PipeTransformer: Automated Elastic Pipelining for Distributed Training of Large-scale Models

Chaoyang He\textsuperscript{1}  Shen Li\textsuperscript{2}  Mahdi Soltanolkotabi\textsuperscript{1}  Salman Avestimehr\textsuperscript{1}

Abstract

The size of Transformer models is growing at an unprecedented rate. It has taken less than one year to reach trillion-level parameters since the release of GPT-3 (175B). Training such models requires both substantial engineering efforts and enormous computing resources, which are luxuries most research teams cannot afford. In this paper, we propose PipeTransformer, which leverages automated elastic pipelining for efficient distributed training of Transformer models. In PipeTransformer, we design an adaptive on-the-fly freeze algorithm that can identify and freeze some layers gradually during training, and an elastic pipelining system that can dynamically allocate resources to train the remaining active layers. More specifically, PipeTransformer automatically excludes frozen layers from the pipeline, packs active layers into fewer GPUs, and forks more replicas to increase data-parallel width. We evaluate PipeTransformer using Vision Transformer (ViT) on ImageNet and BERT on SQuAD and GLUE datasets. Our results show that compared to the state-of-the-art baseline, PipeTransformer attains up to 2.83-fold speedup without losing accuracy. We also provide various performance analyses for a more comprehensive understanding of our algorithmic and system-wise design. Finally, we have modularized our training system with flexible APIs and made the source code publicly available at https://DistML.ai.

1. Introduction

Large Transformer models (Brown et al., 2020; Lepikhin et al., 2020) have powered accuracy breakthroughs in both natural language processing and computer vision. GPT-3 hit a new record high accuracy for nearly all NLP tasks. Vision Transformer (ViT) (Dosovitskiy et al., 2020) also achieved 89\% top-1 accuracy in ImageNet, outperforming state-of-the-art convolutional networks ResNet-152 (He et al., 2016) and EfficientNet (Tan & Le, 2019). To tackle the growth in model sizes, researchers have proposed various distributed training techniques, including parameter servers (Li et al., 2014; Jiang et al., 2020; Kim et al., 2019), pipeline parallel (Huang et al., 2019; Park et al., 2020; Narayanan et al., 2019), intra-layer parallel (Lepikhin et al., 2020; Shazeer et al., 2018; Shoeybi et al., 2019), and zero redundancy data parallel (Rajbhandari et al., 2019).

Figure 1. Interpretable Freeze Training: DNNs converge bottom up (Results on CIFAR10 using ResNet). Each pane shows layer-by-layer similarity using SVCCA (Raghu et al., 2017).

Existing distributed training solutions, however, only study scenarios where all model weights are required to be optimized throughout the training (i.e., computation and communication overhead remains relatively static over different iterations). Recent works on freeze training (Raghu et al., 2017; Morcos et al., 2018; Shen et al., 2020) suggest that parameters in neural networks usually converge from the bottom-up (i.e., not all layers need to be trained all the way through training). Figure 1 shows an example of how weights gradually stabilize during training in this approach. This observation motivates us to utilize freeze training for distributed training of Transformer models to accelerate training by dynamically allocating resources to focus on a shrinking set of active layers. Such a layer freezing strategy is especially pertinent to pipeline parallelism, as excluding consecutive bottom layers from the pipeline can reduce computation, memory, and communication overhead.

In this paper, we propose PipeTransformer, an elastic pipelining training acceleration framework that automatically reacts to frozen layers by dynamically transforming the scope of the pipelined model and the number of pipeline replicas. To the best of our knowledge, this is the first paper...
that studies layer freezing in the context of both pipeline and data-parallel training. Figure 2 demonstrates the benefits of such a combination. First, by excluding frozen layers from the pipeline, the same model can be packed into fewer GPUs, leading to both fewer cross-GPU communications and smaller pipeline bubbles. Second, after packing the model into fewer GPUs, the same cluster can accommodate more pipeline replicas, increasing the width of data parallelism. More importantly, the speedups acquired from these two benefits are multiplicative rather than additive, further accelerating the training.

The design of PipeTransformer faces four major challenges. First, the freeze algorithm must make on the fly and adaptive freezing decisions; however, existing work (Raghu et al., 2017) only provides a posterior analysis tool. Second, the efficiency of pipeline re-partitioning results is influenced by multiple factors, including partition granularity, cross-partition activation size, and the chunking (the number of micro-batches) in mini-batches, which require reasoning and searching in a large solution space. Third, to dynamically introduce additional pipeline replicas, PipeTransformer must overcome the static nature of collective communications and avoid potentially complex cross-process messaging protocols when onboarding new processes (one pipeline is handled by one process). Finally, caching can save time for repeated forward propagation of frozen layers, but it must be shared between existing pipelines and newly added ones, as the system cannot afford to create and warm up a dedicated cache for each replica.

PipeTransformer is designed with four core building blocks to address the aforementioned challenges. First, we design a tunable and adaptive algorithm to generate signals that guide the selection of layers to freeze over different iterations (Section 3.1). Once triggered by these signals, our elastic pipelining module AutoPipe, then packs the remaining active layers into fewer GPUs by taking both activation sizes and variances of workloads across heterogeneous partitions (frozen layers and active layers) into account. It then splits a mini-batch into an optimal number of micro-batches based on prior profiling results for different pipeline lengths (Section 3.2). Our next module, AutoDP, spawns additional pipeline replicas to occupy freed-up GPUs and maintains hierarchical communication process groups to attain dynamic membership for collective communications (Section 3.3). Our final module, AutoCache, efficiently shares activations across existing and new data-parallel processes and automatically replaces stale caches during transitions (Section 3.4).

Overall, PipeTransformer combines the Freeze Algorithm, AutoPipe, AutoDP and AutoCache modules to provide a significant training speedup. We evaluate PipeTransformer using Vision Transformer (ViT) on ImageNet and BERT on GLUE and SQuAD datasets. Our results show that PipeTransformer attains up to 2.83-fold speedup without losing accuracy. We also provide various performance analyses for a more comprehensive understanding of our algorithmic and system-wise design. Finally, we have also developed open-source flexible APIs for PipeTransformer which offer a clean separation among the freeze algorithm, model definitions, and training accelerations, allowing for transferability to other algorithms that require similar freezing strategies. The source code is made publicly available.

2. Overview

2.1. Background and Problem Setting

Suppose we aim to train a massive model in a distributed training system where the hybrid of pipelined model parallelism and data parallelism is used to target scenarios where either the memory of a single GPU device cannot hold the model, or if loaded, the batch size is small enough to avoid running out of memory. More specifically, we define our settings as follows:

Training task and model definition. We train Transformer models (e.g., Vision Transformer (Dosovitskiy et al., 2020), BERT (Devlin et al., 2018)) on large-scale image or text datasets. The Transformer model $F$ has $L$ layers, in which the $i$th layer is composed of a forward computation function $f_i$ and a corresponding set of parameters, $w_i$. With this definition, the overall model is $F = f_0(w_0) \circ \ldots \circ f_{L-1}(w_{L-1})$. The model size is $S$, and the batch size is set to $N_{bs}$.

Training infrastructure. Assume the training infrastructure contains a GPU cluster that has $N$ GPU servers (i.e. nodes). Each node has $I$ GPUs. Our cluster is homoge-
neous, meaning that each GPU and server have the same hardware configuration. Each GPU's memory capacity is $M_{\text{GPU}}$. Servers are connected by a high bandwidth network interface such as InfiniBand interconnect.

**Pipeline parallelism.** In each machine, we load a model $F$ into a pipeline $\mathcal{P}$ that has $K$ partitions ($K$ also represents the pipeline length). The $k$th partition $p_k$ consists of consecutive layers $p_k = f_1(w_1) \circ \ldots \circ f_j(w_j)$, and $\mathcal{P} = p_0 \circ \ldots \circ p_{K-1}$. We assume each partition is handled by a single GPU device. $1 \leq K \leq I$, meaning that we can build multiple pipelines for multiple model replicas in a single machine. We assume all GPU devices in a pipeline belong to the same machine. Our pipeline is a synchronous pipeline, which does not involve stale gradients, and the number of micro-batches is $M$. In the Linux OS, each pipeline is handled by a single process. We refer the reader to Pipelime (Huang et al., 2019) for more details.

**Data parallelism.** DDP (Li et al., 2020) is a cross-machine distributed data parallel process group within $R$ parallel workers. Each worker is a pipeline replica (a single process). The $r$th worker’s index (ID) is rank $r$. For any two pipelines $\mathcal{P}(r_i)$ and $\mathcal{P}(r_j)$ in DDP, $r_i$ and $r_j$ can belong to either the same GPU server or different GPU servers, and they can exchange gradients with the AllReduce algorithm. Under these settings, our goal is to accelerate training by leveraging freeze training, which does not require all layers to be trained throughout the duration of the training. Additionally, it may help save computation, communication, memory cost, and potentially prevent overfitting by consecutively freezing layers. However, these benefits can only be achieved by overcoming the four challenges of designing an adaptive freezing algorithm, dynamical pipeline re-partitioning, efficient resource reallocation, and cross-process caching, as discussed in the introduction. We next describe our overall design, named **PipeTransformer**, which can address these challenges.

### 2.2. Overall Design

PipeTransformer co-designs an on the fly freeze algorithm and an automated elastic pipelining training system that can dynamically transform the scope of the pipelined model and the number of pipeline replicas. The overall system architecture is illustrated in Figure 3. To support PipeTransformer’s elastic pipelining, we maintain a customized version of PyTorch Pipe (Kim et al., 2020). For data parallelism, we use PyTorch DDP (Li et al., 2020) as a baseline. Other libraries are standard mechanisms of an operating system (e.g., multi-processing) and thus avoid specialized software or hardware customization requirements. To ensure the **generality** of our framework, we have decoupled the training system into four core components: freeze algorithm, AutoPipe, AutoDP, and AutoCache. The freeze algorithm (grey) samples indicators from the training loop and makes layer-wise freezing decisions, which will be shared with AutoPipe (green). AutoPipe is an elastic pipeline module that speeds up training by excluding frozen layers from the pipeline and packing the active layers into fewer GPUs (pink), leading to both fewer cross-GPU communications and smaller pipeline bubbles. Subsequently, AutoPipe passes pipeline length information to AutoDP (purple), which then spawns more pipeline replicas to increase data-parallel width, if possible. The illustration also includes an example in which AutoDP introduces a new replica (purple). AutoCache (orange edges) is a cross-pipeline caching module, as illustrated by connections between pipelines. The source code architecture is aligned with Figure 3 for readability and generality.

### 3. Algorithm and System Design

This section elaborates on the four main algorithmic and system-wise design components of PipeTransformer.

#### 3.1. Freeze Algorithm

The freeze algorithm must be lightweight and able to make decisions on the fly. This excludes existing layer-wise training approaches such as SVCCA (Raghu et al., 2017) which require full training states and heavy posterior measurements. We propose an adaptive on the fly freeze algorithm to define $L_{\text{frozen}}^{(T)}$ at timestep $T$ as follows:

$$
\min \left( L_{\text{frozen}}^{(T-1)} + \alpha(L - L_{\text{frozen}}^{(T-1)}), \arg\min_{\ell \in \{L_{\text{frozen}}^{(T-1)}, \ldots, L\}} \| g_{\ell}^{(T)} \| \right)
$$

where $T \geq 1$, $L_{\text{frozen}}^{(0)} = 0$, and $\alpha \in (0, 1)$

(1)
where $g^{(T)}_{\ell}$ is the gradient for layer $\ell$ at iteration $T$, and $\|g^{(T)}_{\ell}\|$ is its norm. The intuition behind the second term in the $\min$ function is that the layer with the smallest gradient norm converges first. To stabilize training, we enforce an upper bound $L_{\text{frozen}}^{(T-1)} + \alpha (L - L_{\text{frozen}}^{(T-1)})$ for the number of frozen layers, which is a geometric sequence containing a hyper-parameter $\alpha$. This essentially freezes an $\alpha$ fraction of the remaining active layers. To illustrate the impact of $\alpha$, we rewrite the equation as: $L_{\text{frozen}}^{(T)} = (1 - \alpha)^T \left[ \frac{\alpha L}{1 - \alpha} + \sum_{t=2}^{T} \frac{\alpha L}{(1 - \alpha)^t} \right]$ (see Appendix for the derivation), and draw the curve of this function in Figure 4. As we can see, a larger $\alpha$ leads to a more aggressive layer freezing. Therefore, Equation 1 calculates the number of frozen layers at timestep $T$ using both the gradient norm and a tunable argument $\alpha$.

Balancing computation time across partitions is critical to pipeline training speed, as skewed workload distributions across stages can lead to stragglers, forcing devices with lighter workloads to wait (demonstrated by Section 4.3.1). However, maintaining optimally balanced partitions does not guarantee the fastest training speed because other factors also play a crucial role:

1. **Cross-partition communication overhead.** Placing a partition boundary in the middle of a skip connection leads to additional communications since tensors in the skip connection must now be copied to a different GPU. For example, with BERT partitions in figure 5, partition $k$ must take intermediate outputs from both partition $k-2$ and partition $k-1$.

   In contrast, if the boundary is placed after the addition layer, the communication overhead between partition $k-1$ and $k$ is visibly smaller. Our measurements show that having cross-device communication is more expensive than having slightly imbalanced partitions (see the Appendix). Therefore, we do not consider breaking skip connections (highlighted separately as an entire attention layer $f_{\text{ATT}}$, and MLP layer $f_{\text{MLP}}$, in green at line 7 in Algorithm 1).

2. **Frozen layer memory footprint.** During training, AutoPipe must recompute partition boundaries several times to balance two distinct types of layers: frozen layers and active layers. The frozen layer’s memory cost is a fraction of that in active layers, given that the frozen layer does not need backward activation maps, optimizer states, and gradients. Instead of launching intrusive profilers to obtain thorough metrics on memory and computational cost, we define a tunable cost factor $\lambda_{\text{frozen}}$ to estimate the memory footprint ratio of a frozen layer over the same active layer. Based on empirical measurements in our experimental hardware, we set $\lambda_{\text{frozen}}$ to $\frac{1}{6}$.

   Based on the above two considerations, AutoPipe balances pipeline partitions based on parameter sizes. More specifically, AutoPipe uses a greedy algorithm to allocate all frozen and active layers to evenly distribute partitioned sub-layers into $K$ GPU devices. Pseudo code is described as the $\text{load_balance()}$ function in Algorithm 1. The frozen layers are extracted from the original model and kept in a separate model instance $f_{\text{frozen}}$ in the first device of a pipeline. Note that the partition algorithm employed in this paper is not the only option; PipeTransformer is modularized to work with any alternatives.

### 3.2. AutoPipe: Elastic Pipelining

Triggered by the freeze algorithm, AutoPipe can accelerate training by excluding frozen layers from the pipeline and packing the active layers into fewer GPUs. This section elaborates on the key components of AutoPipe that dynamically partition pipelines, minimize the number of pipeline devices and optimize mini-batch chunk size accordingly. Algorithm 1 presents the pseudo-code.

#### 3.2.1. Balanced Pipeline Partitioning

![Figure 5. Partition boundary is in the middle of a skip connection](image)

**Remark:** Our system design idea can be generalized to many other progressive training algorithms. See Section 6 for more discussions.

#### 3.2.2. Pipeline Compression

Pipeline compression helps to free up GPUs to accommodate more pipeline replicas and reduce the number of cross-device communications between partitions. To determine the timing of compression, we can estimate the memory cost of the largest partition after compression, and then compare it with that of the largest partition of a pipeline at timestep...
Algorithm 1 AutoPipe Algorithm

1: def m_partition(F, L, Lfrozen): //see 3.2.1
2: for i = 0,0 to K - 1 do
3: \( S_{pipe[i]} = 0 \)
4: for \( L_{frozen} \) do
5: \( F_{pipe} = Sequential() \)
6: \( S_{pipe} = 0 \)
7: \( f_{\text{att}}, f_{\text{MLP}} \leftarrow f_i \)
8: \( F_{pipe}.append(f_{\text{att}}); S_{pipe}.append(f_{\text{size}}(f_{\text{att}})) \)
9: \( F_{pipe}.append(f_{\text{MLP}}); S_{pipe}.append(f_{\text{size}}(f_{\text{MLP}})) \)
10: return \( F_{pipe}, S_{pipe} \)
11: for sublayer index \( i = L_{\text{assigned}} \) to len \( S_{pipe} \) do
12: var=np.var \((S_{pipe}[L_{\text{assigned}}])/(K - k) \)
13: if criterion = B\(_S\)[i] - Lfrozen(1.0 - \( \lambda \)frozen) + \( S_k \)
14: \( B_S += S_k; B_L += 1; \text{Lassigned} += 1; \text{Stotal} = S_k \)
15: \( \text{break} \)
16: \( \text{return} B_L, B_S \)
17: \( M_{GPU} = \max(B_S, K/2) \)
18: \( M_{GPU}^{(T)} = \max(B_S, K/2) \)
19: \( \text{if criterion < mean + var then} \)
20: \( \text{K} = \text{K}/2 \)
21: \( \text{else} \)
22: \( \text{break} \)
23: \( \text{load} F_{\text{pipe}}, S_{\text{pipe}} \) to K GPUs using \( B_S \) and \( B_L \)
24: \( \text{Pipe} (F_{\text{pipe}}, \text{chunks} = \text{get_optimal_chunks} (K)) \)

Therefore, it is clear that shorter pipelines have smaller bubble sizes.

### 3.2.3. Dynamic Number of Micro-batches

Prior pipeline parallel systems use a fixed number of micro-batches per mini-batch \( M \). GPipe suggests \( M \geq 4 \times K \), where \( K \) is the number of partitions (pipeline length). However, given that that PipeTransformer dynamically configures \( K \), we find it to be sub-optimal to maintain a static \( M \) during training. Moreover, when integrated with DDP, the value of \( M \) also has an impact on the efficiency of DDP gradient synchronizations. Since DDP must wait for the last micro-batch to finish its backward computation on a parameter before launching its gradient synchronization, finer micro-batches lead to a smaller overlap between computation and communication (see Appendix for illustration). Hence, instead of using a static value, PipeTransformer searches for optimal \( M \) on the fly in the hybrid of DDP environment by enumerating \( M \) values ranging from \( K \) to \( 6K \). For a specific training environment, the profiling needs only to be done once (see Algorithm 1 line 35). Section 4 will provide performance analyses of \( M \) selections.

### 3.3. AutoDP: Spawning More Pipeline Replicas

As AutoPipe compresses the same pipeline into fewer GPUs, AutoDP can automatically spawn new pipeline replicas to increase data-parallel width.

Despite the conceptual simplicity, subtle dependencies on communications and states require careful design. The challenges are threefold: 1. **DDP Communication**: Collective communications in PyTorch DDP requires static membership, which prevents new pipelines from connecting with...
existing ones; 2. State Synchronization: newly activated processes must be consistent with existing pipelines in the training progress (e.g., epoch number and learning rate), weights and optimizer states, the boundary of frozen layers, and pipeline GPU range; 3. Dataset Redistribution: the dataset should be re-balanced to match a dynamic number of pipelines. This not only avoids stragglers but also ensures that gradients from all DDP processes are equally weighted.

To tackle these challenges, we create double communication process groups for DDP. As in the example shown in Figure 7, the message process group (purple) is responsible for light-weight control messages and covers all processes, while the active training process group (yellow) only contains active processes and serves as a vehicle for heavy-weight tensor communications during training. The message group remains static, whereas the training group is dismantled and reconstructed to match active processes. In T0, only process 0 and 8 are active. During the transition to T1, process 0 activates processes 1 and 9 (newly added pipeline replicas) and synchronizes necessary information mentioned above using the message group. The four active processes then form a new training group, allowing static collective communications adaptive to dynamic memberships. To redistribute the dataset, we implement a variant of DistributedSampler that can seamlessly adjust data samples to match the number of active pipeline replicas.

The above design also naturally helps to reduce DDP communication overhead. More specifically, when transitioning from T0 to T1, processes 0 and 1 destroy the existing DDP instances, and active processes construct a new DDP training group using $F_{\text{pipe}}$ (AutoPipe stores $F_{\text{frozen}}$ and $F_{\text{pipe}}$ separately, introduced in Section 3.2.1). Discussion of communication cost can be found in Appendix.

### 3.4. AutoCache: Cross-pipeline Caching

Caching activation maps from frozen layers can help further speed up training. This idea appears to be straightforward, but several caveats must be carefully addressed.

**Cross-process caching.** The cache must be shared across processes in real time, as creating and warming up a dedicated cache for each model replica slow down the training. This is achieved by spawning a dedicated daemon process to hold cache in shared memory that all training processes can access in real time. Figure 8 shows an example of the transition from T1 to T2, assuming T1 freezes 3 layers, T2 freezes 4 layers, and 5 layers remain active in T2. Immediately after the transition by AutoDP, the cache still holds cached activations from layer 3, which must be replaced by activations from layer 7. Therefore, all processes read their corresponding activations from the cache, feed them to the next 4 layers to compute activations for layer 7, then replace the existing cache with new activations for their samples accordingly. In this way, AutoCache can gradually update cached activations without running any sample through any frozen layers twice.

When the activations are too large to reside on CPU memory, AutoCache will also swap them to the disk and perform prefetching automatically. More details on the cross-process cache design can be found in the Appendix.

**Timing of cache** is also important, as the cache can be slower than running the real forward propagation, especially if frozen layers are few and activations are large. To ensure that our training system can adapt to different hardware, model architecture, and batch size settings, AutoCache also contains a profiler that helps evaluate the appropriate transition to enable caching, and it only employs cached activations when the profiler suggests caching can speed up the forward pass. Performance analysis is provided at Section 4.3.5.

### 4. Experiments

This section first summarizes experiment setups and then evaluates PipeTransformer using computer vision and natural language processing tasks. More comprehensive results can be found in the Appendix.

#### 4.1. Setup

**Hardware.** Experiments were conducted on 2 identical machines connected by InfiniBand CX353A (5GB/s), where each machine is equipped with 8 NVIDIA Quadro RTX 5000 (16GB GPU memory). GPU-to-GPU bandwidth within a machine (PCI 3.0, 16 lanes) is 15.754GB/s.

**Implementation.** We used PyTorch Pipe as a building block, which has not yet been officially released at the time.
of writing of this paper. Hence, we used the developer version 1.8.0.dev20201219. The BERT model definition, configuration, and related tokenizer are from HuggingFace 3.5.0. We implemented Vision Transformer using PyTorch by following its TensorFlow implementation. More details can be found in our source code.

Models and Datasets. Experiments employ two representative Transformers in CV and NLP: Vision Transformer (ViT) and BERT. ViT was run on an image classification task, initialized with pre-trained weights on ImageNet21K and fine-tuned on ImageNet and CIFAR-100. BERT was run on two tasks, text classification on the SST-2 dataset from the General Language Understanding Evaluation (GLUE) benchmark, and question answering on the SQuAD v1.1 Dataset (Stanford Question Answering) which is a collection of 100k crowdsourced question/answer pairs.

Training Schemes. Given that large models normally would require thousands of GPU-days (e.g., GPT-3) if trained from scratch, fine-tuning downstream tasks using pre-trained models has become a trend in CV and NLP communities. Moreover, PipeTransformer is a complex training system that involves multiple core components. Thus, for the first version of PipeTransformer system development and algorithmic research, it is not cost-efficient to develop and evaluate from scratch using large-scale pretraining. Therefore, experiments presented in this section focuses on pre-trained models. Note that since the model architectures in pre-training and fine-tuning are the same, PipeTransformer can serve both. We discussed pre-training results in the Appendix.

Baseline. Experiments in this section compares PipeTransformer to the state-of-the-art framework, a hybrid scheme of PyTorch Pipe (PyTorch’s implementation of GPipe (Huang et al., 2019)) and PyTorch DDP. Since this is the first paper that studies accelerating distributed training by freezing layers, there are no perfectly aligned counterpart solutions yet.

Hyper-parameters. Experiments use ViT-B/16 (12 transformer layers, 16 × 16 input patch size) for ImageNet and CIFAR-100, BERT-large-uncased (24 layers) for SQuAD 1.1, and BERT-base-uncased (12 layers) for SST-2. With PipeTransformer, ViT and BERT training can set the per-pipeline batch size to around 400 and 64 respectively. Other hyperparameters (e.g., epoch, learning rate) for all experiments are presented in Appendix.

4.2. Overall Training Acceleration

We summarize the overall experimental results in Table 1. Note that the speedup we report is based on a conservative α ( 1 2 ) value that can obtain comparable or even higher accuracy. A more aggressive α ( 5 2 , 1 2 ) can obtain a higher speedup but may lead to a slight loss in accuracy (See section 4.3.3). Note that the model size of BERT (24 layers) is larger than ViT-B/16 (12 layers), thus it takes more time for communication (see Section 4.3.2 for details).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Baseline</th>
<th>PipeTransformer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Training time</td>
</tr>
<tr>
<td>ImageNet</td>
<td>80.83 ± 0.05</td>
<td>26h 30m</td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>91.21 ± 0.07</td>
<td>35m 6s</td>
</tr>
<tr>
<td>SQuAD 1.1</td>
<td>90.71 ± 0.18</td>
<td>7h 7m</td>
</tr>
</tbody>
</table>

*Note: 1. the accuracy is the mean and variance of three independent runs with the same random seed; 2. the training time among different runs are relatively stable (the gap is less than 1 minute); 3. GLUE (SST-2)’s training time is relatively small, thus we mainly used it for debugging without reporting a few minutes result. 4. accuracy metric: ImageNet/CIFAR-100: top-1 accuracy; SQuAD: F1 score.

4.3. Performance Analysis

This section presents evaluation results and analyzes the performance of different components in PipeTransformer. More experimental results can be found in the Appendix.

4.3.1. Speedup Breakdown

![Figure 9. Speedup Breakdown (ViT on ImageNet)](attachment:image)

To understand the efficacy of all four components and their impacts on training speed, we experimented with different combinations and used their training sample throughput (samples/second) and speedup ratio as metrics. Results are illustrated in Figure 9. Key takeaways from these experimental results are: 1. the main speedup is the result of elastic pipelining which is achieved through the joint use of AutoPipe and AutoDP; 2. AutoCache’s contribution is amplified by AutoDP; 3. freeze training alone without system-wise adjustment even downgrades the training speed (discussed in Section 3.2). We provide additional explanations of these results in the Appendix.

4.3.2. Communication Cost

We also analyzed how communication and computation contribute to the overall training time. Since PyTorch DDP overlaps communication with computation, the tim-
ference between a local training iteration and distributed training iteration does not faithfully represent the communication delay. Moreover, as DDP also organizes parameters into buckets and launches an AllReduce for each bucket, recording the start and finish time of overall communications also falls short, as there can be time gaps between buckets. To correctly measure DDP communication delay, we combined the DDP communication hook with CUDAFuture callback. More details of this measurement are documented in the Appendix. Key takeaways: 1. larger model cost more time on communication (BERT on SQuAD); 2. a higher cross-machine bandwidth can further speedup the training, especially for larger model.

Table 2. Communication Cost v.s. Computational Cost

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Overall Cost</th>
<th>Communication Cost</th>
<th>Computation Cost</th>
<th>Communication to Computation Cost Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageNet</td>
<td>9h 21m</td>
<td>34m</td>
<td>8h 47m</td>
<td>5.9 %</td>
</tr>
<tr>
<td>SQuAD</td>
<td>2h 26m</td>
<td>16m 33s</td>
<td>2h 9m</td>
<td>8.8%</td>
</tr>
</tbody>
</table>

4.3.3. Tuning α in Freezing Algorithm

We ran experiments to show how the α in the freeze algorithms influences training speed. The result clearly demonstrates that a larger α (excessive freeze) leads to a greater speedup but suffers from a slight performance degradation. In the case shown in Figure 10(a), where α = 1/5, freeze training outperforms normal training and obtains a 2.04-fold speedup. We provide more results in the Appendix.

4.3.4. Optimal Chunks in Elastic Pipeline

We profiled the optimal number of micro-batches M for different pipeline lengths K. Results are summarized in Figure 10(b). As we can see, different K values lead to different optimal M, and the throughput gaps across different M values are large (as shown when K = 8), which confirms the necessity of an anterior profiler in elastic pipelining.

4.3.5. Understanding the Timing of Caching

To evaluate AutoCache, we compared the sample throughput of training that activates AutoCache from epoch 0 (blue) with the training job without AutoCache (red). Figure 10(c) shows that enabling caching too early can slow down training, as caching can be more expensive than forward propagation on a small number of frozen layers. After freezing more layers, caching activations clearly outperforms the corresponding forward propagation. As a result, AutoCache uses a profiler to determine the proper timing to enable caching. In our system, for ViT (12 layers), caching starts from 3 frozen layers, while for BERT (24 layers), caching starts from 5 frozen layers.

5. Related Works

PipeTransformer combines pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019; 2021; Park et al., 2020; Yang et al., 2021) and data parallelism (Li et al., 2020). Both techniques have been extensively studied in prior work. GPipe (Huang et al., 2019) parallelizes micro-batches within a mini-batch and enforces synchronizations between consecutive mini-batches. The synchronization barrier creates execution bubbles and it exacerbates if the model spans across more devices. PipeDream (Narayanan et al., 2019; 2021), Megatron-LM (Narayanan et al., 2021), HetPipe (Park et al., 2020) and PipeMare (Yang et al., 2021) remove or mitigate execution bubbles by allowing a configurable amount of staleness. Although evaluations show that models can still converge with high accuracy, it breaks the mathematical equivalence to local training. PipeTransformer builds on top of PyTorch pipeline parallel and distributed data-parallel APIs (Li et al., 2020). Compared to prior solutions, PipeTransformer reduces the size of bubbles during training by dynamically packing the active layers into fewer GPUs. Moreover, the communication overhead for data-parallel training, which is the dominant source of delay, also drops when the active model size shrinks.

6. Discussion

Pretraining v.s. Fine-tuning: Given that the model architectures in pre-training and fine-tuning are the same, we do not need to change the system design. Running larger Transformers (over 32 layers) is straightforward because almost all giant Transformer models are designed by simply stacking more transformer encoder layers. PipeTransformer can serve as a training system for both pre-training and fine-tuning training. We plan to run our training system on more models and datasets in both
settings.

**Designing better freeze algorithm:** Our proposed algorithm is simple, yet it proves to be effective on various tasks. However, we believe that further developments to the freeze algorithm may lead to better generalization and obtain higher accuracy.

**Versatility:** PipeTransformer training system can also be used on other algorithms that run progressive training (Gong et al., 2019) or gradually fix portions of neural network. For example, cross-silo federated learning, layer-by-layer neural architecture search, and pruning large DNNs are all potential use cases of our training system. We will explore the training acceleration for these scenarios in our future works.

### 7. Conclusion

This paper proposes PipeTransformer, a holistic solution that combines elastic pipeline-parallel and data-parallel for distributed training. More specifically, PipeTransformer incrementally freezes layers in the pipeline, packs remaining active layers into fewer GPUs, and forks more pipeline replicas to increase the data-parallel width. Evaluations on ViT and BERT models show that compared to the state-of-the-art baseline, PipeTransformer attains up to 2.83× speedups without accuracy loss.

### Acknowledgments

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under FASTNICS program Contract Number HR001120C0088. Mahdi Soltanolkotabi is additionally supported by the Packard Fellowship in Science and Engineering, a Sloan Research Fellowship in Mathematics, an NSF-CAREER under award #1846369, DARPA Learning with Less Labels (LwLL) program, and NSF-CIF awards #1813877 and #2008443. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

### References


PipeTransformer: Automated Elastic Pipelining for Distributed Training of Large-scale Models


