TSPipe: Learn from Teacher Faster with Pipelines

Hwijoon Lim[1] **Yechan Kim**[2] **Sukmin Yun**[1] **Jinwoo Shin**[1 2] **Dongsu Han**[1 2]

Abstract

The teacher-student (TS) framework, training a (student) network by utilizing an auxiliary superior (teacher) network, has been adopted as a popular training paradigm in many machine learning schemes, since the seminal work—Knowledge distillation (KD) for model compression and transfer learning. Many recent self-supervised learning (SSL) schemes also adopt the TS framework, where teacher networks are maintained as the moving average of student networks, called the momentum networks. This paper presents TSPipe, a pipelined approach to accelerate the training process of any TS frameworks including KD and SSL. Under the observation that the teacher network does not need a backward pass, our main idea is to schedule the computation of the teacher and student network separately, and fully utilize the GPU during training by interleaving the computations of the two networks and relaxing their dependencies. In case the teacher network requires a momentum update, we use delayed parameter updates only on the teacher network to attain high model accuracy. Compared to existing pipeline parallelism schemes, which sacrifice either training throughput or model accuracy, TSPipe provides better performance trade-offs, achieving up to 12.15x higher throughput. [1]

1. Introduction

Knowledge distillation (KD) (Hinton et al., 2015) has shown remarkable success with the teacher-student (TS) framework in transferring knowledge from a teacher network to a student network. Motivated by this, the TS framework has been used in a broader range of applications—vision (Pham et al., 2021), natural language processing (Sanh et al., 2019), and deep reinforcement learning (Yin & Pan, 2017). In particular, many recent studies in self-supervised learning (SSL) for vision (He et al., 2020; Chen et al., 2021; Grill et al., 2020; Li et al., 2021a; Zhou et al., 2021) have successfully learned visual representations from a large number of unlabeled data using the TS framework.

[1]School of Electrical Engineering, KAIST, Daejeon, Republic of Korea [2]Kim Jaechul Graduate School of AI, KAIST, Daejeon, Republic of Korea. Correspondence to: Dongsu Han <dhan.ee@kaist.ac.kr>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

[1]The source code is available at `https://github.com/kaist-ina/TSPipe`.

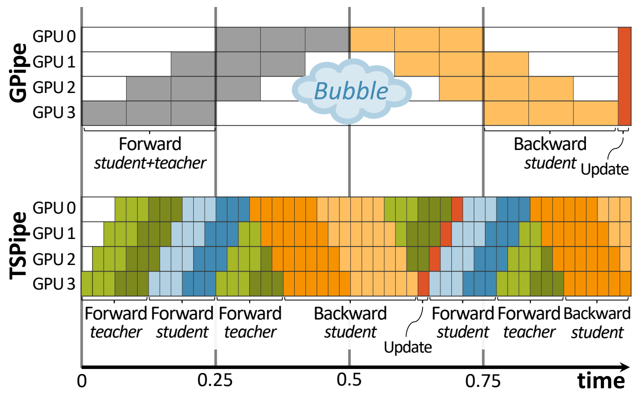

Figure 1. TSPipe achieves high training throughput by eliminating pipeline bubbles. **Top:** With GPipe, GPUs are idle, exhibiting pipeline bubbles. Gray blocks indicate the forward pass, and orange blocks indicate the backward pass. **Bottom:** Timeline for TSPipe. Green, blue, and orange blocks indicate the forward pass of the teacher network, the forward pass of the student network, and the backward pass of the student network, respectively. Note two figures share the same time axis, which is normalized.

However, both KD and SSL often suffer from the extensive amounts of resource requirements (*e.g.,* GPU memory and computation) for training. For example, KD often leverages large teacher networks—in Natural Language processing (NLP), the state-of-the-art pre-trained language models have up to 175B parameters (Brown et al., 2020; Zhang et al., 2022), which requires 700 GB of GPU memory only for the model itself. Many recent SSL methods also employ a large-scale of architectures for better representation learning, e.g., MoCo-v3 (Chen et al., 2021) adopting ViT (Dosovitskiy et al., 2020), a transformer-based model, takes 128 GPU-days with ViT-B (86 M parameters) to converge. In addition, SSL methods often require extensive training epochs for model convergence—BYOL (Grill et al., 2020) needs an order of magnitude more training epochs to achieve the accuracy proximal to the supervised learning counterpart.

In some cases, even a cutting-edge GPU cannot accommodate such large models, which led to the adoption of Model Parallelism (MP) that splits a model into multiple GPUs. However, MP suffers from either serious under-utilization due to the dependency among layers (inter-layer MP) (Huang et al., 2019) or extreme slowdown in multi-node environments due to the inter-node communication overhead (intra-layer MP) (Narayanan et al., 2021). Thus, increasing the number of GPUs hardly contributes to speed-up in training with MP.

To overcome this issue, many recent works introduce pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019; 2021; Park et al., 2020; Li et al., 2021b). However, existing solutions still fail to achieve high training throughput while maintaining scalability and model accuracy. For example, in Figure 1, GPipe (Huang et al., 2019), one of the well-known pipeline parallelism schemes, only utilizes 57% of GPU time and fails to fully schedule training tasks. This is due to the fundamental challenge that pipeline parallelism faces—dependencies between layers cannot be eliminated without changing training semantics at the risk of accuracy degradation.

This paper presents TSPipe, a novel approach to accelerate the training of any TS framework including KD and SSL by pipelining multiple GPUs. TSPipe is the only training scheme that achieves three highs—*high training throughput, high model accuracy, and high scalability*. We achieve these by leveraging the following unique properties of the TS framework; 1) The teacher network does not need a backward pass. 2) The parameters of the teacher network are never updated, or updated in a steady and stable manner (i.e, momentum coefficient, $\tau = 0.996$).

Unlike other schemes that do not distinguish the teacher from the student, TSPipe schedules them separately to take advantage of the property of the TS framework. TSPipe interleaves the computations of the teacher network between the computations of the student, which enables us to eliminate all pipeline bubbles without additional memory footprint for activation stashing. This allows TSPipe to train larger models since activation memory accounts for the majority of total memory usage.

TSPipe further applies delayed parameter updates as in other schemes (Narayanan et al., 2021; Xu et al., 2020; Park et al., 2020), to fully schedule the pipeline. However, unlike the existing schemes, TSPipe considers asymmetric parameter update. To be specific, we suggest applying delayed parameter update *only on the teacher network*, as the slow update of the teacher network parameters allows TSPipe to mitigate the performance drop of the student network. As a result, TSPipe provides high training throughput and a better trade-off between the GPU memory footprint and utilization, without loss of the model performance.

We demonstrate the efficiency of TSPipe by training various KD and SSL schemes. For example, When we train MoCo-v3 under multiple-sized ViT architectures with 16 GPUs, TSPipe achieves up to 12.15x higher training throughput compared to inter-layer MP (Shoeybi et al., 2019). When we perform KD from ViT networks to ResNet with 8 GPUs, TSPipe achieves up to 4.68x higher training throughput over inter-layer MP. We also evaluate the learned representation quality for SSL where we adopt asymmetric parameter update. TSPipe preserves the same accuracy as the inter-layer MP under ResNet-18 with respect to the linear evaluation protocol (Chen et al., 2020). To the best of our knowledge, TSPipe is the first framework for training parallelism that targets the TS framework.

2. Background and Related Work

This paper focuses on the general teacher-student framework of Knowledge distillation (Hinton et al., 2015), which is an effective learning scheme to transfer the knowledge from a powerful teacher network to a student. We remark many recent SSL frameworks (He et al., 2020; Chen et al., 2021; Grill et al., 2020; Roh et al., 2021) also belong to a form of TS framework with a slight variation, which leverages two encoder networks in training: the online (student) network θ and the target (teacher) network ξ. The former is the primary network for encoding the final representations directly updated by the loss gradients, and the parameters ξ of is the latter target (momentum) network (Tarvainen & Valpola, 2017) updated by an exponential moving average of parameters θ of the former as:

$$\xi \leftarrow \tau\xi + (1-\tau)\theta, \tag{1}$$

where $\tau \in [0, 1]$ is a momentum coefficient. One key idea of our work is that such TS frameworks do not require back-propagating gradients of the target (or teacher) network during training.

Many KD and SSL models feature 100M+ parameters (e.g., ViT (Dosovitskiy et al., 2020)), which cannot be trained with a single GPU due to the memory constraint. Pure data parallelism that does not split a model across GPUs cannot be used to train large models that do not fit in a single GPU's memory. Mechanisms for distributed training are discussed below.

Model parallelism (MP) (Shoeybi et al., 2019; Shazeer et al., 2018; Chilimbi et al., 2014) splits a model into multiple partitions and places each partition into a single GPU. This enables training larger models. Specifically, MP can be further classified into inter-layer MP and intra-layer MP. Inter-layer MP partitions a model layer-wise, and each parti-

		DP (Data Parallelism)	Inter-layer MP	GPipe (Huang et al., 2019)	**TSPipe**
Model + Optimizer states		$2W$	$\frac{2W}{N}$	$\frac{2W}{N}$	$\frac{2W}{N}$
Batch + Activations		$\sum_{i<n} \frac{A_i}{Nk}$	$\max_j \sum_{L_i \in P_j} \frac{A_i}{k}$	$\max_j \sum_{L_i \in P_j} \frac{A_i}{k}$	$\max_j \sum_{L_i \in P_j} \frac{A_i}{k}$
Ideal pipeline utilization		1	$\frac{1}{N}$	$\frac{u}{u+N-1}$	1
KD DistilBERT BERT-xxlarge	Total memory	39.70 GiB	**24.78 GiB**	**24.78 GiB**	**24.78 GiB**
	Ideal utilization	Out of Memory	12.5%	53%	**100%**
SSL MoCo-v3 ViT-Large	Total memory	34.24 GiB	**28.02 GiB**	**28.02 GiB**	**28.02 GiB**
	Ideal utilization	Out of Memory	12.5%	53%	**100%**

Table 1. Peak GPU memory footprints and ideal GPU utilization. W is the model size, N is the number of GPUs, A_i is the activation memory for a layer L_i, P_j is the model partition, k is the degree of gradient accumulation, and u is the number of microbatches per batch. In the lower part of the table, we give the example of training models with eight V100 GPUs, each with 32 GB of GPU memory.

tion holds subsets of layers. After each GPU completes computing a batch using the partition of the model, it hands over activations and gradients to the next GPU holding the immediate subsequent partition. However, due to this dependency between partitions, MP suffers from the under-utilization of GPUs (Huang et al., 2019).

Meanwhile, in intra-layer MP, each layer is sharded over multiple GPUs (e.g., tensor parallelism (Shoeybi et al., 2019)). Although intra-layer MP does not suffer from the dependency issue of the inter-layer MP, calculating each layer requires all-to-all communication between all sharded workers, which brings significant performance overhead (Narayanan et al., 2021). When intra-layer MP is applied over multiple nodes, all-to-all communication remarkably slows down the training due to limited bandwidth, making the strategy not scalable.

Pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019; Park et al., 2020; Narayanan et al., 2021; Li et al., 2021b) tackles the GPU resource under-utilization problem of inter-layer MP by pipelining computations. The key difference with inter-layer MP is that pipeline parallelism splits given (mini-)batches into microbatches. Instead of waiting for the computation of a whole (mini-)batch to be completed, each GPU sends activations and gradients immediately after they are done on computations for a single microbatch. This relaxes communication dependencies so that the next GPU can start computing without waiting for the previous GPUs to complete the whole batch.

However, proposed solutions for pipeline parallelism still exhibit undesirable trade-offs between the model staleness, memory footprint, and training throughput. GPipe (Huang et al., 2019) achieves better GPU utilization and higher throughput than MP by pipelining the computation, but still fails to fully utilize GPUs. The maximum theoretical training efficiency of GPipe only reaches 50.3% with 64 GPUs and 64 microbatches. DSP (Xu et al., 2020) has flexibility in its scheduling with layer-wise staleness. However, it makes both student and teacher networks stale worsening on lower layers, and requires activation recomputation which adds 25% more computation.

PipeDream (Narayanan et al., 2019) and PipeDream-2BW (Narayanan et al., 2021) employ pipeline parallelism. However, they do not support SSL with momentum networks out-of-the-box. This is because both schemes highly depend on their planners' algorithms to partition models, but the planners do not support the unique model structure of momentum networks. Even if PipeDream and PipeDream-2BW are redesigned to support the structure of momentum networks as in TSPipe, they would still suffer from a large memory footprint and final accuracy drop, respectively.

Asynchronous pipeline parallelism (Niu et al., 2011; Yang et al., 2021; Xu et al., 2020) tries to achieve speedup by allowing discrepancy between weight versions used in the forward and backward passes. However, this incurs a varying degree of staleness across partitions, which requires extra effort (i.e., activation recomputation (Xu et al., 2020), learning rate rescheduling (Yang et al., 2021)) to prevent accuracy drops. To avoid the additional cost and retain high accuracy, we do not take the approach.

3. Method

Challenge. Due to the dependencies between computations, pipeline parallelism cannot fully schedule the computations, resulting in GPU under-utilization. Prior approaches to override the dependencies and fully schedule the computations came with side effects, including larger memory footprint (Narayanan et al., 2019), reduced computing efficiency (Xu et al., 2020), and degraded model accuracy (Narayanan et al., 2021). In our paper, we ask if

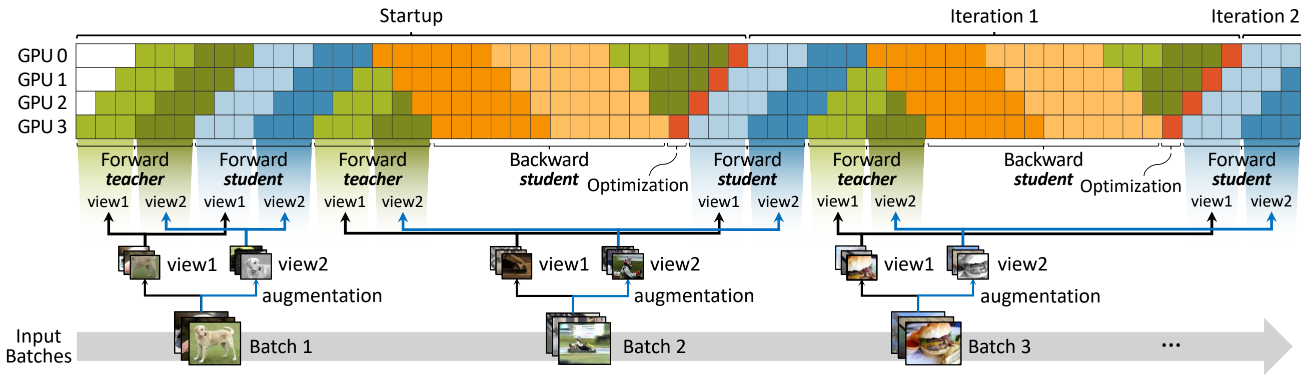

Figure 2. Overview image of TSPipe that shows how TSPipe trains self-supervised learning network. TSPipe achieves full utilization of GPU pipelines by scheduling the teacher network's forward pass between computations of the student network.

it is possible to achieve full scheduling without using extra memory footprint and model accuracy loss.

Our observation regarding the TS framework provides a key for this challenge; a teacher network does not require a backward pass, so its forward pass can be scheduled more leniently than the student's, without worrying about the activation stashing, which doubles the memory footprint and halves the maximum model sizes. Thus, unlike other schemes that do not distinguish the two networks, we schedule the teacher and student networks separately. This allows us to fully schedule computations without increasing the memory footprint (§3.1).

In addition, separating student and teacher networks also benefits in terms of accuracy. Since we schedule these two separately, we can choose to apply staleness only to the teacher network to maintain model accuracy. Therefore, SSL, in which the teacher network is updated slowly, can mitigate the loss of accuracy, and the KD whose teacher network remains unchanged has no loss of accuracy at all (§3.2).

3.1. Achieving high GPU utilization without a large memory footprint

TSPipe introduces a novel pipeline parallelism scheme for the TS framework. Unlike other works that do not take into account the structure of the TS framework, TSPipe separates the scheduling of the student and teacher network from its design. This enables TSPipe to benefit from the property that the teacher network does not need a backward pass, allowing TSPipe to interleave the teacher network's forward pass between the computation of the student network without concern about activation stashing. This simple but clever idea enables TSPipe to achieve 100% GPU utilization. Figure 2 illustrates the pipeline of TSPipe with 4 GPUs.

During the startup phase, TSPipe first calculates a teacher network's forward pass and then calculates a student network's forward pass. With the loss calculated from the two forward passes, TSPipe computes a student network's backward pass. Between the forward and backward passes, GPU time slots naturally become idle, creating a bubble as depicted in Figure 1. In the bubble, TSPipe *inserts* a teacher network's forward pass for the next batch. Once the backward pass for the current batch finishes, TSPipe updates the model parameters using an optimizer (student network) and weighted average (teacher network) if needed.

After the first batch, TSPipe enters steady-state, where pipelines are fully scheduled. TSPipe computes the backward pass of the student network, using the teacher network's forward pass from the previous iteration and the student network's forward pass from the current iteration. The teacher network's forward pass for the next batch is pre-computed in the current iteration so that it can be used at the next iteration. Note that we schedule the teacher network's forward pass in the pipeline bubbles between the student network's forward and backward passes. This way, we maximize the pipeline utilization and reduce the training time.

Strategy for splitting a batch for full GPU scheduling. Unlike existing works (Huang et al., 2019; Park et al., 2020), TSPipe manages two different microbatch sizes for the forward and backward passes, u_f and u_b, respectively. This accommodates a longer processing time taken in backward passes compared to forward passes in processing the same tensor size (Narayanan et al., 2021). During the backward passes, the autograd engine needs to accumulate all relative tensors and propagates in addition to computing the gradients. Empirically, we estimate that backward passes take twice the longer time than the forward pass, and thus we set $u_b = 2 \times u_f$.

To completely fill up the pipeline, TSPipe splits a single batch into $u_f = N - 1$ microbatches, where N is the number of GPUs. In the prior work (Huang et al., 2019), the computation of a single batch takes $n_v(2u_f + u_b + N - 1)$ time slots per GPU, where n_v is the number of views per batch, generally 2 for SSL networks and 1 for others. This is composed of the computation time $n_v(2u_f + u_b)$ and the pipeline bubbles $n_v(N - 1)$. TSPipe eliminates this bubble by filling in the teacher network's forward pass for the next batch, which takes $n_v u_f$ time slots.

Table 1 shows the ideal GPU utilization for each scheme. For inter-layer MP, it is $\frac{1}{N}$, where N is the number of GPUs. This is because only one GPU can be active at a time. GPipe's (Huang et al., 2019) ideal GPU utilization is $\frac{u}{u+N-1}$, where u is the number of microbatches per batch. Although GPipe can achieve higher utilization with high u, it comes with a reduced size of microbatches. This brings significant scheduling overhead, as well as inefficient utilization of CUDA cores in GPUs. The ideal GPU utilization of TSPipe is 1 in steady-state, which means we can achieve up to $\frac{u+N-1}{u}$x throughput gain over GPipe; e.g., with 8 GPUs and $u = 8$, this gives 1.88x improvement.

High GPU utilization without additional memory cost. TSPipe achieves high GPU utilization without additional GPU memory cost compared to the other pipeline parallelism schemes. As shown in Table 1, TSPipe uses the same amount of model and activation memory as MP. Unlike stashing approaches (Narayanan et al., 2019; 2021) that keep multiple versions of activation or parameters in memory, TSPipe holds exactly one version of activation and parameters in each GPU, which enables us to train a very large network by splitting it into multiple GPUs. In order to train larger batches, TSPipe also leverages gradient accumulation, where the model accumulates gradients for k iterations without updating parameters. This allows TSPipe to keep a low memory footprint even when training large batches, requiring the activation memory of $\frac{A}{k}$, where A is the activation size for a partition. Note that TSPipe's memory footprint can be further reduced with activation checkpointing (Chen et al., 2016), but it comes at the cost of increased computation.

3.2. Attaining high accuracy

To achieve fast training throughput and high utilization, many existing pipeline parallelism schemes (Narayanan et al., 2021; Park et al., 2020) make changes in training semantics. They schedule the computation for the current batch before the model parameters are updated from the previous batch computations, which we call "early computation". This results in the degraded model accuracy, as their early computation cannot reflect the gradient from the previous batch. In contrast, although TSPipe also introduces the early computation, TSPipe preserves the model accuracy leveraging the property that the teacher network is updated slowly (momentum network-based SSL) or never updated (KD).

Preserving accuracy with asymmetric parameter update. Leveraging the fact that the teacher network is updated slowly, TSPipe performs early computation only for the teacher network. We use the current student network (θ_n) and the stale teacher network (ξ_{n-1}) to compute the loss $\mathcal{L}_{\theta_n,\xi_{n-1}}$:

$$\theta_{n+1} \leftarrow \text{optimizer}(\theta_n, \nabla_{\theta_n}\mathcal{L}_{\theta_n,\xi_{n-1}}, \eta) \tag{2}$$

where θ_{n+1} is the student network's parameter at $n+1$-th iteration, ∇_{θ_n} is the gradient calculated from $\mathcal{L}_{\theta_n,\xi_{n-1}}$ with respect to θ_n, and η is the learning rate. The asymmetric update has minimal impact on model performance because the teacher is updated as the exponential moving average of the student network (Equation (1)) with τ close to 1, which implies $\xi_n \approx \xi_{n+1}$, and thus $\mathcal{L}_{\theta_n,\xi_n} \approx \mathcal{L}_{\theta_n,\xi_{n-1}}$. In §4.2, we demonstrate that this is indeed the case for real-world workloads.

Note KD is a special case of momentum-based SSL where the momentum $\tau = 1$ and thus, $\xi_n = \xi_{n-1}$ for all n. Substituting ξ_{n-1} for ξ_n into Equation (2) shows that TSPipe exactly preserves the original training semantic of KD, results in the model accuracy preservation.

On the other hand, generic pipelined approaches (Ren et al., 2021; Narayanan et al., 2021) do not differentiate the student network from the teacher network and perform early computation for both networks, updating the parameter as:

$$\theta_{n+1} \leftarrow \text{optimizer}(\theta_n, \nabla\mathcal{L}_{\theta_{n-1},\xi_{n-1}}, \eta). \tag{3}$$

The difference between θ_n and θ_{n-1} is significant. This discrepancy gradually propagates throughout parameter updates and eventually results in model accuracy degradation. This is even worse on recent SSL approaches where they often adopt a high learning rate due to large batch sizes(He et al., 2020; Grill et al., 2020). Note that KD also suffers from accuracy degradation with this approach, a substituting ξ_{n-1} for ξ_n into Equation (3) still differs from the original training semantic of KD.

3.3. Discussion

Model partitioning. We use a simple model partitioning method similar to one that appears in Narayanan et al. (2021). Our method aims to find a schedule $s : L \rightarrow D$, where L is a set of Layers L_i and D is a set of devices D_j. We denote a function s as a vector $(s_1, s_2, \cdots, s_m)$ where $S(L_i) = D_{s_i}$. We consider three dominant factors:

- **Memory footprint:** We measure the actual activation and model parameter size required to compute the layer.

- **Transfer time $t_i^{(t)}$:** We estimate the transfer time $t_i^{(t)}$ by dividing the output batch size of layer L_i into the bandwidth between the two device s_i and s_{i+1}. Here we consider if P2P technology is available between GPUs (i.e., NVLink).
- **Computing time $t_i^{(c)}$:** We measure the time required for the forward pass of layer L_i. We assume the computing time is identical for all GPUs.

We exhaustively search for the optimal schedule $\hat{s}$ that minimizes the bottleneck processing time $\max_i \max(t_i^{(t)}, t_i^{(c)})$ such that the memory footprints of partitions fit into the assigned device memory.

Batch normalization. TSPipe normalizes input batches and keeps track of the mean and variance on every microbatch. This may potentially degrade performance with models with many batch normalization layers. To mitigate the problem, we adopt deferred batch normalization from GPipe.

Combining with DP. As with other pipeline parallelism schemes, TSPipe can be combined with DP (Data Parallelism), which enables us to run multiple parallel pipelines, for better scalability (Li et al., 2020). For example, with 64 GPUs, we can run 4 parallel pipelines, each composed of 16 GPUs. It is especially useful when training a relatively small model with a large number of GPUs, e.g., when the number of GPUs are similar to or larger than the number of layers. For small models, it would be more efficient to partition the model into a small number of partitions and apply DP, rather than partitioning the model to fit the number of GPUs. Increasing the number of parallel pipelines will reduce the overhead of transmitting intermediate activations across GPUs.

4. Evaluation

We compare TSPipe with prior work in KD and SSL models with momentum networks. We summarize our findings:

- TSPipe boosts up training throughput up to **12.2x** compared to inter-layer MP and **1.88x** compared to GPipe, achieving near-ideal performance that we expect from our design.
- TSPipe preserves the final model accuracy of the original training semantics. Meanwhile, applying the conventional strategy for parameter updates significantly degrades the accuracy (up to -5.8%p).

Implementation. We implement TSPipe on PyTorch (Paszke et al., 2019). Multiple GPUs cannot be fully utilized with multi-threaded design on PyTorch due to the Python global interpreter lock (Beazley, 2010). Thus, we use a multi-process design where we implement CPU-CPU communication with PyTorch RPC and GPU-GPU communication via NCCL (NVIDIA, 2021). We implement communication and computation overlapping to hide data transfer latency and improve throughput.

Baselines. We compare TSPipe with inter-layer MP and GPipe (Huang et al., 2019). For fair comparisons, we apply the same model partitioning and configurations. We also implement an extended version of inter-layer MP and GPipe that support gradient accumulation. Accordingly, the same architectures and network configurations are applied.

Models. We evaluate soft target (Hinton et al., 2015) and DistilBERT (Sanh et al., 2019) for KD, and BYOL (Grill et al., 2020) and MoCo-v3 (Chen et al., 2021) for SSL models. During the self-supervised training (pre-training) of SSL models, we use the same configurations from the original papers except for the batch size.

For soft target, we use ViT (Large and Huge) (Dosovitskiy et al., 2020) as the backbone architecture for teacher networks, and ResNet-101, Resnet-152 (He et al., 2016) as the backbone architecture for student networks. For DistilBERT, we use BERT-xlarge and BERT-xxlarge (Shoeybi et al., 2019) as the backbone architecture for its teacher networks. Corresponding to the teacher network architecture, we resized the student model to DistilBERT-xlarge and DistilBERT-xxlarge.

For BYOL, we use four different sizes of ResNet (He et al., 2016) as its backbone architecture. LARS (You et al., 2017) optimizer is used with base learning rate of $lr = 0.2$ which linearly scales w.r.t the batch size($lr \times (\text{BatchSize})/256$) (Goyal et al., 2017). We apply a cosine-annealing learning rate scheduling (Loshchilov & Hutter, 2016) with weight decay of 1.5×10^{-6}. On momentum constant τ, cosine-annealing was applied starting from $\tau = 0.996$ to 1. We train for 200 epochs each with 10 warm-up epochs.

For MoCo-v3, we use ViT (Small, Base, Large, and Huge) (Dosovitskiy et al., 2020) as its backbone architecture. Following (Chen et al., 2021), AdamW (Loshchilov & Hutter, 2017) optimizer is used with linearly scaled learning rate, $lr = 1.5 \times 10^{-4}$. We apply weight decay of 0.1 and momentum of 0.99 with cosine-annealing and train for 100 epochs with 10 warm-up epochs.

Setup. We follow the training procedure described in Grill et al. (2020); Chen et al. (2021). Given an image, we apply SimCLR (Chen et al., 2020)'s image augmentation. Then, two backbone architectures are trained to learn good image representations with different views of the image. After training, we extract the representations by removing the final MLP layers and attaching a linear classifier with the size of 4096 hidden dimensions and an output dimension of 256 (512 and 128 for ResNet-18-based models). We utilize the linear evaluation protocol described in Grill et al. (2020); Chen et al. (2020); Oord et al. (2018). We evalu-

				Training Throughput (Seq/s)		
Method		Architecture	Param.	Inter-layer MP	GPipe	**TSPipe** (Ours)
KD	Soft Target (Hinton et al., 2015)	ViT-Large / ResNet-101	303 M / 43 M	57.41	136.8	204.4 **(3.56x)**
		ViT-Large / ResNet-152	303 M / 58 M	47.24	126.6	180.7 **(3.82x)**
		ViT-Huge / ResNet-101	631 M / 43 M	35.65	100.6	148.5 **(4.17x)**
		ViT-Huge / ResNet-152	631 M / 58 M	30.30	84.03	141.8 **(4.68x)**
	DistillBERT (Sanh et al., 2019)	BERT-xlarge	1.3 B / 480 M	62.82	113.3	193.4 **(2.00x)**
		BERT-xxlarge	3.9 B / 1.2 B	30.36	75.22	98.82 **(3.25x)**
SSL	BYOL (Grill et al., 2020)	ResNet-18	11 M	346.3	585.1	728.5 **(2.10x)**
		ResNet-50	26 M	102.0	232.0	295.8 **(2.90x)**
		ResNet-101	45 M	71.25	162.7	243.0 **(3.41x)**
		ResNet-152	60 M	53.33	136.9	201.6 **(3.78x)**
	MoCo-v3 (Chen et al., 2021)	ViT-Small	22 M	99.42	259.9	365.7 **(3.68x)**
		ViT-Base	86 M	35.06	106.7	176.6 **(5.04x)**
		ViT-Large	307 M	11.31	33.95	54.70 **(4.84x)**
		ViT-Huge	632 M	5.496	18.71	35.26 **(6.42x)**

Table 2. Training throughput (seq/s) evaluated on various architectures using TSPipe with 8 GPUs. Numbers in parenthesis show improvement over inter-layer MP. Note that there exists speedup with respect to the size of the models (# of parameters).

Method	MP	GPipe	**TSPipe**
Soft Target (ViT-Huge / ResNet-152)	26.5	136	191 **(7.21x)**
MoCo-v3 (ViT-Huge)	5.90	40.0	71.7 **(12.15x)**

Table 3. Training throughput (seq/s) evaluated on KD (Soft Target (Hinton et al., 2015)) and SSL (MoCo-v3 (Chen et al., 2021)) using TSPipe with 16 GPUs. Numbers in parenthesis show improvement over inter-layer MP.

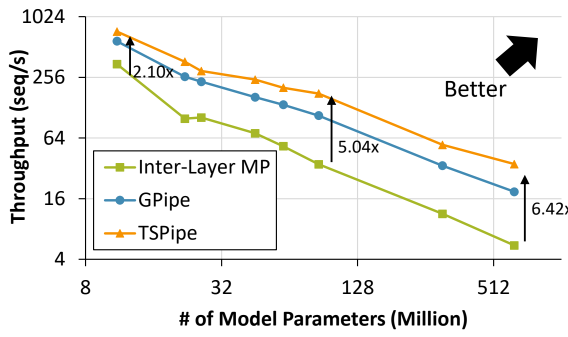

Figure 3. Training throughput vs. # of Model Parameters (Million). TSPipe delivers the best training throughput, up to 6.42x better than the inter-layer MP. TSPipe shows the greatest performance increase with a larger model size.

ate TSPipe on a DGX-1 machine, which features 8 V100 GPUs (32GB memory) with 2 NVLink connecting adjacent GPUs (NVIDIA, 2017). To further evaluate TSPipe with 16 GPUs, we use two Azure ND40rsv2 VMs (8 V100 GPUs each) with GPUDirect RDMA for faster inter-node communication.

4.1. Throughput Analysis

We evaluate the training throughput of KD (soft target (Hinton et al., 2015) and DistilBERT (Sanh et al., 2019)) and SSL (BYOL (Grill et al., 2020) and MoCo-v3 (Chen et al., 2021)) models. To avoid out-of-memory, we vary the batch size between 128 and 2048.

Throughput gain over baselines. Tables 2 and 3 and Figure 3 illustrate the training throughput according to models and their architectures. With 8 GPUs, TSPipe achieves improvement in training throughput up to 6.42x compared to inter-layer MP and 1.88x compared to GPipe. With 16 GPUs, TSPipe achieves even greater speedup reaching 12.15x compared to inter-layer MP. TSPipe shows the best performance improvement in MoCo-v3 with ViT-Huge backbone, while evaluation with BYOL shows relatively low throughput gain. Figure 3 shows that TSPipe tends to get more performance improvement with larger-sized models. Such tendency results from higher utilization of internal computing resources in GPUs (CUDA cores) when larger tensors are computed. ViT-Huge is literally a "huge" model with 632M parameters, so it benefits the most from TSPipe design with full pipelines. Compared to GPipe, we achieve performance gain up to 1.88x with 8 GPUs, which is close to the estimated ideal performance gain of our design (§3.1).

4.2. Accuracy Analysis

According to the common practice, we evaluate the linear classification accuracy over frozen representations to evaluate the performance of self-supervised learning models.

Dataset	Vanilla Top1	Vanilla Top5	TSPipe Top1	TSPipe Top5
STL10 (Coates et al., 2011)	81.73 ±0.27	99.41 ±0.06	81.75 ±0.32 (+0.02)	99.40 ±0.03 (-0.01)
CIFAR10 (Krizhevsky et al., 2009)	74.76 ±0.34	98.60 ±0.08	75.24 ±0.52 (+0.48)	98.73 ±0.09 (+0.13)
CIFAR100 (Krizhevsky et al., 2009)	48.54 ±0.34	78.46 ±0.16	49.79 ±0.32 (+1.25)	79.22 ±0.50 (+0.76)
ImageNet100 (Russakovsky et al., 2015)	64.18 ±0.61	88.12 ±0.33	64.24 ±0.23 (+0.06)	88.24 ±0.22 (+0.12)

Table 4. Linear evaluation accuracy (%) of ResNet-18 (He et al., 2016), pre-trained with BYOL (Grill et al., 2020) for 200 epochs. Numbers in parenthesis indicate the gain over the vanilla scheme. TSPipe achieves almost identical accuracy as the vanilla scheme.

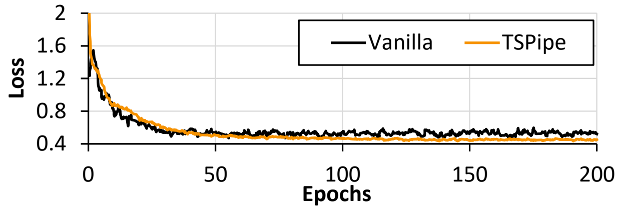

Figure 4. Training loss curve of TSPipe and vanilla during the pre-training of BYOL (Grill et al., 2020). Trained with ResNet-50 (He et al., 2016) for 200 epochs, batch size of 1024 and LARS optimizer (You et al., 2017) are used.

Method	Vanilla	TSPipe	TSPipe without ASP
STL10	81.7%	81.7%	79.1% **(-2.69%p)**
ImageNet100	64.2%	64.2%	58.3% **(-5.89%p)**

Table 5. Ablation study on asymmetric parameter update (§3.2). Linear evaluation accuracy (%) of ResNet-18 (He et al., 2016), pre-trained on BYOL (Grill et al., 2020) with vanilla, TSPipe, and TSPipe without asymmetric parameter update(ASP).

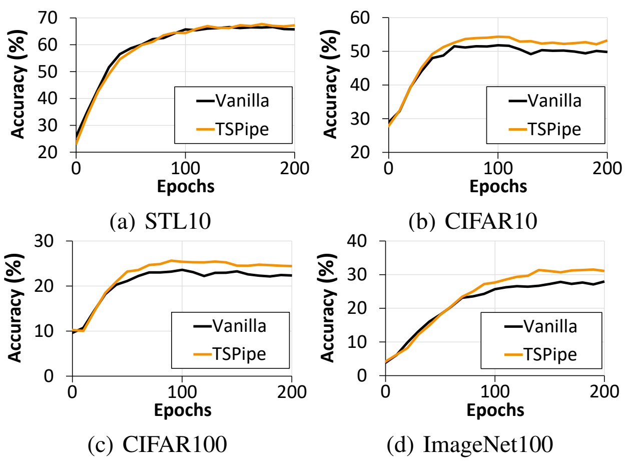

Figure 5. Validation accuracy of the k-NN classifier (Wu et al., 2018) on various datasets during the pre-training of BYOL (Grill et al., 2020) with ResNet-18 (He et al., 2016)

Since inter-layer MP and GPipe share the same training semantics with vanilla training scheme, we only compare the accuracy of TSPipe with the vanilla training scheme in this section.

Linear classification accuracy. After training the BYOL model with ResNet-18 architecture over 200 epochs using ImageNet100 (pre-training), we further train the linear classifier over 90 epochs to evaluate the test accuracy. Unlike pre-training, the linear classifier was trained on a single GPU. We use SGD with momentum and a linearly scaled learning rate. We utilize four image datasets, STL10 (Coates et al., 2011), CIFAR10/CIFAR100 (Krizhevsky et al., 2009), and ImageNet100 (Russakovsky et al., 2015). We report the average accuracy and the standard deviations of five runs with random seeds.

As shown in Table 4, TSPipe shows almost identical test accuracy compared to the vanilla training schemes. The accuracy differences between vanilla and TSPipe are very marginal, where the accuracy of TSPipe is better than the vanilla scheme for some datasets, such as STL10 and Imagenet100.

k-NN classifier accuracy. We also show the validation accuracy of a k-nearest-neighbor (k-NN) classifier (Wu et al., 2018), with which we can monitor the performance of the model as the pre-training progresses. Figure 5 (a) to (d) shows the validation accuracy of the k-NN classifier evaluated using four different datasets (STL10, CIFAR10, CIFAR100, and ImageNet100), during the pre-training of BYOL with ResNet-18. It shows that TSPipe achieves nearly identical or better accuracy than the vanilla during the entire process of pre-training.

Loss curve. In Figure 4, we train BYOL under ResNet-50 architecture and the batch size of 1024 using TSPipe and vanilla (inter-layer MP). We show the loss curve during the 200 epochs of pre-training. TSPipe exhibits an almost similar loss curve to that of the vanilla training schemes. Actually, the curve of TSPipe is even more stable since the asymmetric parameter update of TSPipe contributes to stabilizing the training. We think this is another strong property of TSPipe as self-supervised training schemes are often highly unstable (Chen et al., 2021).

Asymmetric parameter update ablation study. We evaluate whether limiting the early computation only to the teacher network helps preserve the accuracy. More specifically, we evaluate if TSPipe can preserve its accuracy without incorporating the asymmetric parameter update (§3.2). The conventional strategy for parameter update used in previous works (Narayanan et al., 2021; Ren et al., 2021) (Equation (3)) does not differentiate between the student network and the teacher network, making both networks stale. However, TSPipe's strategy for parameter update (Equation (2)) clearly differentiates the two networks, and TSPipe only make the teacher network stale. We compare the result from the conventional and TSPipe parameter update strategies. We observe that under the conventional strategy, the linear evaluation accuracy significantly drops up to -5.9%p (Table 5), i.e., the update of the student network is unstable, and the difference between θ_n and θ_{n-1} is not negligible (θ_n is the parameters of the student network at n-th iteration).

5. Conclusion

TSPipe presents a framework that enables faster training of large models with the TS framework without risking any performance degradation of the model. We demonstrate it is possible to utilize 100% of GPU pipelines for training KD and SSL with momentum networks, as the teacher network does not need a backward pass. We show TSPipe can mitigate potential model accuracy degradation coming from delayed parameter update, using that the parameters of the teacher network are updated in a steady and stable manner. TSPipe exhibits better performance than any other pipeline parallelism schemes, providing near-optimal training throughput without sacrificing the model accuracy. TSPipe achieves up to 12.15x higher training throughput than inter-layer model parallelism, while preserving the model accuracy.

Acknowledgements

We appreciate Jinwoo Park and anonymous reviewers for providing helpful feedbacks and suggestions to improve our paper. This work was mainly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00117, Development of low-power computational storage system technology for low-latency large-scale data processing; No.2019-0-00075, Artificial Intelligence Graduate School Program (KAIST)).

References

Beazley, D. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia*, 2010.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020.

Chen, X., Xie, S., and He, K. An empirical study of training self-supervised visual transformers. *arXiv e-prints*, pp. arXiv–2104, 2021.

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 571–582, 2014.

Coates, A., Ng, A., and Lee, H. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 215–223. JMLR Workshop and Conference Proceedings, 2011.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Grill, J.-B., Strub, F., Altché, F., Tallec, C., Richemond, P. H., Buchatskaya, E., Doersch, C., Pires, B. A., Guo, Z. D., Azar, M. G., et al. Bootstrap your own latent: A new approach to self-supervised learning. *arXiv preprint arXiv:2006.07733*, 2020.

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

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9729–9738, 2020.

Hinton, G., Vinyals, O., Dean, J., et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images, 2009.

Li, C., Yang, J., Zhang, P., Gao, M., Xiao, B., Dai, X., Yuan, L., and Gao, J. Efficient self-supervised vision transformers for representation learning. *arXiv preprint arXiv:2106.09785*, 2021a.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. *arXiv preprint arXiv:2102.07988*, 2021b.

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

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021.

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

NVIDIA. Nvidia DGX-1 with tesla v100 system architecture white paper, 2017. URL `https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf`.

NVIDIA. Nvidia collective communications library (nccl), Dec 2021. URL `https://developer.nvidia.com/nccl`.

Oord, A. v. d., Li, Y., and Vinyals, O. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Pham, H., Dai, Z., Xie, Q., and Le, Q. V. Meta pseudo labels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11557–11568, 2021.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

Roh, B., Shin, W., Kim, I., and Kim, S. Spatially consistent representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1144–1153, 2021.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3): 211–252, 2015.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

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

Tarvainen, A. and Valpola, H. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. *arXiv preprint arXiv:1703.01780*, 2017.

Wu, Z., Xiong, Y., Yu, S. X., and Lin, D. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3733–3742, 2018.

Xu, A., Huo, Z., and Huang, H. On the acceleration of deep learning model parallelism with staleness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2088–2097, 2020.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C., and De Sa, C. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3, 2021.

Yin, H. and Pan, S. J. Knowledge transfer for deep reinforcement learning with hierarchical experience replay. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhou, J., Wei, C., Wang, H., Shen, W., Xie, C., Yuille, A., and Kong, T. ibot: Image bert pre-training with online tokenizer. *arXiv preprint arXiv:2111.07832*, 2021.