers approach 64, suggesting optimal resource utilization and system's ability to leverage additional computational resources effectively.

Both the dispatcher and preprocessing services contribute significantly to this scalability, ensuring efficient task allocation and data preparation, respectively. These results confirm

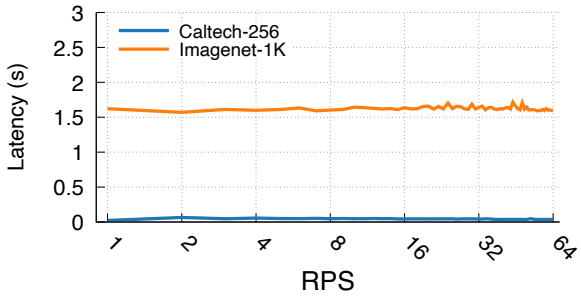



Fig. 10: RPS versus dispatch latency.

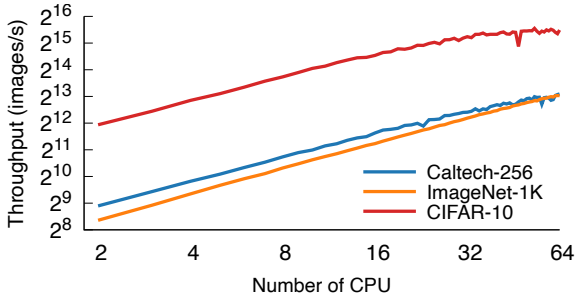



Fig. 11: Throughput scalability from 1 to 64 workers.

FastMatch's robust scalability, capable of adapting to and efficiently processing large volumes of data in demanding computational environments.

## VII. Related Work

**Data Preprocessing Frameworks.** Several frameworks optimize input data processing for DLT. `tf.data` [29], DALI [22], and Cachew [8] provide mechanisms to alleviate data bottlenecks by distributing data processing and caching transformations. PyTorch DataLoader and MXNet DataLoader offer similar functionalities [27], [5]. However, these frameworks lack policies to efficiently leverage cache and eviction mechanisms for optimizing ML training performance and cost. FastMatch is inspired by `tf.data` paradigm, introducing novel policies for efficient caching and resource management in ML training jobs.

**ETL Caching.** Caching mitigates I/O bottlenecks in DLT. Cachew dynamically scales resources, leveraging autocache for caching all data used. Quiver [28] focuses on distributed caching for source data, while CoorDL and OneAccess optimize reuse within coordinated jobs [20], [13]. DeepIO [32] introduces entropy-based batch selection but lacks robust eviction policies. Hoard [25] accelerates DNN training without prioritizing sample importance. SHADE [15] utilizes importance sampling but mini-batch sampling may bring more overhead when in remote scenario. iCACHE [6] shares similarities reducing performance. FastMatch, however, enhances caching by considering cache the intermediate data to ensure the sample variance, significantly outperforming other systems.

**Autoscaling and Resource Matching.** ML resource management systems like Gandiva, Tiresias, Optimus, and MARBLE focus mainly on GPU allocation [31], [10], [24], [11],

leaving users to manually optimize multi-dimensional resources for data processing, which is challenging in complex ML environments. Decima and Gavel propose scheduling techniques for DNN training jobs but do not address data processing optimizations [18], [21]. In contrast, FastMatch jointly optimizes data caching and resource scaling through application-specific metrics, dynamically adjusting resources to match each job's requirements while avoiding over-provisioning.

## VIII. Conclusion

FastMatch redefines computational efficiency in machine learning, focusing on quickly achieving target accuracies and efficiently utilizing resources. It surpasses traditional caching methods by adapting data prioritization and resource allocation in real-time, significantly reducing operational costs and expediting model training. This dual approach optimizing for both accuracy and resource management positions FastMatch as a key innovation for enhancing the speed and resource-effectiveness of machine learning projects.

## References


[1] "CIFAR-10 and CIFAR-100 datasets." [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[3] CEPH. Ceph file system — ceph documentation. [Online]. Available: https://docs.ceph.com/en/latest/cephfs/

[4] T. Chatzinikolaou, E. Vogiatzi, A. Kousis, and C. Tjortjis, "Smart healthcare support using data mining and machine learning," in *IoT and WSN based Smart Cities: A Machine Learning Perspective*. Springer, 2022, pp. 27–48.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[6] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X. Sun, and G. Chen, "icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2023, pp. 220–232. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA56546.2023.10070964

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[8] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, "Cachew: Machine learning input data processing as a service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 689–706.

[9] G. Griffin, A. Holub, and P. Perona, "Caltech 256," Apr 2022.

[10] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[11] J. Han, M. M. Rafique, L. Xu, A. R. Butt, S.-H. Lim, and S. S. Vazhkudai, "Marble: A multi-gpu aware job scheduler for deep learning on hpc systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 272–281.

[12] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, X. Ning, Y. Ma, H. Yang, B. Yu, H. Yang, and Y. Wang, "Machine learning for electronic design automation: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 26, no. 5, jun 2021. [Online]. Available: https://doi.org/10.1145/3451179

[13] A. Kakaraparthy, A. Venkatesh, A. Phanishayee, and S. Venkataraman, "The case for unifying data loading in machine learning clusters," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/kakaraparthy

[14] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," 2019.

[15] R. I. S. Khan, A. H. Yazdani, Y. Fu, A. K. Paul, B. Ji, X. Jian, Y. Cheng, and A. R. Butt, "{SHADE}: Enable fundamental cacheability for distributed deep learning training," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 135–152.

[16] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 2, pp. 712–733, 2021.

[17] T. Lahire, "Importance sampling for stochastic gradient descent in deep neural networks," *arXiv preprint arXiv:2303.16529*, 2023.

[18] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," 2019.

[19] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," 2024.

[20] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *Proc. VLDB Endow.*, vol. 14, no. 5, p. 771–784, jan 2021. [Online]. Available: https://doi.org/10.14778/3446095.3446100

[21] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 481–498. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[22] NVIDIA. NVIDIA DALI Documentation — NVIDIA DALI 1.27.0 documentation. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/index.html

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[24] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190517

[25] C. Pinto, Y. Gkoufas, A. Reale, S. Seelam, and S. Eliuk, "Hoard: A distributed data caching system to accelerate deep learning training on the cloud," 2018.

[26] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via i/o analysis and optimization," *ACM Trans. Parallel Comput.*, vol. 6, no. 2, jul 2019. [Online]. Available: https://doi.org/10.1145/3331526

[27] PyTorch. Datasets & DataLoaders — PyTorch tutorials 2.0.1+cu117 documentation. [Online]. Available: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[28] Z. Tan, X. Yuan, C. He, M.-K. Sit, G. Li, X. Liu, B. Ai, K. Zeng, P. Pietzuch, and L. Mai, "Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness," 2023.

[29] tf.data. Better performance with the tf.data API | TensorFlow core. [Online]. Available: https://www.tensorflow.org/guide/data_performance

[30] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, mar 2020. [Online]. Available: https://doi.org/10.1145/3377454

[31] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[32] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. 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)*, 2018, pp. 145–156.