SmartPipe: Intelligently Freezing Layers in Pipeline Parallelism for Distributed DNN Training

Nadia Niknami, Abdalaziz Sawwan, and Jie Wu
Center for Networked Computing, Temple University, USA

***Abstract*—Deep Neural Network (DNN) models have been widely utilized in various applications. However, the growing complexity of DNNs has led to increased challenges and prolonged training durations. Despite the availability of high-performance computing systems, certain DNNs still require several days for successful training. This study aims to address this issue by proposing a method for significantly reducing the training time of deep learning models while maintaining test accuracy. Existing approaches primarily concentrate on optimizing training efficiency through computational and communication overlap/scheduling. In contrast, this research takes a step further by inspiring transfer learning. Transfer learning is a useful way to quickly retrain a model on new data without having to retrain the entire network. During transfer learning, the first layers of the network are frozen while leaving the end layers open to modification. By doing so, computation and communication requirements in these frozen layers are eliminated. This intelligent approach involves freezing some of the specific DNN layers and allocating resources to the remaining active layers during the training process, thereby minimizing DNN training time. To achieve this objective, we propose an intelligently freezing DNN using pipeline parallelism. Through trace-based simulation results, our scheme has demonstrated its effectiveness in efficiently reducing the time cost of a training iteration.**

***Index Terms*—DNN training, gradient descent, intelligent freezing, machine learning, neural networks, pipeline.**

I. Introduction

Recent advancements in deep learning (DL) have greatly benefited from training larger deep neural networks (DNNs) on extensive datasets. Larger DNNs with increased model sizes have shown improved inference accuracy and enhanced generalization capabilities. However, deep learning models require larger training times as the depth of a model increases and suffers from vanishing gradients. As model sizes continue to grow and datasets become larger, the computational cost associated with large-scale training has become a significant concern. Recent research in DL has focused on enhancing DNN training techniques. These efforts include leveraging parallelism and pipelining, and employing sophisticated methods such as computation-communication overlap or scheduling, in order to build more efficient systems and reduce training time [1]. However, while achieving linear scalability can decrease the time required to train a model, the overall computation requirements remain unchanged. Numerous research studies have explored different approaches include adjusting learning rates, parallelizing training tasks, reducing model sizes, optimizing convolution computations, employing batch normalization, and fine-tuning to reduce training time for DL models.

This research was supported in part by NSF grants CNS 2214940, CPS 2128378, CNS 2107014, CNS 2150152, CNS 1824440, and CNS 1828363.

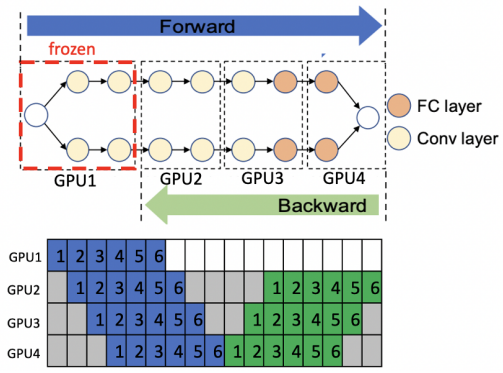


Fig. 1: Freezing layers in pipeline parallelism.

Data parallelism stands as a widely employed technique, involving the division of training data, which is then allocated across distinct training devices [2]. Model parallelism divides the parameters of a DNN across layers, with each device responsible for a specific parameter subset. Model parallelism tends to exhibit relatively lower resource utilization efficiency. *Pipeline model parallelism* (Pipeline) techniques have been proposed to enhance the resource utilization of model parallelism through pipeline-like scheduling. One significant advantage of pipeline is its ability to reduce GPU idle time during model parallelism training. Its benefits include reducing overall training time, minimizing GPU idle time while waiting for predecessor or successor GPU outputs, and adaptability to various types of DNN models [3].

Transfer learning involves leveraging feature representations from a pre-trained model, eliminating the need to train a new model from scratch. It is called fine-tuning referring to using the weights of an already-trained network as the starting values for training a new network. By incorporating pre-trained models into a new model, the training time is reduced, and generalization error is lowered. Nevertheless, fine-tuning for a few epochs can still be quite time-consuming, even when GPUs are employed. This is primarily attributable to the fact that pre-trained models are generally computationally intensive [4]. A natural approach to enhance the efficiency of fine-tuning involves constraining the extent to which the model's layers are updated, effectively aligning it with the principles

of transfer learning. In transfer learning, one approach is freezing layers. In DNN, it is a frequent observation that the initial layers converge swiftly, while the deeper ones demand significantly more training time. By strategically freezing these early-converging DNN layers, we can effectively reduce their associated computational and communication overheads without sacrificing model accuracy [5].

An instance of layer freezing in a pipeline is exemplified in Fig. 1. The blue and green colors in the diagram represent forward and backward propagation operations, respectively. Each block contains a number denoting the input data ID. A row of these blocks represents the operations conducted by a specific worker during a given time slot. Each GPU initially carries out forward propagation for input data 1 to 6, followed by running backward propagation for input data 1, contingent on receiving the output from the next GPU. In this example, we operate under the assumption that the layers allocated to $GPU1$ have been frozen. Consequently, no backward propagation occurs from $GPU2$ to $GPU1$ in this scenario. While freezing layers can help decrease training costs, careful consideration must be given to ensure that the freezing process is applied at an appropriate stage in the training process to maintain optimal accuracy.

We propose the expansion of a pipeline approach to enhance the efficiency of DNN training on mobile devices. Our emphasis on the pipeline methodology stems from its capacity to diminish training duration by concealing the communication phase behind the computation phase. The process of freezing a layer involves preserving its weights without making any modifications. By avoiding the backward pass to a frozen layer, computational speedups can be achieved. While freezing layers can be beneficial in reducing training costs, prematurely freezing under-trained layers can have a negative impact on the final accuracy. The key challenge in extending layer freezing to general DNN training lies in maintaining accuracy by selectively freezing only the converged layers. This objective is accomplished by employing a mathematical formula to identify the layers that require less training and are suitable for freezing while ensuring that the overall accuracy of the model is preserved.

To summarize, the key contributions of this paper include:

- We introduce an efficient system to implement the idea of freezing layers during training. In order to reduce the computation time overhead, we selectively and periodically calculate the gradient information during the training process.
- We utilize semantic knowledge to optimize DNN training by freezing layers, thereby reducing both backward computation and communication while upholding accuracy.
- We introduce the concept of conservative and aggressive freezing rate to determine the number of frozen layers for DNNs.
- Our scheme's effectiveness in efficiently reducing training time and enhancing resource utilization is evident from our trace-based simulations conducted on various widely adopted DNN models.

II. Background and Motivation

A. DNN Training

Modern DNNs comprise numerous layers that execute mathematical operations. Each layer takes an input tensor of features and produces corresponding activations. The DNN training procedure entails iterating through a large dataset multiple times while minimizing a loss function. The dataset is divided into smaller mini-batches, and a complete pass through the entire dataset is referred to as an *epoch*. The initialization phase involves setting parameters to random weights and initializing biases to zero. This is succeeded by a forward pass of the data through the network to generate the model's output. Finally, the backward propagation process is conducted. The model training process typically entails several iterations. Within each training iteration, three main steps are performed: (1) forward pass (FP), (2) backward pass (BP), and (3) parameter synchronization.

$$T_{training} = \sum\nolimits_{i \in D} t_{fp}^{(i)} + \sum\nolimits_{i \in D} t_{bp}^{(i)} + t_{com}, \qquad (1)$$

where $t_{fp}^{(i)}$ and $t_{bp}^{(i)}$ denote the time cost for forward propagation and the time cost for backward propagation for i-th batch of data D, respectively. t_{com} is the completion time of parameter updating. Both the forward and backward passes require computational resources, typically executed on GPUs. During the forward pass, a mini-batch of data is processed layer-by-layer within the model, calculating the loss with respect to the target labels and the defined loss function. The backward pass of a DNN layer involves two steps: computing the error or gradient of the current layer based on the error or gradient propagated from the subsequent layer, and updating the model parameters of the current layer accordingly. Gradient g_l can start its forward propagation only when the previous gradient finishes the forward propagation and g_l completes its parameter update process.

The backward pass calculates the parameter gradients from the last layer to the first layer using the chain rule of derivatives with respect to the loss. In contrast, the forward pass of a DNN layer only computes the output values for the subsequent layer, which typically has a similar computational

TABLE I: Main notations

Symbol	Meaning
$L = \{l_1, ..., l_k\}$	Layer of DNN
l_s/ l_t	First/ Last layer of partition
$p_i = (l_s, l_t)$	Partition of DNN layers assigned to v_i
g_l	Gradient of layer l
η	Gradient norm change
λ_i	Freezing rate in i^{th} iteration
t_i	i^{th} iteration
s	Pipeline depth (number of stages/GPUs)
m	Mini-batch size
t_{fp}/t_{bp}	Cost of forward/backward propagation
t_{com}	Completion time of parameter updating

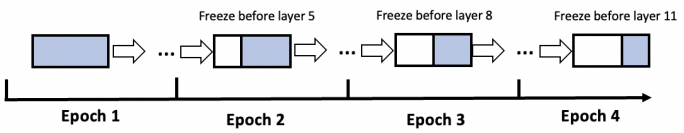


Fig. 2: Freezing layers during the training process.

complexity to calculating the error or gradient. Therefore, the time required for the backward pass is generally considered to be approximately twice that of the forward pass.

The weights of the model are updated layer-by-layer in a backward manner using an error estimate, such as the gradient, which guides how each weight should be updated within a layer. This error estimate assumes that the input to a layer remains unchanged from the previous layer. However, this assumption can hinder the learning process, necessitating the use of smaller learning rates and potentially requiring more epochs to achieve convergence. To address this issue, batch normalization is employed. Batch normalization normalizes the inputs of a layer for each batch, ensuring that the input distribution does not vary significantly. By standardizing the inputs, batch normalization stabilizes the learning process, reducing the time and number of epochs required for convergence. It helps mitigate the negative impact of changing input distributions and allows for more efficient and stable training of deep models.

B. Freezing Layers

Recent efforts have shown that it is not necessary to have every unit in a network participate in the training process at every training step. The front layers primarily extract general features of the raw data and often become well-trained much earlier, while deeper layers are more task-specific and capture complicated features outputted by front layers. Not all layers need to undergo training for the entire training duration. We can reduce computation and prevent overfitting by consecutively freezing layers (Fig. 2). For instance, consider a scenario where only the most recent k layers of the model are subject to updates. This approach is called static freezing [6]. In this approach, we train the last portions of the layers, specifically targeting the final $25\%, 50\%, 75\%$, or only the very last layer, for complete fine-tuning. This approach implies that avoiding gradient computation for specific layers (freezing) can notably shorten the training duration.

According to findings in [4], static freezing methods result in a reduction in accuracy ranging from 0.2% to 1.7%. However, employing too low a percentile value may result in an overly conservative approach, causing fewer layers to be frozen than desired. Conversely, using too high a percentile value can lead to an overly aggressive approach. Therefore, while techniques such as static freezing and cosine annealing can diminish backward computation costs, it is important to be aware of the common side effects of accuracy loss. There is no specific way to decide how many layers should be frozen confidently. However, depending on how much target data we have and how similar they are to the source data, we can approximately decide how many layers should be fixed and how many layers should be fine-tuned on the target data.

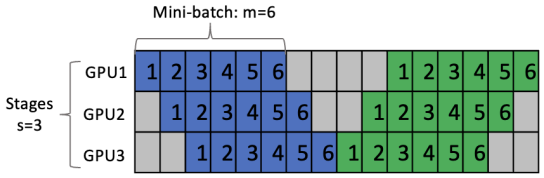


Fig. 3: Pipeline with three GPUs.

C. Pipeline Model Parallelism (Pipeline)

In Pipeline, the model is partitioned into multiple *stages* each including a consecutive set of neural network layers. These stages are allocated to specific GPUs or workers. To maximize the system throughput, each training batch is further split into mini-batches, enabling a pipeline-like execution of computations. This pipeline operation necessitates point-to-point communication for the exchange of *activations* and their corresponding *gradients* between adjacent stages.

An illustrative example is presented in Fig. 3, where we can observe the forward and backward propagation operations designated by blue and green blocks, respectively. Each block's numerical value corresponds to the input data ID. This scenario involves three stages ($s = 3$), with a mini-batch size of $m = 6$. Each row of blocks represents the operations executed by a specific GPU during each iteration. In the pipeline strategy, each GPU undertakes computations on successive micro-batches and transfers the outcomes to the subsequent GPU. $GPU1$ initially conducts forward propagation for input data 1 to 6. $GPU1$ proceeds to perform backward propagation for input data 1 after receiving the output from $GPU2$. To elucidate the steps of a DNN training iteration, we utilize the labeled blocks in Fig. 3.

The GPUs are interconnected by data dependencies, and computations on each GPU follow the input data order. To ensure optimal training throughput and maintain a continuous pipeline during the entire training process, GPUs must coordinate effectively. Delays or disruptions in data flow can result in GPU idleness ("bubbles" are illustrated by gray areas) and a decrease in training efficiency. The presence of idle *bubbles* within the training pipeline, as depicted in Fig. 3, enlarge the training makespan. There are three mainstream techniques of pipeline to reduce bubbles: (1) Gpipe [7], (2) One forward one backward (1F1B) [8] and (3) Overlapping cost [9]. Gpipe [7] introduced the concept of micro-batches, wherein the traditional mini-batch is subdivided into smaller, equal-sized micro-batches. These micro-batches are introduced into the pipeline simultaneously to minimize idle periods within the distributed system. In 1F1B scheduling scheme, as introduced in Pipedream [8], workers/GPUs execute backward propagation immediately following a forward propagation.

III. Proposed Method

During the typical training of DNN models, all layers are involved in the training process simultaneously. However, it

may not be necessary for all layers to be actively trained at all times, as different layers exhibit varying learning speeds. Layer freezing techniques are used to freeze the weights of specific layers of an already trained DNN in order for them to remain unchanged during training. In our method, the first training iteration starts without freezing, where all layers are updated. In the following iterations, this method then progressively starts freezing from the early layers down to the latest layers in the model in an orderly fashion.

By avoiding the backward pass to a layer whose weights we wish to leave unaltered, we can achieve a substantial acceleration in processing speed. For instance, if we freeze half of the model and proceed to train it, the time required will be approximately half that of a fully trainable model. However, it is crucial to strike a balance as we still need to train the model adequately. Freezing the model prematurely can lead to inaccurate predictions, emphasizing the importance of determining the optimal point at which to freeze the layers.

For stability, we adopt an initial training phase where layers are allowed to train without freezing for a specific number of epochs before the freezing process commences. Nevertheless, two crucial questions require attention: 1) The determination of the optimal number of layers to freeze initially, and 2) The appropriate frequency at which these layers should be frozen. To address these questions, a gradual identification and freezing process during training is implemented, accompanied by the allocation of resources to train the remaining active layers. Notably, excluding consecutive bottom layers from the pipeline proves beneficial as it effectively reduces computation, memory, and communication overhead. By conducting a meticulous analysis of the freezing rates and applying suitable freezing strategies, we can achieve optimization of the training process, leading to improved efficiency of DNN models.

A. Model

Consider a distributed training setup with multiple workers or GPUs, each responsible for updating a subset of the DNN parameters. $L = \{l_1, l_2, ..., l_n\}$ are layers in the DNN. The total number of layers in the training model is denoted by n. Let $\phi(w_1, w_2, ..., w_k)$ denote the loss function, where w_i refers to parameters in layer l_i. We suppose that there are k parameters for each layer. After the backward propagation of iteration t, the model parameters for the following iteration $t+1$ are updated using the following equation:

$$W_x(t+1) = W_x(t) - \alpha \frac{\partial}{\partial W_x} \phi(w_1, w_2, \ldots, w_k), \quad (2)$$

where $W_l(t)$ is the weight matrix of layer l at timestamp t, where $l \in L$ and α is the learning rate. Let $V = \{v_1, v_2, ..., v_n\}$ denote a set of GPUs. We assume each partition is handled by a single GPU device. Mathematically, the architecture of a neural network with three layers can be described as follows:

$$y_1 = xW_1 + b_1, \qquad y_2 = y_1W_2 + b_2, \qquad y_3 = y_2W_3 + b_3,$$

where W_1, W_2, W_3, b_1, b_2 and b_3 are the weights and biases of the network. In this example, x, y_1, and y_2 are inputs to the first, second, and third layers, respectively. We derive the backpropagation equations with the loss $\mathcal{L}$ using the chain rule as follows:

$$\frac{\partial \mathcal{L}}{\partial W_3} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial W_3} = \frac{\partial \mathcal{L}}{\partial y_3} y_2,$$

$$\frac{\partial \mathcal{L}}{\partial b_3} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial b_3} = \frac{\partial \mathcal{L}}{\partial y_3} \times 1 = \frac{\partial \mathcal{L}}{\partial y_3},$$

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T y_1,$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T \times 1 = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T,$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T W_2^T x,$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial L}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T W_2^T \times 1 = \frac{\partial \mathcal{L}}{\partial y_3} W_3^T W_2^T,$$

With the equations provided earlier, updating the network weights (namely W_1, W_2, and W_3) necessitates retaining the values of x, y_1, and y_2 throughout the forward calculations, as these values are essential for subsequent backward computations.

Importantly, the progression of forward and backward operations between workers is organized in a pipeline manner. multiple workers can concurrently execute forward and backward propagation on distinct mini-batches of input data. Let d_i signify the mini-batch with label i. Moreover, we make the assumption that the time taken for both forward and backward propagation on all workers remains constant. Assume that the network architecture is already pre-trained on a large dataset and that the lower layers capture general features, while the higher layers capture more task-specific features. With the freezing algorithm, we freeze the lower layers, allowing the higher layers to adapt to the new task during training. Since the lower layers are frozen, their gradients are not updated, reducing the overall computational load during training. Therefore, during training, the gradient for frozen layers is $g_i = 0$.

B. Learning the Importance of Layers

The proof of convergence for optimization algorithms like SGD [10] demonstrates that the expected gap between parameters and the optimal solution diminishes across training iterations. Consequently, the difference in weights (or the magnitude of weight updates) between consecutive iterations also lessens. As a result, there is a possibility that certain layers might experience minimal parameter adjustments as training nears completion. Detecting and subsequently freezing these layers when their updates are minimal will not compromise accuracy. In the context of transfer learning, pre-trained models show slight adjustments during fine-tuning compared to the initial pre-training phase. Thus, identifying and freezing layers with negligible updates (weight distance

values) should have a limited impact on the fine-tuning process, as well as the ultimate accuracy achieved.

Definition 1. (Freezing Decision) For a layer l, whose weights at timestamp j can be denoted as W_l^j: Given a sequence of its weight history $(W_l^j)_{j=0}^t$ at timestamp t, yield a positive decision to freeze the layer at current epoch if the layer is ready to be frozen, and yield a negative decision if the layer needs further training.

We can evaluate how much a layer adds to the overall training progress. If a layer does not make a big difference, we can freeze it to speed up training. The change in gradient values over time in a layer helps us understand how fast the model's weights are getting updated for that specific layer. We consider the normalized difference of gradients as an important measurement. When weight adjustments happen in both positive and negative directions, and these updates eventually offset each other, it results in time being expended on redundant updates. As a result, these weights can be frozen since the opposing gradients offset each other. The freezing rate of a layer defines the degree to which we opt to freeze it. We collect gradients for each layer in the model and carry out our evaluations at regular time intervals marked by t.

Definition 2. (Gradient Norm Difference) We define the alteration in gradient norm for layer l at time t as η:

$$\eta_l = \frac{\|g_l^{t-1} - g_l^t\|}{\|g_l^{t-1}\|}. \tag{3}$$

Definition 3. (Frozen Layer Sequence) A layer can be marked for freezing if all preceding layers are frozen, and it holds the layer that experiences the slow rate of change. Therefore, to implement the freezing algorithm for layer l_i, two conditions should be considered:

1) Gradient norm difference of l_i should be smaller than given *Threshold* ($\eta_{l_i} < \tau$)
2) For $k = 0$ to $k = i - 1$: gradient of l_k should be zero ($g_{l_k} = 0$).

The core idea involves employing a formula to compute freezing rates, which offers insight into the number of initial layers to freeze. We introduce a test based on the gradient norm that assesses layers according to their rate of change. Subsequently, we choose the least changing layers for freezing. Our aim is to freeze layers as they approach optimality. This signifies that the gradients pertaining to these layers are comparably small. We can find the max prefix of layers for freezing by comparing the gradient difference with the threshold. It is noted that our approach includes a freezing of successively more layers from layer 1.

Definition 4. (Freezing Rate) We propose an adaptive freeze algorithm to define $L_{frozen}^{(t)}$ as the set of frozen layers at time step t where $t \geq 1$ as follows:

$$|L_{frozen}^{(t)}| = |L_{frozen}^{(t-1)}| + \lambda_t |(L - L_{frozen}^{(t-1)})|, \tag{4}$$

where λ is a hyperparameter termed the freezing rate, constrained within the interval $\lambda \in (0, 1)$. This effectively freezes λ fraction of the remaining active layers. Increasing λ results in an aggressive layer freezing approach. The λ parameter controls the trade-off between accuracy and training speed.

However, having small gradients does not necessarily mean we are getting close to the best solution. The shape of the function we are attempting to minimize can be quite complex, featuring saddle points and flat regions with high loss. These features can also result in small gradient values.

Lemma 1. If the rate is too aggressive (freezing too many layers too quickly), it leads to suboptimal accuracy. If the rate is too conservative (freezing too few layers), the model may not be able to fully leverage the benefits of freezing and may require more epochs to converge to the desired accuracy.

Definition 5. (Conservative and aggressive freezing Rate) In the case of *Conservative Freezing*, where fewer layers are frozen in each epoch (λ has a small value), it requires more epochs and longer training times to achieve similar accuracy. In the case of *Aggressive Freezing*, where a larger number of layers are frozen in each epoch (λ has a large value), it leads to shorter training times since more layers are frozen, and the model's parameters require fewer updates. However, there is a risk of prematurely freezing, leading to suboptimal accuracy. The more aggressive our freezing policy is, the greater the accuracy loss will be.

The procedure outlined in Algorithm 1 elucidates the details of training deep models through SGD coupled with intelligent freezing. To streamline the process, a learning rate denoted as α and threshold τ are employed within Algorithm 1. In scenarios where SGD with step decay is adopted, the steps delineated in Algorithm 1 are straightforwardly repeated for each distinct learning rate utilized.

Algorithm 1 Intelligent freezing on pipeline parallelism

Require: Network parameters: ω,
Learning rate: α,
Number of training epochs without freezing: e_1,
Number of training epochs: E,
The threshold for freezing rate: τ.

for $epoch = 0$ to e_1 **do**
  train(); test();
end for
Optimizer = *Get-optimizer*(ω, α)
for $epoch = e_1 + 1$ to E **do**
  $L_{candidate}$ = *Get-Layers-to-Freeze*(ω, τ)
  $L_{frozen} = L_{frozen} \cup L_{candidate}$
  Optimizer = *Get-optimizer*$(\omega, L_{candidate})$
  train(); test();
end for

A higher learning rate can expedite training, but it risks overshooting the optimal solution or hindering convergence.

Conversely, a lower learning rate fosters stability but extends training duration. Typically, higher learning rates are applied in initial epochs for acceleration. As training progresses and model weights approach optimality, the learning rate diminishes to encourage convergence. Lowering the learning rate generally results in lower freezing rates for layers. It is important to note that freezing rates cannot directly determine which layers to freeze. When an unfrozen layer feeds into a frozen one, gradients for the frozen layer are still computed to facilitate gradients for preceding unfrozen layers. Consequently, to enhance efficiency, it is advisable to commence freezing from the first layer and proceed sequentially. Non-sequential frozen layers would lead to gradients being computed for layers preceding the unfrozen ones. When a layer's learning rate reaches zero, it transitions to inference mode and is excluded from subsequent backward passes, resulting in an immediate per-iteration speedup proportional to the layer's computational cost.

C. Reduction of Bubbles and Training Time

The training time corresponds to the period taken for the completion of the training process. We can quantify the extent of the pipeline bubble in GPipe, denoted as (t_{pb}). We denote the number of mini-batches in a batch as m, the number of pipeline stages (the number of GPUs) as s, the ideal time per iteration assuming perfect scaling as t_{id}, and the time taken for executing both the forward and backward passes of a single mini-batch as t_{fp} and t_{bp}, respectively. The pipeline bubble encompasses $s-1$ forward passes at the start of batch and $s-1$ backward passes towards the end. The cumulative time spent within the pipeline bubble can be expressed as $t_{pb} = (s-1)(t_{fp}+t_{bp})$. Correspondingly, the ideal processing time for the batch is $t_{id} = m(t_{fp} + t_{bp})$. Therefore, the fraction of ideal computation time spent in the pipeline bubble is $t_{pb}/t_{id} = (s-1)/m$.

Lemma 2. If training data has D batches and the size of mini-batch is denoted by m, in each epoch there are $I = D/m$ iterations. Each GPU needs to wait $(s-1)$ and perform training on m data in each iteration. By considering both forward and backward propagation for each item of data, GPUs spend time for $2(m+s-1)$ in each iteration. The finishing training time and number of bubbles depend on batch size m and number of stages s in each iteration :

$$\text{Training time} = 2(m+s-1) \tag{5}$$

The fraction of time wasted on bubbles depends on the pipeline depth or stage s and mini-batch size m:

$$\text{Bubble} = 1 - \frac{2sm}{2s(m+s-1)} = 1 - \frac{m}{m+s-1} \tag{6}$$

Increasing the size of the mini-batches is necessary for making the bubble fraction small. For the bubble time fraction to be small, we thus need $m >> s$. Large mini-batch sizes require careful learning rate scaling and will increase the memory demand for caching the activations to be kept in memory for all m mini-batches through the lifetime of a training iteration.

Theorem 1. The freezing algorithm in distributed training on DNNs can accelerate convergence by reducing training time. Suppose the training data has D batches and the number of mini-batches in each batch is denoted by m, so that in each epoch, there are $I = D/m$ iterations. By freezing f layers, where $f = \lambda.|L|$ and Gpipe is used as the pipeline, the reduction time is:

$$\text{Reduction Time} = f(2I-1), \tag{7}$$

By freezing f layers, where $f = \lambda.|L|$ and 1F1B is used as pipeline, the reduction time is :

$$\text{Reduction Time} = fI, \tag{8}$$

Proof. In an epoch with f frozen layers, there would be f reduction for forwarding passes and f reduction for backwarding passes in each iteration except the first iteration. In the first iteration, because each layer needs to have its input to start forwarding passes, there is no reduction for forwarding passes. Therefore, the reduction is f for the first iteration and $2f$ for $(D/m) - 1 = I - 1$ iteration: Reduction Time $= f + 2f(I-1) = f(2I-1)$. If we have the 1F1B approach for pipeline and interactions between layers, the GPU performs a backward propagation immediately after a forward propagation. That is why, in this approach, for each iteration there is a f reduction. □

Theorem 2. The freezing algorithm in distributed training on DNNs can accelerate convergence by reducing bubbles. By freezing f layers, where $f = \lambda.|L|$ and Gpipe is used as the pipeline,

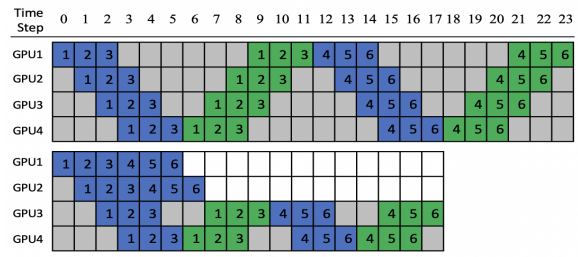
$$\text{Bubbles} = (s-f)(I+1)(s-2) \tag{9}$$

By freezing f layers, where $f = \lambda.|L|$ and 1F1B is used as the pipeline,

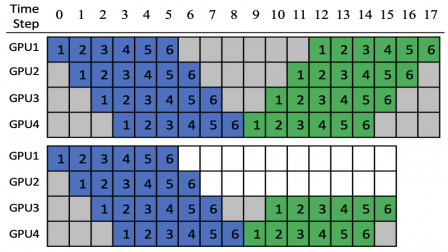
$$\text{Bubbles} = (s-f)I \tag{10}$$

Proof. In the Gpipe pipeline, each active GPU has $s-2$ bubbles for backward passes in each iteration except the first iteration. The first iteration has $(s-2)$ bubbles for forward passes as well. So, by having $s-f$ active GPUs, there are $s(s-2)(I) + s(s-2)$ bubbles in total.

In the 1F1B pipeline, the bubble consists of $s-1$ forward passes at the start of a mini-batch. The bubble that we have in Gpipe for backward passes has been filled by forwarding passes of new data. So, by having s GPUs, there are $s(s-1)$ bubbles in each iteration. By freezing f layers, each GPU, which does not have frozen layers, just has a 1 bubble which is for waiting for the first input from the previous layer. Therefore, by having $(s-f)$ GPUs which are not involved with frozen layers and I iterations, the number of bubbles is $(s-f)I$. Note that GPUs with frozen layers can efficiently process new data during idle periods without the need for backward passes, eliminating any bubble time. □

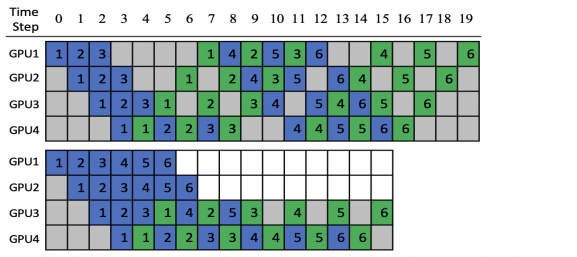


(a) No freeze: (Bubbles: 48, T:24). SmartPipe: (Bubbles: 13, T:18)

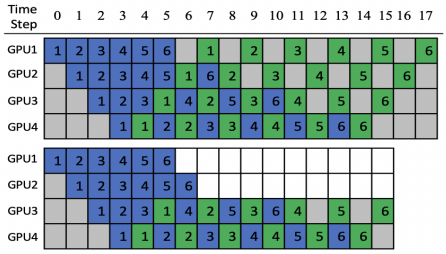


(b) No freeze: (Bubbles: 24, T:18). SmartPipe: (Bubbles: 9, T:16)

Fig. 4: Gpipe: No freezing vs. two frozen layers. Data size $D = 6$, stage $s = 4$, and (a) batch size $b = 3$, (b) batch size $b = 6$.



(a) No freeze: (Bubbles: 32, T:20). SmartPipe: (Bubbles: 10, T:16)



(b) No freeze: (Bubbles: 24, T:18). SmartPipe: (Bubbles: 9, T:16)

Fig. 5: 1F1B: No freezing vs. two frozen layers. Communications are omitted for simplicity where data size $D = 6$, stage $s = 4$ and (a) batch size $b = 3$, (b) batch size $b = 6$.

Consider a dataset of size $D = 6$ and a total of $s = 4$ stages, and present an example to demonstrate the effectiveness of the frozen layer in SmartPipe. In the accompanying diagram, the blue and green colors denote forward and backward propagation operations, respectively. The bubbles are depicted in gray areas. The unoccupied space signifies that the GPU is at liberty for any other tasks. Fig. 4 presents a comparative analysis between the *No Freezing* and *SmartPipe* during the Gpipe approach. In part (a), when the No Freezing method is employed with a batch size of $b = 3$, the following results are observed: The number of bubbles is 48, and the training time amounts to 24 units. Conversely, with the utilization of the SmartPipe technique and by freezing two layers, the number of bubbles is notably reduced to 13, and the training time is correspondingly diminished to 18 units. In part (b), the same scenario is repeated, but with a larger batch size of $b = 6$.

Fig. 4 presents a comparative analysis between the No Freezing and SmartPipe methodologies within the context of the 1F1B approach. Consistently, the results reveal that SmartPipe yields a reduction in both the number of computational bubbles and the overall training time in comparison to the No Freezing method. In both scenarios, utilizing different approaches and distinct batch sizes, the effectiveness of the SmartPipe method in enhancing and optimizing the training process is clearly evident.

D. Resource Allocation

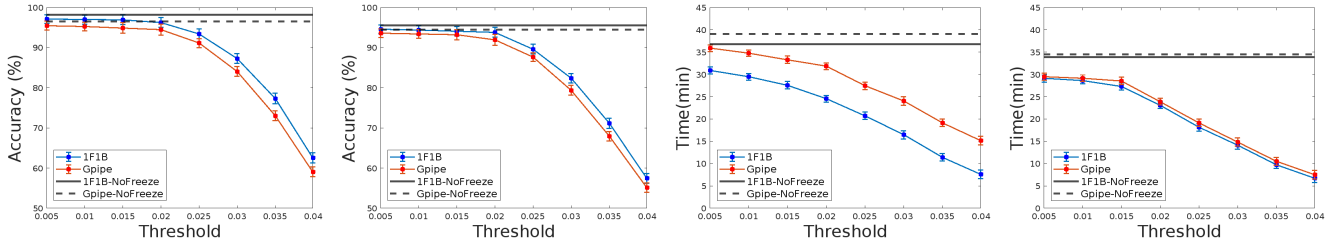
We can complement our approach with the method presented in [9], which incorporates resource allocation considerations. Our primary objective is to minimize the duration of bottleneck operations during DNN training while adhering to resource allocation constraints. In the case of considering resource allocation for pipeline on distributed training, the lengths of forward and backward propagation are mainly related to resource allocation and the DNN model partition. The allocation of resources guarantees that each worker can attain optimal speedup, while the model partition effectively determines the individual workload of every worker. Given the resource allocation, the forward propagation time for a specific DNN partition $p_i = (l_s, l_t)$ can be represented by $f(p_i, \sum_{j=1}^{j} \beta_j r_j)$. Similarly, the backward propagation time for p_i is denoted as $g(p_i - L_{frozen}, \sum_{j=1}^{j} (1-\beta_j) r_j)$. In our model, the communication time is designated as $h(p_i - L_{frozen}, b)$, which exclusively varies with the partition p_i. The problem can be formulated as minimizing the equation:

$$\max_{v_i \in V} \Big\{ f(p_i, \sum_{j=1}^{m} \beta_j r_j), g\big(p_i - L_{frozen}, \sum_{j=1}^{m} (1-\beta_j) r_j\big), h(p_i - L_{frozen}, b) \Big\}$$
$$\text{Subject to} \quad 0 \le \beta_j \le 1, \quad 1 \le j \le m, \quad L_{frozen} \le L - 1. \tag{11}$$

By combining these two approaches, we aim to achieve greater optimization in DNN training, effectively reducing bottlenecks and ensuring efficient utilization of available resources.

IV. Related Work

A significant amount of work has been dedicated to fragmenting the computation of NN models in pursuit of efficient large-scale distributed training [11]. In [12], the



(a) Accuracy for batch size: 1024 (b) Accuracy for batch size: 8192 (c) Training time for batch size: 1024 (d) Training time for batch size: 8192

Fig. 6: Comparing accuracy and training time for different methods under baseline and proposed approaches.

authors explored strategies for accelerating training processes. Several parallel techniques have emerged to expedite the training of distributed deep learning. Megatron [13] introduced model parallelism by dividing layers, facilitating the training of sizable transformer-based language models. Mesh-TensorFlow enables intra-layer model parallelism, with a user-friendly interface for specifying parallelization strategies. Another widely used parallel approach is pipeline parallelism, which involves segmenting extensive models into distinct stages [14], [15].

GPipe [7] employed micro-batches to partition batches, minimizing pipeline bubble size without altering the strict synchronous optimizer semantics. In a bid to further reduce pipeline bubbles, Chimera [16] introduced bidirectional pipelines that amalgamate two pipelines in opposing directions. Both pipelines encompass a complete DNN model and leverage micro-batch strategies. Chimera not only fills idle pipeline time but also reduces bubble count by up to 50% compared to GPipe. A framework based on elastic averaging has been explored in [17], which incorporates elastic averaging to introduce multiple parallel pipelines.

PipeDream [8] presented a solution called 1F1B, which introduces asynchronous updates to mitigate the issue of bubbles in micro-batch-based methodologies. Under the 1F1B scheduling within the pipeline, workers execute backward propagation immediately after forward propagation, utilizing distinct data samples for each computation. When pipeline balance is maintained, 1F1B ensures a bubble-free pipeline. Addressing the storage problem of PipeDream, PipeDream-2BW [18] adopts double-buffered gradients to alleviate storage overhead arising from numerous gradient copies. TeraPipe [19] put forth a fine-grained pipeline approach across tokens tailored for auto-regressive models. PipeTransformer [20] dynamically adjusts pipeline and data parallelism by freezing certain layers and allocating resources to active layers. Deepspeed [21] combined data parallelism, intra-layer model parallelism, and inter-layer pipeline methods to train exceedingly large models.

DAPPLE [22] is a synchronous training framework that combines data and pipeline parallelism to enhance computational efficiency while ensuring convergence. In [23], the concept of training plasticity is introduced to quantify the progress of internal DNN layers. The authors devise a knowledge-guided DNN training system leveraging semantic knowledge from a reference model to accurately assess the training plasticity of individual layers. This approach safely freezes converged layers, conserving computational resources and communication. Furthermore, in [24], the idea from [5] is harnessed to apply gradient amplification, expediting training and achieving enhanced performance at larger learning rates.

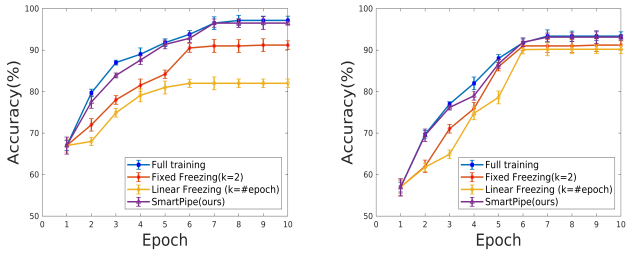
V. Evaluation

In this section, we assess the efficacy of the SmartPipe methodology through its application to standard benchmark datasets, namely CIFAR-100 [25] and ImageNet [26], for the purpose of image classification. Our experimentation incorporates three distinct models: RestNet50 [27], AlexNet [28], and GoogLeNet [29]. We use the `PyTorch` [30] library to implement the DNN training. Models are trained by using `PyTorch` on an $8 \times A100$ GPU server. We adopt the standard data augmentation and SGD optimizer. In PyTorch, to freeze a specific layer, we need to set `requires_grad = False/True` for each parameter that we want to freeze/update. The `requires_grad` flag is a boolean that allows for fine-grained exclusion of subgraphs from gradient computation. Setting the parameters' `requires_grad` flag to False would prevent calculating the gradients for these parameters in the backward step, which in turn prevents the optimizer from updating them.

We measure the execution time for forward and backward propagation on mobile devices. We conduct these tests using a laptop featuring an Intel i7 CPU, a GTX 1650 GPU, and 32GB RAM. This laptop is considered a relatively robust device. Additionally, we treat the laptop's CPU and GPU as separate computing resources, enabling us to separately evaluate the time taken for both forward and backward propagation on each resource [9]. To comprehensively evaluate our approach and make a comparative assessment against baseline methods, we employ three key measurements: accuracy, training time, and average time per epoch. These metrics will enable us to gauge the effectiveness and efficiency of our proposed approach in comparison to other established methods.

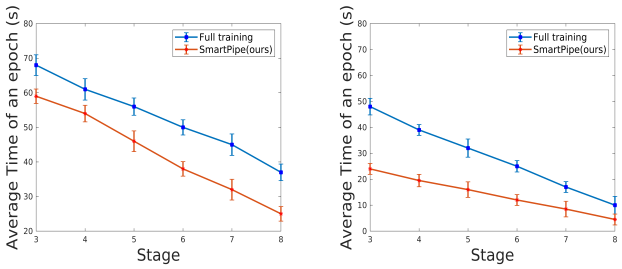
A. Accuracy and Training Time under Different Thresholds

The threshold serves as a critical hyperparameter in our proposed approach. It governs the rate at which layer freezing occurs during training. Achieving the right balance is crucial, as overly aggressive freezing, where too many layers are



(a) AlexNet (b) GoogLeNet

Fig. 7: Comparison accuracy on different numbers of epochs.



(a) AlexNet (b) GoogLeNet

Fig. 8: Average time on different numbers of stages.

frozen too quickly, can undermine accuracy. Conversely, a conservative approach, freezing too few layers, might hinder the model from fully capitalizing on the benefits of freezing and necessitate more epochs to attain the desired accuracy. As depicted in Fig 6, adopting a conservative freezing strategy (small λ) demands a higher number of epochs and longer training times to achieve comparable accuracy levels. On the other hand, an aggressive freezing strategy (large λ) leads to shorter training times due to the increased number of frozen layers, reducing parameter updates. However, it carries the risk of premature freezing, potentially leading to suboptimal accuracy. Certainly, the level of aggressiveness in our freezing approach directly impacts the extent of accuracy reduction. It is crucial to achieve an optimal balance that enables efficient training while maintaining an acceptable level of accuracy.

B. Comparison of Accuracy for Different Numbers of Epochs

As shown in Fig 7, the SmartPipe approach exhibits performance closely comparable to the full training approach in terms of accuracy. Particularly, when applied to the GoogLeNet model, SmartPipe achieves the same level of accuracy and effectively completes the training process. Both the fixed freezing and linear freezing approaches may complete training in fewer epochs, but their lower accuracy compared to our proposed approach renders them unacceptable. Despite the shorter training times, sacrificing accuracy can have significant implications for the overall performance and reliability of the deep learning models. Therefore, our proposed approach, which achieves comparable accuracy while efficiently reducing training time, stands as a more viable and effective solution for DNN training.

C. Average Time of an Epoch for Different Numbers of Stages

Increasing speedup in pipeline parallelism by considering more GPUs involves optimizing the distribution of the workload across the GPUs and minimizing communication overhead. As the number of GPUs increases, the workload can be divided into more stages or subtasks, allowing multiple stages to be executed concurrently. This parallel processing reduces the overall time needed to complete the entire pipeline. With more GPUs, it becomes possible to overlap the computation of one stage with the communication and setup of subsequent stages. This reduces the idle time of GPUs and increases overall throughput. While one GPU is busy processing, others can begin processing the next stage.

As depicted in Fig 8, the number of stages (representing the number of GPUs responsible for pipeline training) significantly influences the average time required for each epoch during training. Specifically, increasing the number of stages reduces training time per epoch. The implementation of SmartPipe further enhances this effect, contributing to a noticeable reduction in the time required for each epoch. By optimizing the training process and leveraging pipeline, SmartPipe effectively accelerates the overall training duration, making it more efficient and time-effective. The reason behind the time reduction is attributed to the freezing process, which effectively limits the number of interconnections between GPUs. This limitation optimizes communication overhead, enabling a more streamlined and efficient training process

TABLE II: Comparison of different freezing methods.

Model	Method	CIFAR-100		ImageNet	
		Accuracy	Time(s)	Accuracy	Time(s)
RestNet50	**Full training**	$96.10\% \pm 0.12\%$	$2,594$	$81.68\% \pm 0.15\%$	$2,500$
	Fixed Freezing (k=2)	$96.05\% \pm 0.25\%$	$1,980$	$81.48\% \pm 0.29\%$	$1,844$
	Linear Freezing (k=#epoch)	$95.03\% \pm 0.21\%$	$2,424$	$78.30\% \pm 0.48\%$	$1,963$
	SmartPipe (ours)	$96.02\% \pm 0.11\%$	$1,955$	$81.63\% \pm 0.22\%$	$1,787$
AlexNet	**Full training**	$97.48\% \pm 0.20\%$	$14,603$	$85.03\% \pm 0.28\%$	$14,628$
	Fixed Freezing (k=2)	$81.06\% \pm 0.23\%$	$8,760$	$73.89\% \pm 0.19\%$	$9,956$
	Linear Freezing (k=#epoch)	$90.56\% \pm 0.22\%$	$10,368$	$76.10\% \pm 0.32\%$	$10,654$
	SmartPipe (ours)	$97.35\% \pm 0.16\%$	$8,662$	$84.82\% \pm 0.25\%$	$9,599$
GoogLeNet	**Full Training**	$93.36\% \pm 0.17\%$	$2,698$	$74.95\% \pm 0.11\%$	$2,703$
	Fixed Freezing (k=2)	$92.35\% \pm 0.19\%$	$1,587$	$73.26\% \pm 0.23\%$	$1,846$
	Linear Freezing (k=#epoch)	$92.08\% \pm 0.19\%$	$10,291$	$72.26\% \pm 0.23\%$	$1,846$
	SmartPipe (ours)	$93.28\% \pm 0.25\%$	$1,554$	$74.66\% \pm 0.29\%$	$1,831$

across multiple GPUs in the pipeline. As a result, the overall training time is significantly reduced.

D. Comparison of Different Models and Different Datasets

In Table II, we present a comprehensive comparison of accuracy and training time for the RestNet50, AlexNet, and GoogleNet models on the CIFAR-100 and ImageNet datasets. The evaluation involves different approaches: 1) Full Training (No Freezing): All layers are trained without any freezing, 2) Fixed Freezing: A fixed number of layers are frozen in each epoch, 3) Linear Freezing: The number of frozen layers varies linearly with the number of epochs, 4) SmartPipe. The results indicate that the SmartPipe approach achieves accuracy close to those obtained by full training on both datasets. Moreover, our proposed approach significantly reduces the training time compared to the other evaluated methods.

VI. Conclusion

This study delved into the proposing a method for significantly reducing the training time of deep learning models while maintaining test accuracy by using of layer freezing in DNNs through pipeline parallelism. Layer freezing involves excluding specific layers from backpropagation, thus retaining their unchanged weights and reducing the computation time during backpropagation. Building upon the benefits of freezing layers, we propose an intelligent freezing approach that takes into account the training progress of layers throughout the epochs. By dynamically selecting layers to freeze during the training process, we aim to optimize the training efficiency further. To evaluate the effectiveness of our proposed scheme, we conducted trace-based simulations. The results of our approach clearly demonstrate its efficiency in reducing the time required for a single training iteration without significantly compromising accuracy.

References

[1] L. Cui, Z. Qu, G. Zhang, B. Tang, and B. Ye, "A bidirectional dnn partition mechanism for efficient pipeline parallel training in cloud," *Journal of Cloud Computing*, vol. 12, no. 1, p. 22, 2023.

[2] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *Proc. of the IEEE Conf. on Computer Communications (INFOCOM)*, 2020, pp. 626–635.

[3] H. Wang, C. Imes, S. Kundu, P. A. Beerel, S. P. Crago, and J. Paul Walters, "Quantpipe: Applying adaptive post-training quantization for distributed transformer pipelines in dynamic edge environments," in *Proc. of the Intl. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2023, pp. 1–5.

[4] Y. Liu, S. Agarwal, and S. Venkataraman, "Autofreeze: Automatically freezing model blocks to accelerate fine-tuning," *arXiv preprint arXiv:2102.01386*, 2021.

[5] X. Xiao, T. B. Mudiyanselage, C. Ji, J. Hu, and Y. Pan, "Fast deep learning training through intelligently freezing layers," in *Proc. of Intl. Conf. on Internet of Things (iThings)*, 2019, pp. 1225–1232.

[6] J. Lee, R. Tang, and J. Lin, "What would elsa do? freezing layers during transformer fine-tuning," *arXiv preprint arXiv:1911.03090*, 2019.

[7] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] 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 *Proc. of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[9] Y. Duan and J. Wu, "Optimizing resource allocation in pipeline parallelism for distributed dnn training," in *Proc. of the 28th IEEE Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 161–168.

[10] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[11] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[12] L. Shen, Y. Sun, Z. Yu, L. Ding, X. Tian, and D. Tao, "On efficient training of large-scale deep learning models: A literature review," *arXiv preprint arXiv:2304.03589*, 2023.

[13] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[14] J. Geng, D. Li, and S. Wang, "Elasticpipe: An efficient and dynamic model-parallel solution to dnn training," in *Proc. of the 10th Workshop on Scientific Cloud Computing*, 2019, pp. 5–9.

[15] J. Zhan and J. Zhang, "Pipe-torch: Pipeline-based distributed deep learning in a gpu cluster with heterogeneous networking," in *Proc. of the 7th IEEE Intl. Conf. on Advanced Cloud and Big Data (CBD)*, 2019, pp. 55–60.

[16] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[17] Z. Chen, C. Xu, W. Qian, and A. Zhou, "Elastic averaging for efficient pipelined dnn training," in *Proc. of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, p. 380–391.

[18] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *IEEE Intl. Conf. on Machine Learning (PMLR)*, 2021.

[19] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. of the IEEE Intl. Conf. on Machine Learning (PMLR)*, 2021, pp. 6543–6552.

[20] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "Pipetransformer: Automated elastic pipelining for distributed training of large-scale models," in *Proc. of the 38th Intl. Conf. on Machine Learning (PMLR)*, 2021, pp. 4150–4159.

[21] D. Team and R. Majumder, "Deepspeed: Extreme-scale model training for everyone," 2020.

[22] 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 *Proc. of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[23] Y. Wang, D. Sun, K. Chen, F. Lai, and M. Chowdhury, "Egeria: Efficient dnn training with knowledge-guided layer freezing," in *Proc. of the 18th European Conf. on Computer Systems*, 2023, pp. 851–866.

[24] S. Basodi, K. Pusuluri, X. Xiao, and Y. Pan, "Intelligent gradient amplification for deep neural networks," *arXiv preprint arXiv:2305.18445*, 2023.

[25] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

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

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

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