2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-6654-7315-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS56603.2022.00029

Optimizing Resource Allocation in Pipeline Parallelism for Distributed DNN Training

Yubin Duan
Temple University, USA
Email: yubin.duan@temple.edu

Jie Wu
Temple University, USA
Email: jiewu@temple.edu

***Abstract*—Deep Neural Network (DNN) models have been widely deployed in a variety of applications. Driven by privacy concerns and great improvement in the computational power of mobile devices, the idea of training machine learning models on mobile devices has become more and more important. Directly applying parallel training frameworks designed for data center networks to train DNN models on mobile devices may not achieve the ideal performance, since mobile devices usually have multiple types of computation resources such as ASIC, neural engine, and FPGA. Moreover, the communication time is not negligible when training on mobile devices. With the objective of minimizing DNN training time, we propose to extend the pipeline parallelism, which can hide the communication time behind computation for DNN training by integrating the resource allocation. Fine-tuning the ratio of resources allocated to forward and backward propagation can improve resource utilization. We focus on homogeneous workers and theoretically analyze the ideal cases where resources are linearly separable. We also discuss the model partition and resource allocation for a more realistic case. Additionally, we investigate the heterogeneous worker case. Trace-based simulation results show that our scheme can efficiently reduce the time cost of a training iteration.**

***Index Terms*—DNN training, pipeline parallelism, machine learning, model partition, resource allocation**

I. Introduction

Machine learning models, especially deep neural networks (DNNs), have been widely deployed in a variety of applications. With the rapid growth of available training data volume, sizes of DNN models continuously increase to improve the model accuracy. A large number of parameters in DNN models makes the training process time-consuming. Usually, the large DNN models are trained in data center clusters. However, the service provider would acquire users' privacy data when training their DNN models, which raises data security concerns. Moreover, the computation power of mobile devices has been greatly improved in recent years. Driven by these factors, the idea of training machine learning models on a group of mobile devices is proposed [1]. For example, a federated learning framework [2] keeps data locally and trains DNN models on mobile devices. It is necessary to develop an efficient distributed training scheme to speed up the training process on a group of mobile devices.

Training DNN models is an iterative process. Each iteration has two phases: forward and backward propagation. In *forward propagation*, training data is passed from input layers to output layers. Each intermediate layer processes the output from the previous layer using its corresponding model parameters and activation functions. *Backward propagation* traverses the DNN in reverse order. It calculates the gradient of model parameters and updates the parameters accordingly. The backward propagation cannot start until the result of the forward propagation is calculated. Efficiently training DNN models in parallel is challenging because of the bi-directional propagation property and the fact that model parameters are frequently modified.

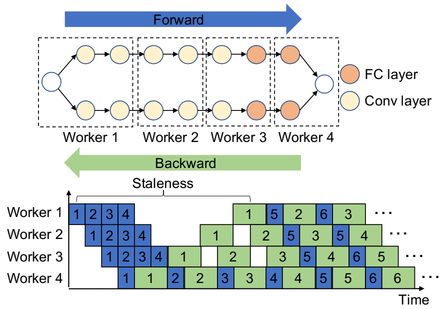


Fig. 1. A DNN training pipeline for AlexNet.

There are many parallelisms that are proposed to efficiently train DNN models in data center networks. Data parallelism is a commonly used approach, in which the training data is divided and assigned to training devices. Workers need to frequently synchronize the model parameters via parameter servers [3] or AllReduce communication [4]. However, data parallelism suffers from a large communication cost [5]. Model parallelism is another efficient approach to organize multiple devices. In model parallelism, DNN model parameters are partitioned by layers. Each device is assigned to a subset of parameters. However, the resource utilization ratio is not high in model parallelism.

Notably, by expanding on the model parallelism, [5]–[7] introduce pipeline parallelism that allows scalable DNN training. Similar to model parallelism, the DNN model is divided among workers. Each worker has a set of consecutive DNN layers. Importantly, the forward and backward propagation phases of different input data are overlapped in a pipelined manner. An example of pipeline parallelism is illustrated in Fig. 1. The forward and backward propagation operations are labeled by blue and green blocks, respectively. The number in each block indicates the input data ID. A row of blocks shows the operations taken by the corresponding worker in each time

This research was supported in part by NSF grants CNS 2128378, CNS 2107014, CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, and CNS 1651947.

978-1-6654-7315-6/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS56603.2022.00029

slot. Worker 1 first performs the forward propagation for input data 1 to 4 and then runs the backward propagation for input data 1 after waiting for the output of worker 2. We use the blocks in Fig. 1 labeled by 1 to explain the procedures of a DNN training iteration.

Existing distributed training frameworks are usually designed for data center networks. Directly applying those frameworks to train DNNs on mobile devices may not achieve the ideal performance, considering that mobile devices have unique properties compared to data center clusters. Mobile devices usually have a different type of computation unit, which usually contains a CPU and one or more cores for specialist computation such as GPU, FPGA, and neural engines [8], [9]. Therefore, we need to consider resource allocation when training DNN models on mobile devices. Without proper allocation, there would be idle bubbles in the training pipeline as shown in Fig. 1, which enlarges the training makespan. In addition, the communication latency of mobile communication is usually larger than that in data center networks. Moreover, mobile devices may have different hardware configurations.

Considering those observations, we propose to extend pipeline parallelism for efficient DNN training in mobile devices. We focus on pipeline parallelism since it can reduce the training time by hiding the communication phase behind the computation phase. Compared to data parallelism, using pipeline parallelism can reduce the communication overhead. Noticing that each worker device could have multiple types of computational resources, we put the resource allocation into consideration when partitioning DNN models among workers. We theoretically analyze the model partition for homogeneous workers where all worker devices are identical. For the ideal case where all resources are linearly separable, an optimal model partition algorithm based on binary search is demonstrated. Then, we extend the algorithm to a more realistic case where each device has two fixed types of computational resources. Additionally, we discuss the pipeline parallelism for heterogeneous workers that have different computational power. Inspired by [10], we organize the pipeline into different waves to fit different devices.

The contributions of our paper are summarized as follows:

- We investigate the pipeline parallelism for training DNNs on mobile devices, which have multiple types of computational resources.
- We theoretically analyze the DNN model partition problem among homogeneous workers and take the resource allocation into consideration.
- We discuss the pipeline parallelism for heterogeneous training devices and illustrate a clustering algorithm to organize those devices.
- Our trace-based simulation on several widely deployed DNN models shows that our scheme can efficiently improve resource utilization and reduce the training time.

II. Related Work

Data parallelism distributes training data samples among workers. Workers need to frequently synchronize local parameters with each other. The synchronization frequency can adjust the trade-off between communication overhead and model convergence or training accuracy. Bulk synchronous parallel [11] requires workers to wait for each other to finish each training iteration. Asynchronous Parallel [12] allows a worker to use stale versions of model parameters without waiting, but the training may not converge. Stale synchronous parallel [13] allows parameter staleness but it is bounded by a pre-defined threshold. The communication overhead during synchronization is usually large. Existing approaches that minimize the communication cost include optimizing data and parameter allocation [14] and model compression [15]. Different from those approaches, we use pipeline parallelism to avoid the massive communication volume caused by transmitting the whole set of parameters.

Model parallelism splits DNN models among available workers. Workers only need to send gradients across each other instead of the whole parameter set, which helps to reduce the communication burden. Challenges in model parallelism include efficient model partition [16] and resource allocation [17]. Moreover, the model parallelism may underutilize available resources [5], [6]. To overcome the shortcomings of the model parallelism, GPipe [6] presents pipeline parallelism, in which forward and backward propagation is overlapped in a pipeline. PipeDream [5] further improves the resource utilization in the pipeline parallelism by avoiding frequent pipeline flushes. HetPipe [10] extends the pipeline parallelism for heterogeneous GPU clusters. Different from those approaches, we consider the resource allocation problem when partitioning DNN models for pipeline parallelism.

Another approach to accelerate DNN training is from the perspective of system architecture optimization. PatDNN [18] presents an architecture-aware compiler to optimize the DNN training on mobile devices. This approach focuses on improving the DNN training speed for each individual device. In contrast, we consider the DNN training among a group of mobile devices. Notably, we assume each device can have multiple types of computation resources. It has been shown that integrating multiple types of computation units to hardware architecture can accelerate DNN-related applications [19]. Integrating different types of computation resources could speed up DNN training.

III. Model

A. Pipeline Parallelism of DNN Training

Before formulating the model partition problem, we first introduce the procedures of DNN training in pipeline parallelism. Training a DNN model consists of repeated iterations of *forward* and *backward propagation*. Let L denote a DNN model and l_i denote the $i-$th layer in the DNN model. Let l_0 and l_k denote the input and output layers, respectively. Training a DNN model can be viewed as repeatedly adjusting parameters in functions represented by DNN layers such that a loss function is minimized. Let $\phi(w_1, w_2, \ldots, w_k)$ denote the loss function, where w_i refers parameters in layer l_i. After the backward propagation of iteration t, the model parameter for the following iteration $t+1$ is updated using the following equation: $w^{(t+1)} = w^{(t)} - \alpha \cdot \nabla\phi(w_1, w_2, \ldots, w_k)$, where α is the learning rate, and $\nabla\phi$ denotes the gradient of ϕ.

In pipeline parallelism, the model parameters are partitioned into multiple parts and are allocated among available workers. Let $V = \{v_1, v_2, \ldots\ v_n\}$ denote a set of workers. It is worth noting that forward and backward operations among workers are organized in a pipelined manner. Multiple workers could simultaneously run forward and backward propagation on input data of *different* minibatches. Let d_i denote the minibatch of input data with label i. Additionally, we assume forward and backward propagation on all workers takes a fixed period of time. Then, in the startup phase of the pipeline, i.e., before the backward propagation of d_1 is completed, v_i can start forward propagation of input d_{j+1} when v_{i+1} is executing the forward propagation of input d_j. After the startup phase, v_i may work on the forward propagation of d_{j+n-i} or the backward propagation of d_{j-1} when v_{i+1} is running the backward propagation of input d_j, where we assume the pipeline depth is n.

B. Network Model

We consider training DNN models among mobile devices using pipeline parallelism. We first focus on the case of homogeneous workers that have identical hardware configurations. Let $R = \{r_1, r_2, \ldots, r_m\}$ denote the set of different types of computational resources available on workers, where m is the total number of resource types. The value of r_j represents the computational power of the j-th type of resource. The computational power can be quantified by the number of floating operations per second. Let b_i denote the communication bandwidth of worker v_i. For homogeneous workers, we omit the subscript and use b for simplicity. Both the computational power and the communication bandwidth can be evaluated by running benchmark tests in advance. Therefore, we assume that the values of r_j and b are known before partitioning the DNN model among workers.

When using pipeline parallelism, we need to partition the DNN layers. Let q denote the pipeline depth. For homogeneous workers, we assume $n = q$. This assumption is feasible even if the number of mobile devices is greater than q since we can evenly group available devices into q virtual workers. If the mobile devices cannot be evenly divided, we treat such cases as heterogeneous settings, which are discussed in Section V. When $n = q$, the DNN model needs to be partitioned into q parts. Let $p_i = (l_s, l_t)$ denote the partition of DNN layers assigned to v_i, where l_s and l_t represent the first and last layer of v_i's partition, respectively. The partition p_i inclusively contains the DNN layers from l_s to l_t.

Our objective is to minimize the DNN training time. We use τ to denote the overall makespan of DNN training. The formulation of the makespan is based on the computation time and communication time costs of workers. Let $f(p_i, r_j)$ denote the computation time of executing forward propagation of the DNN partition p_i on resource r_j of v_i. Similarly, let $g(p_i, r_j)$ denote the backward propagation time of p_i on r_j for worker v_i. We use the function $h(p_i, b_i)$ to denote the communication time of transmitting the output of p_i on the communication channel with bandwidth b_i. The communication volume consists of the output of forward or backward propagation, and is fixed once the partition p_i is determined. In general cases, it is difficult to formulate these functions. We first theoretically analyze an ideal case where these functions are linear and the resources are linearly separable. Additionally, we discuss a more realistic case where we profile the computation and communication time on two types of resources and predict the function values with regression models.

To simplify the theoretical analysis, we assume that the computation and communication time are linear models. For each type of resource r_j, the computation time is proportional to the ratio of computational workload. The workload of a DNN partition can be measured by the number of floating operations (flops) needed before completion. Formally, for forward propagation, $f(p_i, r_j) \propto p_i / r_j$, where we use p_i to represent the workload of the corresponding DNN partition in order to reduce the number of notations. Similarly, for backward propagation, $g(p_i, r_j) \propto p_i / r_j$. The communication time is also a linear function with respect to the bandwidth b.

We assume that the computational resources are linearly separable. We can assign a portion of an arbitrary type of resource r_j to either forward or backward operations, without causing additional computation overhead. Then, we have the following definition for the resource allocation ratio.

Definition 1: For any worker, let β_j $(0 \leq \beta_j \leq 1)$ denote the percentage of resource r_j which is allocated for forward propagation. The ratio of r_j that is allocated for backward propagation is $1 - \beta_j$.

The ratio β_j does not change over time since the DNN partition is fixed during training in pipeline parallelism. In the linear model, the time consumption for forward and backward proportion is enlarged by $1/\beta_j$ when the computation power is reduced to $\beta_j r_j$. Similarly, we assume that the computation workload p_i can be linearly divided among different resources. Formally, $f(p_i, \beta_j r_j)/f(p_i, r_j) = g(p_i, \beta_j r_j)/g(p_i, r_j) = 1/\beta_j$. Note that $f(p_i, \beta_j r_j) \neq g(p_i, \beta_j r_j)$ since the same type of resources may have different efficient processing speeds when calculating forward and backward propagation. By adjusting the ratio of β_j, we may have a better origination of pipeline parallelism and reduce the resource idle time.

In our model, workers can simultaneously perform forward and backward propagation on its available computational resources in parallel. Different from the traditional pipeline parallelism, where a worker can at most perform one operation at a time, our scheme further enhances the parallelism level for workers with multiple types of resources. Fig. 2 shows the timetable for our extended pipeline parallelism. We allow the parallel execution of forward and backward propagation on the same worker. Data $1'$, $2'$, $3'$, and so on is inserted to fulfill the pipeline. Allowing the parallel execution not only makes full use of computational resources, but also provides opportunities to fine-turn the duration of forward and backward operations. We also enable parallel execution on works with one type of resource using time sharing. This

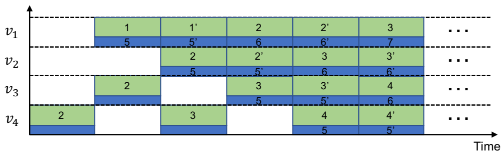


Fig. 2. An illustration of pipeline parallelism.

gives us an opportunity to adjust the resource allocation ratio. A proper resource allocation can reduce the resource idle time and training makespan. We need to carefully adjust the ratio of resource allocation on forward and backward operations. Therefore, we propose to consider the resource allocation problem when partitioning a DNN model among workers.

C. Problem Formulation

Our objective is to minimize the DNN training time. The training makespan is mainly determined by the number of iterations and the duration of each iteration. We assume that the number of iterations is a constant value and focus on the duration of forward/backward propagation and communication. The convergence properties of our training scheme are analyzed in Section IV. As shown in Fig. 2, the forward propagation, backward propagation, and communication time in our extended pipeline parallelism are overlapped. Fig. 3 provides a closer look at the pipeline execution on worker v_i. Except during the pipeline startup phase, the DNN training time is repeatedly built by the time period shown in Fig. 3. We notice that the operation that has the largest length dominates the training time and is the bottleneck of the training pipeline and we aim to minimize the longest operations.

In our scheme, the lengths of forward and backward propagation are mainly related to resource allocation and the DNN model partition. Specifically, model partition determines the workload of each worker and the resource allocation is used to ensure each worker can achieve the maximum speedup. Notably, the forward propagation time for DNN partition p_i can be formulated as $f(p_i, \sum_{j=1}^{m} \beta_j r_j)$, when considering the resource allocation. Accordingly, the backward propagation time of p_i is $g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)$. The communication time is $h(p_i, b)$, which is only varied with partition p_i in our model.

Our problem can be formulated as the following equations:

$$\min \max_{v_i \in V} \Big\{ f(p_i,\sum\nolimits_{j=1}^{m}\beta_j r_j), g(p_i,\sum\nolimits_{j=1}^{m}(1-\beta_j)r_j), h(p_i,b) \Big\} \quad (1)$$

$$s.t.\ 0 \le \beta_j \le 1, \forall 1 \le j \le m \quad (2)$$

$$\cup_{v_i \in V}\, p_i = L \quad (3)$$

Eq. (1) shows our objective of minimizing the duration of bottleneck operations in DNN training. Eq. (2) shows the resource allocation constraint. We can split each type of resources for forward and backward propagation operations. β_j is the ratio of resources allocated to forward propagation and cannot exceed 1. Eq. (3) is the partition constraint. The union of partitions allocated to all workers should cover all of the layers of the DNN model L.

IV. Pipeline for Homogeneous Workers

A. Resource Allocation

The resource allocation aims to balance the time consumption of forward and backward propagation. When analyzing

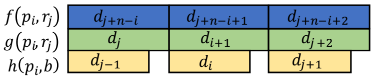


Fig. 3. Closer look at pipeline execution. (Blue for forward propagation, green for backward propagation, and yellow for communication)

the resource allocation, we assume that the DNN partition p_i for worker v_i is given. To simplify theoretical analysis, we focus on the ideal case where resources are linearly separable. When workers are homogeneous, the available resources are identical on all workers. We denote w_i as a representative in our discussion. We build the linear programming model to calculate the optimal β_j for each type of resource r_j.

We only consider the forward and backward propagation time for resource allocation since the communication volume does not change with allocation ratios. The objective of resource allocation is to minimize $\max\Big\{f(p_i, \sum_{j=1}^{m}\beta_j r_j), g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)\Big\}$. The following lemma shows that $f(p_i, \sum_{j=1}^{m}\beta_j r_j)$ equals to $g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)$ for β_j in the optimal solution.

Lemma 1: In the optimal resource allocation, $f(p_i, \sum_{j=1}^{m}\beta_j r_j) = g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)$.

Proof: We use proof by contradiction. Assume $f(p_i, \sum_{j=1}^{m}\beta_j r_j) \neq g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)$ in the optimal solution. W.l.o.g, we assume $f(p_i, \sum_{j=1}^{m}\beta_j r_j) > g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j)$. Then, the objective function becomes $f(p_i, \sum_{j=1}^{m}\beta_j r_j)$. We can reduce $f(p_i, \sum_{j=1}^{m}\beta_j r_j)$ by increasing β_j (for any $beta_j > 0$) by a small value ϵ. In our linear model, the forward propagation time f monotonically decreases with more resources or larger β_j. The value of the objective function decreases, accordingly. β_j is not the optimal solution and there is a contradiction. Therefore, the lemma holds in the optimal solution. ■

According to Lemma 1, the resource allocation is equivalent to the following linear programming problem.

$$\min \quad f(p_i, \sum\nolimits_{j=1}^{m}\beta_j r_j) \quad (4)$$

$$s.t. \quad f(p_i, \sum\nolimits_{j=1}^{m}\beta_j r_j) = g(p_i, \sum\nolimits_{j=1}^{m}(1-\beta_j)r_j) \quad (5)$$

$$0 \le \beta_j \le 1, \forall 1 \le j \le m \quad (6)$$

The problem shown by Eq. (4) can be efficiently solved by linear programming solvers. Additionally, it has a closed-form solution when $f(p_i, r_j)/g(p_i, r_j) = c, \forall 1 \le j \le m$, where c is a constant. The value c represents the ratio between the forward and backward propagation time which together use the whole portion of resource r_j. It is a common assumption that the ratio is fixed when analyzing the pipeline parallelism [5], [6]. The following theorem shows the optimal resource allocation in such cases.

Theorem 1: The optimal resource allocation ratio $\beta_j = c/(c+1)$, if $f(p_i, r_j)/g(p_i, r_j) = c, \forall 1 \le j \le m$, where c is a constant.

Proof: We use the KKT conditions in the optimization theory to show that $\beta_j = c/(c+1)$ for $1 \le j \le m$ is the optimal solution to the problem. Specifically, in the linear model, $f(p_i, \sum_{j=1}^{m}\beta_j r_j) = c_1 p_i/(\sum_{j=1}^{m}\beta_j r_j)$ and $g(p_i, \sum_{j=1}^{m}(1-\beta_j)r_j) = c_2 p_i/(\sum_{j=1}^{m}(1-\beta_j)r_j)$, where c_1

and c_2 are constant coefficients. From $f(p_i, r_j)/g(p_i, r_j) = c$, we know that $c_1/c_2 = c$. For simplicity, we omit function parameters in the following proof and use f and g to denote $f(p_i, \sum_{j=1}^{m} \beta_j r_j)$ and $g(p_i, \sum_{j=1}^{m}(1 - \beta_j)r_j)$, respectively. Then, we show that the stationarity condition holds when $\beta_j = c/(c + 1)$. Specifically, $\frac{\partial f}{\partial \beta_j} = \frac{c_1 r_j p_i}{(\sum_{j=1}^{m} \beta_j r_j)^2}$ and $\frac{\partial g}{\partial \beta_j} = \frac{c_2 r_j p_i}{(\sum_{j=1}^{m}(1-\beta_j) r_j)^2}$. Using $\beta_j = c/(c + 1)$ and the dual parameter $u = -2/c$, we have $2\frac{\partial f}{\partial \beta_j} - u\frac{\partial g}{\partial \beta_j} = 0$ for all $1 \le j \le m$, which shows that the stationarity condition is satisfied. What's more, the complementary slackness, primal feasibility, and dual feasibility also hold. According to the KKT conditions, $\beta_j = c/(c+1)$ is the optimal solution to the primal problem. ■

B. DNN Model Partition

The resource allocation balances the forward and backward propagation time for a individual worker. From Fig. 2, we notice that the it is also important to balance workload among different workers. Otherwise, workers with an extremely large workload would be stragglers and enlarge the training makespan. In DNN partitioning, we also need to consider the communication time in different partition besides measuring the computation time of forward and backward propagation. We take the resource allocation into consideration when partitioning DNN models and illustrate an efficient DNN partition algorithm based on binary search.

DNN partitioning aims to balance the workload among different workers and reduce the training makespan, which can be achieved by calculating the resource allocation β_j for available resources of each worker. The communication time is evaluated by the function $h(p_i, b)$. The forward and backward communication volume of a partition p_i is determined by the dimensions of the first and last layer in the partition, respectively. Those values can be quickly retrieve from a lookup table given a partition p_i. It is difficult to find a closed-form solution, and exploring all possible partitions costs exponential time.

Although it is difficult to directly find the optimal partition, we can verify if a feasible partition exists given a partition limitation. Partition limitation is an upper bound restriction on the longest operations among all workers. Formally, given a limitation x, a feasible partition plan should guarantee that $\max_{v_i \in V} \left\{ f(p_i, \sum_{j=1}^{m} \beta_j r_j), g(p_i, \sum_{j=1}^{m}(1 - \beta_j) r_j), h(p_i, b) \right\} \le x$. The feasibility check can be done in linear time. Starting from the input layer l_0 and empty partitions, we greedily find the longest continuous DNN layers such that the time costs of forward, backward, and communication operations on the partition consists of those layers which do not exceed the limitation. The greedy partition repeats until the all layers are processed. If the greedy strategy can partition the DNN in to n parts or less (but more than 0), the given partition limitation is feasible. Otherwise, the limitation is increased. DNN layers would at most be scanned n times, and the feasibility check can be completed in $O(nk)$ time.

Considering the feasibility check is efficient, we propose to use binary search to find the smallest feasible partition limitation. Specifically, the lower bound of the feasible partition limitation is $\min_{l_i \in L}\{f(l_i, r_j), g(l_i, r_j), h(l_i, b)\}$, since such a partition would at least contain one DNN layer. There does not exist a feasible partition limitation that is lower than this value. Similarly, the upper bound of the feasible partition limitation is $\max_{l_i \in L}\{f(L, r_j), g(L, r_j), h(l_i, b)\}$. $f(L, r_j), g(L, r_j)$ represent the longest computation time when the whole DNN model is assigned to a single worker. $\max_{l_i \in L} h(l_i, b)$ finds the worst communication time. We can use binary search to find the smallest feasible partition limitation in the range with granularity ϵ, where ϵ is a small constant. Through the value of ϵ, we can adjust the trade-off between time efficiency and performance of the partition algorithm.

Algorithm 1 DNN Partition

```
Input:  DNN model $L$
Output: The partition of DNN layers for each worker
1: Initialize $x \leftarrow \min_{l_i \in L}\{f(l_i, r_j), g(l_i, r_j), h(l_i, b)\}$
2: Initialize $y \leftarrow \max_{l_i \in L}\{f(L, r_j), g(L, r_j), h(l_i, b)\}$
3: while $x < y$ do
4:     $mid \leftarrow (x + y)/2$
5:     if $P \leftarrow$ partition DNN with limitation $mid$ is feasible then
6:         $x \leftarrow mid + \epsilon$
7:     else
8:         $y \leftarrow mid$
9: return $P$
```

The procedure of our DNN partition algorithm is shown in Alg. 1. Lines 1 and 2 initialize the upper and lower bounds of the partition limitation. Variables x and y store lower and upper bounds, respectively. Then, the following loop searches for the smallest feasible limitation. Specifically, we iteratively update the partition limitation using the average value of the current bounds x and y. We attempt to partition the DNN model using mid as the partition limitation with our greedy strategy. If there is no feasible partition, it means the partition limitation mid is too small. In line 6, we increase the lower bound x to $mid + \epsilon$, where ϵ a constant hyperparameter. Otherwise, the limitation mid is too large and we decrease y to mid in line 8. The while loop terminates when $x = y$. Finally, we return the last feasible set P which contains partition p_i for each worker v_i. The smallest feasible partition limitation is stored in x.

We use a loop invariance to show the correctness of our DNN partition algorithm. In Alg. 1, the while loop guarantees that $x - \epsilon$ is always an infeasible partition limitation and r always stores a feasible partition limitation. Specifically, in the while loop, if the branch at line 6 is taken, then we know mid is an infeasible limitation. After assigning $mid + \epsilon$ to x, $x - \epsilon$ is infeasible. On the other hand, if the branch at line 8 is taken, then mid is a feasible limitation. After updating in line 8, $y = mid$ is also a feasible limitation. Those invariant properties hold during the while loop. Finally, when the loop terminates, we have $x = y$. According to the loop invariance, $x - \epsilon$ is infeasible and $x = y$ is feasible. This means that

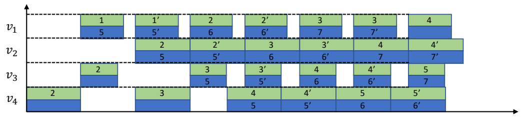


Fig. 4. Idle blocks in heterogeneous settings.

x is the smallest feasible limitation under search granularity ϵ. Therefore, the up-to-date partition P corresponding to x stores the optimal DNN partition that minimizes our objective function shown in Eq. (1).

C. Convergence Analysis

In pipeline parallelism, there are staleness issues when considering parameter updates. Specifically, when a worker performs a forward propagation, it may not use the most up-to-date model parameters. For example, in Fig. 2, when worker 1 performs the forward propagation on input data 6, it can only use the model parameters updated by data 1 instead of all first five pieces of data. It is because the backward propagation on data 2, 3, and 4 have not been passed to worker 1 yet. Staleness is related to model convergence. If staleness is not bounded, the DNN model may not converge during training.

Staleness refers to the number of missing updates when processing data from the current minibatch. The staleness can be bounded by using the weight stashing techniques proposed in [5]. Specifically, a worker may keep multiple versions of model parameters, and use the most up-to-date version available when performing the forward propagation. After each round of forward propagation, a copy of the model parameters is stored in the memory. After k steps delay, the copy is used to calculate the backward propagation on the same set of data minibatch. There is a trade-off between memory cost and the staleness bound or the model accuracy. We show that the model still converges with a staleness bound.

Following the analysis in [13], we show that the DNN model regret is bounded with an assumption that the component function ϕ is convex and has bounded subdifferential $||\nabla\phi(W)|| \leq c_1$, where c_1 is a constant. The new characteristic used in the proof is that the distance between model updates are bounded by another constant c_2. Let s denote the staleness of the model update, the regret can be bounded as shown in the following theorem.

***Theorem** 2:* Suppose the component function ϕ is convex, its subdifferential $||\nabla\phi(W)|| \leq c_1$, and the distance between two updates is bounded by c_2, where c_1 and c_2 are constants. Then, the regret is bound as $R[W] \leq 4c_1c_2\sqrt{2ns/T}$, where T is the number of updates.

Proof: The proof follows procedures similar to those shown in [13]. The key characteristic is that the the staleness is bounded by a constant value in pipeline parallelism. ■

V. Pipeline for Heterogeneous Workers

If a worker in the pipeline has different computation abilities, there will be resource idle slots during training as shown in Fig. 4. It would reduce the resource utilization and enlarge the training makespan. Therefore, we aim to group heterogeneous devices into q groups, where q is the pipeline depth, such that the difference of computation abilities among workers is minimized. We assume devices in the same group can collaborate without additional overhead. Formally, each device $v \in V$ has a resource vector that indicates the computation speed of each resource r_j. The set of heterogeneous devices are grouped into q workers. To separate mobile devices and workers, we use $V_i \subset V$ to denote a worker which contains multiple devices. We assume that the resource vector of V_i is the summation of resource vectors of devices $v \in V_i$. Formally, the computation ability of a worker V_i is evaluated by $\max f(p_i, \sum_{v\in V_i} \sum_{j=1}^{m} r_j), g(p_i, \sum_{v\in V_i} \sum_{j=1}^{m} r_j)$. A higher value means a longer computation time and a weaker computation power.

The objective of our device group problem is to minimize the training duration of the slowest worker. Formally, we aim to find a set of workers to $\min \max_{V_i}\{f(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j),g(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j)\}$. It is not trivial to evenly group the heterogeneous devices. Even if the cost function is submoduler, it is challenging to achieve an average partition [20]. We follow a local search heuristic to group heterogeneous devices. Start from an empty set, we iteratively choose the worker that has the smallest computation ability (or have the largest value of $\max\{f(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j), g(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j)\})$ and then give it an unassigned worker that would induce the largest decrease on the objective function value.

The detailed procedure of our device group algorithm is shown in Alg. 2. Specifically, line 1 initializes workers as an empty set, i.e, contains no devices. Then, the loop in lines 2-3 initializes the cost function for each worker. The cost function value represents the time duration of the overlapped forward and backward propagation. The following while loop in lines 4-8 updates the device partition. Line 5 chooses the worker with the largest cost which is the current bottleneck. If there are several workers that have the same cost, we arbitrarily choose one. Line 6 picks a device that has not been assigned yet. The device which can reduce the cost function the most is chosen. Line 7 adds the chosen device to worker V_i. Line 8 removes it from the unsigned device list. Then, the loop repeats until all devices are properly assigned to workers. Eventually, workers that group heterogeneous devices are returned.

***Theorem** 3:* The time complexity of the worker grouping algorithm is $O(n^2m)$.

Proof: Lines 1-3 use linear time to initialize the worker set and the cost functions. Within the while loop, line 5 takes at most $O(q)$ time to find the slowest worker. Line 6 needs at most $O(nm)$ time to choose the device that can induce the largest reduction on the cost function. Specifically, evaluating the value of $\max\{f(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j),\ g(p_i,\sum_{v\in V_i}\sum_{j=1}^{m}r_j)\}$ costs linear time according to our analysis in the previous subsection. At most we need to evaluate the cost function value n times, once for each device in V. Lines 7 and 8 costs constant time if we use hash set to store V and V_i for $i = 1, 2, \ldots, q$. The while loop at most has $O(n)$ iterations. Therefore, the time complexity of the worker grouping algorithm is $O(q + n^2m) = O(n^2m)$. ■

Algorithm 2 Grouping Heterogeneous Devices

```
Input: Heterogeneous device set $V$, depth of the pipeline $q$
Output: Workers that group heterogeneous devices $V_i, i = 1, 2, \ldots, q$
$V_i \leftarrow \emptyset$ for all $i = 1, 2, \ldots, q$
for $i = 1, 2, \ldots, q$ do
    initialize the cost function of each worker, $cost(V_i) \leftarrow \max\{f(p_i, \sum_{v \in V_i} \sum_{j=1}^{m} r_j), g(p_i, \sum_{v \in V_i} \sum_{j=1}^{m} r_j)\}$
while $V$ is not empty do
    choose the worker $V_i$ with the largest cost
    $v^* \leftarrow \arg\max_{v \in V} cost(V_i) - cost(V_i \cup v^*)$
    assign $v^*$ to $V_i$.
    remove $v^*$ from $V$
return $V_i, i = 1, 2, \ldots, q$ as workers
```

VI. Evaluation

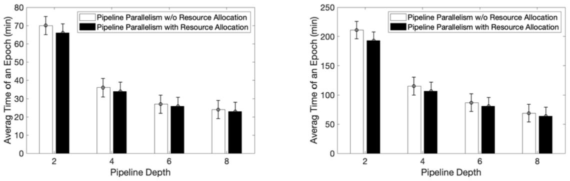
A. Simulation Setup

We use trace-based simulation to evaluate the performance of our resource allocation and model partition algorithms. We use the `PyTorch` library to implement the DNN training. We profile the computation time of forward and backward propagation on mobile devices. Specifically, we use a laptop that has an Intel i7 CPU, a GTX 1650 GPU, and 32GB RAM as a relatively powerful device. We treat the CPU and GPU in the laptop as different computation resources and separately profile the forward and backward propagation time on each resource. Other types of mobile devices include a laptop with Intel i5 CPU, a Intel Iris Xe GPU and a Raspberry Pi 4B with a Cortex-A72 CPU. The forward propagation output and time consumption of each layer can be easily measured using the `torch.nn` package. The forward and backward propagation time of different layers on different resources is recorded in a lookup table. We use the table and our linear model to simulate the time consumption of forward and backward propagation when only a part of resource is used.

We also measure the actual communication volume among devices during training. The distributed computing package of `PyTorch` does not provide APIs to measure the communication time. We build the communication interface between devices using the `gRPC` package. Each `gRPC` message contains a field to store forward or backward tensors and a field to store timestamps. Usually, machine learning engines send encoded tensors instead of plain tensors in communication. To measure the accurate communication volume, we use `tensor.save()` and `tensor.load()` APIs to encode and decode tensors for communication. A virtual interface `BytesIO` is used to flushed the encoded tensors in memory.

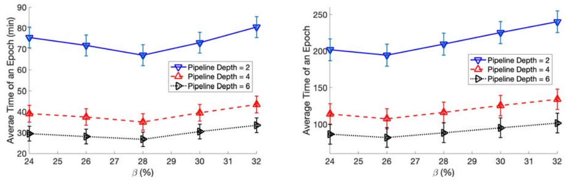
B. Simulation Results

We illustrate the result under different pipeline depths. In this set of simulations, the laptop with an Intel i7 CPU and GTX 1650 GPU is used as a worker. The number of workers equals to the pipeline depth. The DNN model is partitioned based on the pipeline depth. We compare the average time consumption of a training epoch. Without resource allocation, the default computation power allocated on forward propagation is 50%. The simulation results are shown in Fig. 5.



(a) AlexNet (b) GoogLetNet

Fig. 5. Evaluating the impact of pipeline depth.



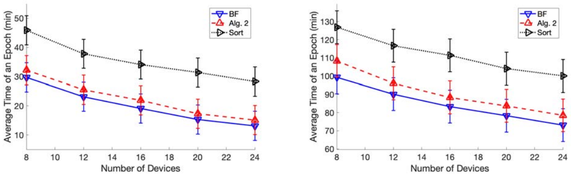
(a) AlexNet (b) GoogLeNet

Fig. 6. Comparison over resource allocation ratios.

Fig. 5(a) illustrates the training time for AlexNet [21]. We can observe the speedup of training time when adding more workers. The speedup is no longer obvious when there are 8 workers. This is because the AlexNet is relatively small. Generating too many partitions is not necessary. Additionally, we find that considering the resource allocation in DNN partition helps reduce the training time. When pipeline depth is 4, the improvement is about 5.7% compared to traditional pipeline parallelism. Fig. 5(b) illustrates the simulation results on GoogLeNet [22]. We find a more obvious speed up when the number of workers increases in this case. Compared to AlexNet, considering the resource allocation can bring more benefits for GoogLeNet. When pipeline depth is 4, the improvement is about 7.3%. Simulation results show that our resource allocation scheme helps to reduce the training time.

Then, we investigate the impact of different resource allocation ratios on homogeneous workers. In this set of simulations, we adjust the ratio of computation resources that are allocated for forward propagation. Formally, we change the value of β_j for each type of resource. In the simulation, the mobile device has two types of computation resources. According to Theorem 1, $\beta_1 = \beta_2$ in the optimal allocation. Hence, we changes the value of β_1 and β_2 in the experiment, but keep $\beta_1 = \beta_2$. The simulation results are shown in Fig. 6. From the figure, we find that allocating too much or too little computation power on forward propagation increases the time consumption of each training epoch. Backward propagation usually needs more computation power. Allocating more power on forward propagation has a more significant impact on the batch training time compared to allocating less power.

Comparing Fig. 6(a) and Fig. 6(b), we find that training GoogLeNet requires a greater proportion of computation resources compared to training AlexNet. Additionally, from the figure, we find that the pipeline depth does not have a significant impact on the resource allocation ratio. For a specific DNN model, it is usually built with some repeated blocks. For example, GoogLeNet repeats inception models



(a) AlexNet (b) GoogLeNet

Fig. 7. Evaluating the impact of heterogeneous workers.

multiple times. Despite different partitions, the ratio between the time consumption of forward and backward propagation is similar. The simulation results show that it is necessary to optimize the resource allocation, which helps reduce the makespan of DNN training in pipeline parallelism.

Additionally, we evaluate the performance of our scheme with heterogeneous devices. We generate heterogeneous device sets with different sizes. To build each set, we randomly pick from three types of devices. Then, we group heterogeneous devices together. Each group of devices is treated as a worker for training. We assume that the resources on different devices in the same worker group can be shared without additional overhead since we aim to test if our worker grouping algorithm can evenly partition heterogeneous devices. The pipeline depth equals the number of groups in the simulation. We compare our device grouping algorithm with a brute force (BF) method and a sorting-based method. BF explores all possible combinations of devices and finds the optimal worker cluster. However, its time complexity is exponential. In the sorting approach, devices are sorted based on their overall computation power in ascending order. Start from the slowest device, we continuously add devices to a group until the group's total computation power exceeds the average computation power of all devices.

For heterogeneous settings, we investigate the performance of our algorithm on different DNN models. The pipeline depth is 4 in this set of simulations. Fig. 7 shows the average training time of each epoch of AlexNet and GoogLeNet. From the figure, we find that the performance of our device grouping algorithm is close to the optimal one. Notably, our algorithm does not explore all possible combinations and has polynomial time complexity. The sorting-based algorithm does not perform well especially when the number of workers is large. The performance gap between the sorting-based approach and our algorithm is more obvious for GoogLeNet. This is because that the computation workload of GoogLeNet is larger compared to AlexNet. Evenly grouping mobile devices can balance the training pipeline among workers.

VII. Conclusion

We discuss the resource allocation and model partition problem in pipeline parallelism. Different from existing pipeline parallelism, we consider the resource allocation problem in DNN model partition. For homogeneous workers, we theoretically analyze the resource allocation for the ideal case where the resources are linearly separable. We show a partition algorithm based on the resource allocation scheme to split DNN models among workers. Moreover, we investigate the case of heterogeneous mobile devices. We present a local-search-based algorithm to group heterogeneous devices into workers. We try to balance the computation power among workers when grouping devices. Our simulation results show that considering resource allocation for model partitioning can improve resource utilization. Grouping heterogeneous workers helps to reduce the training time.

References

[1] R. Gu, S. Yang, and F. Wu, "Distributed machine learning on mobile devices: A survey," *arXiv preprint arXiv:1909.08329*, 2019.

[2] J. Konečnỳ, B. McMahan, and D. Ramage, "Federated optimization: Distributed optimization beyond the datacenter," *arXiv preprint arXiv:1511.03575*, 2015.

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.

[4] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed DNN training," in *IEEE INFOCOM*, 2020.

[5] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *ACM SOSP*, 2019, pp. 1–15.

[6] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[7] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *ACM PPoPP*, 2021, pp. 431–445.

[8] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, G. Huang, and X. Liu, "The case for fpga-based edge computing," *IEEE TMC*, 2020.

[9] Z. M. Fadlullah and N. Kato, "Hcp: Heterogeneous computing platform for federated learning based collaborative content caching towards 6g networks," *IEEE Transactions on Emerging Topics in Computing*, 2020.

[10] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *ATC 20*, 2020, pp. 307–321.

[11] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[12] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *arXiv preprint arXiv:1106.5730*, 2011.

[13] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *NIPS*, 2013.

[14] Y. Duan, N. Wang, and J. Wu, "Minimizing training time of distributed machine learning by reducing data communication," *IEEE Transactions on Network Science and Engineering*, pp. 1–1, 2021.

[15] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[16] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[17] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *ICML'17*, pp. 2430–2439.

[18] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *ASPLOS*, 2020, pp. 907–922.

[19] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, 2017.

[20] K. Wei, R. K. Iyer, S. Wang, W. Bai, and J. A. Bilmes, "Mixed robust/average submodular partitioning: Fast algorithms, guarantees, and applications." in *NIPS*, 2015, pp. 2233–2241.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE CVPR*, 2015, pp. 1–9.