NDPipe: Exploiting Near-data Processing for Scalable Inference and Continuous Training in Photo Storage

Jungwoo Kim*
jungwoo@dgist.ac.kr
DGIST
South Korea

Seonggyun Oh*
sungkyun123@dgist.ac.kr
DGIST
South Korea

Jaeha Kung
jhkung@korea.ac.kr
Korea University
South Korea

Yeseong Kim
yeseongkim@dgist.ac.kr
DGIST
South Korea

Sungjin Lee
sungjin.lee@dgist.ac.kr
DGIST
South Korea

Abstract

This paper proposes a novel photo storage system called NDPipe, which accelerates the performance of training and inference for image data by leveraging near-data processing in photo storage servers. NDPipe distributes storage servers with inexpensive commodity GPUs in a data center and uses their collective intelligence to perform inference and training near image data. By efficiently partitioning deep neural network (DNN) models and exploiting the data parallelism of many storage servers, NDPipe can achieve high training throughput with low synchronization costs. NDPipe optimizes the near-data processing engine to maximally utilize system components in each storage server. Our results show that, given the same energy budget, NDPipe exhibits 1.39× higher inference throughput and 2.64× faster training speed than typical photo storage systems.

CCS Concepts: • **Information systems** → *Distributed storage*; • **Computing methodologies** → *Neural networks.*

Keywords: Systems for AI, near-data processing, and photo storage systems

ACM Reference Format:
Jungwoo Kim, Seonggyun Oh, Jaeha Kung, Yeseong Kim, and Sungjin Lee. 2024. NDPipe: Exploiting Near-data Processing for Scalable Inference and Continuous Training in Photo Storage. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3620666.3651345

*These authors equally contributed to this work.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651345

1 Introduction

The demand for storing and retrieving voluminous image data has seen an unprecedented rise in the contemporary digital ecosystem. Daily, approximately 400 million photos are uploaded across social media including Facebook and Instagram [47, 48]. Concurrently, 1.2 billion photos are archived in online repositories such as Google Photos every day [2]. In response to this surging demand, substantial infrastructure resources are dedicated to photo services, e.g., Facebook allocates about 28% of its data center capacity [1].

Modern photo storage platforms integrate deep learning (DL) to enhance user experience, notably outperforming traditional algorithms [23, 87] in feature extraction [54], image categorization [104] , and recommendation delivery [41, 93]. However, incorporating DL into storage platforms incurs significant computation and I/O costs. Developing Deep Neural Network (DNN) necessitates extensive computing capacity and frequent data transfers for large datasets, extending training periods to weeks [17, 70]. Moreover, the growing volume of images [73] requires more intensive image inference, placing pressure on computational and I/O resources [42].

The integration of DL into storage systems manifests two primary concerns. The first is the *outdated model problem*, where trained DNN models gradually lose accuracy over time due to data evolution or *drift* [34, 69, 71]. Strategies such as regular retraining [42, 72] or initiating training upon significant accuracy drops [15, 33] have been explored to mitigate this issue. However, these methods come with high computing and IO costs as they usually involve full retraining from scratch. *Fine-tuning* [46, 49, 97], which updates specific model parts, offers a more efficient alternative, yet still demands substantial data traffic to feed training datasets, hindering timely adaptation to changing data [17, 70, 100]. The second concern, the *outdated label problem*, arises when

new models render older labels obsolete [99]. In photo storage systems, image labels are indexed in databases for user queries [32, 36, 77], thus demanding updates through *offline inference* with model changes. It leads to considerable I/O costs to process historical images. Consequently, typical photo storage systems either selectively apply offline inference or turn to alternative labeling strategies [20, 66, 67].

In this paper, we propose a novel photo storage system, *NDPipe*, which enables fine-tuning and offline inference by leveraging near-data processing (NDP) to address the outdated model and label problems. The main idea of NDPipe stems from the insight that the computing demands for fine-tuning and offline inference are not excessively high, and can be effectively managed by low-end GPUs. This approach shifts the focus from computational power to the challenge of data communication; the main bottleneck is not the processing capability, but rather the extensive data transfer between storage and compute servers. In NDPipe, we thereby incorporate cost-effective and energy-efficient commodity GPUs into storage servers to perform fine-tuning and offline inference locally, significantly reducing the data transfer. NDPipe also enhances these processes by utilizing high aggregate throughputs of many storage servers operating in parallel.

The potential of NDP has been explored previously with various initiatives seeking to enhance DL algorithms [56, 66, 94]. However, its direct application within the context of photo storage introduces unique challenges. The following are the three key components of NDPipe, each addressing a targeted challenge of photo storage systems:

1. Parallel Training Strategy Across Storage Servers. The primary challenge is how to efficiently perform fine-tuning-based training tasks across numerous storage servers. We found that merely offloading entire fine-tuning tasks to storage servers does not yield optimal performance, primarily due to the significant *weight synchronization costs* across the network. NDPipe tackles this with a new training strategy, *fine-tuning-based data and model parallelism* (FT-DMP). It segments a DNN model into two parts: one for storage servers with weight-freeze layers and the other one for a training server with layers needing updates. Since it makes weight updates happen locally, i.e., only in the training server, we can reduce the synchronization overhead significantly.

2. Load Balancing with Model Partitioning. Another challenge is achieving a load balance between the training server and storage servers, requiring careful choices like the number of storage servers for training and effective DNN model partitioning. These decisions depend on specific model types and hardware performance, including GPUs and network bandwidth. A critical consideration here is the risk of resource underutilization: deploying too many storage servers might lead to marginal performance gains but increase energy inefficiency. To navigate this complexity, we developed the *Automated model Partitioning and Organization (APO)* tool. It is designed to identify the most suitable partitioning points for DNN models and ascertain the suitable number of storage servers for training.

3. Advanced Scheduling and Optimization for End-to-End DL Pipeline. In developing NDPipe, we recognized that optimizing each subprocedure in the end-to-end DL pipeline of photo storage applications is key to achieving maximized efficiency and performance. This encompasses not just computation parts of DL inference/training but also preprocessing stages and data communications needed for fine-tuning. For example, we observed a notable bottleneck in the process of reading and preprocessing image data to supply the necessary intermediate data for training and inference. Our approach thus led to the development of a comprehensive optimization software framework called *near-data processing engine* (NPE). It collectively enhances data handling, reduces bottlenecks, and ensures optimal GPU utilization across the system, addressing challenges in the full spectrum of DL operations, from data ingestion to model updating.

We have implemented a proof-of-concept prototype of NDPipe with two main components: PipeStore and Tuner. PipeStore is a storage server equipped with a low-end GPU or an inference accelerator for near-data training and inference. Tuner is a training server that manages distributed PipeStores. It is also responsible for a part of training using a more powerful GPU.

To evaluate NDPipe, we conduct a case study with an image classification system that employs five popular DNN models. Our evaluation using CIFAR100 [62], ImageNet-1K [91], and ImageNet-21K [25] shows the following key results. First, NDPipe provides scalable inference performance without being bottlenecked by data transfers, fully utilizing GPUs distributed over storage servers. When using 4–7 PipeStores, we achieve the same level of inference throughput as a typical centralized inference system using two V100 GPUs. In this configuration, NDPipe shows 1.39× higher energy efficiency. Second, ten PipeStores and one Tuner provide 1.64× faster training speed and 2.02× higher energy efficiency than a centralized training server that performs training using two Tesla V100 GPUs. Third, compared to the outdated model, NDPipe exhibits 1.7% and 2.4% higher top-1 and top-5 accuracy, on average, respectively. When compared to the model created from scratch, NDPipe suffers from 2.3% and 1.5% accuracy drops at top-1 and top-5, on average, but provides over 300× faster training time, which facilitates continuous model updates.

This paper is organized as follows: We give background in §2 and analyze the impacts of the outdated model and label problems in §3. Then, we discuss the technical issues of adopting NDP to photo storage systems in §4. In §5, we present the design of NDPipe. After showing experimental results in §6 and discussing NDPipe in §7, we review prior work in §8 and conclude in §9.

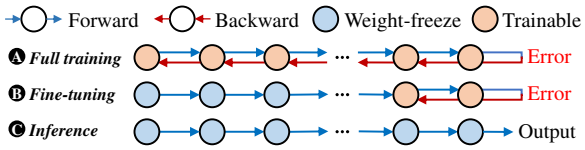
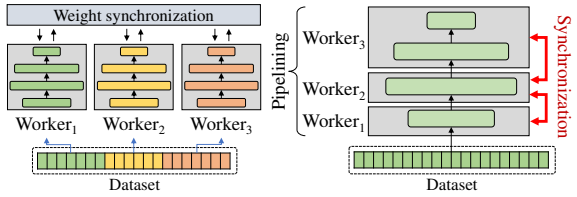


Fig. 1. Deep learning training and inference process



(a) Data parallelism **(b)** Model parallelism

Fig. 2. Distributed training technique

2 Background

2.1 Training Strategy for DNN

Fine Tuning. Fine-tuning is a strategy used to enhance a pre-trained model on a new or related task [46, 49, 68, 97]. Training a model from scratch is time-consuming as it requires updating all weights in every layer through forward and backward passes on a large general dataset (Ⓐ in Fig. 1). Fine-tuning adjusts only a few weights of the original model to adapt it to specific tasks or new datasets. This approach not only improves the model's accuracy but requires shorter training time. Throughout the paper, training from scratch is referred to as *full training* to differentiate it from fine-tuning.

A commonly used fine-tuning strategy is to train a pre-trained model's last few Multi-Layer Perceptron (MLP) layers [97, 105]. Specifically, for Convolutional Neural Network (CNN)-based models, these MLP layers are often termed the classifier; for transformer-based models, they are known as the task module. This approach entails freezing the initial layers of the pre-trained model, focusing only on updating the weights of the subsequent layers during the backward pass, as illustrated in Ⓑ. The layers with frozen weights are referred to as *weight-freeze layers*, while the ones being updated are the *trainable layers*. In this process, input batches are processed by the weight-freeze layers, which extract relevant features. These features are then utilized in training the model's trainable layers. Notably, the fine-tuning procedure for weight-freeze layers is identical to the inference process (Ⓒ), thus making it more resource-efficient than full training.

Distributed Training. Distributed training is a technique to speed up training neural networks in a distributed manner. Two common approaches exist: *data parallelism* (DP) and *model parallelism* (MP). DP distributes batches over multiple servers (workers), each of which has a replica of a DNN model and processes assigned batches in parallel with other servers (see Fig. 2(a)) [64, 65, 92]. DP can reduce training time by utilizing the parallelism of multiple workers, but it suffers from high synchronization penalties to update the trained model across workers [89, 108]. While various approaches are proposed [64, 86], synchronization still causes non-trivial communication overheads, limiting scalability [85].

MP is used for training large DNN models that cannot fit into the memory of a single system. It involves partitioning the model into multiple parts and distributing them across different machines to be processed in a pipelined manner (see Fig. 2(b)) [79, 85]. However, high synchronization costs can negatively impact the training process, as each machine must keep the resulting weights in memory until the backward pass of the same batch. This often results in low utilization of multiple machines because only one or a few can be active [24, 50].

2.2 Drift Problem and Its Mitigation

A trained model provides high accuracy immediately after it is created, but the accuracy may drop as data changes over time. This phenomenon is called drift. Drift occurs when the relationship between inputs and outputs changes and/or the model's input distribution changes.

In order to mitigate the drift problem, many approaches have been proposed. Online learning (or incremental learning) updates the model on the fly as it processes one sample at a time. Online learning enables learning from data streams and seamlessly adapting to changing data. A critical challenge is that it is likely to forget past knowledge. The use of online learning is thus limited to specific streaming applications (e.g., weather forecasts and stock predictions), where historical data is less important in making decisions.

Another approach is to reflect historical and recent data through full training. The full training can be triggered when drift is detected or is performed regularly. The detection-based training may degrade the prediction quality as the training starts after sufficient drift is observed. Detecting drift is also challenging due to the presence of hidden factors. On the other hand, regular full training addresses these limitations by creating new models regularly. However, it consumes excessive computing and I/O resources, resulting in higher power consumption and the need for more servers and infrastructure. The long training time, which can reach 2 to 4 weeks, also impedes timely model updates.

3 Outdated Model and Label Problems

Processing complex data types, such as images, and developing applications that utilize these data requires advanced methods, not traditional analysis techniques. In this context, modern photo storage systems are increasingly integrating DL algorithms to enhance user experiences [12, 37, 76]. These systems employ DNN models to extract valuable information from images, facilitating applications such as content

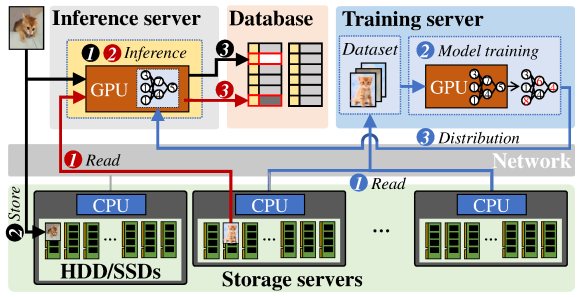
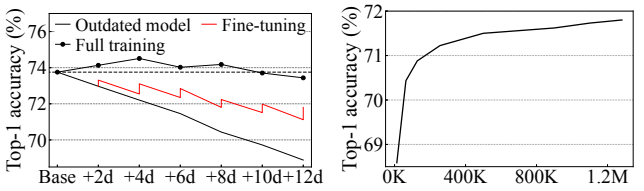


Fig. 3. Training and inference in photo storage systems



(a) Outdated model problem (d: day) **(b)** Impact of dataset size on accuracy of fine-tuning

Fig. 4. Challenges in DL integrated photo storage system

recommendation, categorization, and retrieval [40, 93, 104]. However, despite the many benefits of DNN integration, it also gives rise to specific challenges, i.e., (i) outdated models due to the 'drift' and (ii) outdated labels resulting from inferences made by now obsolete models. In this section, we discuss our empirical investigation that quantifies the effects of these outdated model and label problems on accuracy and the potential bottlenecks to address these challenges.

3.1 System Organization

Fig. 3 illustrates our target photo storage system, which is modeled after the production systems of Google Photos [36–38] and Amazon Photos [5, 12]. The system comprises four components: a training server, an inference server, a database, and storage servers. Using training datasets in storage servers (❶ in Fig. 3), the training server trains DNN models regularly (❷). The trained model is then delivered to the inference server (❸). When a new image comes, the inference server extracts its label using the model, which we call online inference (❶). The image is stored in the storage server (❷), and its label and location are indexed in the database to serve image search requests (❸). If necessary, the system performs offline inference to refresh outdated labels of previously stored images. This process involves reading images from storage servers (❶), extracting labels from a new model (❷), and updating the database (❸). For experiments, we use ResNet50 [43], which is widely deployed in production systems. As a benchmark, we use ImageNet-1K [91].

3.2 Impact of Outdated Model on Accuracy

To understand the impacts of drift on accuracy, we conduct experiments in which new images with new categories are added to the storage server. Following the approach shown in [70], the training server creates an up-to-date model biweekly and transfers it to the inference server. Based on the growth rate of data [73], we assume that the number of images stored increases by 1.78% daily, with 5.3% of newly added images belonging to new categories.

Fig. 4(a) shows the top-1 accuracy of the model over two weeks (a similar trend to this result is also observed in the top-5 accuracy (see Table 2)). We create an initial model using a training dataset with 937K images. The accuracy, measured using a test dataset with 50K images, is 73.8%. As new images are continuously added to the server over time, we evaluate the model accuracy every other day using new test datasets that reflect changes in the stored images. The accuracy gradually drops from 73.8% to 68.9%. It shows that the trained model becomes stale and fails to reflect recent changes. Performing full training more frequently may prevent the accuracy drop. Fig. 4(a) shows the model's accuracy in training it using the latest images every other day. The retrained model operates fairly well, offering the equivalent accuracy. However, it is infeasible to use in practice by the long training time.

Fine-tuning that adjusts the last few layers is a viable alternative to mitigate the burden of full training. We can prevent accuracy degradation by fine-tuning before its major update by full training. Fig. 4(a) depicts how much accuracy drops can be prevented by fine-tuning. We observe only an accuracy drop of 1.95% compared to the initial model. Fine-tuning cannot achieve the same accuracy as regular full training, but it still maintains a high enough accuracy. Unfortunately, even though fine-tuning is known to be less expensive than full training, it still involves many I/Os to feed a sufficient dataset to DNN models for higher accuracy [57]. As shown in Fig. 4(b), to provide noticeable accuracy improvement through fine-tuning, a large training dataset (e.g., more than 500K images) must be fed to the model.

3.3 Impact of Outdated Label on Accuracy

Image labels in a database become outdated as the model gets updated to reflect new relationships between inputs and outputs and to classify wider categories.

To understand the extent to which image labels become outdated labels as the model gets updated, we carry out experiments under the same setup in §3.1. We create the initial model, M_0, and assign labels to a set of 50K images using the model. As previously described in §3.2, we add new images to the storage server and perform full training biweekly to create up-to-date models, $M_1, ..., M_4$. We then use these new models to re-perform inference on the same 50K images, assigning more accurate labels. Finally, we evaluate

Table 1. % of labels fixed by new models

	M_0	M_1	M_2	M_3	M_4
% of fixed labels	0%	6.67%	7.29%	7.96%	8.98%

how many images incorrectly labeled by M_0 are fixed by $M_1, ..., M_4$. Table 1 shows the percentages of labels corrected by the new ones. 6.67% of the images have incorrect labels and are corrected by M_1. With the latest model, M_4, this number increases to 8.98%. It implies that the database contains a significant number of outdated and incorrect labels that should be updated.

The most fundamental way to reflect the enhancement of the newer DNN model is to perform offline inference again on previously stored images. Offline inference, however, involves significant data traffic because it requires fetching images from storage and performing inference on them.

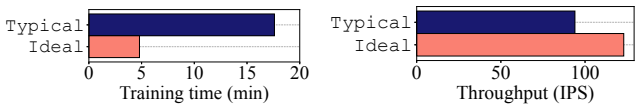
3.4 Bottleneck Analysis

As outlined in §3.1, NDPipe employs fine-tuning and offline inference as strategic solutions to the challenges of outdated models and labels, respectively. While these tasks are not computationally intensive, they entail significant data transfers, potentially creating a major bottleneck. To evaluate the extent of this bottleneck, we conducted comparative experiments between two system setups: `Typical` and `Ideal`.

`Typical`: This configuration represents a standard setup where a host server is networked with storage servers. For our experiments, we utilized a `p3.8xlarge` instance from Amazon Web Services (AWS) as the host system. This instance is equipped with four V100 GPUs and 32 vCPUs operating at 2.7GHz where we employed two of the V100 GPUs for the experiments. The host server was paired with four `g4dn.4xlarge` storage server instances, each originally featuring 16 vCPUs at 2.5GHz and a T4 GPU; but we disabled the GPUs for the experiments. The storage servers were configured with `st1` storage volumes, consisting of 16x HDDs organized in a RAID-5 array. The host and storage servers are communicated via a 10Gbps network.

`Ideal`: The `Ideal` system mirrors the `Typical` setup in terms of host server specifications but loads images directly from local memory, thus eliminating network traffic. This idealized scenario allows for the complete avoidance of network-related delays. Training in both systems utilizes a specifically curated set of images, separate from preprocessing steps, to align the input data with the DNN model. For fine-tuning purposes, we used the preprocessed ImageNet-1K dataset, averaging 0.59 MB per image. The offline inference tests were conducted using 1,000 images, each a typical 2.7MB JPEG file, to represent common photo storage system formats [51, 61].

Fig. 5 shows how the fine-tuning and offline inference procedures behave differently on the two setups. During the fine-tuning phase, the `Typical` system faces significant network overheads, leading to substantially extended training



(a) Fine-tuning **(b)** Offline inference

Fig. 5. Impact of network bottleneck

durations as shown in Fig. 5(a). For example, the training throughput in this configuration is 3.7× slower compared to the `Ideal` system, despite both setups employing identical GPUs. A similar trend is observed in the offline inference process as illustrated in Fig. 5(b). The `Typical` system processes only 94 images per second (IPS), which is considerably lower than the 123 IPS managed by the `Ideal` system.

It is important to note that these results also account for the impact of image preprocessing, a substantial contributing factor to the overall overhead, which will be explored in greater detail in §4. From these observations, we find the two following key insights: (i) network limitations are a decisive factor in hindering the efficiency of fine-tuning and offline inference in the typical systems, and (ii) merely utilizing additional GPUs for the host server does not correspondingly increase training or inference throughput.

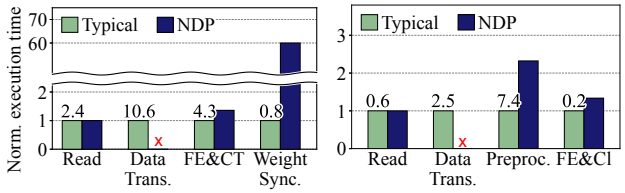
4 Challenges with NDP for Photo Storage

As identified in our bottleneck analysis in §3.4, data transfer is the main limiting factor in photo storage systems. In this context, NDP could be a potentially attractive solution to alleviate network bottlenecks and increase aggregate throughput by decentralizing computational capabilities and positioning them closer to data storage [18, 103]. However, we found that integrating NDP into photo storage systems is not straightforward, rather presenting unique challenges particularly when applied to fine-tuning and offline inference.

To illustrate our findings, we present our comparative analysis for the fine-tuning and offline inference procedures under two system configurations: `Typical` and `NDP`. The `Typical` system, as outlined in §3.4, consists of a host server connected to storage servers via a 10Gbps network. For the `NDP` system, we enabled GPUs in the storage servers, consisting of four `g4dn.4xlarge` instances, and processed images within these servers, bypassing the host server. This configuration allows direct data processing at the storage site, circumventing the network bottleneck. For these experiments, we utilize the same workloads as those used in §3.4.

4.1 Fine-tuning with NDP

In our exploration of fine-tuning with `NDP`, we break down the execution time of the process within a `Typical` setup, focusing on several key phases: (i) reading images from storage (`Read`), (ii) transferring them via a network (`Data Trans.`), (iii) executing DNN operations, including feature extraction



(a) Fine-tuning **(b)** Offline inference

Fig. 6. Execution times of DL tasks normalized to `Typical`. (The numbers above the bars are the actual times when conducted in a `Typical` setup: (a) in minutes and (b) in seconds.)

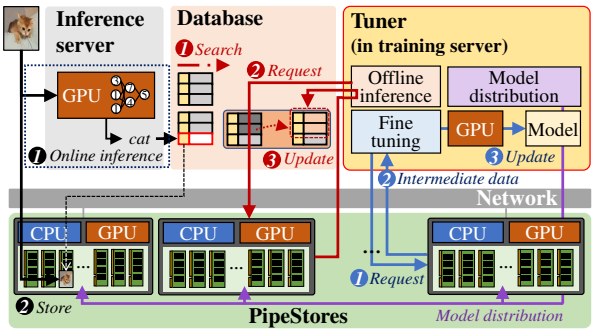


Fig. 7. Overall architecture and operations of NDPipe

and classifier training on the training server (`FE&CT`), and (iv) performing weight synchronization (`Weight Sync.`).

Fig. 6(a) presents our experimental findings, with the execution time for each subprocess in `NDP` normalized against that of `Typical`. The results highlight that `NDP` successfully bypasses the need for network-based data transfer. Additionally, we observed that DNN operations within the `NDP` system can be efficiently executed using low-end GPUs thanks to (i) the less demanding nature of fine-tuning compared to full training and (ii) the utilization of aggregated computational power from multiple storage servers. In FE&CT, only a 36% increase in execution time is observed with low-end GPUs.

However, we observe another challenge in using `NDP` for fine-tuning: the requirement for weight synchronization across the GPUs in multiple storage servers. This process incurs considerable overhead, forming a new bottleneck. A critical aspect exacerbating this challenge is the linear increase in synchronization costs as more storage servers are incorporated into fine-tuning. This finding indicates a scaling limitation within `NDP`: merely adding more NDP devices does not linearly enhance the fine-tuning performance.

4.2 Offline Inference with NDP

Our analysis of offline inference within the `NDP` framework involves a detailed timing study of each component during the processing of 1,000 images. We categorize the offline inference process into four stages, typically observed in most systems: (i) reading the raw image from disk (`Read`), (ii) transferring the image over a network to the host server (`Data Trans`), (iii) preprocessing the image for compatibility with the DNN model (`Preproc.`), and (iv) extracting the feature and then classifying the image using a DNN model (`FE&Cl`). In the `NDP` setup, we allocated a single CPU core for preprocessing tasks in each storage server, contrasting with the `Typical` system that uses eight cores for these tasks.

Fig. 6(b) illustrates the performance differences of each subprocess in `NDP` compared to `Typical`. Consistent with our findings from the fine-tuning analysis, the `NDP` system centralizes all subprocesses within the storage server, eliminating the data transfer overhead. `NDP` also takes advantage of the aggregate computational power of multiple GPUs within the storage servers, resulting in computation times that are only 1.33× longer than those of the `Typical` system.

Yet, `NDP` introduces a new challenge in preprocessing due to the limited CPU resources in the storage servers, which significantly impacts the overall inference speed. Allocating additional CPU resources for preprocessing in common storage servers is impractical, as it may adversely affect the fundamental operations of the storage services, such as handling read/write requests. This situation necessitates a systematic approach to address the preprocessing bottleneck without compromising the core functions of the storage servers.

5 Design and Implementation of NDPipe

In addressing the identified challenges in NDP within photo storage systems, we propose NDPipe, a system designed to effectively manage the outdated model and label problems with minimal overhead. NDPipe harnesses the potential of NDP in storage servers to enhance system performance. Also, NDPipe utilizes the same computational devices and software engines between near-data processing tasks (i.e., fine-tuning and offline inference).

Fig. 7 shows the architecture of NDPipe with two main components: a computational storage server (*PipeStore*) and a fine-tuning server (*Tuner*). PipeStore operates as a standard storage server but is equipped with a commodity GPU, enabling it to perform the fine-tuning and offline inference tasks near the data. This design choice is practically feasible, given existing storage servers capable of GPU integration [95]. Tuner manages many PipeStores and triggers fine-tuning and offline inference. It also performs part of the fine-tuning by utilizing a host-side GPU. We may run Tuner on the existing training server or a dedicated machine.

NDPipe uses fine-tuning that is expensive to run entirely on a storage server. To address the issue, NDPipe splits the model into two partitions, assigning a data-intensive one to a group of PipeStores and a computing-intensive one to Tuner. When training a model, Tuner sends training requests to multiple PipeStores (❶ in Fig. 7). PipeStores then extract *intermediate data* from local images, sending them to Tuner

(❷). Using the data received from PipeStores, Tuner tunes the model and generates an up-to-date model (❸). Since the intermediate data are much smaller than the original images, NDPipe can drastically reduce network traffic.

Once the model is updated, it is redistributed to each PipeStore, ensuring they possess the most current versions (as indicated by the purple lines in Fig. 7). To mitigate the overheads associated with distributing the entire model, we implement the Check-N-Run approach [29], which only transfers the compressed deltas (or differences) between models rather than the entire models themselves. Since changes in the fine-tuned model are confined to the last few layers, this method significantly reduces network traffic, achieving up to a 427.4× data traffic reduction.

With the fine-tuned models, NDPipe performs inference procedure in two contexts: online inference for newly uploaded images and offline inference to update labels for stored images based on the newly updated models. While the online inference happens in the typical DL system (❶~❷ in Fig. 7), offline inference is entirely performed in PipeStores without expensive external I/Os (❶~❸). For offline inference, Tuner sends inference requests to PipeStores that have target images. PipeStores perform offline inference locally and return extracted labels. Since only small labels are delivered, NDPipe eliminates the significant network traffic.

In designing NDPipe, we have addressed several technical issues existing in current photo storage systems. First, we have developed an FT-DMP mechanism designed to perform fine-tuning effectively with minimal overhead. FT-DMP enhances the parallelism of multiple PipeStores while reducing synchronization costs for separately running partitioned models (see §5.1). Second, we propose a pipelining strategy for FT-DMP, aimed at optimizing resource utilization and enhancing training performance, complete with a formal analysis of its feasibility (see §5.2). Third, we propose an automation tool called APO, enabling easy deployment of NDPipe and ensuring high performance and energy efficiency in fine-tuning. It identifies the optimal model partitioning points for FT-DMP and determines the number of storage servers that participate in training (see §5.3). Lastly, we have developed an NPE, specifically optimized for storage server environments, to facilitate fast training and inference in PipeStore with a storage-side accelerator (see §5.4).

5.1 Data and Model Parallelism in NDPipe

There have been previous attempts to partition DNN models to exploit data and model parallelism [56, 63, 90]. These studies have focused on accelerating only full training [90] or inference [56, 59, 63]. Fine-tuning, however, has unique architectures, which require us to take a different approach to partition models and execute tasks in parallel.

Data & Model Parallelism for Fine-tuning. Fig. 8 shows how FT-DMP partitions and distributes the model across

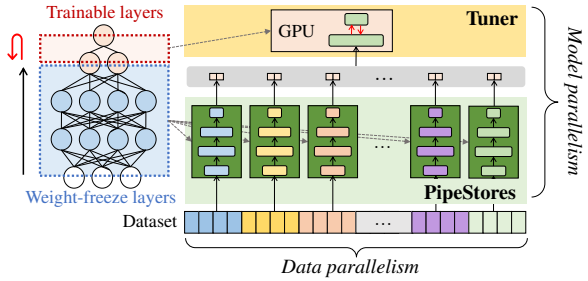


Fig. 8. Data and model parallelism of FT-DMP

PipeStores. NDPipe exploits the unique property of fine-tuning where the model's layers are split into weight-freeze and trainable layers. NDPipe assigns a replica of some or all of the weight-freeze layers to PipeStores, executing multiple workers. The rest of the layers, including trainable layers, are assigned to Tuner and processed by a single worker on Tuner. The individual PipeStores extract features for local batches and deliver them to Tuner, which processes the trainable layers. This design offers three benefits.

First, it is lightweight to run in a commodity GPU. The weight-freeze layers only require processing the forward pass of the network, similar to the inference process. Since low-end GPUs and accelerators are well suited for inference [14, 84], PipeStore can offer enough computing and memory capabilities for local training. Second, synchronization across PipeStores is unnecessary. No weight updates for weight-freeze layers exist, so workers need not synchronize weights. This property allows multiple workers across many PipeStores to execute in parallel without interfering with each other. As a result, NDPipe's performance scales linearly by adding more PipeStores. Trainable layers that need weight synchronization run locally on Tuner and thus produce up-to-date models without any network traffic. Third, it reduces network traffic. Instead of sending images to train, NDPipe transfers only the outputs from the last weight-freeze layer to Tuner. The output size of a layer typically decreases as the layer is located deeper into the model since it assimilates only fewer meaningful features and passes to the next layer.

Impact of Partitioning. To maximize FT-DMP's benefits, we need to make careful decisions about the number of layers offloaded to PipeStores. This choice affects both training time and network traffic. Offloading too many layers to PipeStores may overload them, so finding the right balance is crucial.

To gain insights, we conduct experiments with ResNet50, which comprises 50 layers grouped into five convolution layers (Conv1 to Conv5) and a fully-connected layer (FC) that serves as a classifier. We apply fine-tuning that only updates the weights of the FC layer. We use the optimized PipeStore (as explained in §5.4) with one Tesla T4 GPU. We connect a group of four PipeStores to Tuner with one V100 GPU using 10Gb Ethernet. Throughout the experiments, we

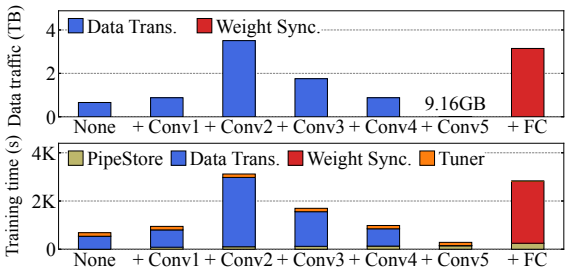


Fig. 9. Impact of layer offloading and data traffic

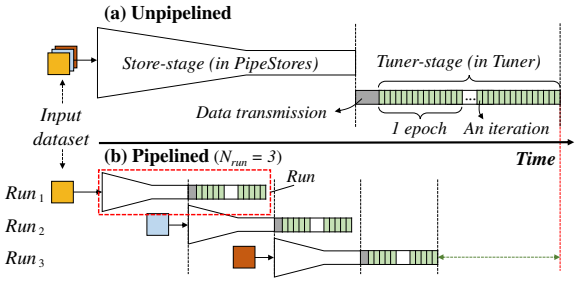


Fig. 10. Unpipelined (a) and pipelined (b) FT-DMP

measure the training time and the amount of data transferred over the network. For detailed experimental setups, see §6.1.

Fig. 9 shows the experimental results. None means that no layers are offloaded. PipeStore forwards raw images to Tuner, and the entire training is done by Tuner. +Conv1 offloads the Conv1 layers onto PipeStores, and Tuner handles the rest. +FC is an extreme case where PipeStore runs all the layers. As mentioned earlier, the network traffic to send extracted features to Tuner gradually decreases as more layers are offloaded to PipeStore. However, traffic surges after offloading the FC layer (+FC) due to high weight synchronization costs. As a result, for ResNet50, NDPipe achieves the shortest training time after offloading +Conv5. We discuss how the model partitioning can be automatized in §5.3.

5.2 Pipelined Training of Partitioned Model

In NDPipe, all the batches are distributed across multiple PipeStores. Each PipeStore trains independently using its local batches and then transfers intermediate results to Tuner. Tuner gathers and stores the results in its local storage temporarily before starting the training epoch. Following the DNN training procedure that requires the entire training data [58, 78], we may wait for the intermediate results corresponding to all data samples from the participated PipeStores. This results in the serial execution of PipeStores (Store-stage) and Tuner (Tuner-stage), as shown in Fig. 10(a).

Inspired by pipelined model parallelism [50, 79, 85] that executes split partitions concurrently, we propose a pipelined training strategy for FT-DMP. As shown in Fig. 10(b), the pipelined FT-DMP executes multiple runs simultaneously over separate sub-datasets. We denote the number of runs by N_{run}. The pipelined FT-DMP starts the training in Tuner for the current run, while PipeStores are processing local batches for the next run. The training throughput improves as more PipeStores participate. However, assigning too many PipeStores wouldn't be beneficial as the bottleneck moves to Tuner, making PipeStores idle and thus wasting energy. It is best practice to balance the pipeline stages (Store- and Tuner-stages) to achieve high throughput and energy efficiency at a low HW cost. To this end, we have devised an automated tool that identifies the best number of PipeStores for a given model and hardware specifications for PipeStore and Tuner. We will explain the details in §5.3.

A potential concern with the pipelined training strategy is the impact on the learned model quality compared to the vanilla model trained without pipelining. In fine-tuning, it is a common issue known as catastrophic forgetting [21, 60], where knowledge obtained from previous training is forgotten while updating the model with newly observed data. However, our theoretical analysis confirms that the pipelined FT-DMP can still guarantee the convergence of training and not seriously affect the final model quality with a reasonable number of pipeline runs, N_{run}.

Converge Analysis. Before the proof, we assume the following conditions are satisfied: (i) the weights of the pre-trained model are well-trained, (ii) the unpipelined training process (i.e., $N_{run} = 1$) converges if performed, and (iii) sub-datasets used over different runs have similar distributions.

Let us consider a vanilla neural network having N fully-connected layers, with $W_1, ..., W_N$ where $W_j \in \mathbb{R}^{d_j \times d_{j-1}}$. The analysis in [13] showed that a vanilla network converges under the following assumptions. (A) The dimensions of hidden layers are at least the minimum of the input and output dimensions (i.e., $d_i \leq \min\{d_0, d_N\}$ where $1 \leq i \leq N-1$). (B) The training starts with the approximately balanced weights with a parameter of δ, called δ-balanced and defined by $\|W_{i+1}^\top W_{i+1} - W_i W_i^\top\|_F \leq \delta$ and $\|A\|_F$ is the Frobenius norm of a matrix A. (C) The initial loss is bounded by the loss of the rank-deficient solution, denoted by ϵ.

In our case of pipelined training, baseline assumptions (A) and (B) are intuitively met. The classifier with hidden layers larger than the input/outputs satisfies (A). Also, (B) holds by our condition (i) for the FT-DMP convergence, which is aforementioned as the *well-trained* model in §5.2. We define that a model is well-trained if it is δ-balanced. The work in [13] shows that the weights remain balanced during gradient descent if they start δ-balanced, ensuring approximate balance at the start of each pipelined training run.

In the rest of this section, we detail the proof for the last baseline assumption (C). By condition (ii), we assume convergence with an initial loss of ϵ for unpipelined training. For the pipelined training, the first run will also converge since it satisfies the baseline assumptions (A) and (B) and

starts with the same initial loss of ϵ. We denote by $l^1(T_1)$ the loss converged at iteration T_1 during the first run, and $l^1(T_1) < \epsilon_1 (= \epsilon)$. With the conditions above, we claim that the second run starting with initial loss $l^1(T)$ also converges:

Theorem 5.1. *Assume that the gradient descent is initialized such that the weights have a deficiency margin [13], $c > 0$ and are δ-balanced. Also, suppose that the second run starts with the weights converging in the first run by $l^1(T_1)$ and the inter-run loss difference, Δ. Then, with the minimum learning rate requirement for η in Eq. 7 of [13], $\forall \epsilon_2 > 0$ and*

$$T_2 \geq \frac{1}{\eta \cdot c^{2(N-1)/N}} \cdot \log(\frac{l^1(T_1) + \Delta}{\epsilon_2}), \tag{1}$$

the loss at iteration T_2 is no greater than ϵ_2. In other words, the second run is guaranteed to converge with the loss of ϵ_2.

Theorem 5.1 applies to both the first and second runs, simplifying the explanation, but it also ensures convergence for all other runs through its inductive step. That is, as long as a p-th run converges at T_p with the final loss $l^p(T_p)$, any $(p+1)$-th run starting with the weights trained in the previous run will also converge. Generalizing Theorem 5.1 with $l^1(T_1) = l^p(T_p)$ and $l^2(T_2) = l^{p+1}(T_{p+1})$ accomplishes the induction, and this completes the proof of the convergence of our pipelined training. To establish the claim, we describe the inter-run loss difference Δ in the following lemma.

Lemma 5.2. *For a given confidence level θ,*

$$l^2(0) \leq l^1(T_1) + \Delta$$

where $\Delta = \sqrt{\frac{1}{2m} log(\frac{2P}{\theta})}$, P is the number of total weights in the model, and m is the number of training dataset samples.

Proof of Lemma 5.2. By Hoeffding's inequality [45], the final loss of the first run and the initial loss of the second run satisfy the following relationship with a probability bound ε:

$$\mathbf{P}(|l^2(0) - l^1(T_1)| \geq \varepsilon) \leq 2\exp(-2m\varepsilon^2). \tag{2}$$

We can write the union bound of the right side of Eq. 2 to generalize over all the potential weights in the model, P:

$$2P \cdot \exp(-2m\varepsilon^2) \leq \theta \Rightarrow \sqrt{\frac{1}{2m}\log(\frac{2P}{\theta})} \leq \varepsilon. \tag{3}$$

where θ is the confidence of the union bound, which we earlier denoted in Lemma 5.2. Finally, if we incorporate Eq. 3 into Eq. 2, this completes the proof of Lemma 5.2. □

Proof of Theorem 5.1. By substituting Lemma 5.2 and Eq. 13 in [13] into Eq. 1, we then get that

$$\begin{aligned} \epsilon_2 &\geq (l^1(T_1) + \Delta) \cdot \exp(-\eta \cdot c^{\frac{2(N-1)}{N}} \cdot T_2) \\ &\geq l^2(0) \cdot \exp(-\eta \cdot c^{\frac{2(N-1)}{N}} \cdot T_2) \\ &\geq l^2(T_2) \end{aligned} \tag{4}$$

By Equation 4, we can ascertain that the l^2 can be bounded by an iteration T_2, thereby concluding our proof. □

Note that the actual loss difference, Δ, largely corresponds to the difference between the two sub-datasets used in the first and second runs. For example, when the sub-datasets have very similar distributions, we will have a small Δ. Thus, our condition (iii) is necessary to make Δ small enough, resulting in a small initial loss for each run and eventually achieving high training accuracy comparable to the unpipelined training. In §6.3, we will also empirically show that the pipelined FT-DMP provides fairly high accuracy.

Algorithm 1 Find the best number of PipeStores

Input: DNN model architecture (M), FLOPS of PipeStore and Tuner (F_P and F_T), network bandwidth (BW), the maximum number of PipeStores (N_{ps}^{max})

Output: the best number of PipeStores (N_{ps}^{best})

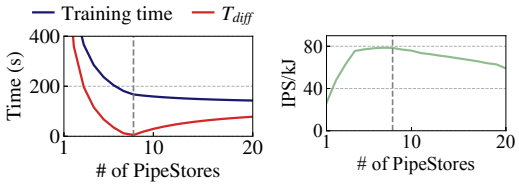
1: $N_{ps}^{best} \leftarrow 0$; $T_{min} \leftarrow \infty$
2: **for** $N_{ps} \leftarrow 1$ **to** N_{ps}^{max} **do**
3: $\quad p, T_{ps}, T_{tuner} \leftarrow$ FindBestPoint(M, F, N_{ps}, BW)
4: $\quad T_{diff} \leftarrow |T_{ps} - T_{tuner}|$
5: $\quad$ **if** $T_{diff} < T_{min}$ **then**
6: $\quad\quad T_{min} \leftarrow T_{diff}$; $N_{ps}^{best} \leftarrow N_{ps}$
7: $\quad$ **end if**
8: **end for**
9: **return** N_{ps}^{best}

5.3 Identifying Best Organization for NDPipe

Deploying NDPipe efficiently necessitates a strategic combination of methodologies discussed in §5.1 and §5.2, particularly in structuring its core components to optimize training throughput and energy efficiency. In pursuit of this balance, we developed the Automated model Partitioning and Organization (APO) tool. The primary objective of APO is to determine the best model partitioning point and the most suitable number of PipeStores required for use. This ensures that both PipeStores and Tuner operate efficiently with minimal pipeline bubbles, factoring in specific DNN models and respective hardware specifications of PipeStore and Tuner. The operational flow of APO is detailed in Algorithm 1.

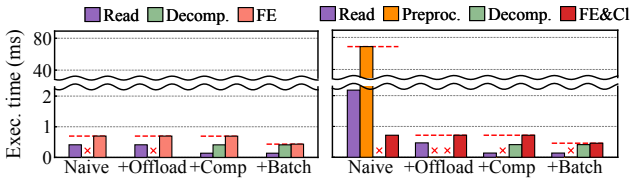
APO begins by setting N_{ps}^{best} to zero and T_{min} to infinity, then iteratively considers various PipeStore numbers (Lines 1–2). In each iteration, it calls `FindBestPoint()` (Line 3), which determines the best partitioning point by evaluating four key input parameters: the DNN model's architecture (M), the FLOPS of the PipeStore-side accelerator (F_P) and Tuner-side GPU (F_T), the network bandwidth between PipeStore and Tuner (BW), and the number of PipeStores engaged in fine-tuning (N_{ps}).

`FindBestPoint()`, inspired by prior model partitioning studies for inference [56] or full training [53], is uniquely tailored for the fine-tuning process in NDPipe to identify the best suitable partitioning segments for the given inputs. It identifies partitionable points in the model, which do not include areas with residual blocks and skip connections [43].



(a) Training time **(b)** Energy efficiency

Fig. 11. Training and energy efficiency by # of PipeStores



(a) Fine-tuning **(b)** Offline inference

Fig. 12. Elapsed time for each task on a PipeStore

APO calculates FLOPs at each partition point to estimate execution times for the partitioned segments on the PipeStore-side and Tuner-side using their respective FLOPS parameters. It also measures the output data size for each model layer.

For each candidate, we predict the training time, factoring in data transfer times over the network based on the assessed output data sizes. This estimation includes the impact of the pipelining strategy detailed in §5.2, where a group of PipeStores and Tuner operate in a pipelined fashion. To prevent weight synchronization among the PipeStores, the trainable layer is assigned to be in the Tuner. The optimal partitioning point is then determined as the one that results in the shortest training time.

`FindBestPoint()` then returns the best partitioning point p, and the execution times T_{ps} for PipeStores and T_{tuner} for Tuner. APO updates T_{min} and sets N_{ps}^{best} to the current iteration value if the difference in execution times, T_{diff}, is smaller than the current T_{min} (Lines 4–7). This process is repeated until N_{ps} equals N_{ps}^{max}, the maximum number of PipeStores. Finally, APO outputs the most effective number of PipeStores, N_{ps}^{best}, that ensures optimized performance.

To explain how APO determines the best organization, as an example study, Fig. 11 shows the training time and energy efficiency of NDPipe with varying number of PipeStores for ResNet50. The results present that when employing a few PipeStores, NDPipe suffers from prolonged training time due to the limited data supplied by PipeStores, leading to underutilization of Tuner. When the number of PipeStores is 8, which is chosen by APO, T_{min} approaches zero, significantly reducing the training time. Adding more PipeStores beyond this number is redundant as the training time remains constant since Tuner becomes a bottleneck. Consequently, the energy efficiency, measured as training throughput-per-joule (IPS/J), decreases as more PipeStores are underutilized.

5.4 Optimization of Near-data Processing Engine

PipeStore is a crucial component for the system efficiency since it handles fine-tuning and offline inference on the same hardware. To optimize the PipeStore's performance, we analyzed the execution times of its major tasks with a comparison study to the results reported in Fig. 6.

Fig. 12 shows our findings. `Naive` represents the average execution time of each task on one PipeStore, without any optimizations. For fine-tuning, the FE task becomes the main bottleneck, as weight synchronization, which is the major bottleneck in the naive NDP as discussed in §4.1, is now migrated to Tuner. In contrast, offline inference, which processes large raw images, experiences longer I/O time and needs more CPU resources for preprocessing. As a result, image loading and preprocessing tasks occupy a substantial portion of the total inference time. Based on these observations, we design the NPE with the following techniques.

3-stage Pipelining. Pipelining is a straightforward yet highly effective method to improve the throughput of NDPipe. We divide the fine-tuning and offline inference process into three stages: data loading (in SSD or HDD), preprocessing (in CPU; only needed for offline inference), and FE&Cl (in GPU), which can run in parallel across different HW components. The execution of each stage is organized in a pipeline format for batches of input data. That is, while the data-loading stage is reading data for $batch_{n+2}$, the preprocessing stage simultaneously processes $batch_{n+1}$, delivering the intermediate data to the FE&Cl stage processing with $batch_n$. Such a pipelined execution maximizes the utilization of each system unit and, as a result, improves the throughput of NDP.

Offloading Preprocessing to the Host. To alleviate the preprocessing bottleneck in PipeStore, we have implemented a strategy of offloading these tasks to an inference server. When new images arrive, the inference server performs online inference, applying the preprocessing step. We modify the inference server to deliver the preprocessed data and its raw image originally stored in storage servers. This approach removes the preprocessing overhead on the PipeStore side (see `+Offload` of Fig. 12). Nevertheless, it comes at the cost of storage overheads due to the preprocessed binaries stored additionally. In our analysis, preprocessed binaries account for 17.5% of the total storage space when the average image size is 2.7MB. To mitigate the storage waste, we instead store compressed data in PipeStore using a deflate algorithm [26]. Note that this compression also contributes to a reduction in I/O time for both the fine-tuning and inference processes.

While data compression benefits storage efficiency, it introduces computational demands for decompression within PipeStore. However, in our observation, dedicating a modest amount of computing resources to this task, i.e., allocating a

maximum of two CPU cores for decompression, can mitigate these demands since FE&Cl hides the overhead.

Enlarging Batch Size. The use of offloading techniques shifts the primary bottleneck to FE&Cl on the GPU (see `+Comp` of Fig. 12). We shorten the execution time of FE&Cl by enlarging a batch size to enhance the throughput of the FE&Cl process. For instance, with ResNet50, we set the batch size to 128 to balance each stage's duration (See `+Batch`). In PipeStore, where only weight-freeze layers are processed, there is flexibility in selecting the batch size. However, choosing a reasonable batch size is important as larger batch sizes require more memory space. We will explore the implication of various batch sizes on the system performance in §6.4.

6 Evaluation

6.1 Experimental Setup

We implement NDPipe on Amazon EC2 [6]. For PipeStore, we utilize the `g4dn.4xlarge` instance with a Tesla T4 GPU [82]. `st1` with 16× HDDs is used for a storage volume. A T4 GPU is chosen as it is the most cost-effective option available on EC2 for GPU. Lightweight inference accelerators could be a viable option for PipeStore. To prove this, we also implement PipeStore on the AWS Inferentia [8] and evaluate its performance (see §6.4). The AWS Inferentia has a similar hardware specification as `g4dn.4xlarge` but with an inference accelerator, NeuronCoreV1 [10], instead of T4 GPU.

For Tuner, we use the `p3.2xlarge` instance with one Tesla V100 GPU [81]. We use at most 20 PipeStores and one Tuner. We set the network bandwidth to 10Gbps by referring to commercial cloud systems [7, 98] that provide network bandwidth of 1–16Gbps. We also carry out experiments while varying the network bandwidth from 1Gbps to 40Gbps (see §6.4). The inference and fine-tuning engines of PipeStore are TensorRT 7.1.3 [83] for T4 and PyTorch-Neuron [11] for NeuronCoreV1. For Tuner, we use TensorFlow 2.6.0 [3] to implement the host-side fine-tuning engine. The batch size is set to 128 and 512 for inference and training, respectively.

We set up a typical host server system (denoted by SRV) that performs both inference and fine-tuning on the host side. We use the `p3.8xlarge` with two V100 GPUs. This computing capability is 2× higher than Tuner. For a fair comparison, we apply the same optimizations from §5.4 to the inference and fine-tuning engine. The storage server for SRV uses `g4dn.4xlarge` and `st1` with the same specifications as StorePipe. The GPU of the storage server is disabled.

Benchmarks. We use ImageNet-1K [91] as a default dataset. For a more comprehensive analysis of accuracy, we also use the ImageNet-21K [25] and CIFAR100 [62] datasets. Our experiments are conducted using five image classification models (ShuffleNetV2 [107], InceptionV3 [96], ResNet50 [43], ResNeXt101 [101], and ViT [28]). The limited space does not permit us to present all results; instead, we present results from InceptionV3, ResNet50, ResNeXt101, and ViT, which represent small-, middle-, large-scale CNN-based models, and large-scale transformer-based model, respectively.

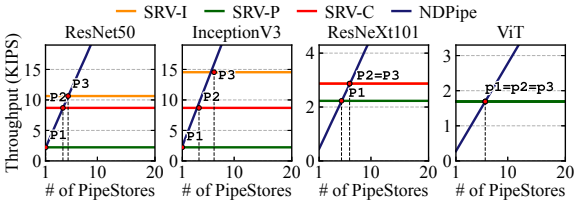


Fig. 13. Comparison of inference throughput

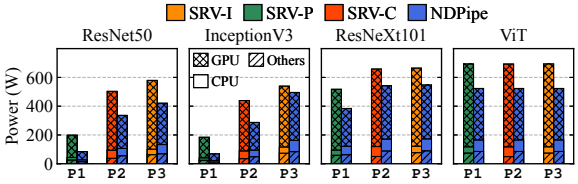


Fig. 14. Comparison of inference power consumption

6.2 Inference Performance Analysis

We evaluate the inference throughput of NDPipe, focusing on scalability and energy efficiency. Tuner was not activated as offline inference is entirely made by PipeStores.

We compare NDPipe with three configurations of the typical DL system: `SRV-I`, `SRV-P`, and `SRV-C`. First, `SRV-I`, an ideal system, keeps all preprocessed image binaries locally to infer in the host storage. `SRV-I` thus does not need to load binaries from the network. `SRV-I` is not a practical setup, but we include it for comparison. Second, `SRV-P` loads preprocessed binaries from the storage server over the network. Third, `SRV-C` is similar to `SRV-P` but uses compressed binaries to reduce network traffic. Thus, inputs need to be decompressed before being fed to GPUs. We assign eight cores to the host server for decompression. We evaluate the systems' throughputs (Images Per Second), varying the number of PipeStores or typical storage servers from 1 to 20.

Fig. 13 shows our results. `SRV-I` exhibits higher throughput compared to the other systems. This is because `SRV-I` can fully exploit the available GPU throughput without being limited by the network. `SRV-C` has higher throughput than `SRV-P` by reducing network traffic through compression.

The performance of NDPipe scales linearly with more PipeStores used. Each PipeStore offers 2,129, 2,439, 449, and 277 IPS for ResNet50, InceptionV3, ResNeXt101, and ViT, respectively. At point `P1`, NDPipe with 1–7 PipeStores surpasses `SRV-P` by sending only small-sized labels, mitigating network overhead. At `P2`, where 4–7 PipeStores are deployed, NDPipe outperforms `SRV-C`. NDPipe shows even higher throughput than `SRV-I` with 5–7 PipeStores. This indicates that the aggregate throughput of PipeStores exceeds that of two V100 GPUs. ResNeXt101 and ViT have different trends from ResNet50 and InceptionV3. ResNeXt101 and ViT

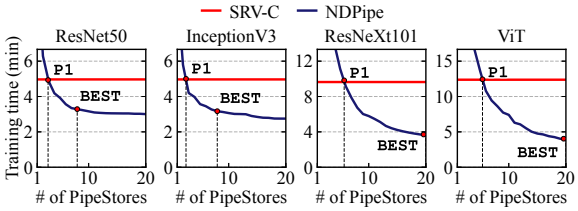


Fig. 15. Comparison of training time

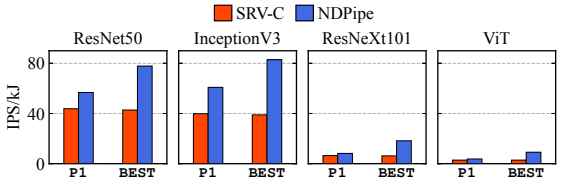


Fig. 16. Comparison of energy efficiency

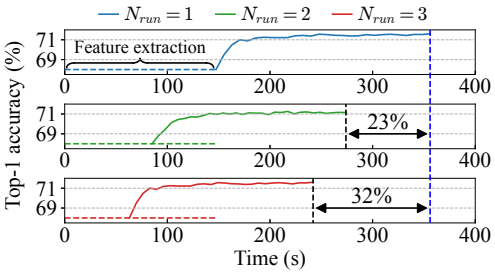


Fig. 17. Accuracy of pipelined FT-DMP

are huge models requiring high computing power to process their layers. Two V100 GPUs are insufficient to process data supplied from storage servers, so the GPUs become a bottleneck. This is why SRV-I, SRV-C, and SRV-P show similar throughputs.

We now analyze the energy efficiency of NDPipe over the typical systems. We measure the power consumption of CPUs, GPUs, and other components (e.g., power supply, SoC, I/O, and so on) while executing inference tasks. To measure the power consumption of GPUs, we use `gpustat`. AWS does not permit us to measure the power consumption of CPUs and other components. We thus set up local server machines that have specifications similar to our AWS instances and then measure the power consumption of CPUs and other components using `powerstat` and `ipmitool`.

Fig. 14 compares the average power consumption of the systems at P1, P2, and P3, where NDPipe exhibits similar performance as SRV-P, SRV-C, and SRV-I, respectively. Fig. 14 shows the portions of three major components. NDPipe shows 1.83× and 1.39× higher power efficiency, on average, than SRV-P and SRV-C, respectively. SRV-P and SRV-C are often bottlenecked by the network and thus fail to fully utilize GPU and CPU resources, leaving them idle, which causes a waste of power. Even compared to SRV-I where the network is not a bottleneck, NDPipe exhibits higher power efficiency due to the power efficiency of commodity GPUs.

6.3 Training Performance Analysis

We next analyze the training throughput of NDPipe. For all the models, we used fine-tuning that trains the classifier. To complete the evaluation in a reasonable time, we use a relatively small training dataset, 1.2M images from ImageNet-1K. We compare NDPipe with SRV-C, which has the most realistic setup among the comparative systems. NDPipe uses Tuner to perform training, and N_{run} is set to 3.

Fig. 15 shows the results. As we add more PipeStores to the system, NDPipe begins to outperform SRV-C (P1 in Fig. 15). For ResNet50 and InceptionV3, NDPipe exhibits faster training speed with three PipeStores. NDPipe outperforms ResNeXt101 when it has six PipeStores. Compared to ResNet50 and InceptionV3, ResNeXt101 has very deep feature extraction layers, requiring high computing capability. By adding more PipeStores, NDPipe can reduce the training time further. BEST in Fig. 15 represents a point where NDPipe offers the maximum training throughput-per-joule (IPS/J). The reduction in the training time becomes marginal beyond a certain number. This is expected because a bottleneck moves to Tuner due to the unbalanced lengths of pipeline stages, as discussed in §5.2. Instead, NDPipe consumes more energy because it has to maintain more PipeStores.

Fig. 16 plots the energy efficiency of the different systems when NDPipe shows a similar training time as SRV-C (P1) and the best throughput-per-joule BEST). Note that, for NDPipe, we also include the energy consumption of Tuner. Compared to SRV-C, NDPipe exhibits on average 1.44× and 2.64× higher energy efficiency at P1 and BEST, respectively.

Impact of Pipelined Training. We evaluate the impact of the pipelined FT-DMP on the accuracy, varying N_{run} from 1 to 3. We stop the training when more than 0.01% accuracy improvement is not observed over three consecutive epochs. We evaluate ResNet50 with four PipeStores.

Fig. 17 shows the training time depending on N_{run}. If N_{run} is 1, the training is the same as the original FT-DMP. With pipelined FT-DMP, NDPipe can reduce the training time by up to 32% with negligible accuracy loss. The accuracy of the original FT-DMP is 71.61%; when the training phase is divided into two and three runs, the accuracy is maintained at 71.55% and 71.52%, respectively. However, when the training phase is split into four runs, the accuracy significantly drops to 70.36%. As we divide the training phase more, the number of images supplied to each run gets smaller, which makes the impact of catastrophic forgetting more serious.

Accuracy. To understand how NDPipe keeps its model accuracy against drift, we conduct experiments under the same setup in §3.2. We simulate a scenario where data accumulates by 1.78% daily over two weeks. To simulate this, our initial model trains with only 78% of the total dataset, and the

Table 2. Comparison of model accuracy (%)

Dataset		ShuffleNet V2		ResNet 50		Inception V3		ResNeXt 101		ViT	
		Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
CIFAR100	Base	67.39	89.43	77.15	94.08	77.21	94.10	78.27	94.56	86.41	96.12
	Outdated	**64.12**	**87.35**	**74.73**	**91.73**	**74.93**	**92.29**	**73.65**	**92.82**	**81.73**	**94.77**
	NDPipe	**65.07**	**88.79**	**75.2**	**93.11**	**76.82**	**93.50**	**76.9**	**94.23**	**84.46**	**95.69**
	Full	69.51	90.64	77.39	93.96	79.00	94.55	77.78	95.01	87.60	96.47
ImageNet 1K	Base	65.32	84.62	73.75	91.38	73.71	91.18	75.15	92.23	80.45	93.43
	Outdated	**62.25**	**79.52**	**68.89**	**85.35**	**68.88**	**85.18**	**70.11**	**86.10**	**77.36**	**89.17**
	NDPipe	**63.12**	**80.45**	**71.80**	**90.44**	**71.82**	**89.92**	**72.86**	**90.81**	**79.37**	**93.88**
	Full	66.90	86.08	73.68	91.42	75.38	92.56	78.74	94.02	81.61	94.46
ImageNet 21K	Base	22.12	45.71	36.24	66.77	36.49	66.79	38.38	69.7	46.11	75.72
	Outdated	**20.72**	**44.10**	**34.31**	**63.76**	**34.63**	**63.24**	**38.07**	**65.31**	**36.59**	**70.14**
	NDPipe	**21.25**	**44.98**	**35.97**	**66.51**	**35.57**	**66.2**	**38.24**	**66.82**	**44.52**	**74.26**
	Full	22.07	45.60	36.78	66.96	36.02	66.55	38.79	68.67	-	-

subsequent model training after two weeks with the whole dataset. Table 2 compares the top-1 and -5 accuracies of NDPipe with three cases: (i) the initial accuracy that is measured when the model is just created (`Base`), (ii) the accuracy after two weeks without any model updates (`Outdated`), and (iii) the accuracy after two weeks with full training (`Full`). `Full` is impractical due to its long training time. The results of `Full` for ViT on ImageNet-21k are not included because of its long training time over large datasets. The results suggest that NDPipe provides fairly good accuracy, outperforming `Outdated` on every dataset. NDPipe also provides similar or slightly lower accuracy than `Full`. This implies that NDPipe solves the outdated model problems cost-effectively.

6.4 Impact of System Parameters

Network Bandwidth. Fig. 18 compares the inference throughput-per-watt (IPS/W) of NDPipe with `SRV-C` while increasing the network bandwidth (BW) from 1 to 40Gbps (we exclude InceptionV3 and ViT because it shows similar results as ResNet50 and ResNeXt101). `SRV-C` suffers from a serious network bottleneck when its bandwidth is limited to 1Gbps. As the network bandwidth increases, the performance of `SRV-C` gets improved. However, beyond 20Gbps, `SRV-C` cannot show any performance improvement. This is owing to high decompression overhead. Eight CPU cores (dedicated to decompression) are not able to provide sufficient throughput to process large amounts of input data fed via the fast network. The simplest solution is assigning more CPU cores for decompression, but it is costly and causes more power consumption. NDPipe is freed from network bottlenecks as small-sized labels or features are transmitted. As a result, it provides fairly good throughput, achieving 3.7× and 1.3× higher performance at 1Gbps and 40Gbps, respectively.

Batch Size. Fig. 19 shows the inference throughput of NDPipe with varying batch size (BS) from 1 to 512. With the batch size of 1, NDPipe shows poor performance as the GPUs in PipeStores are not fully utilized. As the batch size grows, the inference throughput gradually improves. However, beyond a certain batch size, the improvement becomes

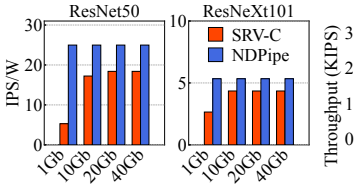


Fig. 18. Impact of BW

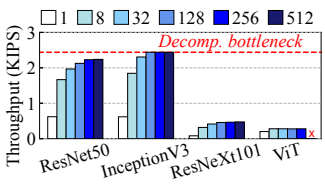
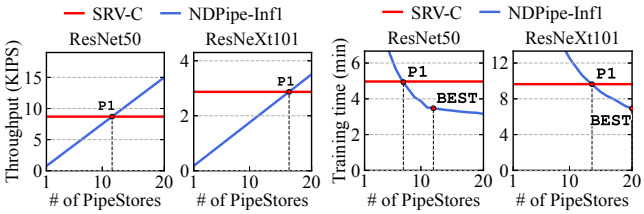


Fig. 19. Impact of BS



(a) Offline inference **(b)** Fine-tuning

Fig. 20. Performance of NDPipe implemented on Inferentia

marginal. For InceptionV3, when the batch sizes are over 128, no performance improvement is observed as decompressing images by CPU becomes a bottleneck in the 3-stage pipelining (see §5.4). For ViT (a large model), PipeStore encounters Out-of-Memory (OOM) errors with large batch sizes. This issue arises not only due to the large batch size but also because of the high memory requirements of the model itself.

Impact of Accelerator. We implement NDPipe with NeuronCoreV1, a low-end but cost-effective AI accelerator in AWS Inferentia instances (`Inf1.2xlarge`). We evaluate the performance and power or energy efficiency of NDPipe with NeuronCoreV1 (denoted by NDPipe-Inf1) for offline inference and fine-tuning. The AWS Inferentia does not provide a way of measuring the power consumption of NeuronCoreV1. We thus estimate its power usage by referring to [52].

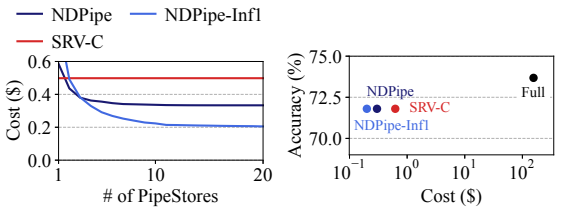
Fig. 20 shows the results. The NeuronCoreV1 has lower computation capabilities than T4. To achieve the same level of performance as SRV-C, NDPipe needs more PipeStores: 11–16 and 8–13 PipeStores for offline inference and fine-tuning, respectively. However, thanks to the energy-efficient design of NeuronCoreV1, NDPipe-Inf1 still provides 1.17× and 1.5× higher power and energy efficiency for offline inference and fine-tuning than SRV-C, respectively, on average.

7 Discussion

7.1 Extension of NDPipe to Other Systems

In this paper, we have focused on enhancing user experience in photo storage systems through efficient inference and fine-tuning. However, the concept of NDPipe can be extended to various media formats such as video, audio, and document.

Video content. One issue with utilizing DL applications for video formats is that they require more resources than images due to larger content sizes. One of the solutions to



(a) Fine-tuning cost **(b)** Cost versus accuracy

Fig. 21. Operational cost analysis of NDPipe

address the issue is frame extraction, which extracts key frames from videos for analysis [39]. These key frames are analyzed using a CNN model to label content, creating a summary vector for further video analysis. NDPipe can adapt to video by preprocessing with frame extraction and processing frames as NDPipe does for photos. This extension enhances the search for videos in online archives [31, 37] and recommendations on streaming platforms [80, 106].

Audio content. NDPipe can be adapted for audio formats through audio spectrogram transformation (AST) [19, 35], converting audio frequency data into visual representations. This process enables the use of image-based CNNs or transformer models for audio applications such as genre classification [88], mood detection [102], and speaker identification [55] by transforming audio into spectrograms.

Document content. For documents, NDPipe uses NLP techniques for enhanced document storage, converting text into analyzable embedding vectors via recent transformer-based models [16, 27]. These embeddings then serve as inputs for various downstream tasks, such as document classification [4] and sentiment analysis [75], conducted by Tuner. This approach can reduce data transfer costs by converting large documents into small embedding vectors.

7.2 Cost Analysis of NDPipe

To assess the cost-effectiveness of NDPipe, we have estimated the operational costs of NDPipe (including NDPipe-Inf1) and SRV-C using the AWS pricing tool [9]. Fig. 21(a) shows the cost of running fine-tuning for ResNet50. For NDPipe, we add up the costs of Tuner and PipeStores, and for SRV-C, the costs of the training and storage servers are included. When the number of PipeStores is too small, the NDPipe's cost is higher than SRV-C owing to the long training time with limited PipeStores. As the number of PipeStores increases, the cost decreases with the reduced time for fine-tuning.

Fig. 21(b) illustrates the cost versus accuracy for ResNet50. We compare two training strategies: fine-tuning and full training. Full training is conducted under the SRV-C setup. We run 90 epochs with the batch size of 128. Full training shows the highest accuracy but is achieved by a long training time. NDPipe and SRV-C, which employ fine-tuning, show slightly lower accuracies but greatly reduce training costs. Compared to SRV-C, NDPipe and NDPipe-Inf1 exhibit 1.5× and 2.5× lower costs, respectively, by shorter training times.

8 Related Work

Near-data processing approaches for optimizing DL performance have been studied by many researchers. Neurosurgeon [56] automatically identifies the model partition points and orchestrates the distribution of computation between the mobile device and the data center to optimize the inference performance. However, it considers neither multi-device scenarios nor fine-tuning for model training. Deep partitioned training [53] leverages NDP and feeds only small features to NPUs to enhance full training performance, yet it has limited scalability due to the high synchronization costs.

In the context of the DL pipelining, NVIDIA Triton [74] pipelines the preprocessing, model execution, and postprocessing for higher throughput. However, all the computation is performed in GPU and does not consider a resource-constrained system. ZeRO-Offload [90] offloads the memory-intensive DL process to CPUs and pipelines the training process over CPU and GPU. However, it does not consider balancing pipeline stages executed by different hardware.

There have been lots of attempts to reduce synchronization overhead for data and model parallelism. Most studies tried to increase synchronization periods while minimizing accuracy drops [22, 44, 89] and to pipeline batch processing to fully utilize computing resources [30, 50, 85]. All these studies have only considered synchronization overhead in a PCIe environment and did not consider large-scale systems like photo storage built over networked environments.

9 Conclusion

In this paper, we proposed a photo storage system, NDPipe, which accelerated training and inference to address the outdated model and label problems. To ensure timely DNN creation and image relabeling, NDPipe distributed storage servers with inexpensive commodity GPUs and used their collective intelligence to train and infer near image data. According to our results, given the same energy budget, NDPipe exhibits 1.39× higher inference throughput and 2.64× shorter training time than typical photo storage systems.

Acknowledgments

We thank our shepherd, Dr. Swami Sundararaman, and the anonymous reviewers for all their helpful comments. This work was supported by the National Research Foundation of Korea (NRF-2018R1A5A1060031), the MOTIE (Ministry of Trade, Industry & Energy) (1415181081), and KSRC (Korea Semiconductor Research Consortium) (20019402). This study was also supported by the Samsung Research Funding Incubation Center of Samsung Electronics under Grant SRFCIT1902-03. (Corresponding author: Sungjin Lee)

A Artifact Appendix

A.1 Abstract

NDPipe is a deep learning system designed to enhance both training and inference performance by embracing the concept of near-data processing within storage servers. At its core, NDPipe utilizes an innovative architecture that distributes storage servers equipped with cost-effective commodity GPUs across a data center. NDPipe is composed of two main elements: PipeStore (storage server equipped with a low-end GPU for near-data training and inference) and Tuner (training server that manages distributed PipeStores). This artifact appendix provides a way to emulate NDPipe and the source code and scripts for a better understanding of the evaluation in our paper. Please refer to the README file at https://github.com/dgist-datalab/NDPipe.

A.2 Artifact check-list (meta-information)

- **Model:** ResNet50.
- **Data set:** CIFAR-100.
- **Hardware:** CUDA-enabled GPU.
- **Run-time environment:** Linux Ubuntu 20.04, Tensorflow 2.13, TensorRT 7.1.3, and so on (pip requirements are included in our GitHub).
- **Execution:** In the case of fine-tuning, the evaluation needs two or more machines.
- **Metrics:** Fine-tuning execution time and inference throughput (Image/sec).
- **Output:** Fine-tuning execution time and inference throughput (Image/sec) table in the console.
- **Experiments:** Manual steps by user.
- **How much disk space required (approximately)?:** 60GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 10 mins.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License v2.0.
- **Data licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10796943

A.3 Description

A.3.1 How to access. We provide the public GitHub URL (https://github.com/dgist-datalab/NDPipe) and Zenodo archive (https://doi.org/10.5281/zenodo.10796943). We also provide the Zenodo URI of the dataset for inference evaluation (https://doi.org/10.5281/zenodo.10796922).

A.3.2 Hardware dependencies. Two or more machines are needed for the experiment. One machine is for the Tuner, and the other machines are for the PipeStores (You only need two machines for functionality). All machines require a CUDA-enabled NVIDIA GPU, and PipeStore requires approximately 60GB or more of free storage space to conduct experiments.

A.3.3 Software dependencies. Our experiments are conducted using Python 3.9 on Ubuntu 20.04 LTS. The required Python packages can be found in `requirements.txt` in our GitHub repository.

A.4 Installation

Refer to the README file in our GitHub repository. Briefly, you first need to clone the required repository (NDPipe) onto multiple machines (Tuner and several PipeStores). Afterward, install the necessary Python packages from requirements.txt located in each directory (`Fine_tuning` and `Offline_inference`).

A.5 Experiment workflow

After completing the required installations, we first initiate Tuner by executing the script with optional parameters: the number of runs for pipelined fine-tuning, the number of PipeStores, and the port number. Then, we begin to run PipeStores by matching the port number on the Tuner side. Once the connection between Tuner and PipeStores is successfully established, Tuner allocates the tasks of fine-tuning or offline inference to the involved PipeStores.

For details, refer to the README in our GitHub repository.

A.6 Evaluation and expected results

The specific execution time and throughput differ on different platforms. The expected results are as follows:

- Fine-tuning

```
Feature extraction time (sec): 31.36
Feature extraction throughput (image/sec): 1913.26
Overall fine-tuning time (sec): 75.19
```

- Offline inference

```
[NDPipe] inference time: 14.78sec
[NDPipe] inference throughput: 2417.53IPS
```

References

[1] 42 Facebook Statistics Marketers Need to Know in 2023. Christina Newberry. https://blog.hootsuite.com/facebook-statistics/, 2023.

[2] 500 million people using Google Photos, and three new ways to share. Anil Sabharwal. https://blog.google/products/photos/google-photos-500-million-new-sharing/, 2017.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu,

and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/, 2015.

[4] Ashutosh Adhikari, Achyudh Ram, Raphael Tang, and Jimmy Lin. Docbert: Bert for document classification. *arXiv preprint arXiv:1904.08398*, 2019.

[5] Amazon Web Services Inc. Amazon S3: Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/?nc1=h_ls, 2022.

[6] Amazon Web Services Inc. Amazon EC2. https://aws.amazon.com/ec2/, 2023.

[7] Amazon Web Services Inc. Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, 2023.

[8] Amazon Web Services Inc. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia/, 2023.

[9] Amazon Web Services Inc. AWS Pricing Calculator. https://calculator.aws/, 2023.

[10] Amazon Web Services Inc. NeuronCore-v1 Architecture. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/neuron-core-v1.html, 2023.

[11] Amazon Web Services Inc. PyTorch Neuron. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/frameworks/torch/index.html, 2023.

[12] Amazon Web Services Inc. Unlimited photo storage. https://www.amazon.com/Amazon-Photos/b?ie=UTF8&node=13234696011, 2024.

[13] Sanjeev Arora, Nadav Cohen, Noah Golowich, and Wei Hu. A Convergence Analysis of Gradient Descent for Deep Linear Neural Networks. In *Proceedings of the International Conference on Learning Representations*, 2019.

[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 *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 387–401, 2021.

[15] Manuel Baena-Garcıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, R Gavalda, and Rafael Morales-Bueno. Early drift detection method. In *Proceedings of International Workshop on Knowledge Discovery from Data Streams*, volume 6, pages 77–86, 2006.

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[17] Yi Cai, Yujun Lin, Lixue Xia, Xiaoming Chen, Song Han, Yu Wang, and Huazhong Yang. Long Live TIME: Improving Lifetime for Training-in-Memory Engines by Structured Gradient Sparsification. In *Proceedings of ACM/ESDA/IEEE Design Automation Conference*, pages 1–6, 2018.

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 29–41, 2020.

[19] Ke Chen, Xingjian Du, Bilei Zhu, Zejun Ma, Taylor Berg-Kirkpatrick, and Shlomo Dubnov. Hts-at: A hierarchical token-semantic audio transformer for sound classification and detection. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 646–650, 2022.

[20] Minmin Chen, Alice Zheng, and Kilian Weinberger. Fast image tagging. In *Proceedings of International Conference on Machine Learning*, pages 1274–1282, 2013.

[21] Xinyang Chen, Sinan Wang, Bo Fu, Mingsheng Long, and Jianmin Wang. Catastrophic forgetting meets negative transfer: Batch spectral shrinkage for safe transfer learning. *Advances in Neural Information Processing Systems*, 2019.

[22] Yangrui Chen, Cong Xie, Meng Ma, Juncheng Gu, Yanghua Peng, Haibin Lin, Chuan Wu, and Yibo Zhu. SAPipe: Staleness-Aware Pipeline for Data Parallel DNN Training. *Advances in Neural Information Processing Systems*, 35:17981–17993, 2022.

[23] Yixin Chen and James Z Wang. Image categorization by learning and reasoning with regions. *The Journal of Machine Learning Research*, 5:913–939, 2004.

[24] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[26] L. Peter Deutsch. DEFLATE compressed data format specification version 1.3. https://www.w3.org/Graphics/PNG/RFC-1951, 1996.

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

[28] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *Proceedings of the International Conference on Learning Representations*, 2021.

[29] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 929–943, 2022.

[30] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *Proceedings of the USENIX Annual Technical Conference*, pages 381–396, 2021.

[31] Flickr. About Flickr. https://www.flickr.com/about, 2024.

[32] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[33] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Brazilian Symposium on Artificial Intelligence*, pages 286–295, 2004.

[34] Joao Gama, Indrundefined Zliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys*, 46(4), 2014.

[35] Yuan Gong, Sameer Khurana, Andrew Rouditchenko, and James Glass. Cmkd: Cnn/transformer-based cross-model knowledge distillation for audio classification. *arXiv preprint arXiv:2203.06760*, 2022.

[36] Google Inc. Vision AI. https://cloud.google.com/vision, 2022.

[37] Google Inc. Google Photos. https://www.google.com/intl/en_uk/photos/about/, 2024.

[38] Google Inc. What is Object storage? https://cloud.google.com/learn/what-is-object-storage, 2024.

[39] Shreyank N Gowda, Marcus Rohrbach, and Laura Sevilla-Lara. Smart frame selection for action recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 1451–1459, 2021.

[40] Guillermo Krovblit. Organize your Google Photos library with these two updates. https://blog.google/products/photos/google-photos-organization-updates-november-2023/, 2023.

[41] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 488–501, 2020.

[42] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 620–629, 2018.

[43] 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.

[44] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in Neural Information Processing Systems*, 26, 2013.

[45] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[46] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of International Conference on Machine Learning*, pages 2790–2799, 2019.

[47] How many pictures are there (2023): Statistics, trends, and forecasts. Matic Broz. https://photutorial.com/photos-statistics/, 2023.

[48] How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. Bernard Marr. https://bernardmarr.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/, 2021.

[49] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[50] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems*, 32:103–112, 2019.

[51] Andrey Ignatov, Luc Van Gool, and Radu Timofte. Replacing Mobile Camera ISP With a Single Deep Learning Model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 536–537, 2020.

[52] Amazon Web Services Inc.

[53] Yongjoo Jang, Sejin Kim, Daehoon Kim, Sungjin Lee, and Jaeha Kung. Deep Partitioned Training From Near-Storage Computing to DNN Accelerators. *IEEE Computer Architecture Letters*, 2021.

[54] Manjunath Jogin, MS Madhulika, GD Divya, RK Meghana, S Apoorva, et al. Feature extraction using convolution neural networks (cnn) and deep learning. In *Proceedings of the IEEE International Conference on Recent Trends in Electronics, information & Communication Technology*, pages 2319–2323, 2018.

[55] Shirali Kadyrov, Cemil Turan, Altynbek Amirzhanov, and Cemal Ozdemir. Speaker recognition from spectrogram images. In *IEEE International Conference on Smart Information Systems and Technologies*, pages 1–4, 2021.

[56] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, page 615–629, 2017.

[57] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[58] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 135–152, 2023.

[59] Minsub Kim, Jaeha Kung, and Sungjin Lee. Towards scalable analytics with inference-enabled solid-state drives. *IEEE Computer Architecture Letters*, 2020.

[60] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming Catastrophic Forgetting in Neural Networks. In *Proceedings of the National Academy of Sciences*, pages 3521–3526, 2017.

[61] John S Koh, Jason Nieh, and Steven M Bellovin. Encrypted cloud photo storage using google photos. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 136–149, 2021.

[62] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, 2009.

[63] Jooyeon Lee, Junsang Park, Seunghyun Lee, and Jaeha Kung. Implication of Optimizing NPU Dataflows on Neural Architecture Search for Mobile Devices. *ACM Transactions on Design Automation of Electronic Systems*, 27(5), 2022.

[64] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, page 583–598, 2014.

[65] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. *Advances in Neural Information Processing Systems*, 31, 2018.

[66] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *Proceedings of the USENIX Annual Technical Conference*, pages 395–410, 2019.

[67] Dong Liu, Shuicheng Yan, Xian-Sheng Hua, and Hong-Jiang Zhang. Image retagging using collaborative tag propagation. *IEEE Transactions on Multimedia*, 13(4):702–712, 2011.

[68] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35, 2022.

[69] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[70] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Manohar Paluri, and Laurens Maaten van der. Advancing state-of-the-art image recognition with deep learning on hashtags. https://ai.facebook.com/blog/advancing-state-of-the-art-image-recognition-with-deep-learning-on-hashtags/, 2018.

[71] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4:77–94, 2022.

[72] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4:77–94, 2022.

[73] Bernard Marr. How Much Data Do We Create Every Day? https://bernardmarr.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/, 2021.

[74] Matthew Radzihovsky and Farzan Memarian and Ethem Can and Burak Yoldemir. Serving ML Model Pipelines on NVIDIA Triton Inference Server with Ensemble Models. https://www.tensorflow.org/lite/guide, 2023.

[75] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal*, 5(4):1093–1113, 2014.

[76] Meta. How Facebook is using AI to improve photo descriptions for people who are blind or visually impaired. https://tech.facebook.com/artificial-intelligence/2021/1/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are-blind-or-visually-impaired/, 2021.

[77] Microsoft Inc. Suggest content tags with NLP using deep learning. https://learn.microsoft.com/en-us/azure/architecture/solution-ideas/articles/website-content-tag-suggestion-with-deep-learning-and-nlp, 2022.

[78] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies*, pages 203–216, 2021.

[79] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[80] Netflix Inc. The Story of Netflix. https://about.netflix.com/en, 2024.

[81] NVIDIA. NVIDIA Tesla V100. https://www.nvidia.com/en-gb/data-center/tesla-v100/, 2017.

[82] NVIDIA. NVIDIA T4 Tensor Core GPUs for Accelerating Inference. https://www.nvidia.com/en-us/data-center/tesla-t4/, 2018.

[83] NVIDIA Developer. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2021.

[84] Mathias Parger, Chengcheng Tang, Christopher D Twigg, Cem Keskin, Robert Wang, and Markus Steinberger. DeltaCNN: End-to-end CNN inference of sparse frame differences in videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12497–12506, 2022.

[85] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young ri Choi. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–321, 2020.

[86] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[87] Dong ping Tian. A review on image feature extraction and representation techniques. *International journal of multimedia and ubiquitous engineering*, 8:385–396, 2013.

[88] Arjun Raj Rajanna, Kamelia Aryafar, Ali Shokoufandeh, and Raymond Ptucha. Deep neural networks: A case study for music genre classification. In *IEEE International Conference on Machine Learning and Applications*, pages 655–660, 2015.

[89] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24, 2011.

[90] Jie Ren, Samyam Rajbhandari, Yazdani Reza Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the USENIX Annual Technical Conference*, pages 551–564, 2021.

[91] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[92] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[93] Devashish Shankar, Sujay Narumanchi, H A Ananya, Pramod Kompalli, and Krishnendu Chaudhury. Deep Learning based Large Scale Visual Recommendation and Search for E-Commerce, 2017.

[94] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Xue. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture*, pages 1056–1070, 2022.

[95] SuperMicro Computer Inc. . Storage SuperServer SSG-121E-NES24R. https://www.supermicro.com/en/products/system/storage/1u/ssg-121e-nes24r, 2024.

[96] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[97] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. 2010.

[98] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 513–527, 2020.

[99] Yuchen Wei, Son Tran, Shuxiang Xu, Byeong Kang, and Matthew Springer. Deep learning for retail product recognition: Challenges and techniques. *Computational intelligence and neuroscience*, 2020, 2020.

[100] 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. *Advances in Neural Information Processing Systems*, 30, 2017.

[101] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1492–1500, 2017.

[102] Pei-Tse Yang, Shih-Ming Kuang, Chia-Chun Wu, and Jia-Lien Hsu. Predicting music emotion by using convolutional neural network. In *International Conference on Human-Computer Interaction*, pages 266–275, 2020.

[103] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment*, 14(11), 2021.

[104] Liang Ye, Zhiguo Cao, and Yang Xiao. Deepcloud: Ground-based cloud image categorization using deep convolutional features. *IEEE Transactions on Geoscience and Remote Sensing*, 55(10):5729–5740, 2017.

[105] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in Neural Information Processing Systems*, 27, 2014.

[106] Youtube. Ever wonder how YouTube works? https://www.youtube.com/intl/en_us/howyoutubeworks/, 2024.

[107] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[108] Shen-Yi Zhao and Wu-Jun Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2379–2385, 2016.