SLIMFIT: Memory-Efficient Fine-Tuning of Transformer-based Models Using Training Dynamics

Arash Ardakani[1] **Altan Haan**[1] **Shangyin Tan**[1] **Doru Thom Popovici**[2]
Alvin Cheung[1] **Costin Iancu**[2] **Koushik Sen**[1]
University of California, Berkeley[1] Lawrence Berkeley National Laboratory[2]
{arash.ardakani,altanh,shangyin,akcheung,ksen}@berkeley.edu
{dtpopovici,cciancu}@lbl.gov

Abstract

Transformer-based models, such as BERT and ViT, have achieved state-of-the-art results across different natural language processing (NLP) and computer vision (CV) tasks. However, these models are extremely memory intensive during their fine-tuning process, making them difficult to deploy on GPUs with limited memory resources. To address this issue, we introduce a new tool called SLIMFIT that reduces the memory requirements of these models by dynamically analyzing their training dynamics and freezing less-contributory layers during fine-tuning. The layers to freeze are chosen using a runtime inter-layer scheduling algorithm. This allows SLIMFIT to freeze up to 95% of layers and reduce the overall on-device GPU memory usage of transformer-based models such as ViT and BERT by an average of 2.2×, across different NLP and CV benchmarks/datasets such as GLUE, SQuAD 2.0, CIFAR-10, CIFAR-100 and ImageNet with an average degradation of 0.2% in accuracy. For such NLP and CV tasks, SLIMFIT can reduce up to 3.1× the total on-device memory usage with an accuracy degradation of only up to 0.4%. As a result, while fine-tuning of ViT on ImageNet and BERT on SQuAD 2.0 with a batch size of 128 requires 3 and 2 32GB GPUs, respectively, SLIMFIT enables fine-tuning them on a single 32GB GPU without any significant accuracy degradation. The code of SLIMFIT is available at https://github.com/arashardakani/SlimFit.

1 Introduction

Over the past few years, various transformer-based models have been developed with the adoption of the attention mechanism that weighs the importance of each part of the input data differently. Pre-training of such transformer-based models on large data has led to a significant boost in accuracy when fine-tuned on various natural language processing (NLP) and computer vision (CV) downstream tasks (Devlin et al., 2018; Dosovitskiy et al., 2021). Despite their great performance in achieving state-of-the-art (SOTA) accuracy, these models are memory intensive and require a considerably large amount of on-device GPU memory during their fine-tuning phase when compared to the conventional convolutional and recurrent neural networks (Jain et al., 2020). The memory requirement of current transformer-based models has made them difficult to fine-tune even on powerful GPUs. With the introduction of larger transformer-based models over the past few years, the on-device GPU memory has become a major bottleneck for their fine-tuning process (Jain et al., 2020; Liu et al., 2022; Chen et al., 2023).

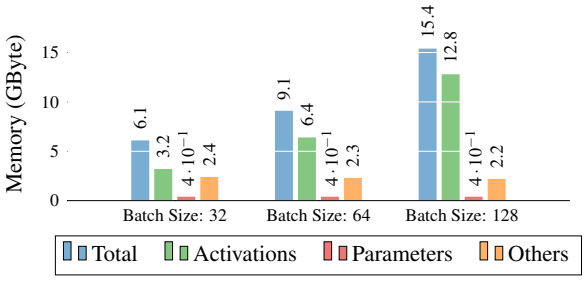


Figure 1: The breakdown of memory usage of BERT when fine-tuned on different batch sizes including 32, 64, and 128.

The total on-device memory usage of GPUs consists primarily of activations, parameters, gradients, optimizer states, and the CUDA context. Among these factors, activations account for most of the memory usage due to batch processing (Liu et al., 2022; Chen et al., 2021; Jain et al., 2020) as shown in Fig. 1. Therefore, activation compressed training (ACT) has emerged as the primary solution for memory-efficient fine-tuning (Chen et al., 2021; Liu et al., 2022). This approach first compresses activations during the forward pass and then decompresses them during the backward pass. In this way, the memory footprint can be significantly reduced by caching the compressed activations. In ACT, quantization (Chakrabarti and Moseley, 2019; Fu et al., 2020; Chen et al., 2021; Liu et al., 2022) has been a popular choice to compress activations among other compressors such as JPEG (Evans et al., 2020) or pruning (Chen et al., 2023). The current SOTA ACT

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 6218–6236
June 16-21, 2024

adaptively assigns quantization bits to each layer for a given architecture (Liu et al., 2022). While the SOTA ACT successfully reduces the memory footprint of activations, its overall on-device GPU memory reduction is not significant. For instance, the total on-device GPU memory reduction of the SOTA ACT is limited to 0.1GB despite its 6.4× reduction in the memory of activations when fine-tuning BERT on CoLA dataset with a batch size of 32. It is worth mentioning that we refer to the memory usage reported by "nvidia-smi" as the overall on-device memory in this paper (see Appendix A for more information on memory management).

Tensor rematerialization (Jain et al., 2020; Chen et al., 2016; Beaumont et al., 2021; Kirisame et al., 2021), also known as gradient checkpointing, is another prominent approach to reducing activation memory by trading computations for memory. In tensor rematerialization, only specific activations are stored during the forward pass, while the rest are recomputed in the backward pass. Of course, recomputing activations requires more operations, resulting in a longer fine-tuning process (Liu et al., 2022). Reduced precision training, as another approach, performs the computations of both forward and backward passes in low-precision (Micikevicius et al., 2017; Wu et al., 2018; Wang et al., 2018b; Banner et al., 2018). While these works can successfully train conventional models, few-bit model fine-tuning is not trivial. For instance, 8-bit quantization of BERT for inference results in a significant precision loss (Zafrir et al., 2019), which makes fine-tuning on few bits a challenging task.

Low-rank adaptation (LoRA) (Hu et al., 2022) is another key approach to reducing the overall on-device GPU memory where the transformer-based models are fine-tuned by inserting a small number of trainable parameters into each layer while keeping the pre-trained model parameters frozen. Such an approach enables fine-tuning transformer-based models with significantly less number of trainable parameters, leading to a reduction in the memory footprint of optimizer states and gradients. Such a memory reduction becomes significant for large transformer models such as GPT (Brown et al., 2020) with billions of parameters.

Different from these methods, we put forward a new approach to reducing the overall on-device memory usage by analyzing training dynamics. More precisely, we dynamically analyze the gradient contributions of layers in transformer-based models and perform parameter updates for specific layers only while the rest of layers are kept frozen. Training dynamics have been used to analyze the behavior of a model during its training/fine-tuning process (Swayamdipta et al., 2020; Teehan et al., 2022; Fang et al., 2022). However, our work uses training dynamics to detect and discard unimportant activations during fine-tuning by freezing their associated layers, leading to a reduction of the memory footprint. Our method is orthogonal to existing approaches including rematerialization, LoRA and fused operations (Dao et al., 2022; Rabe and Staats, 2021), which could be combined for further reductions.

Freezing layers or parameters has been studied in different domains, including transformer-based models to preserve previously learned information during fine-tuning (Lee et al., 2022; Shen et al., 2021). Freezing parameters have also been used to regularize fine-tuning (e.g., over-fitting reduction) in pre-trained models (Ramasesh et al., 2021). Recently, freezing has been used to accelerate fine-tuning by progressively freezing model blocks (Liu et al., 2021; Li et al., 2023; He et al., 2021; Yuan et al., 2022). However, since such an approach starts the fine-tuning process without freezing at least for a few training iterations/epochs, its overall on-device memory requirement remains similar to that of training without freezing. For instance, fine-tuning ViT on ImageNet with a batch size of 128 using such a freezing approach on a single 32GB GPU results in an out-of-memory error (see Appendix B for more details).

To orchestrate effective layer-freezing decisions, we introduce a runtime inter-layer scheduling (ILS) algorithm. Our method finds and freezes a set of layers at each training iteration in transformer-based models that are less contributory, i.e., layers with fewer updates in their parameters, to the fine-tuning process at each iteration. While the ILS algorithm successfully detects and freezes unimportant layers, its memory reduction is not proportional to the freezing rate. The reason behind this disproportionality is twofold: the imbalanced number of activations among layers and the existence of static activations. Static activations refer to those that cannot be discarded regardless of freezing (e.g., activations of non-linear functions such as GELU). We address these two issues using quantization and pruning to even out the number of activations across all layers and to reduce the memory overhead of static activations. We use quantization and pruning for a few specific layers of transformer-based models as opposed to reduced precision training methods where all the layers are quantized. As a result, the

impact of quantization and pruning on accuracy is insignificant in our work. For instance, the accuracy degradation due to quantization and pruning is only 0.1% on the MRPC dataset.

By combining ILS with quantization and pruning, we introduce a performance tool called SLIMFIT for reducing the on-device GPU memory usage of transformer-based models during fine-tuning. We demonstrate the effectiveness of SLIMFIT in reducing the memory footprint on popular models of BERT and ViT. We show that SLIMFIT can freeze up to 95% of layers and reduce the overall on-device memory usage by an average of 2.2× when fine-tuning BERT and ViT models on different benchmarks and datasets, such as GLUE, SQuAD 2.0, CIFAR-10, CIFAR-100 and ImageNet with an average accuracy degradation of 0.2%. More precisely, SLIMFIT reduces the overall on-device memory usage of the fine-tuning process on GLUE from 6.1GB to 4.0GB (1.5× reduction) with a batch size of 32, on SQuAD 2.0 from 58.5GB to 19.1GB (3.1× reduction) with a batch size of 128, on CIFAR-10 from 7.2GB to 4.3GB (1.7× reduction) with a batch size of 32, on CIFAR-100 from 7.2GB to 4.5GB (1.6× reduction) with a batch size of 32, and on ImageNet from 77.4GB to 26.1GB (3.0×) with a batch size of 128 at the cost of up to 0.4% accuracy degradation. As a result, SLIMFIT enables performing memory-intensive fine-tuning processes on a single 32GB GPU such as fine-tuning ViT on ImageNet with a batch size of 128 while this normally requires three 32GB GPUs.

2 Preliminaries

Over the past few years, pre-training of attention-based models has led to significant advances on many NLP and CV tasks with the popular BERT (Devlin et al., 2018) and ViT (Dosovitskiy et al., 2021) models. The pre-training process provides a good initialization point such that these models can better generalize on unseen data of downstream tasks. Therefore, these models can achieve state-of-the-art results by fine-tuning through small adjustments to their parameters. Architecturally, these models consist of an initial embedding layer, followed by repeated blocks of multi-head attention (MHA) fed into a feed-forward network (FFN) module (see Appendix C for more details). The base architectures of BERT and ViT contain over a hundred layers built up in this manner.

Despite the large number of layers, not all need to be updated during fine-tuning to achieve decent performance on downstream tasks, as shown in (Merchant et al., 2020). Notably, the authors found that freezing approximately 60% of early attention layers in BERT led to negligible performance degradation. This suggests that the fine-tuned model tends to preserve generic features learned during pre-training. Motivated by this study, we seek to analyze the training dynamics of pre-trained models and to automatically detect layers with less contributions to the fine-tuning process.

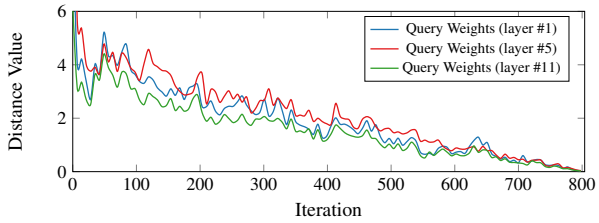
3 Learning the Importance of Layers

Training dynamics is an active field of research that provides insight about the behavior of pre-trained models when fine-tuning on downstream tasks. The convergence proof of optimization algorithms such as stochastic gradient descent (Shalev-Shwartz and Ben-David, 2014) shows that the distance between the parameters and the optimal solution is reduced over training iterations and accordingly, the weight distance (or the weight update amount) between consecutive iterations decreases. Therefore, it is possible that some layers can only receive minimal changes to their parameters as we approach the end of the training process. Of course, detecting and freezing such layers, when they show minimal updates, will not affect accuracy. Since transformer-based models are pre-trained, they already show small updates during fine-tuning compared to pre-training. As such, detecting and freezing layers with minimal updates (i.e., weight distance values) will not significantly affect the fine-tuning process and accordingly the final accuracy. Based on the above observations, we consider the ℓ_1-norm of the update received by parameters of each layer through all the fine-tuning iterations as the training dynamics in this paper. It is also worth mentioning that freezing layers has no impact on training convergence as it causes a pause in the training procedure of frozen layers as shown by our theoretical analysis in Appendix D.1.

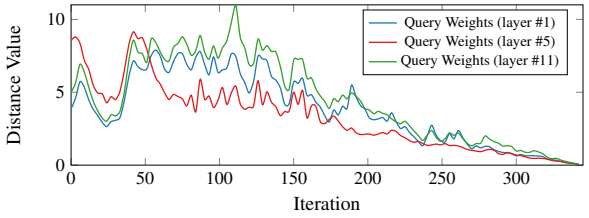
3.1 Training Dynamics

Let us consider a pre-trained model with a set of parameters $\mathbf{W}$ where the parameters associated with the ith layer at iteration t is denoted as $\mathbf{W}_i^t \in \mathbb{R}^{M \times I}$. The training dynamics of for the ith layer at iteration t is defined as the ℓ_1-norm of the distance between $\mathbf{W}_i^{t-1}$ and $\mathbf{W}_i^t$, i.e.,

$$d_i^t = \frac{1}{M \times I} \left\| \frac{W_i^t - W_i^{t-1}}{W_i^{t-1}} \right\|_{\ell_1}, \quad (1)$$



(a) CoLA



(b) MRPC

Figure 2: The distance values of query weight matrix for the first, fifth and eleventh attention layers of BERT-base fine-tuned on (a) CoLA and (b) MRPC datasets for 3 epochs.

where $\mathbf{d}^t \in \mathbb{R}_+^n$ containing all d_is at iteration t is referred to as distance vector, and n denotes the total number of layers. In fact, Eq. (1) calculates the normalized change in the parameters of the ith layer.

3.2 Inter-Layer Scheduling Algorithm

We use the distance values as training dynamics to analyze the fine-tuning behavior of pre-trained models. For instance, consider the distance values across all the fine-tuning iterations for the CoLA (Warstadt et al., 2018) and MRPC (Wang et al., 2018a) datasets. Fig. 2a shows the distance values of the query weight matrix for the first, fifth and eleventh attention layers of BERT-base fine-tuned on CoLA dataset whereas Fig. 2b depicts those of the same layers for BERT-based fine-tuned on MRPC dataset.

We observe the following based on the experimental results of these two datasets. First, the updated amount for each layer becomes smaller over fine-tuning iterations. Second, the updated amount of each layer is task-specific and is independent of its position. Third, there are some layers showing smaller distance values w.r.t. other layers across almost all the iterations. Finally, layers with a higher distance value in the beginning can become smaller over the fine-tuning iterations than layers starting with a lower distance value.

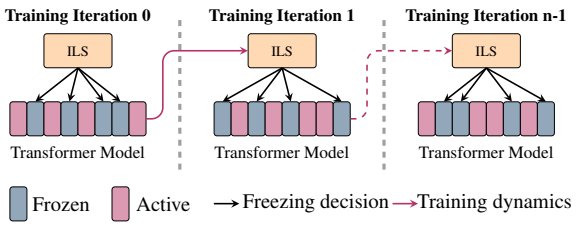


Figure 3: The overview of ILS algorithm. ILS freezes a certain number of layers depending on the freezing rate at every single iteration throughout the fine-tuning process for the total of n training iterations.

Given the above observations, we introduce an ILS algorithm to decide on updating priority of layers using their distance values. Fig. 3 shows an overview of the ILS algorithm. At each iteration ranging from the first iteration to the last iteration, our ILS algorithm selects those layers with large distance values to be updated and those with small distance values to be frozen. More precisely, layers are first ranked based on their distance values at each training iteration and then those with small distance values are kept frozen according to the freezing rate as a hyper-parameter. The intuition is that layers with small distance values are less contributory to the fine-tuning process as their parameters are not being updated much. On the other hand, the layers with large distance values are learning task-specific patterns by making more significant adjustments to their parameters. Note that freezing middle layers does not interrupt the gradient propagation to the early layers of the network as shown through an example in Appendix D.2.

The freezing rate of the ILS algorithm can be decided based on the on-device GPU memory budget. Of course, using an extremely high freezing rate may result in a performance degradation depending on the downstream task, providing a worthwhile trade-off between accuracy and on-device GPU memory. On the other hand, while performance degradation is unlikely with a very small freezing rate, the memory reduction is insignificant as well.

Since there is no prior knowledge about the distance values of each layer at the beginning of the fine-tuning process, our ILS algorithm initializes the distance vector with large random values. Depending on the freezing rate, each layer along with its distance value are updated during the first few iterations once until all random numbers in the distance vector are substituted with an actual distance value. Afterwards, layers are kept frozen according to their actual distance value. The distance value of the active layers is only updated at each iteration while that of the frozen layers remains unchanged. The pseudo code of our ILS algorithm performing iterative freezing is shown in Algorithm 1.

To better understand the ILS algorithm, we illustrate the iterative freezing process using an example as shown in Fig. 4a. Suppose we have an 8-layer transformer-based model and accordingly an

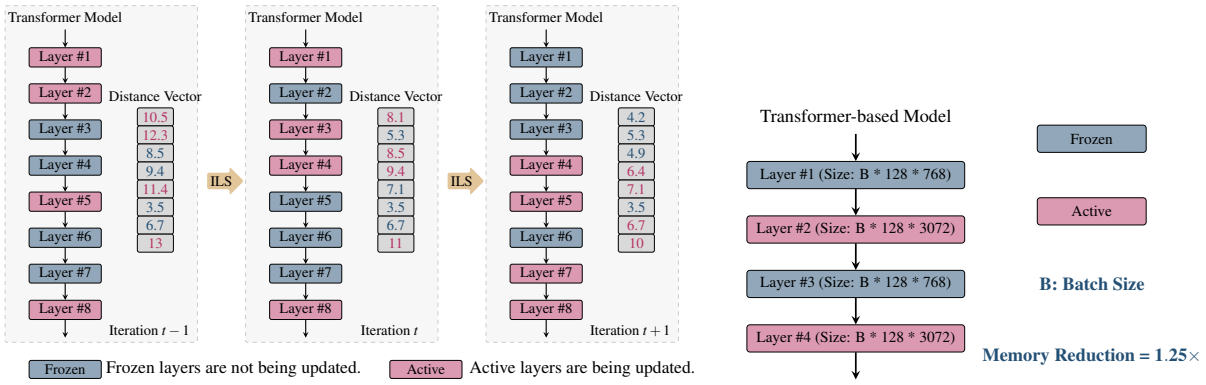


Figure 4: (a) An example of the iterative freezing process using our ILS algorithm. (b) An example of a model with imbalanced number of activations and its impact on the memory reduction.

Algorithm 1 The pseudo code of the ILS algorithm performing iterative freezing.

Input: model, number of iterations as itr, number of layers as L, freezing rate F
d = rand(L)
for i **in** itr **do**
  idx = argsort(d)[:int(L*F)]
  for j **in** idx **do**
    model.layer[j].requires_grad = False
  end for
  model.train()
  Update d
end for

8-element distance vector at iteration t. Considering the freezing rate of 50% for this example, 4 layers with the lowest distance values are kept frozen and the rest are updated at each iteration.

4 Inter-Layer Load-Balancing

So far, we have introduced our ILS algorithm that prioritizes updating particular layers while keeping the rest of layers frozen according to their distance value. For the given freezing rate of 50% as an example, we expect to see a 2× reduction in the memory footprint of activations. However, this is not the case in transformer-based models due to the imbalanced the number of activations across all the layers. In fact, the imbalance in the number of activations undermines the ability of our ILS algorithm in reducing the memory footprint during the fine-tuning as shown in Fig. 4b.

Since the focus of this paper is on transformer-based models such as BERT and ViT, we analyze their architecture for imbalanced layers. Table 1 summarizes the number of activations associated to the input of layers with trainable parameters in BERT or ViT. Among all trainable layers, there is only one imbalanced layer in the attention block which contains 4× more activations than other layers.

Table 1: The number of activations associated to the input of layers with trainable parameters in BERT where B, T, H denote the batch size, sequence length, hidden size, respectively. ViT has the same structure with different descriptions.

Type of Layer	Description	# Activations	Status
Dense	attention.self.query	$B*T*H$	Balance
Dense	attention.self.key	$B*T*H$	Balance
Dense	attention.self.value	$B*T*H$	Balance
Dense	attention.output	$B*T*H$	Balance
LayerNorm	attention.output	$B*T*H$	Balance
Dense	intermediate	$B*T*H$	Balance
Dense	**output**	$\mathbf{B*T*4*H}$	**Imbalance**
LayerNorm	output	$B*T*H$	Balance

To address the load-balancing issue in the number of activations for the aforementioned layer, we use quantization. Since the imbalance factor among layers is 4×, we adopt 8-bit quantization for activations of the imbalanced layer where 4 bits are used for both the integer and fractional parts. In this way, the memory cost of the activations are evened out using quantization. In our quantization scheme, we cache the activations of the imbalanced layer using 8 bits during the forward pass. In the backward pass, we convert the 8-bit activations to 32-bit floating-point format. Therefore, all the forward and backward computations are still performed using single-precision floating-point format. The conversion process between 8-bit fixed-point and 32-bit floating-point formats are provided in Appendix E.

5 Dynamic and Static Activations

Assuming that the backpropagation is performed from the last to the first layer, the type of activations in transformer-based models can be divided into two

categories: dynamic and static. We refer to the activations that can be discarded by freezing their layer as dynamic activations. On the other hand, static activations cannot be discarded regardless of freezing. Among different types of layers, GELU, MatMul, Softmax and LayerNorm contain static activations as shown Table 2. Note that MatMul and Softmax share the same activations. For the backward computations of Softmax, its output during the forward pass is saved as its activations. On the other hand, the input to MatMul is required for its backward computations as activations. Since the output of Softmax is an input to MatMul in the forward pass, they share the same activations.

GELU and MatMul/Softmax do not have any trainable parameters and accordingly cannot be frozen. Therefore, these two layers hold on to their activations throughout the fine-tuning process. The best approach to reduce their memory cost is quantization. We use 4 and 8 bits for quantization of activations in GELU and MatMul/Softmax, respectively. Since there is no 4-bit tensor support in PyTorch, we store each two 4-bit activations as a single 8-bit activations using shift operations. Note that using such bit-levels result in a negligible accuracy degradation while further quantization of those activations incurs a significant accuracy loss.

As opposed to GELU and MatMul/Softmax, LayerNorm contains trainable parameters and can be frozen by the ILS algorithm. However, its activations are still static. The forward pass of LayerNorm is computed by:

$$\widetilde{\mathbf{x}} = \frac{\mathbf{x} - \mathbb{E}(\mathbf{x})}{\sqrt{\mathrm{Var}(\mathbf{x}) + \varepsilon}}, \tag{2}$$

$$\mathbf{y} = \widetilde{\mathbf{x}} * \gamma + \beta, \tag{3}$$

where γ and β are trainable parameters. The input and output to LayerNorm are denoted by $\mathbf{x} \in \mathbb{R}^H$ and $\mathbf{y} \in \mathbb{R}^H$, respectively. $\mathbb{E}(\cdot)$ and $\mathrm{Var}(\cdot)$ compute the average and variance, respectively. The derivative of the loss with respect to γ (i.e., $\widehat{\gamma}$) is computed by

$$\widehat{\gamma} = \widetilde{\mathbf{x}} * \widehat{\mathbf{y}}, \tag{4}$$

and with respect to β (i.e., $\widehat{\beta}$) by:

$$\widehat{\beta} = \widehat{\mathbf{y}}, \tag{5}$$

where $\widehat{\mathbf{y}}$ denotes the derivative of the loss w.r.t. y. We also need to compute the derivative of the loss with respect to $\mathbf{x}$ (i.e., $\widehat{\mathbf{x}}$) as:

$$\mathbf{g} = \frac{\gamma * \widehat{\mathbf{y}}}{H * \sqrt{\mathrm{Var}(\mathbf{x}) + \varepsilon}}, \tag{6}$$

$$\widehat{\mathbf{x}} = H * \mathbf{g} - \sum_H \mathbf{g} - \widetilde{\mathbf{x}} * \sum_H (\mathbf{g} * \widetilde{\mathbf{x}}). \tag{7}$$

When LayerNorm is frozen, there is no need to compute Eq. (4). However, the activations of this layer cannot be discarded since they are still a part of the computations in Eq. (7). More precisely, the standardized version of $\mathbf{x}$ (i.e., $\widetilde{\mathbf{x}}$) is required even when this layer is frozen.

The contribution of the last term in Eq. (7) (i.e., $\sum_H (\mathbf{g} * \widetilde{\mathbf{x}})$) is significant for large values of $\widetilde{\mathbf{x}}$ only. Therefore, the small values of $\widetilde{\mathbf{x}}$ can be discarded. Ideally, we want to have all the activations of this layer to be discarded when this layer is frozen. However, this will results in an accuracy degradation. As such, we prune away the small values in $\widetilde{\mathbf{x}}$ and keep the top 10% largest values. In this way, the memory load of activations is significantly reduced. Of course, when this layer is not frozen, the backpropagation is performed without any approximation. Such a trick converts LayerNorm from a static layer to a semi-static one. It is worth mentioning that the indices to pruned activations are also stored along with activations. The details of the pruning procedure is provided in Appendix F.

Table 2: The type of activations of layers in MHA and FFN of BERT and ViT.

Type of Layer	# Activations	Type of Activations
Dense	$B*T*H$	Dynamic
MatMul	$\mathbf{B*T*H}$ **(2×)**	**Static**
Softmax	$\mathbf{B*T*T}$	**Static**
MatMul	$\mathbf{B*T*H}$ & $\mathbf{B*T*T}$	**Static**
Dense	$B*T*H$	Dynamic
LayerNorm	$\mathbf{B*T*H}$	**Static**
Dense	$B*T*H$	Dynamic
GELU	$\mathbf{B*T*4*H}$	**Static**
Dense	$B*T*4*H$	Dynamic
LayerNorm	$\mathbf{B*T*H}$	**Static**

6 SLIMFIT

SLIMFIT is a performance tool that exploits our ILS algorithm along with quantization and pruning to reduce the memory footprint of activations through an iterative freezing process. The total on-device GPU memory reduction of SLIMFIT is a result of the memory reduction in both dynamic and static activations. Static activations contribute a fixed amount of memory whereas the memory usage of dynamic activations depends on the freezing rate. Given a high freezing rate, the memory footprint of activations and accordingly the total on-device GPU memory usage can be significantly reduced. The choice of freezing rate depends on the memory budget of

the user. By increasing the freezing rate up to a certain point, there will be no performance degradation. However, using an extremely high freezing rate trades off memory for accuracy. Finding the breaking point of the method is task dependent and varies from one dataset to another.

7 Experimental Results

We use the base version of BERT and ViT for our experiments. We fine-tune these two models using SLIMFIT which is implemented on PyTorch. We evaluate BERT (Devlin et al., 2018) using the GLUE benchmark (Wang et al., 2018a) and SQuAD 2.0 (Rajpurkar et al., 2016). For ViT (Dosovitskiy et al., 2021), we use CIFAR-10, CIFAR-100 and ImageNet datasets (Krizhevsky, 2009; Deng et al., 2009) for evaluation purposes. We discuss the memory usage of activations and the overall on-device GPU memory on the 32GB NVIDIA V100 GPU. We report the total on-device GPU memory usage using "nvidia-smi". For all the experiments in this section, we use 3 epochs for fine-tuning. The details about the CV/NLP tasks, measurements and hyper-parameter settings are provided in Appendix G.

7.1 Accuracy Evaluation on GLUE and SQuAD 2.0

To evaluate the language understanding ability of BERT models, the GLUE benchmark is formed by a series of downstream tasks including sentiment classification (SST-2), natural language inference (RTE, QNLI, and MNLI), paraphrase detection (MRPC, QQP, and STS-B), and linguistic acceptability (CoLA). We use Spearman correlation for STS-B, Matthew's correlation for CoLA, percentage accuracy for RTE, MRPC, SST-2, QQP, QNLI and MNLI_m, and F1 score for SQuAD 2.0. In this work, we fine-tune the BERT-base model using SLIMFIT on the downstream tasks of the GLUE benchmark as well as the question answering task on SQuAD 2.0. Table 3 shows the accuracy on the validation set of the aforementioned tasks and memory usage of SLIMFIT compared to the baseline. The results of the baseline were obtained without freezing. We report the results along with its statistics associated with the highest freezing rate that can achieve a similar accuracy to that of the baseline by varying the learning rate over 10 random runs. The experimental results on the GLUE benchmark show that up to 95% of dynamic activations can be discarded with up to 0.4% accuracy degradation, leading to an average of 1.9GB reduction in the total on-device GPU memory usage. On the other hand, while fine-tuning SQuAD 2.0 without freezing requires the minimum of 2 32GB NVIDIA V100 GPUs on a batch size of 128, SLIMFIT enables its fine-tuning on a single 32GB NVIDIA V100 GPU, reducing the total on-device memory requirement of such a task from 58.5GB down to 19.1GB (3.1× reduction).

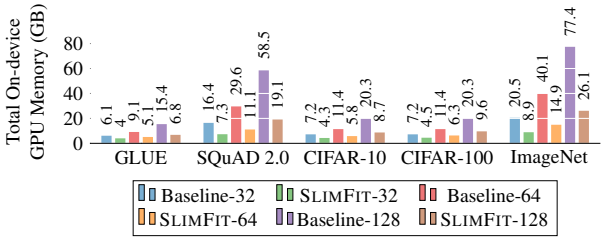


Figure 5: The total on-device GPU memory usage of SLIMFIT compared to the baseline across different batch sizes including 32, 64 and 128 on NLP and CV datasets.

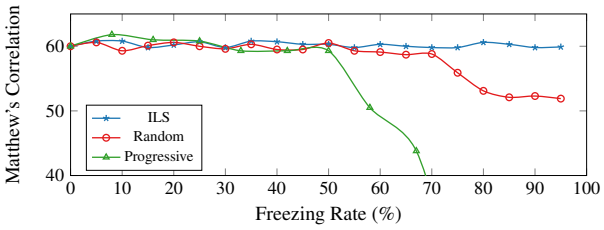
Figure 5 shows the total on-device GPU memory usage of BERT-base when fine-tuned using SLIMFIT for different batch sizes at the freezing rate of 95% on the GLUE benchmark and 80% on SQuAD 2.0. According to the experimental results, SLIMFIT enables a reduction ranging from 1.5× to 3.1× in the total on-device GPU memory on NLP tasks. The reduction in the total on-device memory usage is more significant for larger batch sizes since the activations dominate the memory footprint. It is worth mentioning that the memory benefits of SLIMFIT is not limited to BERT-base for NLP tasks. In fact, similar memory reductions can be obtained when fine-tuning other NLP models such as BERT-large and GPT-2 using SLIMFIT. The experimental results of such models are provided in Appendix H.

7.2 Accuracy Evaluation on CIFAR and ImageNet

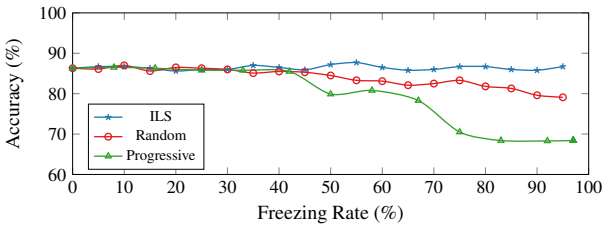
To assess the effectiveness of our method on CV tasks, we fine-tune the ViT-base model on CIFAR-10, CIFAR-100 and ImageNet datasets. We use the test set of CIFAR-10/CIFAR-100 and the validation set of ImageNet to evaluate their accuracy on ViT. Table 3 shows that SLIMFIT can fine-tune the ViT-base model with the freezing rate of up to 95% with up to 0.3% loss in accuracy while significantly reducing the overall on-device GPU memory usage. More specifically, SLIMFIT reduces the overall memory usage of the fine-tuning process on CIFAR-10 from 7.2GB to 4.3GB (1.7× reduction) with a batch size of 32, on CIFAR-100 from 7.2GB to 4.5GB (1.6× reduction) with a batch size of 32, and on ImageNet from 77.4GB to 26.1GB (3× re-

Table 3: The accuracy and memory performance of SLIMFIT on the GLUE benchmark and SQuAD 2.0 using BERT over 10 random runs. The batch size of 32 and 128 were used for GLUE benchmark and SQuAD 2.0, respectively. The top-1 accuracy and memory performance of SLIMFIT are also reported for CV benchmarks with the batch size of 32 for CIFAR datasets and 128 for ImageNet dataset on ViT.

		BERT									ViT		
Method	Metric	MNLI_m	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	SQuAD 2.0	CIFAR-10	CIFAR-100	ImageNet
Basline	Accuracy	83.4	90.8	90.5	92.1	58.9	89.5	86.4	70.2	74.0	98.8	91.2	83.3
	Memory of Activations (GB)	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	55.1	4.5	4.5	69.5
	Total On-chip GPU Memory (GB)	6.1	6.1	6.1	6.1	6.1	6.1	6.1	6.1	58.5 (2 GPUs)	7.2	7.2	77.4 (3 GPUs)
SLIMFIT	Best Accuracy	83.3	90.6	90.6	92.5	59.9	89.6	86.5	70.5	74.2	98.6	91.2	83.4
	Average Accuracy	83.3	90.4	90.3	92.1	58.7	89.2	86.1	70.1	74.0	98.5	91.0	83.3
	Standard Deviation of Accuracy	0.155	0.168	0.173	0.185	1.258	0.323	0.27	0.632	0.115	0.089	0.125	0.071
	Freezing Rate (%)	80	80	95	95	90	85	91	90	80	90	75	95
	Memory of Activations (GB)	0.7	0.7	0.5	0.5	0.6	0.7	0.6	0.6	10	0.8	1.0	11.9
	Total On-chip GPU Memory (GB)	**4.4**	**4.4**	**4.0**	**4.0**	**4.3**	**4.3**	**4.3**	**4.3**	**19.1**	**4.3**	**4.5**	**26.1**



(a) CoLA



(b) MRPC

Figure 6: The trade-off curve between accuracy and freezing rate for three different iterative freezing approaches (i.e., ILS, random and progressive methods) on (a) CoLA and (b) MRPC datasets.

duction) with a batch size of 128. Fig. 5 also shows the total on-device GPU memory usage of SLIMFIT across different batch sizes on CV tasks.

8 Ablation Studies

Due to limited space, we only discuss the impact of the freezing rate on accuracy of SLIMFIT in this section. We provide further discussions on the impact of quantization/pruning, the total wall-clock time and the frequency of update occurrence in Appendices I, J, and K, respectively. For all the experiments in this section, we use a batch size of 32 and 3 epochs for fine-tuning.

Our ILS algorithm orchestrates the freezing schedule based on a simple rule: layers with largest distance values are updated whereas those with lowest distance values are kept frozen for the given freezing rate. Of course, such an iterative freezing approach trades off between accuracy and freezing rate. To better show this trade-off, we measured and illustrated accuracy of CoLA and MRPC datasets across different freezing rates in Fig. 6. The trade-off curve shows our ILS algorithm can maintain the accuracy at the same level of the baseline by freezing up to 95% of layers.

Besides our ILS algorithm, the freezing schedule can be decided using random or progressive freezing approaches. In the random scheduling method, frozen layers are randomly selected at each iteration. In the progressive approach, on the other hand, early layers are progressively kept frozen whereas later layers are being updated throughout the fine-tuning process. Among these approaches, our ILS algorithm significantly stands out in terms of both accuracy and freezing rate as shown in Fig. 6. The reason behind its superior performance is that ILS allows more updates for layers with large distance values by keeping layers with minimal distance values frozen for a specific number of iterations. On the other hand, in the random approach, the layers are randomly selected to be updated. Therefore, layers with large distance values receive less number of updates in the random approach compared to ILS. Of course, the chance of layers with large distance values being randomly selected as active layers decreases as the freezing rate increases, which explains the accuracy gap between ILS and the random approach with freezing rate higher than 70% freezing rate. In the progressive freezing approach, the early layers receive no update during the fine-tuning process, resulting in a significant accuracy degradation for large freezing rates.

9 Comparison With SOTA Techniques

Next, we compare SLIMFIT with state-of-the-art compression methods targeting memory reduction, i.e., 4-bit GACT (Liu et al., 2022) and DropIT (Chen et al., 2023). Table 4 summarizes the comparison results in terms of accuracy, memory and latency. For fair comparison, we measure their performance under the same framework and hyper-parameters (i.e., the batch size and the number of training epochs)

Table 4: Comparison with state-of-the-arts, i.e., 4-bit GACT (Liu et al., 2022) and DropIT (Chen et al., 2023) when fine-tuning BERT on CoLA dataset.

Model	Metric	Baseline	GACT	DropIT	SLIMFIT
BERT	Accuracy (Matthew's Corr.)	58.9	59.0	57.5	**59.9**
	Freezing Rate (%)	NA	NA	NA	90%
	Memory of Activations (GB)	3.2	**0.5**	2.4	0.6
	Total Memory (GB)	6.1	6.0	5.7	**4.3**
	Latency (Seconds)	**251**	455	367	281

during fine-tuning of BERT on CoLA. The experimental results of GACT and DropIT were obtained using their official PyTorch libraries. According to the experimental results, GACT shows the lowest memory amount for activations. However, in terms of on-device GPU memory usage, SLIMFIT outperforms GACT. In terms of accuracy, all models show a comparable accuracy on CoLA w.r.t. the baseline. Finally, in terms of speed, SLIMFIT shows the fastest fine-tuning speed among existing works while it still falls short w.r.t. the baseline (see Appendix J for more details on SLIMFIT's computing speed). It is worth mentioning that our method is orthogonal to commonly-used memory reduction techniques such as activation checkpointing, gradient accumulation and LoRA. In fact, these techniques can be used along with SLIMFIT to further reduce the memory. For instance, activation checkpointing can be applied to SLIMFIT to further reduce the total on-device GPU memory usage by the factor of 1.3× while SLIMFIT equipped with gradient accumulation can reduce it by a factor of 1.8× at the cost of an increase in the wall-clock time when fine-tuning GPT-2. The detailed discussions and experimental findings of these techniques, equipped with SLIMFIT, are presented in Appendix L.

10 Conclusion

In this paper, we presented a performance tool called SLIMFIT that reduces the memory usage of activations and accordingly the overall on-device GPU memory usage of transformer-based models through an iterative freezing of layers during fine-tuning. SLIMFIT adopts an inter-layer scheduling method to orchestrate the freezing schedule at each iteration. To balance the number of activations across all layers and to reduce the memory usage of static activations, SLIMFIT uses quantization and pruning for a few specific layers. We evaluated the performance of SLIMFIT across different NLP and CV tasks. We showed that SLIMFIT significantly reduces the on-device GPU memory usage of the fine-tuning process by up to 3.1× when using a batch size of 128.

11 Acknowledgements

This work is supported in part by the National Science Foundation through grants IIS-1955488, IIS-2027575, DOE awards DE-SC0016260, DE-SC0021982, ARO award W911NF2110339, and ONR award N00014-21-1-2724.

12 Limitations

This paper is an attempt to reduce the total on-device GPU memory usage of the whole fine-tuning process of transformers such as BERT and ViT. Despite the significant reduction in the total on-device GPU memory within the acceptable range of performance cost, our method has the following limitations:

- Introducing a new hyper-parameter: To control the performance of SLIMFIT, we introduced a new hyper-parameter called freezing rate. The choice of freezing rate varies for different downstream tasks and directly impacts the total on-device GPU memory usage and accuracy. Therefore, the freezing rate needs to be adjusted by the user similar to other hyper-parameters in transformers.

- Integration with deep learning optimization libraries developed for large language models (LLMs): Due to the large size of LLMs, training such models using sharding methods such as DeepSpeed library (Rajbhandari et al., 2020) is a common approach. However, the integration of SLIMFIT with such deep learning optimization libraries is non-trivial, preventing us from exploiting the potentials of our tools in reducing the on-device GPU memory usage of LLMs. In the future, we will develop the software required for the aforementioned integration to reduce the memory requirement of sharded LLMs across multiple GPUs.

13 Ethics Statement

This research was conducted in accordance to the academic and professional ethics guidelines. All the datasets used in this paper are publicly available and we ensured to acknowledge their data source using a citation. Moreover, all the results reported in this work are reproducible using the source code provided in the supplementary material. The tool developed in this work is for research purposes only and may not be suitable for production services without further scrutiny.

References

Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks. NIPS'18, page 5151–5159, Red Hook, NY, USA. Curran Associates Inc.

Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Efficient combination of rematerialization and offloading for training dnns. In *Advances in Neural Information Processing Systems*, volume 34, pages 23844–23857. Curran Associates, Inc.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Ayan Chakrabarti and Benjamin Moseley. 2019. Backprop with approximate activations for memory-efficient network training.

Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. 2021. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, pages 1803–1813. PMLR.

Joya Chen, Kai Xu, Yuhui Wang, Yifei Cheng, and Angela Yao. 2023. DropIT: Dropping intermediate tensors for memory-efficient DNN training. In *The Eleventh International Conference on Learning Representations*.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. Cite arxiv:1810.04805Comment: 13 pages.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*.

R. David Evans, Lufei Liu, and Tor M. Aamodt. 2020. Jpeg-act: Accelerating deep learning via transform-based lossy compression. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 860–873.

Zilin Fang, Mohamad Shahbazi, Thomas Probst, Danda Pani Paudel, and Luc Van Gool. 2022. Training dynamics aware neural network optimization with stabilization. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*, pages 4276–4292.

Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. 2020. Don't waste your bits! Squeeze activations and gradients for deep neural networks via TinyScript. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3304–3314. PMLR.

Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. Pipetransformer: Automated elastic pipelining for distributed training of large-scale models. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 4150–4159. PMLR.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511.

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic tensor rematerialization. In *International Conference on Learning Representations*.

Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. pages 32–33.

Yoonho Lee, Annie S Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. 2022. Surgical fine-tuning improves adaptation to distribution shifts. *arXiv preprint arXiv:2210.11466*.

Sheng Li, Geng Yuan, Yue Dai, Youtao Zhang, Yanzhi Wang, and Xulong Tang. 2023. SmartFRZ: An efficient training framework using attention-based layer freezing. In *The Eleventh International Conference on Learning Representations*.

Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, Michael Mahoney, and Alvin Cheung. 2022. GACT: Activation compressed training for generic network architectures. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 14139–14152. PMLR.

Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. 2021. Autofreeze: Automatically freezing model blocks to accelerate fine-tuning. *CoRR*, abs/2102.01386.

Amil Merchant, Elahe Rahimtoroghi, Ellie Pavlick, and Ian Tenney. 2020. What happens to BERT embeddings during fine-tuning? In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 33–44, Online. Association for Computational Linguistics.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2017. Mixed precision training. Cite arxiv:1710.03740Comment: Published as a conference paper at ICLR 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *CUDA semantics - PyTorch 2.0 documentation*.

Markus N. Rabe and Charles Staats. 2021. Self-attention does not need o(n^2) memory. *CoRR*, abs/2112.05682.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. SC '20. IEEE Press.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv e-prints*, page arXiv:1606.05250.

Vinay Venkatesh Ramasesh, Ethan Dyer, and Maithra Raghu. 2021. Anatomy of catastrophic forgetting: Hidden representations and task semantics. In *International Conference on Learning Representations*.

Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press.

Zhiqiang Shen, Zechun Liu, Jie Qin, Marios Savvides, and Kwang-Ting Cheng. 2021. Partial is better than all: revisiting fine-tuning strategy for few-shot learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9594–9602.

Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, Yizhong Wang, Hannaneh Hajishirzi, Noah A. Smith, and Yejin Choi. 2020. Dataset cartography: Mapping and diagnosing datasets with training dynamics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9275–9293, Online. Association for Computational Linguistics.

Ryan Teehan, Miruna Clinciu, Oleg Serikov, Eliza Szczechla, Natasha Seelam, Shachar Mirkin, and Aaron Gokaslan. 2022. Emergent structures and training dynamics in large language models. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 146–159, virtual+Dublin. Association for Computational Linguistics.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018a. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium. Association for Computational Linguistics.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018b. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. 2018. Neural network acceptability judgments. *arXiv preprint arXiv:1805.12471*.

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations*.

Geng Yuan, Yanyu Li, Sheng Li, Zhenglun Kong, Sergey Tulyakov, Xulong Tang, Yanzhi Wang, and Jian Ren. 2022. Layer freezing & data sieving: Missing pieces of a generic framework for sparse training. In *Advances in Neural Information Processing Systems*.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pages 36–39.

Appendix

A Memory Management

The on-device memory of modern GPUs is limited to a few tens of gigabytes depending on their model (e.g., 32GB NVIDIA V100). If the memory requirement of the training/fine-tuning of neural networks goes beyond the available memory on GPUs, an out-of-memory error will occur. The memory requirement of the under-run code on GPUs can be viewed by "nvidia-smi". For a training/fine-tuning process, this memory requirement is determined by the size of the model, cached activations, gradients, gradient

moments from the optimizer and CUDA contents after the first training iterations. It is worth mentioning that the memory usage of the training/fine-tuning process remains constant after the first iteration if the following iterations are performing the same computations (see Fig. 7). If the memory requirement of each iteration is different from others, PyTorch reports the memory requirement of the iteration using the maximum memory among other iterations as the total on-device GPU memory usage of the program in "nvidia-smi" (Paszke et al., 2017). Therefore, the unused memory of tensors in progressive memory optimization over training iterations will still show as used in "nvidia-smi". To reduce the overall memory usage of the training/fine-tuning process given the above explanations, we need to balance the memory usage of all iterations from the first iterations to the last one (see Fig. 7). SLIMFIT aims at reducing the overall memory usage of large transformer-based models during fine-tuning.

B Comparison with Existing Freezing Approaches

Here, we describe the main differences between SLIMFIT and other freezing approaches including SmartFRZ (Li et al., 2023), PipeTransformer (He et al., 2021) and AutoFreeze (Liu et al., 2021). The main difference is that the aforementioned works mainly focus on exploiting freezing to accelerate the training/fine-tuning process. Conceptually, SmartFRZ, PipeTransformer and AutoFreeze progressively freeze layers as the training process proceeds. In these methods, the first training iteration starts without freezing where all layers are updated. In the following iterations, these methods then progressively start freezing from the early layers down to the latest layers in the model in an orderly fashion. For instance, AutoFreeze performs the first epoch without freezing, the second epoch while freezing the first 5 layers, the third epoch while freezing the first 8 layers and the fourth epoch while freezing the first 11 layers when fine-tuning BERT. In this example, the memory and computation requirement of each epoch is different from others as each epoch presents a different degree of freezing. This allows to exploit the unused computing and memory resources to further accelerate the process by increasing batch sizes as the memory decreases throughout the training iterations (Liu et al., 2021) or increasing data-parallel width through pipelining (He et al., 2021). Since the first training iteration (or even epoch) of these methods performs the training process without freezing, their overall memory requirement reported by "nvidia-smi" is similar to that of training without freezing as discussed in Appendix A. In other words, the under-use GPU must still be able to meet the memory requirement of training without freezing. For instance, fine-tuning of ViT with a batch size of 128 on a single 32GB NVIDIA V100 using such methods results in an out-of-memory error.

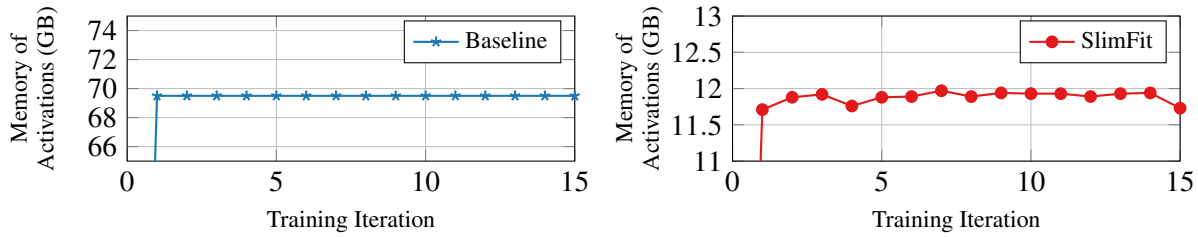
Table 5: The details of layers in MHA and FFN modules of BERT where B, T, H denote the batch size, sequence length, hidden size, respectively. ViT has the same structure with different descriptions.

Module	Type of Layer	Description	# Activations
MHA	Dense	attention.query	$B*T*H$
	Dense	attention.key	$B*T*H$
	Dense	attention.value	$B*T*H$
	MatMul	NA	$B*T*H(2\times)$
	Softmax	NA	$B*T*T$
	MatMul	NA	$B*T*H$ & $B*T*T$
	Dense	attention.output	$B*T*H$
	LayerNorm	attention.output	$B*T*H$
FFN	Dense	intermediate	$B*T*H$
	GELU	NA	$B*T*4*H$
	Dense	output	$B*T*4*H$
	LayerNorm	output	$B*T*H$

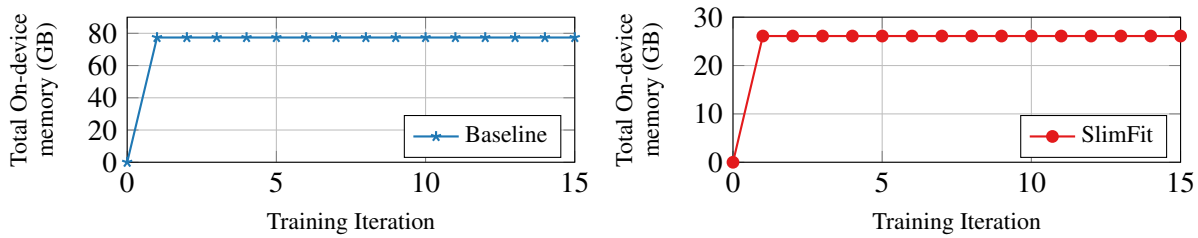
SLIMFIT, on the other hand, focuses on reducing the overall memory requirement of the fine-tuning process using freezing. As opposed to the aforementioned methods (i.e., SmartFRZ, PipeTransformer and AutoFreeze), SLIMFIT freezes layers at every single training iterations from the first iteration to the last one with a fixed freezing rate. With load-balancing using quantization, SLIMFIT ensures that the memory requirement of every single iteration remains roughly the same throughout the fine-tuning process. This enables SLIMFIT performing memory-intensive fine-tuning processes with large batch sizes on a single 32GB GPU such as ViT on ImageNet with a batch size of 128 while this normally requires three 32GB GPUs.

C Architecture of Transformer-based Models

Fig. 8 shows the overall architecture of transformer-based models including an initial embedding layer, followed by repeated blocks of multi-head attention (MHA) and feed-forward network (FFN). The details of each layer inside the MHA and FFN modules are provided in Table 5.



(a) Memory of Activations



(b) Total On-device Memory

Figure 7: The total on-device GPU memory and memory of activations during different training iterations when fine-tuning ViT on ImageNet with a batch size of 128 with the freezing rate of 95% compared to the baseline. SLIMFIT balances the memory usage of activations using freezing to reduce the total on-device memory usage of the fine-tuning process. While the memory usage of activations changes at each iteration when using SLIMFIT, the changes are relatively small thanks to the load-balancing technique described in Section 4.

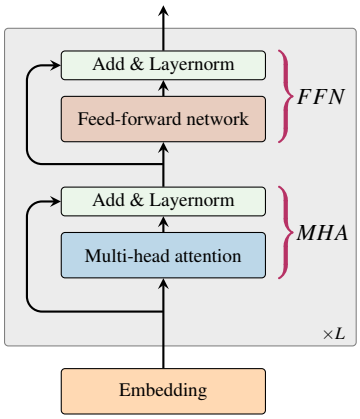


Figure 8: The main architecture of BERT. Note that ViT has a similar architecture with LayerNorms located before the MHA block. L denotes the number of attention layers.

D Theoretical Analysis

D.1 Convergence Analysis

In this section, we provide a convergence analysis for our freezing strategy. More precisely, we prove convergence of stochastic gradient descent (SGD) when considering freezing during update iterations. Given the loss function f, we assume that the parameters are initialized with some value and denoted as the vector $\mathbf{w}_0 \in \mathbb{R}^d$. Given the training example, the parameters are updated by

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma_t \nabla f(\mathbf{w}_t), \tag{8}$$

where $\mathbf{w}_t$ denotes the parameter vector at time t, γ_t is the learning rate, and ∇f represents the gradient of the loss function. We assume that the magnitude of the gradient samples are bounded by a constant $G > 0$ for all $\mathbf{x}$ in the space such that

$$||\nabla f(\mathbf{x})|| \leq G. \tag{9}$$

Also, we assume that there exists a constant $L > 0$ for any vector $\mathbf{u} \in \mathbb{R}^d$ where we have

$$|\mathbf{u}^T \nabla^2 f(\mathbf{x}) \mathbf{u}| \leq L||\mathbf{u}||^2. \tag{10}$$

Given Eq. (9) and Eq. (10), performing Taylor expansion on Eq. (8) similar to (Shalev-Shwartz and Ben-David, 2014) results in

$$\mathbb{E}[f(\mathbf{w}_{t+1})] \leq \mathbb{E}[f(\mathbf{w}_t)] - \gamma_t \mathbb{E}[||\nabla f(\mathbf{w}_t)||^2] + \frac{\gamma_t^2 G^2 L}{2}, \tag{11}$$

where $\mathbb{E}$ denotes the expected value.

Now, let us assume that the layer containing the parameter vector is frozen at the training iteration t. In this case, $\nabla f(\mathbf{w}_t)$ is equal to 0 and consequently $\mathbf{w}_{t+1}$ is equal to $\mathbf{w}_t$. In this freezing scenario, Eq. (11) still holds true since $\frac{\gamma_t^2 G^2 L}{2}$ is greater than 0.

By rearranging the terms in Eq. (11), summing over T iterations and telescoping the sum, we obtain

$$\sum_{t=0}^{T-1} \gamma_t \mathbb{E}[||\nabla f(\mathbf{w}_t)||^2] \leq \sum_{t=0}^{T-1} (\mathbb{E}[f(\mathbf{w}_t)] - \mathbb{E}[f(\mathbf{w}_{t+1})]) + \sum_{t=0}^{T-1} \frac{\gamma_t^2 G^2 L}{2}, \tag{12}$$

$$= f(\mathbf{w}_0) - f(\mathbf{w}_T) + \frac{G^2 L}{2} \sum_{t=0}^{T-1} \gamma_t^2, \tag{13}$$

$$\leq f(\mathbf{w}_0) - f(\mathbf{w}_*) + \frac{G^2 L}{2} \sum_{t=0}^{T-1} \gamma_t^2, \tag{14}$$

where $\mathbf{w}_*$ indicates an optimal solution. Given the above inequality, we showed that the convergence proof of SGD remains intact while introducing freezing for specific training iterations.

D.2 Backpropagation With a Frozen Layer

Here, we provide a simple example demonstrating how gradients are backpropagated to the first layer of a neural network while its middle layer is frozen. To this end, let us perform the backpropagation using a 3-layer network as an example. Mathematically, the architecture of this network can be described as follows:

$$\mathbf{y}_1 = \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1, \tag{15}$$
$$\mathbf{y}_2 = \mathbf{y}_1\mathbf{W}_2 + \mathbf{b}_2, \tag{16}$$
$$\mathbf{y}_3 = \mathbf{y}_2\mathbf{W}_3 + \mathbf{b}_3, \tag{17}$$

where $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{W}_3$, $\mathbf{b}_1$, $\mathbf{b}_2$ and $\mathbf{b}_3$ are the weights and biases of the network. In this example, $\mathbf{x}$, $\mathbf{y}_1$ and $\mathbf{y}_2$ are inputs to the first layer, the second layer and the third layer, respectively. Now, let us derive the backpropagation equations with the loss $\mathscr{L}$ using the chain rule as follows (please note that we obtain $\frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}$ by computing the loss where ∂ denotes the partial derivative):

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}_3} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{W}_3} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{y}_2, \tag{18}$$
$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}_3} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{b}_3} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}1 = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}, \tag{19}$$
$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}_2} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{W}_2} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T\mathbf{y}_1, \tag{20}$$
$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}_2} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{b}_2} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T 1 = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T, \tag{21}$$
$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{y}_1}\frac{\partial \mathbf{y}_1}{\partial \mathbf{W}_1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T\mathbf{W}_2^T\mathbf{x}, \tag{22}$$
$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}_1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{y}_1}\frac{\partial \mathbf{y}_1}{\partial \mathbf{b}_1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T\mathbf{W}_2^T 1 = \frac{\partial \mathscr{L}}{\partial \mathbf{y}_3}\mathbf{W}_3^T\mathbf{W}_2^T. \tag{23}$$

Given the above equations, to update the network weights (i.e., $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_3$), we need to store $\mathbf{x}$, $\mathbf{y}_1$ and $\mathbf{y}_2$ during the forward computations since they are required in Eq. (18), Eq. (20) and Eq. (22) during the back computations.

Now, suppose the middle layer is frozen. In this case, there is no need to compute Eq. (20) and therefore there is no need to store $\mathbf{y}_1$ during the forward computations. Of course, discarding $\mathbf{y}_1$ does not affect the backward computations of the first layer since Eq. (22) and Eq. (23) are independent of $\mathbf{y}_1$.

E Conversion Between 8-bit Integer and 32-bit Floating-point

Algorithm 2 shows the conversion process between 8-bit fixed-point and 32-bit floating-point formats. It is worth mentioning that the same procedure can be used for the conversion between 4-bit fixed-point and 32-bit floating-point formats. Moreover, the quantization function is used to compress the cached tensors during the forward propagation only. Of course, both the forward and backward computations are still performed using 32-bit floating-point computations as shown in Algorithm 4 where the "compress" function in this case is the quantization function (i.e., the conversion from 32-bit floating-point to 8-bit integer) and the "decompress" function performs the reverse computations (i.e., the conversion from 8-bit integer to 32-bit floating-point).

Algorithm 2 The conversion between 8-bit integer and 32-bit floating-point.

Description: number of integer bits as ib, number of fractional bits as fb, input x, output y

32 bits to 8 bits conversion:

$$y = \text{clamp}(\text{round}(x * 2^{fb}), -2^{fb+ib-1}, 2^{fb+ib-1} - 1)$$

8 bits to 32 bits conversion:

$$x = \frac{y}{2^{fb}}$$

F Pruning Algorithm

The pruning algorithm is performed in a few steps. In the first step, the input vector is sorted from largest to smallest values along with their indices and the size of the dense vector. We then only keep and cache the top 10% largest values of the input vector for the backward computations as the second step. It is worth mentioning that pruning beyond 90% results in a significant accuracy degradation. During backpropagation, we create a zero-valued tensor using the size of the dense vector and then replace zero values with the top 10% largest values using their corresponding indices. Algorithm 3 shows the pruning process during the forward computations and the restoring process during the backward computations. It is worth mentioning that the pruning function is used to compress the cached tensors during the forward propagation only. Of course, both the forward and backward computations are still performed using 32-bit floating-point computations as shown in

Algorithm 3 The description of the pruning process during the forward computations and the restoring process during the backward computations.

Description: $\mathbf{x}$: input vector, $\mathbf{x}_s$: sorted input vector, $\mathbf{x}_{idx}$: indices of the sorted input vector, $\mathbf{y}_s$: top 10% largest values, $\mathbf{y}_{idx}$: indices of top 10% largest values, $\mathbf{y}$: output vector, sort: sorting function, zeros: function to create zero-valued tensor, and scatter: function to replace zero values with the tensor values from $\mathbf{y}_s$ according to the indices.

Pruning process:

$\mathbf{x}_s, \mathbf{x}_{idx} = \text{sort}(\mathbf{x})$
$\mathbf{y}_s, \mathbf{y}_{idx} = \mathbf{x}_s[0 : int(\mathbf{x}.numel() * 0.1)], \mathbf{x}_{idx}[0 : int(\mathbf{x}.numel() * 0.1)]$

Restoring process:

$\mathbf{y} = \text{zeros}(\mathbf{x}.numel())$
$\mathbf{y} = \text{scatter}(\mathbf{y}, \mathbf{y}_s, \mathbf{y}_{idx})$

Algorithm 4 where the "compress" function in this case is the pruning function and the "decompress" function performs the restoring computations.

G Details of CV/NLP Tasks, Measurements and Hyper-parameter Settings

For language understanding tasks and CV tasks, we used BERT-base-cased and ViT-base throughout this paper, respectively. The BERT-base and ViT-base pre-trained on ImageNet-21k were configured according to (Devlin et al., 2018) and (Dosovitskiy et al., 2021), respectively. We used AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.999$ and $L2$ weight decay of 0.01) as the optimizer and linear decay of the learning rate with warmup ranging from 0 to 0.1 for both models. We evaluate BERT-base on several downstream tasks from the GLUE benchmark and SQuAD 2.0. We use Spearman correlation for STS-B, Matthews correlation for CoLA, accuracy for RTE, MRPC, SST-2 QQP, QNLI and MNLI_m (matched), and F1 score for SQuAD 2.0. For the downstream tasks from the GLUE benchmark, we used the sequence length of 128 whereas we adopted the sequence length of 384 for the question answering task on SQuAD 2.0. For CIFAR-10, CIFAR-100 and ImageNet, we use top-1 accuracy as our evaluation metric. For the image classification tasks, we used the patch size of 16 with the resolution of 224 for CIFAR-10/CIFAR-100 and the resolution of 384 for ImageNet. Depending on the task, the learning rate varies from 4e-5 to 1.8e-4. For all the experiments in this paper, we used 3 epochs for fine-tuning. The hyper-parameter settings of each task are summarized in Table 6. It is worth mentioning that ViT models can also be fine-tuned using SGD. However, fine-tuning ViT models using SGD requires more epochs w.r.t. AdamW to obtain a similar accuracy.

In this paper, we measured our experimental results directly from 32GB NVIDIA V100 GPU(s) without any memory swapping between CPU and GPU(s). The total on-device GPU memory usage of the fine-tuning process is measured using "nvidia-smi". We measured the wall-clock time (i.e., latency) of the fine-tuning process using the CUDA's event API in PyTorch (i.e., "torch.cuda.Event"). The memory footprint of activations on the GPU(s) was measured using the PyTorch's memory management API (i.e., "torch.cuda.memory_allocated").

H Experimental Results on BERT-large and GPT-2

Here, we provide the results of the fine-tuning experiments with BERT-large on the GLUE benchmark (see Table 7) and GPT-2 for language modeling on WikiText-2 (see Table 8) to demonstrate the benefits of SLIMFIT when applied to different NLP models and tasks. We report the best, average and standard deviation of the results over 10 random runs for our experiments in this section. We used the batch size of 8, the block size of 1024 and the learning rate of 2e-4 for GPT-2 fine-tuning. For BERT-large, we used the batch size of 32 and the learning rate of 5e-5. We fine-tuned both of these models for 3 epochs. For these experiments, SLIMFIT can provide up to 2.1× reduction in the total on-device GPU memory usage without any noticeable performance degradation.

I Impact of Quantization and Pruning

In this work, we used quantization and pruning for a few specific layers to balance the number of acti-

Table 6: The hyper-parameter settings of each NLP/CV task.

Dataset	Model	Optimizer	Learning Rate	Warmup	Evaluation Metric
$MNLI_m$	bert-base-cased	AdamW	4e-5	0	percentage accuracy
QQP	bert-base-cased	AdamW	5e-5	0	percentage accuracy
QNLI	bert-base-cased	AdamW	5e-5	0	percentage accuracy
SST-2	bert-base-cased	AdamW	8e-5	0	percentage accuracy
CoLA	bert-base-cased	AdamW	8e-5	0.1	Matthew's correlation
STS-B	bert-base-cased	AdamW	8e-5	0	Spearman correlation
MRPC	bert-base-cased	AdamW	1.25e-4	0	percentage accuracy
RTE	bert-base-cased	AdamW	1.2e-4	0	percentage accuracy
SQuAD 2.0	bert-base-uncased	AdamW	1.8e-4	0.1	F1 score
CIFAR-10	vit-base-patch16-224-in21k	AdamW	7.5e-5	0	percentage accuracy
CIFAR-100	vit-base-patch16-224-in21k	AdamW	5.5e-5	0	percentage accuracy
ImageNet	vit-base-patch16-384	AdamW	5e-5	0	percentage accuracy

Table 7: The experimental results of SLIMFIT on the GLUE benchmark using BERT-large over 10 random runs.

		BERT							
Method	Metric	$MNLI_m$	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE
Basline	Accuracy	85.9	91.2	92.2	93.8	61.5	89.6	86.9	71.6
	Total On-chip GPU Memory (GB)	15.0	15.0	15.0	15.0	15.0	15.0	15.0	15.0
SLIMFIT	Best Accuracy	86.0	91.3	92.2	94	63.2	89.9	87.7	73.3
	Average Accuracy	85.8	91.1	92.1	93.6	61.3	89.5	86.6	71.5
	Standard Deviation of Accuracy	0.115	0.158	0.097	0.282	0.955	0.363	0.775	1.344
	Freezing Rate (%)	80	80	90	90	90	85	80	80
	Total On-chip GPU Memory (GB)	**8.2**	**8.2**	**7.7**	**7.7**	**7.7**	**8.0**	**8.2**	**8.2**

Table 8: The experimental results of SLIMFIT for language modeling on WikiText-2 using GPT-2 over 10 random runs.

Method	Metric	GPT-2 (WikiText-2)
baseline	Accuracy (Perplexity)	21.3
	Total On-chip GPU Memory (GB)	29.9
SLIMFIT	Best Accuracy (Perplexity)	21.4
	Average Accuracy (Perplexity)	21.3
	Standard Deviation of Accuracy	0.127
	Freezing Rate (%)	75
	Total On-chip GPU Memory (GB)	**14.5**

vations across all layers and to reduce the memory footprint of static activations. We used 8-bit quantization for the activations of the imbalanced linear layer and MatMul. We also quantized the activations of GELU using 4 bits. The pruning of LayerNorm was performed when this layer is kept frozen. It is worth mentioning that both quantization and pruning have no impact on the forward computations. They are only used to compress activations for caching. To show the impact of such compression methods, we report the accuracy evaluation of BERT on CoLA and MRPC datasets with and without quantization or pruning in Table 9. The experimental results show no notable performance loss due to the compression techniques.

Table 9: The impact of quantization and pruning on the accuracy evaluation.

Dataset	Baseline	Quantization of			Pruning of	All
		Linear	MatMul	GELU	LayerNorm	together
CoLA	58.9	58.9	60.6	60.0	59.7	59.7
MRPC	86.4	86.4	86.3	86.3	86.3	86.3

J Discussion on Wall-Clock Time

Compared to training without freezing, SLIMFIT introduces extra computations and also skips weight gradient computations for the frozen layers at the same time. The main source of computational overhead in SLIMFIT is quantization and pruning of activations. The quantization overhead is due to the conversion between different precision levels (i.e., between 8 bits and 32-bit floating-point format) as discussed in Appendix E. Pruning also requires sorting of values to keep their top 10% largest values, which causes an additional computational overhead. Computing the weight distance metric is another source of computational overhead.

On the other hand, SLIMFIT skips the weight gradients computations of frozen layers using PyTorch "requires_grad" as shown in Algorithm 4. When an activate layer is frozen, there is no need to compute its weight gradients as discussed in Appendix D.2, which reduces the wall-clock time. The amount of

Algorithm 4 The description of skipping weight gradient computations when the layer is frozen. In this example, we assume the activations of the frozen layer require compression (e.g., an imbalanced linear layer or LayerNorm). Activations are denoted as "input" and are cached using either quantization or pruning depending on the type of the layer as a compression method. The compression and decompression functions are denoted as "compress" and "decompress". Since weights are defined as "Parameter" in PyTorch, caching weights does not introduce any extra memory.

```
class ILSFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input: torch.Tensor, weight: torch.nn.Parameter, requires_grad):
        # Compute forward computations to obtain out
        if requires_grad:
            ctx.save_for_backward(compress(input), weight)
        else:
            ctx.save_for_backward(weight)
        ctx.requires_grad = requires_grad
        return out
    @staticmethod
    def backward(ctx, grad_output: torch.Tensor):
        if ctx.requires_grad:
            input, weight = decompress(ctx.saved_tensors[0]), ctx.saved_tensors[1]
            # Compute backward computations to obtain grad_input and grad_weight
        else:
            weight = ctx.saved_tensors[1]
            grad_weight = None
            # Compute backward computations to obtain grad_input
        return grad_input, grad_weight, None
```

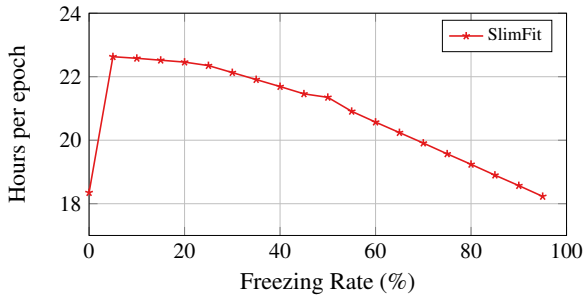
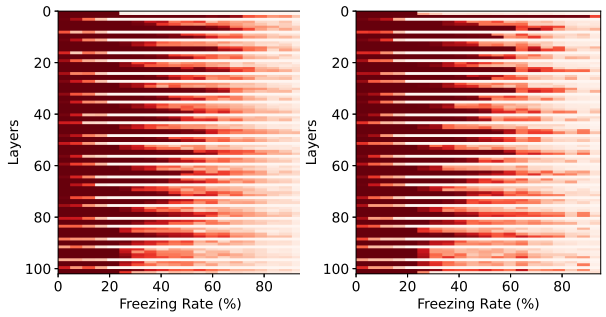


Figure 9: Wall-clock time of fine-tuning ViT on ImageNet with a batch size of 32 across different freezing rates.



(a) CoLA (b) MRPC

Figure 10: The frequency of update occurrence for each layer as a heatmap on (a) CoLA and (b) MRPC datasets.

speedup due to the skipped computations highly depends on the hyper-parameters of the networks such as freezing rate. Therefore, the wall-clock time of each network varies from one to another depending on the hyper-parameters. For instance, Fig. 9 shows the wall-clock time of fine-tuning ViT on ImageNet using a batch size of 32 across different freezing rates. According to the experimental results, the computational overhead of SLIMFIT is dominant for small freezing rates. However, as the freezing rate increases, the speedup of the skipped gradient computations overcomes the computational overhead of SLIMFIT where SLIMFIT with the freezing rate of 95% results in a similar wall-clock time as of the baseline. It is worth mentioning that the baseline is the point at the freezing rate of 0 where no freezing was used during the fine-tuning process.

K Frequency of Update Occurrence

To visualize the frequency of update occurrence for each layer, we use a heatmap as shown in Fig. 10 for both CoLA and MRPC datasets where larger counts are associated with darker colorings. As shown in

Algorithm 5 The description of layers associated to the indices in Fig. 10.

bert.embeddings.word_embeddings.weight
bert.embeddings.position_embeddings.weight
bert.embeddings.token_type_embeddings.weight
bert.embeddings.LayerNorm.weight
for $i = 0$ to 11: **do**
  bert.encoder.layer[i].attention.self.query.weight
  bert.encoder.layer[i].attention.self.key.weight
  bert.encoder.layer[i].attention.self.value.weight
  bert.encoder.layer[i].attention.output.dense.weight
  bert.encoder.layer[i].attention.output.LayerNorm.weight
  bert.encoder.layer[i].intermediate.dense.weight
  bert.encoder.layer[i].output.dense.weight
  bert.encoder.layer[i].output.LayerNorm.weight
end for
bert.pooler.dense.weight
classifier.weight

the heatmap, the dense layers inside the MHA module receive more updates than other layers for both datasets. Moreover, the update patterns of these datasets are similar for small freezing rates whereas they become more task-specific for high freezing rates. In fact, the ILS algorithm prioritizes the update of some specific layers over others for high freezing rates.

The description of layers associated to the indices in Fig. 10 is provided in Algorithm 5. It is worth mentioning that the layers denoted by "bert.encoder.layer[i].attention" belong to the MHA module whereas the remaining layers inside the loop belong to the FFN module.

L Comparison With Memory-Efficient Techniques

In this section, we provide experimental results of SLIMFIT applied to basic memor-efficient training methods including gradient accumulation (GA), gradient checkpointing (GC), and parameter-efficient fine-tuning (PEFT). We fine-tune the GPT-2 model on WikiText-2 for three epochs using the batch size of 8 with the freezing rate of 75% for comparison purposes.

GA is a technique which allows dividing the training data into smaller micro-batches and then accumulating the gradients from each micro-batch before applying them to update the model. We set the size of the micro-batch for GA to one which yields the lowest GPU memory usage. With this configuration, the GA step size is equal to 8. While GA can significantly reduce the memory usage of the fine-tuning process at a slight increase in run-time performance, its reduction is limited by the size of the micro-batch being 1 as shown in Table 10. SLIMFIT equipped with GA can further reduce the GPU memory usage from 5.7GB down to 3.2GB.

Table 10: The experimental results of GA equipped with SLIMFIT when fine-tuning GPT-2 on WikiText-2.

Method	Metric	GPT-2 (WikiText-2)
SLIMFIT	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	14.5
	Time per epoch (s)	241.7
GA	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	5.7
	Time per epoch (s)	257.3
SLIMFIT + GA	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	3.2
	Time per epoch (s)	261.5

Table 11: The experimental results of GC equipped with SLIMFIT when fine-tuning GPT-2 on WikiText-2.

Method	Metric	GPT-2 (WikiText-2)
SLIMFIT	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	14.5
	Time per epoch (s)	241.7
GC	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	10.2
	Time per epoch (s)	307.4
SLIMFIT + GC	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	8.1
	Time per epoch (s)	319

GC can reduce activation memory by trading computations for memory. In this method, only specific activations are stored during the forward pass, while the rest are recomputed in the backward pass. Of course, the recomputation of activations comes at the cost of an increase in the run-time performance while significantly reducing the memory usage as shown in the table below. SLIMFIT equipped with GC can further reduce the memory usage of GC by

Table 12: The experimental results of LoRA (with the rank of 16) equipped with SLIMFIT when fine-tuning GPT-2 on WikiText-2.

Method	Metric	GPT-2 (WikiText-2)
SLIMFIT	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	14.5
	Time per epoch (s)	241.7
PEFT (LoRA)	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	29.5
	Time per epoch (s)	215.3
SLIMFIT + PEFT (LoRA)	Accuracy (Perplexity)	21.4
	Total On-chip GPU Memory (GB)	14.2
	Time per epoch (s)	233.1

21% when fine-tuning GPT-2 on WikiText-2 (See Table 11).

PEFT approaches rely on updating the prepended trainable parameters to the input of the layers. In other words, adapter modules containing a small number of parameters are inserted to each layer of the model and only the parameters of these modules are adjusted during the fine-tuning process. It is worth mentioning that the number of activations for such a method remains the same. The main memory saving of PEFT approaches comes from the memory saving in optimizer states and gradients. AdamW, which is commonly used as the optimizer for fine-tuning, stores two states of the trainable parameters of the model. Since PEFT approaches train the model on the prepended parameters, the size of optimizer states would be the same as the size of the prepended parameters. As such, the memory saving of PEFT methods is significant only when the GPU memory usage is dominated by the size of the model parameters, which is the case for LLMs. However, for small-size models where the memory of activations is dominant, the memory reduction of PEFT approaches is not significant. For instance, fine-tuning the GPT-2 model on WikiText-2 results in a non-significant reduction in the GPU memory usage as shown in Table 12. As such, PEFT equipped with SLIMFIT does not also provide a significant reduction in memory compared to regular fine-tuning using SLIMFIT as shown in Table 12.