FLEET: FLEXIBLE EFFICIENT ENSEMBLE TRAINING FOR HETEROGENEOUS DEEP NEURAL NETWORKS

Hui Guan [1] **Laxmikant Kishor Mokadam** [2] **Xipeng Shen** [1] **Seung-Hwan Lim** [3] **Robert Patton** [3 4]

ABSTRACT

Parallel training of an ensemble of Deep Neural Networks (DNN) on a cluster of nodes is an effective approach to shorten the process of neural network architecture search and hyper-parameter tuning for a given learning task. Prior efforts have shown that data sharing, where the common preprocessing operation is shared across the DNN training pipelines, saves computational resources and improves pipeline efficiency. Data sharing strategy, however, performs poorly for a heterogeneous set of DNNs where each DNN has varying computational needs and thus different training rate and convergence speed. This paper proposes FLEET, a flexible ensemble DNN training framework for efficiently training a heterogeneous set of DNNs. We build FLEET via several technical innovations. We theoretically prove that an optimal resource allocation is NP-hard and propose a greedy algorithm to efficiently allocate resources for training each DNN with data sharing. We integrate data-parallel DNN training into ensemble training to mitigate the differences in training rates and introduce checkpointing into this context to address the issue of different convergence speeds. Experiments show that FLEET significantly improves the training efficiency of DNN ensembles without compromising the quality of the result.

1 INTRODUCTION

Recent years have witnessed rapid progress in the development of Deep Neural Networks (DNN) and their successful applications to the understanding of images, texts, and other data from sciences to industry (Patton et al., 2018; Mathuriya et al., 2018; Ratnaparkhi & Pilli, 2012).

An essential step to apply DNNs to a new data set is hyper-parameter tuning—that is, the selection of an appropriate network architecture and hyper-parameters (e.g., the number of layers, the number of filters at each layer, and the learning rate scheduling). It is called Neural Architecture Search (NAS) when the tuned parameters determine a DNN's architecture. Many different search strategies have been proposed such as random search (Bergstra & Bengio, 2012; Li & Talwalkar, 2019), reinforcement learning (Zoph & Le, 2016; Zoph et al., 2018), evolutionary methods (Salimans et al., 2017), and Bayesian Optimization (Kandasamy et al., 2018). Most existing methods used today need to train a large set of DNN candidates with different architectures (e.g. 450 networks being trained concurrently in (Zoph et al., 2018)) to identify the best model for a particular task.

[1]North Carolina State University [2]Google Inc. (The work was done when he was at NCSU.) [3] Oak Ridge National Laboratory [4]Rensselaer Polytechnic Institute. Correspondence to: Hui Guan <hguan2@ncsu.edu>, Xipeng Shen <xshen5@ncsu.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020.

An effective strategy for shortening the process of hyper-parameter tuning and NAS is to concurrently train a set of DNNs on a cluster of nodes[1], which is referred to as *ensemble training* of DNNs. We refer to an ensemble of DNNs with the same architecture as a *homogeneous ensemble*. Otherwise, the ensemble is called *heterogeneous ensemble*.

A common ensemble training strategy is to duplicate a training pipeline on multiple nodes to train DNNs in parallel. A typical DNN training pipeline is an iterative process including data fetching, preprocessing, and training. For the ease of description, we refer to data fetching and preprocessing together as preprocessing. In ensemble training, training steps are not identical because we train models with different architectures and configurations. However, preprocessing is redundant across the pipelines, resulting in unnecessary CPU usage and even poor pipeline performance.

To eliminate the redundancies, Pittman et al. (Pittman et al., 2018) proposed data sharing where the common preprocessing operations are shared across training pipelines of all DNNs in an ensemble. They demonstrated that data sharing is an effective strategy to reduce computational resource utilization and improve pipeline efficiency. Their solution, however, assumes relatively homogeneous computational needs for DNNs in an ensemble. It may perform poorly

[1]A "node" in this paper refers to a machine in a cluster; one node may contain one or more CPUs and GPUs

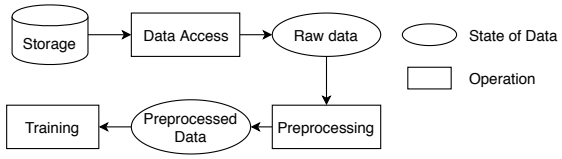

Figure 1: A DNN training pipeline (Pittman et al., 2018).

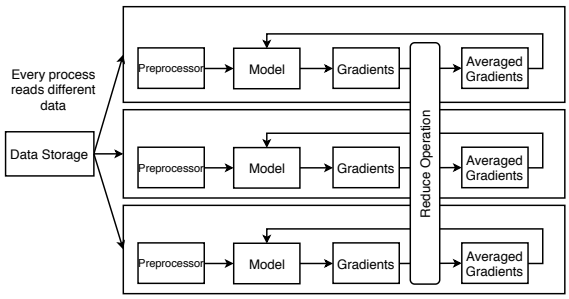

Figure 2: An illustration of data-parallel DNN training (Sergeev & Del Balso, 2018).

for an heterogeneous ensemble due to the variance of DNN model training from two algorithmic characteristics.

The first algorithmic characteristic is *varying training rate*. Training rate of a DNN is the compute throughput of processing units (e.g., CPUs and GPUs) used for training the DNN. Each DNN in an heterogeneous ensemble could have varying computational needs and thus different training rates with the same computing resources (Canziani et al., 2016; Sze et al., 2017). If a DNN consumes preprocessed data slower than other DNNs, others will have to wait for the slower one before evicting current set of cached batches when we employ synchronized data fetching for data sharing to ensure that each DNN is trained using the entire dataset. This waiting lowers the utilization of computing resources in the cluster and delays the overall training time of the ensemble.

The second one is *varying convergence speed*. Due to the differences in network architecture or hyper-parameter settings, some DNNs may require a larger number of epochs (one epoch goes through all data samples once) to converge than others (Krizhevsky et al., 2012; He et al., 2016; Huang et al., 2017; Zagoruyko & Komodakis, 2016). There can be scenarios where a subset of DNNs in the ensemble have already converged while the shared preprocessing operations have to keep prepossessed data for the remaining DNNs. Resources allocated to these converged DNNs will be underutilized until the training of all the DNNs is completed.

To address the issues, we propose FLEET, a flexible ensemble training framework for efficiently training a heterogeneous set of DNNs. We build FLEET via several technical innovations. First, we formalize the essence of the problem into an *optimal resource allocation problem*. We analyze the computational complexity of the problem and present an efficient greedy algorithm that groups a subset of DNNs into a unit (named *flotilla*) and effectively maps DNNs to GPUs in a flotilla on the fly. The algorithm incurs marginal runtime overhead while balancing the progressing pace of DNNs. Second, we develop a set of techniques to seamlessly integrate distributed data-parallel training of DNN, preprocessing sharing, and runtime DNN-to-GPU assignments together into FLEET, the first ensemble DNN training framework for heterogeneous DNNs. We introduce checkpointing into this context to address the issue of different convergence speeds. FLEET features flexible and efficient communications and effective runtime resource allocations.

Experiments on 100 heterogeneous DNNs on SummitDev, the Oak Ridge Leadership Computing Facility (Sec 6.1), demonstrate that FLEET can speed up the ensemble training by 1.12-1.92X over the default training method, and 1.23-1.97X over the state-of-the-art framework that was designed for homogeneous ensemble training.

2 Background

This section provides the necessary background of DNN training pipeline and data-parallel DNN training.

DNN Training Pipeline. As shown in Figure 1, a typical DNN training pipeline is an iterative process containing three main stages: data fetching, preprocessing, and training. In each iteration, data is fetched to the main memory and then run through a sequence of preprocessing operations such as decoding, rotation, cropping, and scaling. The preprocessed data is arranged into batches and consumed by the training stage. The *batch size* is the number of data samples used simultaneously per step.

The modern computing clusters and data centers have evolved into a hybrid structure that contains both CPUs and GPUs on each node. These heterogeneous CPU-GPU clusters are particularly useful for DNN training as CPUs and GPUs can work together to accelerate the training pipeline. Compared to the training stage, preprocessing is usually less computation intensive. To pipeline the preprocessing and DNN training, typically preprocessing is performed on CPUs while training on another batch of data happens simultaneously on GPUs.

Data-Parallel DNN Training. Data-parallel DNN training trains a single DNN using multiple training pipelines where each pipeline handles a different subset of data. As illustrated in Figure 2, each pipeline fetches a different subset of data from storage and prepossesses data independently. In the training stage, gradients are calculated by each pipeline and are reduced so that every pipeline has the same averaged gradients. The averaged gradients are used to update the model to make sure each pipeline has the same copy of

Table 1: The job of different processes.

Process Type	Job Description
Preprocesser	fetch data from storage, preprocess the data, and send the preprocessed data to its paired training group master.
Training Group Master	receive the preprocessed data from its paired preprocesser, scatter it within its training group, broadcast the data to other training group masters, and train the DNN using the assigned batch of data.
Training Worker	receive the assigned batch of data from its training group master and use it to train the DNN.

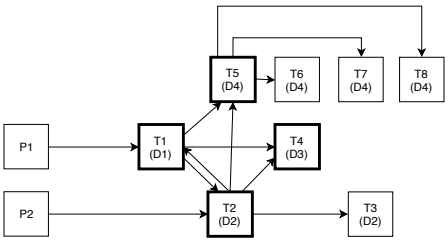

Figure 3: An illustration of the ensemble training pipeline in FLEET. P1 and P2 are preprocessors and T1-T8 are trainers. There are four training groups, (T1), (T2, T3), (T4), (T5, T6, T7, T8), which train the four DNNs D1-D4 respectively. Edges indicate transfers of preprocessed images.

model parameters.

Pipelines in data-parallel DNN training can run either on the same computing node using intra-node communication (single node multiple GPU training) or different nodes using inter-node communication (multiple node multiple GPU training). For the existing communication interfaces (e.g., MPI), the intra-node communication is usually more efficient than inter-node communication. Thus, it is preferred to allocate pipelines on the same computing node rather than on different nodes. As it is common to run only one pipeline on a single GPU, the number of GPUs available to train a DNN model practically limits the maximum number of pipelines that can be created in data-parallel DNN training.

3 Overview of FLEET

This section gives an overview of FLEET. FLEET is a flexible pipeline software architecture for efficient ensemble training of heterogeneous DNNs. It provides flexibility for configuring the scheduling of DNNs on nodes and GPUs via separation of preprocessing and training into different processes and a collection of communication schemes. It creates efficiency via heterogeneity-conscious runtime resource allocation and scheduling, plus sharing of preprocessing results among DNNs.

FLEET uses two types of processes, called *preprocessor* and *trainer*, to perform preprocessing and training separately. A *trainer group* contains at least one trainer processes and is responsible for training one DNN in the ensemble. A trainer process uses one GPU for training. When a trainer group contains more than one trainer process, they perform data-parallel DNN training for one DNN. Each trainer group has a trainer as the *training group master* and zero or more trainers as the *training workers*. The preprocessors communicate directly with only some master trainers, and those master trainers forward the preprocessed data to other trainers. Figure 3 illustrates the ensemble training pipeline in FLEET. The job of each process is summarized in Table 1.

Efficiency and Flexibility. Two important features of FLEET are its efficiency and flexibility.

The efficiency of FLEET comes from its novel resource allocation strategy developed for DNN ensemble training. The strategy is powered by some fundamental understanding of this resource allocation problem, and a greedy scheduling algorithm designed specifically to heterogeneous ensemble training. The algorithm seamlessly integrates data-parallel distributed training with ensemble training. As illustrated in Figure 3, different number of GPUs can be allocated to each DNN so that the DNNs can reach a similar training rate, avoiding the pipeline inefficiency caused by the slowest DNNs. It overcomes the NP-hardness of the resource allocation problem through a greedy design, grouping DNNs into multiple flotillas and periodically (re)allocate GPUs to remaining DNNs in a global efficient manner. It further leverages check-pointing to mitigate the issue of varying convergence speeds among DNNs. Together, FLEET is able to achieve efficient ensemble training while enabling data sharing to save CPU usage.

The flexibility of FLEET is in two aspects. First, decoupling preprocessing and training using different processes[2] provides the flexibility in configuring the number of preprocessors such that the preprocessing throughput can match the trainers' throughput without creating too many preprocessors that may waste computing resource and power. Second, as each trainer is associated with one GPU, resources for training can be allocated in the granularity of GPUs (rather than nodes as in prior work (Pittman et al., 2018)). Each GPU in a node can be assigned independently to DNNs. Each DNN in an ensemble can be trained using different numbers of GPUs concurrently, giving flexibility for handling the heterogeneity in DNNs.

Two-fold Enabling Techniques. The key technical contributions that make FLEET possible are two-fold. The first is theoretical, consisting of a deep understanding of the resource allocation problem and some novel algorithms for assigning DNNs to GPUs. The second is empirical, consisting of a set of solutions to the various challenges for implementing FLEET above the array of complex software

[2]The reason we used processes instead of threads is due to the Global Interpreter Lock in Python. As FLEET is built on TensorFlow which is in Python, multi-processing brings maximum parallelism into the training pipeline.

Table 2: Notations.

Notation	Description
N	the number of DNNs in an ensemble.
M	the number of GPUs available in a cluster.
K	the number of DNN flotillas.
$\mathcal{D}$	the list of DNNs in an ensemble, $\mathcal{D} = [D_1, \cdots, D_N]$.
$\mathcal{F}$	the list of flotillas of DNNs, $\mathcal{F} = [\mathcal{F}_1, \cdots, \mathcal{F}_K]$.
$\mathcal{F}_k$	the k-th flotilla of DNNs, $\mathcal{F}_k = [D_1^{(k)}, \cdots, D_{N_k}^{(k)}]$.
$D_i^{(k)}$	the i-th DNN in the k-th flotilla.
N_k	the number of DNNs in the k-th flotilla.
$\mathcal{A}$	the list of GPU allocations, $\mathcal{A} = [A_1, \cdots, A_K]$
A_k	a N_k-by-M matrix, the GPU allocations for the k-th flotilla of DNNs.
$a_{i,j}^{(k)}$	whether the j-th GPU is assigned to $D_i^{(k)}$.
$m_i^{(k)}$	$\sum_{j=1}^{M} a_{i,j}^{(k)}$, the number of GPUs assigned to $D_i^{(k)}$.
$r_i^{(k)}(m)$	the training rate of $D_i^{(k)}$ trained with m GPUs.

components (TensorFlow, Horovod, Python, MPI, etc.) on a heterogeneous Multi-GPU supercomputer like Summit-Dev (Sum, 2019). We present the two-fold contributions in the next two sections respectively.

4 Resource Allocation Algorithms

Efficient ensemble training is essentially an optimal resource allocation problem. The resources involve CPUs and GPUs in the modern heterogeneous computing clusters. Under the context of data sharing, an optimal CPU allocation sets the number of preprocessors to be the one that just meets the computing requirement of training DNNs. GPU allocation, however, is much more complex and determines the pipeline efficiency. We formalize it as an *optimal resource allocation problem* and analyze its computational complexity; the understanding motivates our later designs of the practical algorithms and the FLEET architecture. We next start with the problem definition.

4.1 Problem Definition

There are two possible paradigms for scheduling DNNs on GPUs. A **local** paradigm assigns a DNN to a GPU immediately when the GPU becomes vacant. A **global** paradigm periodically examines the remaining DNNs and does a global (re)assignment of the DNNs to all GPUs. The *local* paradigm is relatively easy to understand; the *global* paradigm has the potential to avoid the local optimal but is more difficult to design. Particularly, to effectively realize the global paradigm, several open questions must be answered: Is an optimal scheduling algorithm feasible? If so, what is it? If not, how to efficiently approximate it? This section focuses on the *global* paradigm and explores these open questions. For easy reference, we put into Table 2 the important notations used in the rest of this paper.

In this scheduling problem, the entire execution trains N DNNs on M GPUs in K rounds. The beginning of a round is the time for globally (re)scheduling remaining DNNs on GPUs. The set of DNNs being trained in each round is called a *flotilla*. So there are K flotillas being trained in the execution, one flotilla a round.

Theoretically, a round can be a time period of an arbitrary length. We first focus on a simple case where a round finishes when and only when the training of all the DNNs in a flotilla finishes (e.g., converges or the maximum training epochs reached). In this setting, the GPUs that are done with its work in the current flotilla earlier than other GPUs would have some idle waiting time. The simplicity of this setting, however, makes the analysis easy to understand. We will briefly discuss the complexities of the more general settings at the end of Section 4.2.

We now give a formal definition of the resource allocation problem in the focused setting. Each DNN in the ensemble are placed into at least one of the flotillas $\mathcal{F}_k, k = 1, \cdots, K$ such that a list of K flotillas $\mathcal{F} = [\mathcal{F}_1, \cdots, \mathcal{F}_K]$ cover all the DNNs. Each flotilla, $\mathcal{F}_k = [D_1^{(k)}, \cdots, D_{N_k}^{(k)}]$, contains no more than M DNNs (i.e., $N_k \leq M$) such that each DNN in the flotilla can have at least one GPU. Let $\mathcal{A} = [A_1, \cdots, A_K]$ be the GPU assignment for the K flotillas of DNNs. Each assignment A_k is a N_k-by-M matrix $(a_{i,j}^{(k)})$ with:

$$a_{i,j}^{(k)} = \begin{cases} 1, & \text{if the } j\text{-th GPU is assigned to the model } D_i^{(k)}, \\ 0, & \text{otherwise,} \end{cases}$$

$$s.t. \quad \sum_{i=1}^{N_k} a_{i,j}^{(k)} \leq 1 \quad (j = 1, 2, \cdots, M),$$

$$\sum_{j=1}^{M} a_{i,j}^{(k)} \geq 1 \quad (i = 1, 2, \cdots, N_k).$$

An optimal resource allocation is an allocation strategy of available GPUs in a cluster to DNNs in an ensemble such that the end-to-end training time of the DNNs is minimized. The definition is as follows:

Definition 1 ***Optimal Resource Allocation.*** *Given a DNN ensemble $\mathcal{D}$ and a cluster of nodes with totally M GPUs, let $T(\mathcal{D}|\mathcal{F}, \mathcal{A})$ be the end-to-end time to finish the training of all the DNNs according to the list $\mathcal{F}$ and the corresponding GPU assignment $\mathcal{A}$. The optimal resource allocation problem is to find a schedule $(\mathcal{F}^*, \mathcal{A}^*)$ such that*

$$\mathcal{F}^*, \mathcal{A}^* = \arg\min_{\mathcal{F}, \mathcal{A}} T(\mathcal{D}|\mathcal{F}, \mathcal{A}) \tag{1}$$

$$= \arg\min_{\mathcal{F}, \mathcal{A}} \sum_{k=1}^{K} T(\mathcal{F}_k|A_k), \tag{2}$$

where, $T(\mathcal{F}_k|A_k)$ is the time spent on training the DNNs in the $\mathcal{F}_k$ with the assignment A_k for some epochs.

Algorithm 1 Greedy Algorithm

Input: $\mathcal{D}, M$ // DNN ensemble and the number of GPUs
Output: $\mathcal{F}, \mathcal{A}$ // A list of flotillas and GPU assignments
1: $R = profile(\mathcal{D})$ // Profile training rates of each DNN trained using $m = 1, \cdots, M$ number of GPUs.
2: $\mathcal{F}, \mathcal{A}, cands, k = [], [], \mathcal{D}, 1$
3: **while** $|cands| > 0$ **do**
4: $\mathcal{F}_k, \mathbf{m_k}$ = createFlotilla($cands, R, M$) // Step 1: Create a new flotilla from candidate DNNs; return the flotilla of DNNs ($\mathcal{F}_k$) and GPU count vector ($\mathbf{m_k}$)
5: A_k = getGPUAssignment($\mathcal{F}_k, \mathbf{m_k}$) // Step 2: Figure out a GPU assignment for the flotilla
6: $dels$ = train($\mathcal{F}_k, A_k$) // Step 3: Load the latest checkpoint if available; train DNNs in the flotilla for some epochs; return converged models ($dels$).
7: $cands- = dels$ // Remove converged models from candidates ($cands$)
8: $\mathcal{F}$.append($\mathcal{F}_k$); $\mathcal{A}$.append(A_k); $k+ = 1$
9: **end while**

4.2 Complexity Analysis

In this part, we argue that the Optimal Resource Allocation problem is NP-hard in general. The argument comes from the classic results in Parallel Task System Scheduling. As Du and Leung have proved (Du & Leung, 1989), finding an optimal non-preemptive schedule for a Parallel Task System with the precedence constraints consisting of chains is strongly NP-hard for each $n > 2$ (n is the number of processors). And, when the precedence constraints are empty, the problem is strongly NP-hard for each $n \geq 5$. The Optimal Resource Allocation problem can be viewed as a parallel task system scheduling problem with each DNN as a task and each GPU as a parallel processor. One subtle aspect is that even though the DNNs are independent, to leverage shared preprocessing data among DNNs, a newly freed GPU does not take on a new DNN until the new round starts. It could be viewed as there are some pseudo precedence constraints between the DNNs in two adjacent rounds. So in general, the optimal solution is unlikely to be found in polynomial time. Recall that our discussion has been assuming that a new round starts only when the training of the DNNs in the previous round is all done. If the condition is relaxed such that a round can be a time period of an arbitrary length, the problem becomes even more complex to solve.

4.3 Greedy Allocation Algorithm

Motivated by the complexity in finding optimal solutions to the problem, we have designed a greedy algorithm for FLEET to assign DNNs to GPUs efficiently. It is worth noting that, even though the Optimal Resource Allocation problem connects with the classic Parallel Task System Scheduling, several special aspects of it make it unique and demand new algorithm designs. First, unlike what is often assumed in the classic scheduling problems, the length of a task (DNN training) is hard if ever possible to predict: There is no known method that can accurately predict the number of epochs (and hence the time) needed for a DNN to converge. Second, the relations among tasks (DNNs) are "fluid". The training of two DNNs are theoretically independent: One does not depend on another's data or control. But when they are put into the same flotilla, they become related: They would share the same preprocessed data and hence need to keep a similar progressing pace. These special aspects make the problem different from prior problems and call for new algorithms to be designed.

This section describes our algorithm. It first introduces four principles we followed in developing the solution and then elaborates our greedy algorithm. We will explain the solution in the context of the *global* paradigm and discuss how it is also applicable to the *local* paradigm at the end of this section.

4.3.1 Principles

A resource allocation strategy involves grouping the DNNs into flotillas and assigning the DNNs in each flotilla to the GPUs. We develop our solution by following four principles. The core of these principles is to organize tasks with less variation and dependencies at the flotilla level (Principles 1 and 2) and at the node level (Principles 3 and 4).

Principle 1 *DNNs in the same flotilla should be able to reach a similar training rate (e.g., images per sec) if a proper number of GPUs are assigned to each of the DNNs.*

This principle helps ensure a balanced pace of all GPUs, which helps the DNNs in consuming the shared preprocessed data in a similar rate to minimize the waiting time of certain GPUs. This may result in multiple flotillas to be created if not all DNNs in the ensemble are similar.

Principle 2 *Packing into one flotilla as many DNNs as possible.*

The reason for this principle is two-fold. First, the throughput of multi-GPU training scales sublinearly[3] with the number of GPUs due to the communication overhead of exchanging gradients. The principle is to help maintain good efficiency of the DNNs. Second, it allows more DNNs to share preprocessed data.

Principle 3 *When assigning multiple GPUs to a DNN, try to use GPUs in the same node.*

[3]If all DNNs in an ensemble has a perfect linear scaling in throughput, training the DNNs one after another would be the optimal strategy. It is however often not the case in our observation. Another practical reason for concurrently training multiple DNNs is hyperparameter tuning. By checking the intermediate training results of those DNNs, the unpromising ones can be discarded.

This principle is to reduce the variation in communication latency: inter-node communications are slower and have more variations than intra-node communications.

Principle 4 *Try to assign DNNs that need a small number of GPUs to the same node.*

This principle is similar to Principle 2 but at the node level. The rationale is that, although it is hard to reduce the communication overhead of DNNs that need to be trained using multiple nodes, we can minimize the communication overhead of DNNs that need a small number of GPUs by assigning them to the GPUs in the same node.

Based on the four principles, we propose a greedy algorithm to solve the resource allocation problem, as described below.

4.3.2 Algorithm

The greedy algorithm is shown in Algorithm 1 (Complexity analysis in Appendix A). It uses training rates of the DNNs, $R = \{r_i(m)\}, i = 1, \cdots, N, m = 1, \cdots, M$, which are attained through a short profiling process (line 1). We propose profiling of fewer than 50 batches of training for each DNN. We defer the detailed profiling process to Section 6.1.

The greedy algorithm dynamically determines the grouping of the DNNs in an ensemble based on whether the DNN is converged or not and the training rate of each DNN. Once a flotilla is created, an optimal GPU assignment can be derived. Initially, all DNNs are considered as candidates (*cands*) when a new flotilla needs to be created (line 2). The greedy algorithm then iterates over three main steps, *flotilla creation* (line 4), *GPU allocation* (line 5), and *training* (line 6), until all the DNNs in the ensemble are converged (i, e, *cands* are empty).

We next describe the three steps in detail.

Flotilla Creation. This first step selects a set of DNNs from candidates to create a new flotilla whose DNNs are trained concurrently with data sharing, following Principles 1 and 2. The Pseudo-code is in Appendix A. The algorithm first identifies the largest training rate with a single GPU, $r_{fast} = \max\{r_1(1), \cdots, r_{|cands|}(1)\}$, and the corresponding DNN, D_{fast}, from the candidate set of DNNs. Then r_{fast} is used as the reference training rate to search for other DNNs that can be placed in the same flotilla. Mathematically, the algorithm searches for the next DNN that can be placed into the flotilla by solving the following optimization problem:

$$\begin{aligned} &\min_{\substack{D_i \in cands - \mathcal{F}_k, \\ m=1,\cdots,M}} |r_i(m) - r_{fast}|, \\ s.t. \quad &|r_i(m) - r_{fast}| \leq \delta, \\ &m \leq M - M_k, \end{aligned} \tag{3}$$

where δ is the threshold that determines if two training rates are close, and M_k is the total number of GPUs that are already assigned to DNNs. In our experiments, δ is set to 20 (images/sec). The algorithm stops adding DNNs to a flotilla if no solution exists to Eq. 3.

After a flotilla is formed, if there are still GPUs available, we assign the next GPU to the DNN in the flotilla that has the smallest training rate iteratively until all the GPUs are assigned. The DNN with the smallest training rate determines the pipeline efficiency. Assigning extra GPUs to the slowest DNN can improve pipeline efficiency.

The flotilla creation step produces a flotilla of DNNs as well as the GPU count vector that specifies the number of GPUs assigned to each DNN. We next explain how to properly assign GPUs to each DNN based on the GPU count vector and considering GPU locality.

GPU Assignment. This procedure assigns GPUs to DNNs in a flotilla, following Principles 3 and 4. The goal of this procedure is to find an assignment A_k to minimize the number of nodes involved in training each DNN. Let $c(.)$ be the function that counts the number of nodes involved in training a DNN given its GPU assignment $\mathbf{a}_i^{(k)}$, which is the i-th row of the assignment matrix A_k, the GPU assignment is an optimization problem:

$$\begin{aligned} &\min_{A_k} \sum_{i=1}^{N_k} \frac{c(\mathbf{a}_i^{(k)})}{m_i^{(k)}}, \\ s.t. \quad &\sum_{j=1}^{M} a_{i,j}^k = m_i^{(k)}, i = 1, \cdots, N, \end{aligned} \tag{4}$$

where $\frac{c(\mathbf{a}_i^{(k)})}{m_i^{(k)}}$ is the number of nodes involved to train the i-th DNN, scaled by $m_i^{(k)}$, the number of GPUs assigned. The solution space is as large as $\frac{M!}{\prod_{i=1}^{N_k}(m_i^{(k)}!)}$.

Instead of exhaustively searching for an optimal solution in the space, we propose a greedy approach that assigns GPUs to each DNN in an incremental fashion. For example, if the j-th GPU is already assigned to a DNN, then the next GPU to be assigned to the DNN is the $j+1$-th GPU. The solution space is reduced to the space of possible permutations of the GPU count vector ($N_k!$). This algorithm assumes the number of GPUs per node is the same among nodes, which holds in major supercomputers. It assigns GPUs to DNNs in the following order: (1) the DNNs whose required number of GPUs is a multiple of the number of GPUs per node; (2) the pairs of DNNs whose sum of the required number of GPUs is a multiple of the number of GPUs per node; (3) the remaining DNNs by searching for an optimal assignment of GPUs. The Pseudo-code is in Appendix A.

The flotilla creation and GPU assignment steps ensure that DNNs in the same flotilla can achieve similar training rate

to improve GPU utilization. We next describe how the training step addresses the varying convergence speed issue via check-pointing.

Training. The training step trains the DNNs on their assigned GPUs concurrently with data sharing. Due to the architectural difference of DNNs in a heterogeneous ensemble, these DNNs require a different number of epochs to converge. With data sharing, converged models need to wait for the un-converged models to complete, leading to the waste of computing resources. We leverage check-pointing to address the varying convergence speed issue. Specifically, each flotilla is trained until only $\alpha \cdot M$ GPUs remain active for training. α is set to 0.8 in all our experiments. We monitor whether a model is converged at the end of each epoch. Once a model is converged, it is marked as complete and its GPUs are released. If the total number of GPUs that are not released falls below $\alpha \cdot M$, the training of all the DNNs in the flotilla stops. The parameters, loss history, and epoch count of all the DNNs are check-pointed for recovering their training later. A DNN marked as complete will not be packed into any of the following flotillas.

4.3.3 Application in the Local Paradigm

Although the discussion has been assuming the *global* paradigm, the greedy algorithm applies the local paradigm of resource allocation as well. The training proceeds as follows: (1) At the beginning, the algorithm forms the first flotilla of DNNs and starts training them. (2) Whenever a DNN is done, the algorithm fills the released GPUs with new DNNs. If no DNN remains untrained, terminate when all current training is done.

5 IMPLEMENTATION

This section describes an efficient training pipeline implementation of FLEET. We focus on the following two main implementation challenges:

Challenge 1: Recall that FLEET has two types of processes, preprocessor and trainer. The number of preprocessors needs to be set to meet the requirement of trainers' throughput. Thus, it is necessary for FLEET to support creating a different number of processes per node on a cluster and also enable flexible communications between preprocessors and trainers.

Challenge 2: With data-parallel DNN training, preprocessed data from a processor is received by its paired training group master, scattered to trainers within the group (including the training group master), and broadcasted to the other training group masters. How do we build the dataflow to enable efficient training pipeline?

We next describe the solutions and the implementation details.

Communications between Preprocessors and Trainers. A preprocessor process is created through the fork operation. The number of preprocessors can be controlled by the number of trainer group masters that execute the fork operation. We establish the communications between a preprocessor and its paired trainer group master through server process. A server process holds Python objects and allows other processes to manipulate them using proxies. A proxy is an object in the `multiprocessing` package of Python and refers to a shared object which lives (presumably) in another process. A preprocessor sends the processed data to its training group master by writing to a Numpy object using the object's proxy.

Dataflow Implementation. Pipeline is the essential scheme organizing the different stages of DNN processing together. It allows the stages to run in parallel. For example, while a DNN is trained on some set of data, preprocessors can be preprocessing another set of data. Figure 4 illustrates the dataflow implementation in FLEET.

The dataflow contains three pipelined steps: (1) Training group masters receive preprocessed data from their paired preprocessor and put the data into a preprocessed queue Q_P. (2) Preprocessed data from Q_P are broadcast to all the training group masters through MPI. Each training group master receives all the preprocessed data, but handles the data differently, depending on whether data-parallel training is used. If a training group contains only one trainer (i.e., only one GPU is used to train a DNN), the training group master puts all the data into its trainer queue Q_T. Otherwise, the training group master scatters the data to its trainer queue Q_T and the distribution queue Q_{D^*}. The data in the distribution queue is sent to the trainer queue Q_T of each training group worker via MPI point-to-point communication in a separate thread. (3) Each trainer (T1-T4) reads preprocessed data from the trainer queue Q_T to Q_D, creates batches, and feeds each batch to the DNN model for training.

6 EVALUATION

We conduct a set of experiments to examine the efficacy of FLEET by answering the following questions: (1) How much speedup can FLEET bring to ensemble training of heterogeneous DNNs? (2) How do the pros and cons of the two paradigms in FLEET designs, *local* and *global*, play out in handling the variations among DNNs? More specifically, does the greedy scheduling algorithm in FLEET produce favorable schedules? How much waiting time does the round-by-round scheme in FLEET cause, compared to eager scheduling schemes? (3) What is the overhead of runtime profiling, scheduling, and checkpointing in FLEET?

We first describe the experiment settings (machines, base-

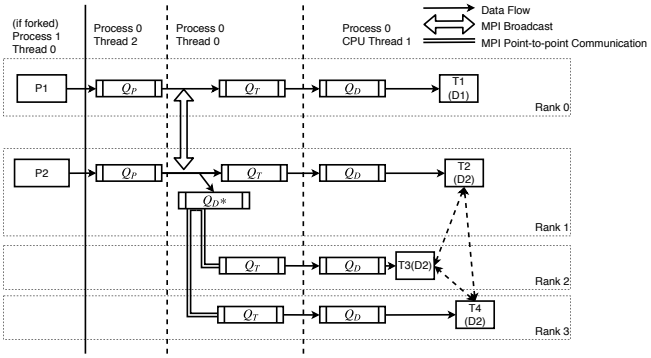

Figure 4: Illustration of the dataflow implementation. Two DNNs, $D1$ and $D2$, are trained using four GPUs (Ranks 0-3) by two training groups, $(T1)$ and $(T2, T3, T4)$. $T1$ and $T2$ are training group masters. Sizes of Q_P, Q_T and Q_D are 2048 images, 2048 images and 10 batches respectively.

lines, etc.) in Section 6.1 and then report our experiment results in Sections 6.2 and 6.3 to answer the questions.

6.1 Experiment Settings

DNNs. The DNNs used in this experiment are derived from DenseNets (Huang et al., 2017) and ResNets (He et al., 2016). Both models are the state-of-the-art network architectures that achieve high performance in various learning tasks. We select these networks as the basis because, as structural DNNs, they are composed of many Convolutional blocks, which have a standard interface making a block ready to be connected with any other blocks. As a result, it is easy to derive new DNNs from them—one just needs to remove or insert some Convolutional blocks. Based on the public DNNs, we derive 100 experimental DNNs (50 from DenseNet and 50 from ResNet) by randomly changing the block size of DenseNet and ResNet variations. The sizes of the DNN models vary from 232MB to 1.19GB. The distribution of their training rates on a single GPU which vary from 21 to 176 images/sec. (DNN details in Appendix B.1)

System. All experiments are conducted on SummitDev (Sum, 2019) at Oak Ridge National Lab. Each node is equipped with two IBM POWER8 CPUs, 256GB DRAM, and four NVIDIA Tesla P100 GPUs. FLEET is built on Tensorflow 1.12 (as the core training engine), Horovod v0.15.2 (Sergeev & Del Balso, 2018) (as the basis for distributed DNN training), and mpi4py v3.0.0 (for the pipeline construction). CUDA version is 9.2. (System details in Appendix B.2)

Datasets. The datasets used in the experiments are ImageNet (Deng et al., 2009) and Caltech256 Object Category Dataset (Cal, 2007). ImageNet contains 1,261,406 training images and Caltech256 contains 30,606 training images. We use ImageNet whenever possible (e.g.,for throughput comparisons), but use Caltech256 for the measurement of end-to-end ensemble training times such that the training can converge within the maximum 240 minutes limit that SummitDev permits.

Profiling. To minimize the overhead of profiling, we only profile the training rates of each DNN in the ensemble with the number of GPUs varying from one to $M_t (M_t < M)$, where M_t is determined based on the training rates of each DNN on a single GPU. For profiling on m $(m = 1, \cdots, M_t)$ GPUs, we train a DNN for a maximum of 48 batches and use the training time of the last 20 batches to calculate the exact training rate: $r_i(m), i = 1, \cdots, N$. Based on the profiled training rates, we estimate the training rates of each DNN when $m > M_t$. Profiling details are in Appendix B.3. The profiling process also measures the throughput of a range of preprocessors and uses it to set the number of preprocessors.

Counterparts for Comparisons.

- **Baseline** The baseline uses the default TensorFlow to train each DNN on one GPU independently. Each DNN trainer has a preprocessor that preprocesses data for itself independently. A GPU randomly picks one yet-to-be-trained DNN whenever it becomes free until there are no DNN left.
- **Homogeneous Training** This is the state-of-the-art framework recently published (Pittman et al., 2018) for ensemble DNN training. This framework allows the DNNs that get trained at the same time to share the preprocessed data. But it is designed for homogeneous DNN training, assuming no variations among DNNs or the situation where the number of DNNs is no greater than the number of GPUs. In our experiments, when there are more DNNs than GPUs, the framework randomly picks a subset of the remaining DNNs to train, one DNN per GPU with shared preprocessed data. After that subset is done, it picks another subset and repeats the process until all DNNs are done.
- **FLEET-G** This is FLEET in the global paradigm.
- **FLEET-L** This is FLEET in the local paradigm as described in Section 4.3.3. Its difference from FLEET-G is that as soon as a DNN is done, the released GPUs are immediately used to train some remaining DNNs; which DNNs are picked is determined by the greedy algorithm as in FLEET-G, but only locally (for the newly released GPUs) rather than globally.

6.2 End-to-End Speedups

Figure 5 reports the speedups of the three methods over the baseline method, in terms of the end-to-end ensemble training time of the 100 DNNs. All runtime overhead for

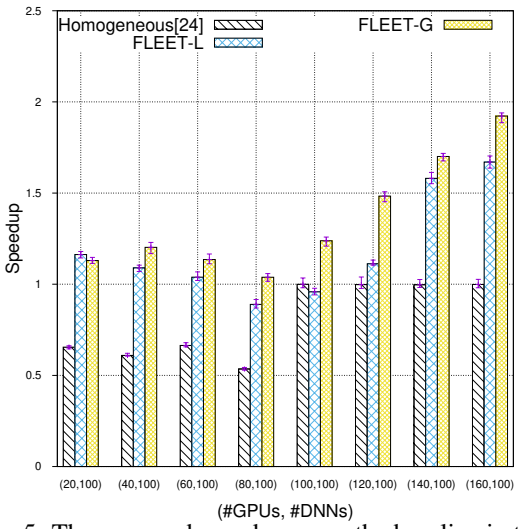

Figure 5: The averaged speedups over the baseline in terms of the end-to-end time for training a 100-DNN ensemble. The error bars show the variations.

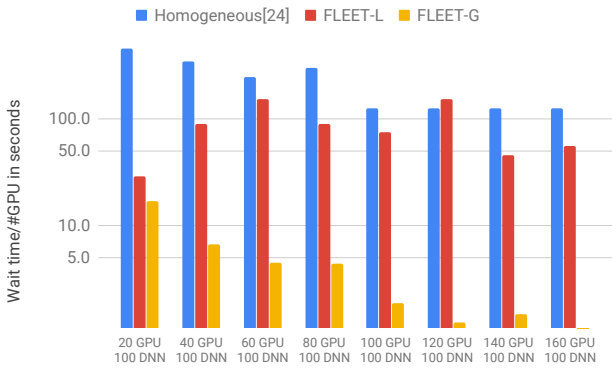

Figure 6: Waiting time per GPU.

FLEET is included. We repeat each measurement multiple times and report the average and error bars.

It shows the results in eight settings. The prior homogeneous framework shows large slowdowns in the first four settings where the number of GPUs is less than the number of DNNs. The slowdowns are due to the waiting of other GPUs for the slowest DNN to finish in each round, shown in Figure 6. In the other four settings, the homogeneous framework performs similarly as the baseline does: As there are more GPUs than DNNs, there is only one round, in which, the two methods use resource similarly. The sharing of preprocessing in the homogeneous framework does not generate speedups for these DNN trainings because the preprocessing is not the bottleneck for them.

FLEET-G gives the best overall performance, producing 1.12-1.92X speedups over the baseline. The primary reason for the speedups come from its better resource allocation to the DNNs. The bottom of Table 3 reports the mean and standard deviations of the running lengths of DNNs in the

Table 3: Mean and standard deviation of the running length of DNNs in seconds. (80 GPUs, 100 DNNs)

Technique	Flotilla ID	Mean	Std. Dev.
Baseline	-	10372.2	4178.9
FLEET-L	-	6213	3580.0
FLEET-G	0	2067.9	54.7
	1	335.6	48.4
	2	2291.9	26.0
	3	415.5	51.9
	4	1072.2	364.0
	5	2322.3	216.0

Table 4: Scheduling and checkpointing overhead.

(#GPU, #DNN)	Total Training Time (in sec)	Scheduling Overhead		Checkpointing Overhead	
		in sec	in %	in sec	in %
(20,100)	55200.1	20.1	0.037	1496.0	2.7
(40,100)	30204.8	15.8	0.054	1156.0	3.8
(60,100)	24495.0	14.0	0.060	986.0	4.0
(80,100)	21891.0	12.0	0.057	816.0	3.7
(100,100)	18359.1	10.1	0.058	782.0	4.3
(120,100)	15323.9	9.9	0.068	680.0	4.4
(140,100)	13366.3	9.3	0.073	680.0	5.1
(160,100)	11825.2	10.2	0.092	748.0	6.3

first five flotillas in FLEET-G (80GPU,100DNN). In comparison to the data in the baseline and FLEET-L (top rows in Table 3), the DNNs show much smaller variations in length, which indicate the effectiveness of the GPU allocations in FLEET-G in evening out the differences among DNNs. At the beginning, we thought that the catch to FLEET-G is the waiting time of some GPUs after they are done with their work in a round. Our experiments, however, show the opposite effects. As Figure 6 shows, the average waiting time per GPU is smallest for FLEET-G. The reason is that the other methods all suffer long waiting time at the end; because of their suboptimal resource allocation, some GPUs have to work long after others to finish up the last few DNNs. FLEET-L gives notable but fewer speedups for its less favorable decisions at resource allocation due to the local view.

Overall, FLEET gives larger speedups when #GPU $>$ #DNN. It is worth noting that in such a setting, there are still many flotillas to schedule and FLEET scheduler plays an important role. The reason is that in many cases, the FLEET scheduler assigns multiple GPUs to one DNN. For instance, 20 flotillas were created when training 100 DNN on 120 GPUs and 22 flotillas were created when training 100 DNNs on 160 GPUs. When #GPU$<$#DNN, the speedups from FLEET are not that significant but still substantial: 113–120% for three out of the four such settings in Figure 5.

6.3 Overhead

Table 4 reports the breakdown of the runtime overhead of FLEET-G. The overhead of scheduling and checkpointing is at most 0.1% and 6.3% of the end-to-end training time in all the settings. Recall that, due to wall-clock-time limitation of SummitDev, we have used the small Caltech256 dataset.

For large datasets (e.g., ImageNet), the overhead would become negligible. The profiling overhead is independent of dataset size and solely depends on ensemble size. Recall that, profiling needs the DNN to train for only a few steps in parallel. Its overhead is marginal for typical DNN trainings on large datasets that take hours or days to train. On recent GPUs, a feature called Multi-Process Services (MPS) could potentially allow multiple DNNs to be co-scheduled to a single GPU and run concurrently. It is not considered in the current FLEET. To consider it, some co-run predictive models could help, which could quickly predict the performance of a DNN when it co-runs with a set of other DNNs on one GPU. The predictive performance can then be combined with the existing performance models in FLEET to guide the scheduling of DNNs.

7 Related Work

Much research has been done to accelerate the training of a single DNN over distributed systems, such as Tensorflow from Google (Dean et al., 2012; Abadi et al., 2016), Project Adams from Microsoft (Chilimbi et al., 2014), FireCaffe (Iandola et al., 2016), PipeDream (Harlap et al., 2018), and GPipe (Huang et al., 2018). All those studies have focused on improving the training speed of an individual DNN rather than ensemble training.

Recently, ensemble training starts drawing more attention. Pittman et al. (Pittman et al., 2018) explored flexible communication strategies for training DNN ensembles and proposed data sharing to eliminate pipeline redundancy and improve pipeline efficiency. The work is designed for DNNs with similar computational needs. There are some other efforts (Garipov et al., 2018; Loshchilov & Hutter, 2016) on ensemble training, but they focus on designing lightweight methods to form high-performing ensembles instead of improving pipeline efficiency of ensemble training. HiveMind (Narayanan et al., 2018) is a system designed for accelerating the training of multiple DNNs on a single GPU by fusing common operations (e.g., preprocessing) across models. It, however, lacks the essential support for distributed DNN training.

Another line of research that is relevant to this work is task scheduling on clusters or workflow management systems. The scheduling of a set of tasks or workloads on clusters or multiprocessor systems has been extensively studied in the literature (Turek et al., 1992; Shmoys et al., 1995; Urgaonkar et al., 2002; Augonnet et al., 2011; Zaharia et al., 2008; Grandl et al., 2015; Chowdhury et al., 2016; Xu et al., 2018; Ousterhout et al., 2013; Cheng et al., 2016; Delimitrou & Kozyrakis, 2014; Feitelson et al., 1997). Recent work including Gandiva (Xiao et al., 2018) and Tiresias (Gu et al., 2019) design GPU cluster managers tailored for DNN workloads. They, however, lack the flexibility supported in FLEET. First, they treat different jobs as independent black boxes. Without the MPI communication mechanisms we put into FLEET to enable flexible data exchanges of TensorFlow-based workers and preprocessors, these schedulers cannot flexibly adjust the number of workers for a DNN training. Second, as they treat the DNNs as separate jobs, they cannot support the coordinations across DNNs in an ensemble, such as, the sharing of preprocessed data, cooperated checkpointing at the appropriate times.

Load balancing techniques for parallel computers such as nearest neighbor assignment to dynamically distribute workloads have been studied in (Kumar et al., 1994). The way FLEET distributes DNN training workloads are fundamentally different because an initial task assignment is not set at the beginning but dynamically determined based on both the convergence status and the training rate of each DNN.

8 Conclusions

This paper presents a systematic exploration on enabling flexible efficient ensemble training for heterogeneous DNNs. It addresses two-fold challenges. First, it formalizes the essence of the problem into an optimal resource allocation problem, analyzes its computational complexity, and presents an efficient greedy algorithm to effectively map DNNs to GPUs on the fly. Second, it develops a set of techniques to seamlessly integrate distributed data-parallel training of DNN, preprocessing sharing, and runtime DNN-to-GPU assignments together into a software framework, FLEET. Experiments on 100 heterogeneous DNNs on SummitDev demonstrate that FLEET can speed up the ensemble training by 1.12-1.92X over the default training method, and 1.23-1.97X over the state-of-the-art framework that was designed for homogeneous ensemble training.

Acknowledgement

We thank the anonymous MLSys reviewers for the helpful comments. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609 and CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

Caltech256 object category dataset caltechauthors. https://authors.library.caltech.edu/7694/, 12 2007. Accessed On 4/3/2019.

Summit user guide oak ridge leadership computing facility. https://www.olcf.ornl.gov/for-users/system-user-guides/summitdev-quickstart-guide/, 2019. Accessed 3/3/2019.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Canziani, A., Paszke, A., and Culurciello, E. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

Cheng, D., Rao, J., Guo, Y., Jiang, C., and Zhou, X. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, 2016.

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 571–582, 2014.

Chowdhury, M., Liu, Z., Ghodsi, A., and Stoica, I. {HUG}: Multi-resource fairness for correlated and elastic demands. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 407–424, 2016.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

Delimitrou, C. and Kozyrakis, C. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGARCH Computer Architecture News*, volume 42, pp. 127–144. ACM, 2014.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Du, J. and Leung, J. Y.-T. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2 (4):473–487, 1989.

Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., and Wong, P. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 1–34. Springer, 1997.

Garipov, T., Izmailov, P., Podoprikhin, D., Vetrov, D. P., and Wilson, A. G. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*, pp. 8789–8798, 2018.

Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44 (4):455–466, 2015.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 485–500, 2019.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., and Keutzer, K. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600, 2016.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pp. 2016–2025, 2018.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Kumar, V., Grama, A. Y., and Vempaty, N. R. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed computing*, 22(1):60–79, 1994.

Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Mathuriya, A., Bard, D., Mendygral, P., Meadows, L., Arnemann, J., Shao, L., He, S., Kärnä, T., Moise, D., Pennycook, S. J., et al. Cosmoflow: using deep learning to learn the universe at scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 819–829. IEEE, 2018.

Narayanan, D., Santhanam, K., Phanishayee, A., and Zaharia, M. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 69–84. ACM, 2013.

Patton, R. M., Johnston, J. T., Young, S. R., Schuman, C. D., March, D. D., Potok, T. E., Rose, D. C., Lim, S.-H., Karnowski, T. P., Ziatdinov, M. A., et al. 167-pflops deep learning for electron microscopy: from learning physics to atomic manipulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pp. 50. IEEE Press, 2018.

Pittman, R., Shen, X., Patton, R. M., and Lim, S.-H. Exploring Flexible Communications for Streamlining DNN Ensemble Training Pipelines. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.

Ratnaparkhi, A. A. and Pilli, E. Networks. *2016 International Conference on Emerging Trends in Communication Technologies (ETCT)*, pp. 1–6, 2012. doi: 10.1109/ETCT.2016.7882969.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Shmoys, D. B., Wein, J., and Williamson, D. P. Scheduling parallel machines on-line. *SIAM J. Comput.*, 24(6):1313–1331, December 1995. ISSN 0097-5397. doi: 10.1137/S0097539793248317. URL http://dx.doi.org/10.1137/S0097539793248317.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Turek, J., Wolf, J. L., Pattipati, K. R., and Yu, P. S. Scheduling parallelizable tasks: Putting it all on the shelf. In *ACM SIGMETRICS Performance Evaluation Review*, volume 20, pp. 225–236. ACM, 1992.

Urgaonkar, B., Shenoy, P., and Roscoe, T. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI): 239–254, 2002.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 595–610, 2018.

Xu, L., Butt, A. R., Lim, S.-H., and Kannan, R. A heterogeneity-aware task scheduler for spark. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 245–256. IEEE, 2018.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, pp. 7, 2008.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

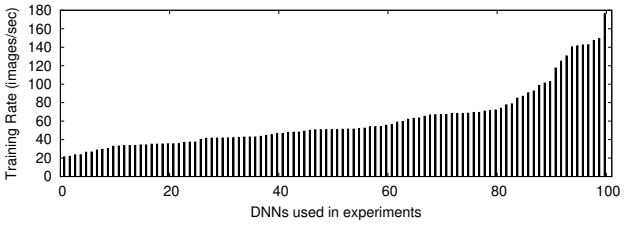

Figure 7: Training rate of each DNN on a single GPU.

A Complexity Analysis of the Greedy Allocation Algorithm

Algorithm 2 shows the flotilla creation algorithm. Flotilla creation first searches for the reference training rate (line 1, time complexity $\mathcal{O}(N)$), then iteratively finds the best candidate DNN to add in the flotilla (lines 3-11, time complexity $\mathcal{O}(N_k \times N \times M)$, and finally assigns all the remaining GPUs available to the DNNs in the flotilla (lines 12- 15, time complexity $\mathcal{O}(N_k \times M)$). So the time complexity of flotilla creation is $\mathcal{O}(N_k \times N \times M)$.

GPU assignment first prunes the factorial solution space by identifying and assigning GPUs to the DNNs whose training rate meets certain requirements in $\mathcal{O}(N_k)$ time complexity. It then searches for the optimal GPU assignment strategy for the remaining DNNs. The algorithm is shown in Algorithm 3. This algorithm assumes the number of GPUs per node is the same among nodes (*GPUsPerNode*), which holds in major supercomputers. It assigns GPUs to DNNs in the following order:

1. the DNNs whose required number of GPUs is a multiple of the number of GPUs per node; (lines 5-11)
2. the pairs of DNNs whose sum of the required number of GPUs is a multiple of the number of GPUs per node; (lines 12-24)
3. the remaining DNNs by searching for an optimal assignment of GPUs. (lines 25-39)

Let N'_k be the number of remaining DNNs. The solution space is $N'_k!$. Most of the time, N'_k is a small number less than five. However, enumerating all the possible solutions is still in factorial time complexity. We set the maximum number of solutions to explore as 1024, reducing the time complexity to $O(1)$. The time complexity of GPU assignment is thus $\mathcal{O}(N_k)$.

B Experiment Details

B.1 Characteristics of Experimental DNNs

The DNNs used in this experiment are derived from six popular DNNs, DenseNet-121, DenseNet-169, DenseNet-201, ResNet-50, ResNet-101 and ResNet-152. The first three are variations of DenseNet (Huang et al., 2017). The three variations share the same structure, but differ in the number of DNN layers, indicated by their suffixes. The latter three are variations of ResNet (He et al., 2016).

The 100 DNNs used in our experiments have a range of model sizes, from 232 MB to 1.19GB. Different DNNs have different GPU memory requirements and thus require different batch sizes to maximize GPU utilization. For each, we use the maximum batch size that can fit into GPU's memory. Figure 7 shows the distribution of their training rates on a single GPU which vary from 21 to 176 images/sec.

Figure 8 outlines the relations between the training rates and model sizes of the DNNs, as well as the relations between convergence rates (i.e., the number of epochs needed for the

Algorithm 2 createFlotilla

Input: $cands, R, M$ // The indices of DNNs that are not converged, the training rates of the DNNs, and the number of GPUs available
Output: $\mathcal{F}_k, \mathbf{m}_k$ // the k-th flotilla and the GPU count vector
1: D_{fast}, r_{fast} = fastestDNN($cands, R$) // Find the DNN with the largest training rate with a single GPU
2: $\mathcal{F}_k, M_k, \mathbf{m}_k = [D_{fast}], 1, [1]$
3: **while** $|\mathcal{F}_k| < |cands|$ **do**
4: $D_{best}, r_{best}, M_{best}$ = findNext($r_{fast}, R, cands, \mathcal{F}_k, M - M_k$) // Find the next DNN, its training rate and required GPU count
5: **if** $D_{best} == -1$ **then**
6: **break**
7: **end if**
8: $\mathcal{F}_k$.append(D_{best})
9: $\mathbf{m_k}$.append(M_{best})
10: $M_k += M_{best}$
11: **end while**
12: **while** $M_k < M$ **do**
13: $D_{slow} = slowestDNN(\mathcal{F}_k, \mathbf{m}_k, R)$ // in terms of speed on the currently assigned GPUs
14: $\mathbf{m_k}[slow] += 1$
15: $M_k += 1$
16: **end while**

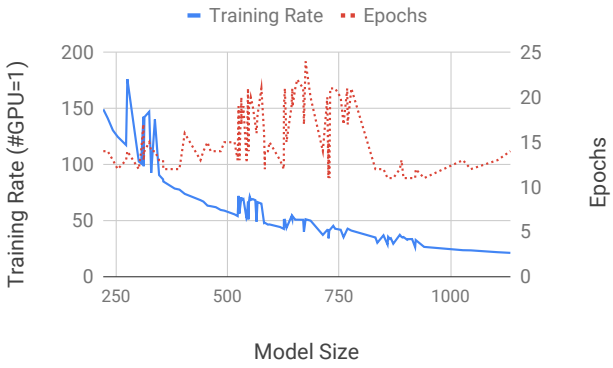

Figure 8: Correlations between model size of a DNN and the training rate and the number of epochs until convergence.

Algorithm 3 getGPUAssignment

Input: $\mathcal{F}_k, \mathbf{m_k}$ // The k-th flotilla and the GPU count vector
Output: A_k

$j = 1$ // The current available GPU with the smallest index.
$A_k = 0_{N_k,M}$ // The GPU assignment matrix of dimension $N_k \times M$
$remaining = \{1, \cdots, N\}$ // The indices of DNNs to allocate GPUs
$assigned = \{\}$ // The indices of DNNs that have assigned GPUs
for all $i \in remaining$ **do**
  if $m_i^{(k)} \% GPUsPerNode == 0$ **then**
    $assigned$.add(i)
    j = assignGPUs($A_k, i, j, m_i^{(k)}$)
  end if
end for
$remaining$ -= $assigned$
memo, assigned = {}, {}
for all $i \in remaining$ **do**
  if $-m_i^{(k)} \% GPUsPerNode$ not in $memo$ **then**
    $memo[m_i^{(k)} \% GPUsPerNode] = i$
  else
    for $ii \in \{i, memo[-m_i^{(k)} \% GPUsPerNode]\}$ **do**
      $assigned$.add(ii)
      j = assignGPUs($A_k, ii, j, m_{ii}^{(k)}$)
    end for
    del $memo[m_i^{(k)} \% GPUsPerNode]$
  end if
end for
$remaining$ -= $assigned$
if $|remaining| > 0$ **then**
  $\tilde{\mathbf{m}}_{(k)}, bestScore, bestA, j_{copy}, A_{copy} = [], \infty, j, clone(A_k)$
  for all $i \in remaining$ **do**
    $\tilde{\mathbf{m}}_{(k)}$.append($(i, m_i^{(k)})$)
  end for
  for $permutation$ in allPermutations($\tilde{\mathbf{m}}_{(k)}$) **do**
    $j, A_k = j_{copy}, clone(A_{copy})$
    for $i, m_i^{(k)}$ in $permutation$ **do**
      j = assignGPUs($A_k, i, j, m_i^{(k)}$)
    end for
    $score$ = calculateScore(A_k) // Score is calculated based on the loss function in Eq. 7
    if $score < bestScore$ **then**
      $bestScore, bestA = score, A_k$
    end if
  end for
  $A_k = bestA$
end if

Algorithm 4 assignGPUs

Input: $A_k, i, j, m_i^{(k)}$
Output: j // The index of the next available GPU to assign

while $m_i^{(k)} > 0$ **do**
  $a_{i,j}^k = 1; j += 1; m_i^{(k)} -= 1$
end while

DNNs to converge) and their model sizes. As model size increases, the training rate tends to drop as more computations are involved in the DNN, but there are no clear correlations with the convergence rate. It is the reason that the resource allocation algorithm in FLEET primarily considers training rate explicitly, and relies on the periodical (re)scheduling to indirectly adapt to the variations of DNNs in the converging rates.

B.2 System Settings

All experiments are conducted on SummitDev (Sum, 2019), a development machine for Summit supercomputer at Oak Ridge National Lab. Each node is equipped with two IBM POWER8 CPUs and 256GB DRAM, and four NVIDIA Tesla P100 GPUs. Each POWER8 CPU has 10 cores with 8 HW threads each. The default SMT level is set to one unless noted otherwise. The number of cores allocated per GPU is five in all the experiments. NVLink 1.0 is the connection among all GPUs and between CPUs and GPUs within a node. EDR InfiniBand connects different nodes in a full fat-tree. The file system is an IBM Spectrum Scale file system, which provides 2.5 TB/s for sequential I/O and 2.2 TB/s for random I/O. Our experiments show that thanks to the large I/O throughput of the file system, I/O is not the bottleneck of DNN training. The used CUDA version is 9.2.

FLEET is built on Tensorflow 1.12 (as the core training engine), Horovod v0.15.2 (Sergeev & Del Balso, 2018) (as the basis for distributed DNN training), and mpi4py v3.0.0 (for the pipeline construction). We set `inter_op_parallelism_threads` and `intra_op_parallelism_threads` to # logical cores for parallel TensorFlow operaitons on CPU. The used CUDA version is 9.2.

B.3 Profiling Details

To minimize the overhead of profiling, we only profile the training rates of each DNN in the ensemble with the number of GPUs varying from one to $M_t (M_t < M)$. For profiling on m ($m = 1, \cdots, M_t$) GPUs, we train a DNN for a maximum of 48 batches and use the training time of the last 20 batches to calculate the exact training rate: $r_i(m), i = 1, \cdots, N$. Based on the profiled training rates, we estimate the training rates of each DNN when $m > M_t$. Specifically, the profiling has three steps:

1. Collect the training rates of each DNN on a single GPU, $R(1) = \{r_i(1)\}, i = 1, \cdots, N$.

2. Estimate the number of GPUs required to make the DNN that has the smallest training rate on a single GPU achieve the largest single-GPU training rate, $M_a = \lceil \frac{\max(R(1))}{\min(R(1))} \rceil$.

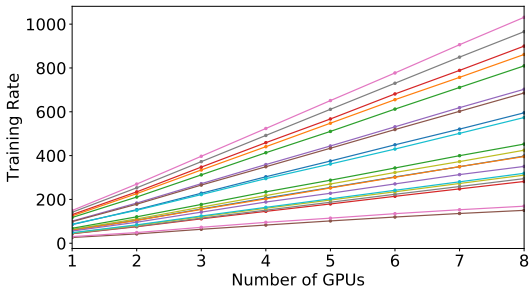

Figure 9: The profiled training rates (images/sec) of 100 DNNs in an ensemble with Imagenet.

3. Collect the training rates of each DNN with the number of GPUs varying from two to $M_t = \max(M_a, M_b)$, where $M_b = 2 \times GPUsPerNode$.

Note that steps 1 and 3 can be done in parallel because the trainings of different DNNs with different number of GPUs are independent. The training rate of the i-th DNN with the number of GPUs higher than M_t is estimated via the following equation:

$$r_i(m) = m \times \frac{r_i(M_b)}{M_b} \times \Big(\frac{r_i(M_b)}{r_i(M_b - 1)} \times \frac{M_b - 1}{M_b}\Big)^{m - M_b}. \quad (5)$$

The formula for M_b and Equation 5 are the result of performance modeling on our observations on the DNN performance trend as illustrated in Figure 9. It achieves a good tradeoff between the profiling cost and the performance prediction accuracy.

The profiling process also measures the throughput of a range of preprocessors (#cores=1, 2, 4, 8, 16, 32) in the pipeline. This step is quick since preprocessing does not exhibit large variations. Based on the profiled information, FLEET calculates the minimum number of preprocessors that can meet the demands of an arbitrary M DNNs (with one running on one GPU), and uses it to set the number of preprocessors.