Sheared Backpropagation for Fine-tuning Foundation Models

Zhiyuan Yu[1], Li Shen[*,2], Liang Ding[2], Xinmei Tian[1], Yixin Chen[3], and Dacheng Tao[4]
[1]University of Science and Technology of China, Hefei, China; [2]JD Explore Academy, Beijing, China;
[3]Washington University in St. Louis, MO, USA; [4]Nanyang Technological University, Singapore
yuzhiyuan@mail.ustc.edu.cn, xinmei@ustc.edu.cn, chen@cse.wustl.edu
{mathshenli, liangding.liam, dacheng.tao}@gmail.com

Abstract

Fine-tuning is the process of extending the training of pre-trained models on specific target tasks, thereby significantly enhancing their performance across various applications. However, fine-tuning often demands large memory consumption, posing a challenge for low-memory devices that some previous memory-efficient fine-tuning methods attempted to mitigate by pruning activations for gradient computation, albeit at the cost of significant computational overhead from the pruning processes during training. To address these challenges, we introduce PreBackRazor, a novel activation pruning scheme offering both computational and memory efficiency through a sparsified backpropagation strategy, which judiciously avoids unnecessary activation pruning and storage and gradient computation. Before activation pruning, our approach samples a probability of selecting a portion of parameters to freeze, utilizing a bandit method for updates to prioritize impactful gradients on convergence. During the feed-forward pass, each model layer adjusts adaptively based on parameter activation status, obviating the need for sparsification and storage of redundant activations for subsequent backpropagation. Benchmarking on fine-tuning foundation models, our approach maintains baseline accuracy across diverse tasks, yielding over 20% speedup and around 10% memory reduction. Moreover, integrating with an advanced CUDA kernel achieves up to 60% speedup without extra memory costs or accuracy loss, significantly enhancing the efficiency of fine-tuning foundation models on memory-constrained devices.

1. Introduction

Nowadays, Transformer-based models [50] have achieved remarkable results in various domains of deep learning [19, 20]. The trend towards scaling up these models for enhanced performance necessitates substantial data and computational resources, often beyond the reach of many researchers [44]. Consequently, fine-tuning pre-trained Transformer models, which are a prominent example of foundation models [5], has become a prevalent strategy. This approach allows for adapting and continuing the training of these models at reduced costs while retaining high performance. Despite the advantages, fine-tuning foundation models on low-end devices, such as personal computers with consumer-level GPUs and other edge devices, presents considerable challenges. These challenges stem primarily from the limited memory and computational capacity of such devices, restricting the feasible model and batch sizes during fine-tuning. This limitation often results in prolonged training periods or compromised performance. Our paper aims to address this issue by exploring efficient methodologies for fine-tuning Transformer models on consumer-level devices, striving to balance performance with training efficiency.

One pervasive challenge in fine-tuning models is effectively managing the training pipeline within the confines of limited device memory [41]. Beyond loading trainable parameters, memory usage in model training includes storing intermediate tensors like gradients and activations. These elements are crucial in backpropagation and become particularly memory-intensive with large input batch sizes or model scales. Although efficient training techniques, such as low-precision and pruning, have been proposed to save memory on model parameters and are widely adopted [23, 31], recent research has shifted focus towards reducing memory consumption of activations, identified as a primary memory bottleneck [7]. Notably, since activations are stored solely for backward computation, recent studies on activation reduction [8, 24, 37] have proposed an asymmetric backpropagation pipeline. This approach maintains a dense forward pass and compresses activations only for gradient computation during backpropagation. Among these, BACKRAZOR by Jiang et al. [24] employs top-k pruning in the forward pass to sparsify activa-

*Corresponding Author

tions, thereby reducing memory requirements for their storage. This method then re-densifies them during the backward pass for gradient computation, achieving significant memory savings. However, since activation pruning methods do not retain the original activations, they typically require additional computations to determine optimal pruning locations, in order to maintain accuracy. This introduces a trade-off between memory reduction and computational overhead, necessitating a careful balance. Furthermore, it presents a challenge in enhancing the efficiency of current computation-intensive activation pruning methods.

In our work, addressing this unresolved challenge, we present PreBackRazor, a memory and computation-efficient activation pruning scheme. It aligns with the asymmetric sparsified backpropagation approach, where sparsified activations are preserved solely for the backpropagation process. PreBackRazor operates by dynamically freezing non-essential activation coordinates, thus minimizing the necessity for their sparsification and storage. This method effectively reduces computational load, thereby boosting the overall computation efficiency. Before initiating each iteration's forward pass, we strategically assign a dropout probability to every activation coordinate. This probability is informed by the gradient values associated with these coordinates, derived from prior iterations. This process enables us to selectively freeze certain coordinates during the ensuing backward pass. The chosen probabilities aim to preserve those gradients that are crucial for model convergence. To continuously refine this probability distribution, we utilize a bandit method, an approach inspired by JointSpar [30]. Consequently, during the forward pass, we bypass the pruning and storage of activations that are deemed non-essential for backpropagation. This method effectively diminishes the computational burden typically associated with activation pruning, while simultaneously upholding memory efficiency and training accuracy.

Our experiments, conducted on a single RTX 4090 GPU, have shown significant improvements in training efficiency and memory management during the fine-tuning of Vision Transformer (ViT) and BERT models. For ViT, an extensive comparative analysis using the CIFAR datasets [26] revealed that our method, PreBackRazor, outperforms traditional activation pruning techniques like BackRazor in terms of computational efficiency. Notably, PreBackRazor achieves an up to 1.7× increase in training speed for linear layers, without incurring additional memory overhead. Furthermore, our carefully tailored adaptation of the FlashAttention kernel has significantly accelerated the fine-tuning process, ensuring efficient memory usage. This customized integration results in an impressive 1.7× boost in training speed, maintaining a balanced computational budget without sacrificing accuracy. In the context of BERT fine-tuning, our approach also results in a modest improvement in time efficiency and final accuracy.

In summary, our main contributions are three-fold:

- We propose a novel activation pruning pipeline to further reduce the computational overhead of previous methods. In addition, we establish the convergence analysis for the proposed algorithm.
- We design improved activation storage schemes that jointly reduce the compute and memory. We employ bandit methods to update gradients across activation coordinates of the model layers adaptively. This selective update mechanism efficiently manages activation pruning and gradient computation, thereby reducing memory and computational costs based on BackRazor's asymmetric activation-self-sparsified backpropagation.
- We conduct experiments on fine-tuning Transformer-based ViT and BERT, adapt the current code library, and combine the state-of-the-art FlashAttention 2 CUDA [16] kernels to boost the training, making the pipeline more welcome by nowadays consumer-level devices. Our experiments show that our method reduces over 30% of training time without loss of accuracy.

2. Related Work

In this section, we briefly review existing memory and computation-efficient fine-tuning methods.

Memory-efficient fine-tuning methods. In memory-limited environments, strategies like partitioning and compression leverage trade-offs in computation, communication, and storage to reduce memory usage. Gradient accumulation lowers memory usage at the cost of increased communication, while gradient checkpointing [12] achieves memory savings through recomputation. CPU offloading [42], GPU partitioning [41], and FlashAttention [16, 17] balance memory optimization against communication overhead. Several memory-efficient optimizers [32, 43, 49] have been proposed to approximate backpropagation to minimize stored information. Recent approaches include computing only forward passes with estimated gradients [34] and integrating gradient computation with parameter updates [33]. Additionally, low or mixed-precision training [27, 35] and other sparse backpropagation methods [13] have been integrated, trading some numerical accuracy for enhanced memory or communication efficiency. There are also related works focusing on communication cost reduction in distributed systems that use these memory-efficient methods for gradient compression [1, 4, 45], which is different from our settings. Additionally, while quantization [9, 18] and pruning [47] methods are commonly employed to compress gradients and model weights, our research concentrates on the pruning of activations. In the realm of activation footprint reduction, Cai et al. [7] suggest freezing layer weights while updating biases and adding lightweight residual modules. Other methods involve compression of activations [2, 11, 54]. To maintain accuracy, recent research proposes asymmetric sparsified backpropagation which stores compressed activations, using original dense inputs for forward computation and retrieving activations for backward gradient computations. They employ techniques like low-precision compression [8, 37], or utilizing top-k masks for pruning activations [24].

Computation-efficient fine-tuning methods. Existing computation-efficient fine-tuning methods include PEFT, knowledge distillation [22], few-shot learning [52], as well as quantization and pruning implementations. Our research primarily focuses on techniques that dynamically adjust training and seamlessly integrate with our sparse backpropagation pipeline. In the realm of computation-efficient fine-tuning methods, we find adaptive learning rates [25, 49] relevant for their role in facilitating faster convergence [10, 46, 56]. Also, dynamic sparse algorithms [29, 36] are notable for introducing sparsity during training. For the aspect of sparse backpropagation, the implementation of top-k masking over gradients by Sun et al. [48] to induce sparsity in backpropagation is in line with our approach to asymmetric pruning. Structured sparsity patterns [21, 55] and layer freezing techniques [6, 40], along with selective weight freezing strategies [28], offer valuable insights into reducing computational load. We also draw specific elements from the JOINTSPAR algorithm [30], which selectively drops gradient vectors, a principle we adapt to target the computational challenges in activation pruning.

3. Rethink the Backpropagation Redundancy

In this section, we first present explanations of the memory cost of activations, and the asymmetric self-sparsified activations as introduced in BACKRAZOR algorithm [24]. Then we show the memory and computation redundancy in the current backpropagation scheme.

3.1. Preliminary

Consider the nonconvex optimization problem of fine-tuning foundation models: $\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\theta})$, where $\mathcal{L}$ is the training loss calculated with the model's weight $\boldsymbol{\theta}$, and $\boldsymbol{\Theta}$ is the parameter space. Without loss of generality, the forward pass of a given neural network with n layers can be expressed as follows:

$$f(\mathbf{x}_0;\boldsymbol{\theta}) = f_n(f_{n-1}(\ldots f_0(\mathbf{x}_0;\boldsymbol{\theta}_0)\ldots;\boldsymbol{\theta}_{n-1};\boldsymbol{\theta}_n), \quad (1)$$

$$L = \mathcal{L}(f(\mathbf{x}_0;\boldsymbol{\theta}), y) \quad (2)$$

where $\mathbf{x}_0$ represents the input features at the 0-th layer, and y corresponds to the input label. The functions f_i and parameters $\boldsymbol{\theta}_i$ are associated with the model's i-th layer. The expression $f(\mathbf{x}_0;\boldsymbol{\theta})$ indicates the model's final prediction. In this section, the symbol L stands for the loss between the prediction and the label, computed using the specified loss function $\mathcal{L}$. We define the activation $\mathbf{x}_i$ as the output of the $(i-1)$-th layer, i.e., $\mathbf{x}_i = f_{i-1}(\mathbf{x}_{i-1};\boldsymbol{\theta}_{i-1})$.

Denoting $h_i^{(\mathbf{x})}(\mathbf{x}_i,\boldsymbol{\theta}_i)$ and $h_i^{(\boldsymbol{\theta})}(\mathbf{x}_i,\boldsymbol{\theta}_i)$ as the derivatives of $\mathbf{x}_{i+1}$ with respect to $\mathbf{x}_i$ and $\boldsymbol{\theta}_i$ respectively. Normally, the backward process can be defined as:

$$\frac{\partial L}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}}\frac{\partial \mathbf{x}_{i+1}}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(\mathbf{x})}(\mathbf{x}_i,\boldsymbol{\theta}_i), \quad (3)$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}}\frac{\partial \mathbf{x}_{i+1}}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(\boldsymbol{\theta})}(\mathbf{x}_i,\boldsymbol{\theta}_i). \quad (4)$$

For instance, considering a linear layer with parameters $\boldsymbol{\theta}_i$, which consist of a weight matrix $\mathbf{W}_i$ and a bias vector $\mathbf{b}_i$, its forward pass function for computing the output $\mathbf{x}_{i+1}$ is expressed as $\mathbf{x}_{i+1} = \mathbf{x}_i\mathbf{W}_i^\top + \mathbf{b}_i$. All of the variables $\mathbf{x}_i$, $\mathbf{W}_i$, and $\mathbf{b}_i$ are retained for gradient computation before the layer produces the output $\mathbf{x}_{i+1}$. Subsequently, during the backward pass of this layer in the loss backpropagation process, the gradients are calculated as:

$$\frac{\partial L}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}}\frac{\partial \mathbf{x}_{i+1}}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}}\mathbf{W}_i, \quad (5)$$

$$\frac{\partial L}{\partial \mathbf{W}_i} = (\frac{\partial L}{\partial \mathbf{x}_{i+1}})^\top \mathbf{x}_i, \quad (6)$$

$$\frac{\partial L}{\partial \mathbf{b}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}}. \quad (7)$$

Here, the activation $\mathbf{x_i}$ is involved in computing the gradient for updating the weight $\mathbf{W}_i$. Consequently, the activation value must be stored in memory during the forward pass. This storage occupies a significant portion of memory, often exceeding the memory required for saving the weights. Additionally, the size of the activations increases linearly with the batch size.

Jiang et al. [24] propose BACKRAZOR, a method that only prunes the activation saved for backpropagation and keeps it dense in the forward pass. The activation $\mathbf{x}_i$ is pruned and stored in the context as a sparse version $\tilde{\mathbf{x}}_i$. The backward process with pruned activations turns into:

$$\frac{\partial L}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(\mathbf{x})}(\widetilde{\mathbf{x}}_i,\boldsymbol{\theta}_i), \quad (8)$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(\boldsymbol{\theta})}(\widetilde{\mathbf{x}}_i,\boldsymbol{\theta}_i). \quad (9)$$

As for their activation pruning, a straightforward top-k pruning method is applied, which identifies and removes the k smallest magnitude values within the activation. Denoting v as the total number of values of the activation, to introduce a targeted sparsity level, a fixed parameter λ is set (e.g., $\lambda = 0.9$), and the pruning budget is determined by setting $k = \lambda v$. Consequently, all activation values below the k-th

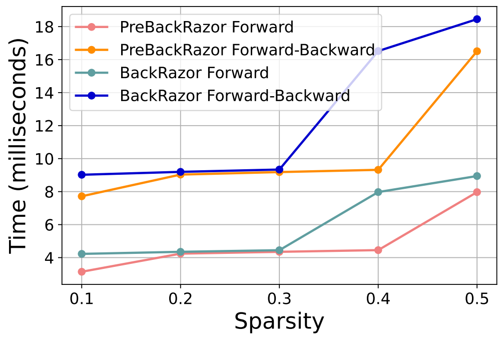


Figure 1. Sparsity vs computation time cost between BackRazor (element-wise, self-sparsified) and proposed PreBackRazor (column-wise, joint-sparsified).

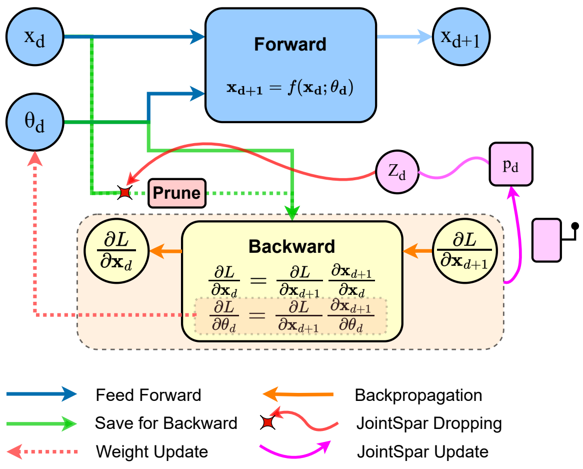
smallest value are zeroed out. After this pruning operation, the pruned activation is stored as a binary bitmap, known as the mask. This mask indicates the positions of the non-zero elements. Additionally, a dense matrix is maintained to retain the values of the non-zero elements.

3.2. Rethink the sparsity in BACKRAZOR

Redundancy in full-gradient computation. Common backpropagation activation sparsification schemes involve restoring compressed activations, saved within the model context, back to their dense format during backpropagation. In the case of BACKRAZOR, this process entails densifying the sparse matrix to its original shape for gradient computation. Consequently, the full gradient is computed using the full-size activation, and the sparsity of the pruned activation is not utilized to expedite computation.

Computation overhead in pruning. A prevalent pruning approach involves generating an element-wise self-sparsified top-k mask to eliminate smaller values. This method, when employed for activation pruning, incurs additional computational costs. These costs include sorting to find the k-th smallest activation value and performing element-wise comparisons across all activation elements to construct a sparse tensor. Collectively, these operations contribute to significant computational overhead, potentially extending training time beyond that of traditional fine-tuning. Such a trade-off for memory reduction may not be justifiable. Exploring alternative sparsity patterns and pruning schemes, such as column-wise and joint-sparse could lead to the discovery of faster algorithms, potentially replacing the current self-sparsified top-k activation pruning method and thus reducing computation costs.

Inspired by the observed inefficiencies in existing methods, we sought to investigate whether alternative sparsity patterns could offer reduced computation. Specifically, we compared the performance of column-wise sparsity with element-wise sparsity within the same linear layer, across various sparsity ratios, as depicted in Figure 1. Our findings indicate notable up to 1.7× speedups, aligning with our initial hypothesis. Additionally, the column-wise activation sparsity pattern facilitates the use of existing joint sparse algorithms, which leverage gradient information to generate sparse masks previous to the BACKRAZOR pruning pipeline. This approach incurs lower computational costs. Given that the original sparsity is unstructured and of a low ratio, the integration of a structured sparsity pattern also holds promise for further acceleration, leveraging modern sparse computation software and hardware advancements [14, 21, 39].



x_d
θ_d
Forward
$x_{d+1} = f(x_d; \theta_d)$
x_{d+1}
Z_d
p_d
Prune
Backward
$\frac{\partial L}{\partial x_d}$
$\frac{\partial L}{\partial x_{d+1}}$
$\frac{\partial L}{\partial x_d} = \frac{\partial L}{\partial x_{d+1}} \frac{\partial x_{d+1}}{\partial x_d}$
$\frac{\partial L}{\partial \theta_d} = \frac{\partial L}{\partial x_{d+1}} \frac{\partial x_{d+1}}{\partial \theta_d}$
Feed Forward
Save for Backward
Weight Update
Backpropagation
JointSpar Dropping
JointSpar Update

Figure 2. Pipeline of the proposed PreBackRazor.

4. Methodology

In this section, we first propose the PREBACKRAZOR method. After that, we introduce the bandit method from JOINTSPAR algorithm [30] that we use to adjust the probability of dropping computations. Lastly, we examine the convergence of our proposed method.

4.1. Overall pipeline of PREBACKRAZOR

Our jointly sparsified activation pruning approach, PREBACKRAZOR, employs asymmetric backpropagation, maintaining dense computations in the forward pass while pruning activations for the backward pass, as depicted in Fig. 2. Inspired by the JOINTSPAR algorithm [30], our method goes a step further by incorporating gradient sparsity into the activation pruning, aiming to significantly boost computational efficiency. In PREBACKRAZOR, we manage a probability distribution over activation coordinates to selectively prune redundant indices. This process effectively reduces the computation of related gradients and halts the updating of corresponding weights, thereby enhancing training efficiency and minimizing computational requirements. The decision-making process hinges on the analysis of gradient norms, leveraging the joint sparsity of gradients and activations. This approach prioritizes the retention of the most impactful gradient coordinates for weight updates. The implementation details of this method are outlined in Algorithm 1.

Algorithm 1 PREBACKRAZOR

```
1:  Initialize all $p_0 = s/D$
2:  for $t$ = 1 to $T$ do
3:      Sample $\mathbf{x}_0, \mathbf{y}$ from $\mathcal{D}$
4:      for $i$ = 1 to $n$ do                                          ▷ Forward
5:          Instantiate $Z_{t,d}$, with $P(Z_{t,d} = 1) = p_{t,d}$
6:          Determine $S_t = \{d : Z_{t,d} = 1\}$
7:          $\mathbf{x}_i = f_i(\mathbf{x}_{i-1}, \boldsymbol{\theta}_i)$
8:          $mask$ = a binary bitmap of size $D$
9:          for parameter $\boldsymbol{\theta}_{[d]}$ of $\boldsymbol{\theta}_i$ do
10:             if $d \in S_t$ then
11:                 $mask_d$ =True
12:             end if
13:         end for
14:         $\tilde{\mathbf{x}}_{i-1}$=Sparsify($\mathbf{x}_{i-1}$, $mask$)
15:         SaveForBackward($\tilde{\mathbf{x}}_{i-1}$, $mask$, $\boldsymbol{\theta}_i$)
16:     end for
17:     Compute loss $L = \mathcal{L}(\mathbf{x}_{n+1}, \mathbf{y})$
18:     for i = $n-1$ to 1 do                                          ▷ Backward
19:         Densify($\tilde{\mathbf{x}}_{i-1}$, $mask$)
20:         $\frac{\partial L}{\partial \mathbf{x}_i} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(x)}(\tilde{\mathbf{x}}_i, \boldsymbol{\theta}_i)$
21:         for parameter $\boldsymbol{\theta}_{[d]}$ with $d \in S_t$ do
22:             $\frac{\partial L}{\partial \boldsymbol{\theta}_{[d]}} = \frac{\partial L}{\partial \mathbf{x}_{i+1}} h_i^{(\theta)}(\tilde{\mathbf{x}}_i, \boldsymbol{\theta}_{[d]})$
23:         end for
24:         $p_{t+1}$ = Update($p_t$,$S_t$,$\{\boldsymbol{g}_{t,[d]}\}_{d \in S_t}$)
25:     end for
26: end for
```

For a given input activation $\mathbf{x}_i$ in the i-th layer of the model, we decompose the activations column-wise, represented as $\mathbf{x}_i = [\mathbf{x}_{i,1}, \cdots, \mathbf{x}_{i,d}, \cdots, \mathbf{x}_{i,D}]$, where d and D respectively denote the d-th coordinate out of all D column indices. Considering a linear layer, its weight parameter $\boldsymbol{\theta}$ is similarly decomposed into $[\boldsymbol{\theta}_{[1]}, \cdots, \boldsymbol{\theta}_{[d]}, \cdots, \boldsymbol{\theta}_{[D]}]$. To differentiate from the layer-wise parameter set $\boldsymbol{\theta}_i$ (as discussed in Section 3.1), we use square brackets to denote each coordinate, represented as $\boldsymbol{\theta}_{[d]}$. The gradient of the loss with respect to $\boldsymbol{\theta}_{[d]}$ is denoted by $\boldsymbol{g}_{t,d}$, and we define $\boldsymbol{g}_{t,[d]}$ as a vector with all elements zero except the d-th element: $\boldsymbol{g}_{t,[d]} = [0, \cdots, \boldsymbol{g}_{t,d}, \cdots, 0]$, with $\boldsymbol{g}_t$ being the sum of all such vectors, $\boldsymbol{g}_t = \sum_{d=1}^{D} \boldsymbol{g}_{t,[d]}$.

At each iteration t, a probability distribution $p_t = [p_{t,1}, \cdots, p_{t,d}, \cdots, p_{t,D}]$ is maintained over the joint coordinates of the parameters $[\boldsymbol{\theta}_{[1]}, \cdots, \boldsymbol{\theta}_{[d]}, \cdots, \boldsymbol{\theta}_{[D]}]$ and activations $[\mathbf{x}_1, \cdots, \mathbf{x}_d, \cdots, \mathbf{x}_D]$. For each coordinate, a Bernoulli random variable $Z_{t,d}$ is sampled with $P(Z_{t,d} = 1) = p_{t,d}$ and $P(Z_{t,d} = 0) = 1 - p_{t,d}$, deciding whether to include ($Z_{t,d} = 1$) or exclude ($Z_{t,d} = 0$) the gradient of $\boldsymbol{\theta}_d$ during backpropagation.

Initially, each p_t is set uniformly to $\frac{s}{D}$, aligning with the desired sparsity budget s, where s signifies the number of columns involved in the backward process. At the onset of each iteration t, the Bernoulli random variable $Z_{t,d}$ is sampled to form the active set S_t of coordinates that retain gradients in activations. Post gradient computation in backpropagation, this gradient information informs the update of the probability distribution, employing a bandit method which we will introduce in the subsequent section.

Integrating the FLASHATTENTION kernel. The fine-tuning of Transformer-based models on consumer-grade devices presents a unique opportunity to leverage modern, efficient kernels for improved training efficiency. In our approach, we have adapted and integrated the FLASHATTENTION kernel, substituting the original, more segmented sparsified attention implementation in BACKRAZOR. To address the increased memory usage resultant from this integration, we have incorporated PREBACKRAZOR within the FLASHATTENTION function. This inclusion not only facilitates pruning but also employs recomputation strategies to mitigate precision loss. We designate this enhanced methodology as PREBACKRAZOR++. This innovative approach ensures memory usage remains on par with BACKRAZOR, while markedly accelerating the training process. We further combine JOINTSPAR over model parameters with our method as the PREBACKRAZOR++JOINTSPAR for best memory and computation efficiency. Details on implementation are omitted here for brevity.

4.2. Bandit method for gradient sparsification

To update the probability distribution p_t over D coordinates at each iteration t, we utilize a bandit method akin to that in JOINTSPAR [30], originally introduced by Auer et al. [3], as detailed in Algorithm 2.

Algorithm 2 Bandit [30] for Updating Distribution p_t

```
1:  Algorithm: UPDATE($p_t$,$S_t$,$\{\boldsymbol{g}_{t,[d]}\}_{d \in S_t}$)
2:  Input: $p_t$: probability distribution, $S_t$: set of active parameters, $\{\boldsymbol{g}_{t,[d]}\}_{d \in S_t}$: gradients
3:  for $d = 1$ to $D$ do
4:      if $d \in S_t$ then
5:          $\tilde{l}_{t,d} = -\frac{\|\boldsymbol{g}_{t,[d]}\|^2}{(p_{t,d})^2} + \frac{G^2}{p_{\min}^2}$
6:      else
7:          $\tilde{l}_{t,d} = 0$
8:      end if
9:      $w_{t,d} = p_{t,d} \exp\left(-\alpha_p \tilde{l}_{t,d} / p_{t,d}\right)$
10: end for
11: $p_{t+1} = \arg\min_{q \in \mathcal{P}} D_{\mathrm{KL}}(q \| w_t)$
12: return $p_{t+1}$
```

We introduce a set $\mathcal{P}$ as $\mathcal{P} = \{p \in \mathbb{R}^D : \sum_{d=1}^{D} p_d = s, p_d \geq p_{\min}, \forall 1 \leq d \leq D\}$, where $p_{\min}$ is a constant lower bound. This set helps in guiding the update of the probability distribution. The sum of the probabilities across all parameters equals the desired sparsity budget s, with each probability bounded below by a constant $p_{\min}$. Note that s and $p_{\min}$ are parameters defined by the user, adhering to the conditions $0 < s \leq D$ and $0 < p_{\min} \leq 1$. Recall that $\boldsymbol{g}_{t,[d]}$ is defined as $\boldsymbol{g}_{t,[d]} = [0, \cdots, \boldsymbol{g}_{t,d}, \cdots, 0]$, and $\boldsymbol{g}_t = \sum_{d=1}^{D} \boldsymbol{g}_{t,[d]}$. The term α_p denotes the step size, while $D_{\mathrm{KL}}(q\|w)$ represents the KL divergence between distributions q and w.

4.3. Theoretical results

In this section, we analyze the convergence of our proposed method PREBACKRAZOR++JOINTSPAR in the context of linear networks. We use $\boldsymbol{g}_t$ to represent the gradient of the loss function $\mathcal{L}$ with respect to the parameter θ_t at iteration t, calculated using all samples in the training dataset. In contrast, $\tilde{\boldsymbol{g}}_t$ denotes the stochastic gradient computed using a single training sample. To support our analysis, we adopt the classical assumptions as detailed by Jiang et al. [24]:

Assumption 1. (1) The objective function $\mathcal{L}$ is S_d-smooth with respect to $\boldsymbol{\theta}_{[d]}$; (2) The gradient $\boldsymbol{g}_{t,[d]}$ is upper bounded by G, i.e., $G \geq \|\boldsymbol{g}_{t,[d]}\|$. $\forall t, d$; (3) The objective function $\mathcal{L}$ is bounded below by L^*; (4) For each iteration t, the stochastic gradient $\tilde{\boldsymbol{g}}_t$ adheres to the following: $\mathbb{E}[\tilde{\boldsymbol{g}}_t] = \boldsymbol{g}_t$, $\mathbb{E}[(\tilde{\boldsymbol{g}}_t - \boldsymbol{g}_t)^2_d] \leq \sigma_d^2, \forall d \in \{1, 2, \ldots, D\}$, with D denoting the number of coordinates and the constant σ_d^2 representing the variance corresponding to coordinate d.

Lemma 1 [24]. In the context of linear neural networks, the gradient of parameters at iteration t can be decomposed as $\tilde{\boldsymbol{g}}_t = \tilde{\mathbf{x}}_t \tilde{\mathbf{A}}_t^\top$, where $\tilde{\mathbf{x}}_t$ denotes the precise activation, and $\tilde{\mathbf{A}}_t$ is referred to as the transformation matrix. After pruning the activations, only the pruned activations $\tilde{\mathbf{x}}_t'$ are reserved and used for backpropagation, and the gradient approximation becomes $\tilde{\boldsymbol{g}}_t' = \tilde{\mathbf{x}}_t' \tilde{\mathbf{A}}_t^\top$.

Theorem 1 (Convergence). Denote α as the step size. Under Assumptions 1-4, the following bound holds:

$$\mathbb{E}[\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{n}\|\boldsymbol{g}_{t,i}\|^2] \leq \frac{\mathcal{L}(\theta_0) - L^*}{\alpha T} + \frac{S_\infty \alpha}{2T}\sum_{t=1}^{T}\sum_{i=1}^{n}\mathbb{E}[\sum_{d=1}^{D}\frac{\|\boldsymbol{g}_{t,[d]}\|^2}{p_{t,d}}], \tag{10}$$

where T is the total number of iterations, $S_\infty = \max_d S_d$, $\boldsymbol{\theta}_0$ is the initial model weights.

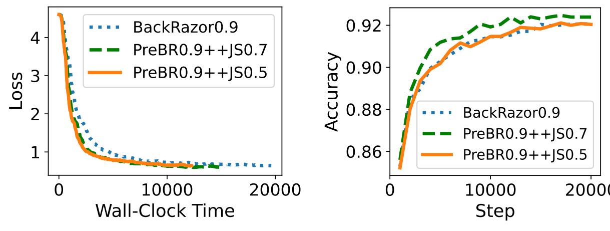
As the training progresses, the first term on the right-hand side of the equation progressively diminishes to zero, whereas the second term remains confined within the bounds set by the bandit method. This theorem establishes that our proposed method converges towards a bounded neighborhood around a stationary point. While our theoretical analysis here is confined to linear networks, the extension to non-linear networks remains an area for future investigation. In the meantime, we have empirically validated that our proposed method effectively accelerates the fine-tuning of Transformer-based models. For those interested in a more thorough exploration of the underlying mathematics, the detailed derivation of these results is provided in the appendix. Additionally, the proofs and analyses in [24] and [30] offer further insight into these concepts.

5. Experiments

We perform experiments on PyTorch [38] with RTX 4090 GPU with 24GB of VRAM. We apply our method on top of BACKRAZOR's code library to report the performance gains, following its training settings on ViT and BERT. We report the actual memory allocation on GPU measured with PyTorch CUDA tools. For better readability, we employ the abbreviation JS to denote JOINTSPAR, and PREBR as a shorthand for PREBACKRAZOR.

5.1. PREBACKRAZOR with Vision Transformer

Settings. For ViT training, we initiate the process by loading the checkpoint of the ViT-B/16 model that has been pre-trained on ImageNet-22K as our starting point. Subsequently, we fine-tune the model using the CIFAR-10 and CIFAR-100 datasets [26]. The fine-tuning process encompasses 20,000 steps and employs cosine learning rate decay, with an initial 500 steps dedicated to learning rate warmup. We utilize the standard SGD optimizer and conduct validation every 1,000 steps. The maximum learning rate is set at 0.01, and the batch size during training is 128.



(a) Training. (b) Validation.

Figure 3. Fine-tuning PREBACKRAZOR++@90% vs. BACKRAZOR@90% on CIFAR-100. (a) Training loss with wall-clock time. (b) Validation accuracy with steps.

Convergence comparison. The results of fine-tuning PREBACKRAZOR++ with parameter-wise JOINTSPAR in comparison to BACKRAZOR at a pruning rate of 90% on CIFAR-100 are illustrated in Fig. 3. In Fig. 3a, the convergence speed of PREBACKRAZOR@90%++JOINTSPAR is comparable to that of BACKRAZOR@90%, with a slight advantage in the early stages of training. Notably, we observe a marginal improvement in accuracy, specifically [0.33%, 0.05%], when adjusting the initial probabilities of JOINTSPAR to [70%, 50%], respectively. This improvement is in contrast to the performance of BACKRAZOR@90%, as illustrated in Fig. 3b.

Table 1. Comparison of fine-tuning methods for ViT.

Method	Memory Allocated	Memory Reduction	Epoch Time	Speedup	CIFAR-100	CIFAR-10	CUB-200
FULL-FT	19980MB	1.0×	2min59s	-	93.0	99.0	85.5
BACKRAZOR	7540MB	2.6×	6min19s	1.0×	92.1	98.6	**86.6**
(ours) PREBR++JS@70%	7440MB	2.7×	4min37s	1.4×	**92.4**	**98.8**	85.0
(ours) PREBR++JS@50%	**6880MB**	**2.9**×	**4min02s**	**1.6**×	92.1	98.6	85.6

Table 2. Performance of different attention implementations fine-tuning ViT on CIFAR-100.

Device	Method	Attention Type	Memory	Epoch Time	Speedup	Accuracy(%)
RTX 4090	FULL-FT	vanilla	19980MB	2min59s	-	93.0
	BACKRAZOR	BACKRAZOR	7540MB	6min19s	1.0×	92.1
	(ours) PREBR++JS	BACKRAZOR	**6570MB**	4min37s	1.4×	91.4
	(ours) PREBR++JS	FLASHATTENTION	7960MB	**3min40s**	**1.7**×	**92.2**
	(ours) PREBR++JS	**sparse** FLASHATTN	6880MB	4min02s	1.6×	92.1
RTX 2080 Ti	FULL-FT	vanilla	out of memory			
	BACKRAZOR	BACKRAZOR	7780MB	19min47s	1.0×	-
	(ours) PREBR+JS	BACKRAZOR	**6870MB**	**15min13s**	**1.3**×	-

Speedup and memory reduction. In Tab. 1, we present a comprehensive comparison among PREBACKRAZOR@90%++JOINTSPAR@[70%, 50%], BACKRAZOR@90%, and FT-Full (fine-tuning the full network) from the perspective of memory usage, speedup relative to BACKRAZOR@90%, and final accuracy on CIFAR-100, CIFAR-10 and CUB-200 for each method. Here we present the average memory allocated by CUDA during training. Compared with FT-full, BACKRAZOR@90% achieves a noteworthy 2.6× reduction in memory usage. However, it demands twice as much training time due to the additional computations required for processing activations. In contrast, PREBR++JS achieves superior computational efficiency by avoiding unnecessary computations through joint sparsification of gradients and activations. This leads to significantly 1.6× faster training without incurring additional memory costs or compromising accuracy. Additionally, Tab. 3 showcases the performance gains of forward and backward computation achieved by transitioning from a self-sparsified top-k pruning pattern to a column-wise joint-sparse pattern as in Fig. 1. These measurements are conducted on a linear layer characterized by input tensors with the shape [128, 197, 768] and of type float32. Tab. 4 illustrates the performance improvements achieved by substituting the element-wise mask in BACKRAZOR with a column-wise mask. This approach, referred to as PREBACKRAZOR-, employs a self-sparsified scheme that eschews the use of the bandit method for predicting pruning masks based on gradient information. Despite this, PREBACKRAZOR- still delivers commendable results, including a modest reduction in memory usage and over a 10% increase in speed, while preserving nearly identical accuracy, particularly at a moderate pruning ratio of 80%.

Table 3. Acceleration on a linear layer.

Sparsity	FWD time (ms) BACKRAZOR	FWD time (ms) PREBR	Speedup	FWD + BWD (ms) BACKRAZOR	FWD + BWD (ms) PREBR	Speedup
0.1	4.23	3.14	1.34×	9.02	7.72	1.17×
0.2	4.35	4.24	1.03×	9.20	9.03	1.02×
0.3	4.45	4.35	1.02×	9.34	9.18	1.02×
0.4	7.97	4.45	1.79×	16.52	9.32	1.77×
0.5	8.94	7.97	1.12×	18.45	16.51	1.12×

Table 4. Comparison of self-sparsified sparsity patterns.

Pruning Ratio	Method	Memory Peak Usage	Epoch Time	Accuracy(%)
-	FULL-FT	19980MB	2min59s	93.0
90%	BACKRAZOR	7540MB	6min19s	92.1
	(ours) PREBR-	7220MB	5min25s	91.7
80%	BACKRAZOR	8210MB	6min28s	92.9
	(ours) PREBR-	8100MB	5min49s	92.9

Integration of the FLASHATTENTION kernel. We present Tab. 2 that details the performance when switching between the original FLASHATTENTION kernel (without memory savings), sparsified FLASHATTENTION (our modified implementation), and no FLASHATTENTION (using BackRazor's attention implementation). We use (PRE)BACKRAZOR@90% with parameter-wise sparsified JOINTSPAR@50% for these settings, note that the incorporation of JOINTSPAR is specifically aimed at offsetting the dense FLASHATTENTION kernel's memory usage. Additionally, we evaluate the performance of these methods on RTX 2080 Ti GPU with 11GB of VRAM, with (PRE)BACKRAZOR@90% and JOINTSPAR@30%. Note that both FULL-FT and FLASHATTENTION-2 are not available for testing on RTX 2080 Ti as they exceed the VRAM capacity and require Ampere architecture [15], respectively. Our initiative to integrate the FLASHATTENTION kernel into the training process aims to adapt a state-of-the-art solution for enhancing training on consumer-level devices. This integration yields a significant speedup of 1.7× during training, effectively mitigating the impact of additional computation introduced when compared to vanilla implementation as in FULL-FT. However, the replacement of BACKRAZOR's sparsified attention implementation does come with more memory costs up to 1GB. In response to this challenge, we have customized FLASHATTENTION by incorporating the PREBACKRAZOR method to prune the gradients and activations. This customization serves to further reduce activations and counterbalance the increased memory usage, ultimately falling below the memory footprint of BACKRAZOR.

5.2. PREBACKRAZOR with BERT

Settings. For BERT training, we conduct fine-tuning on the *bert-base-uncased* model based on BACKRAZOR adaptations that are implemented within Huggingface's Transformers Repository [53]. We note that the model currently does not support FLASHATTENTION-2 to the best of our knowledge, from the involvement of an input attention mask in its attention algorithm which is not compatible with FLASHATTENTION kernel. We closely monitor and record the model's performance throughout 5 training epochs with batch size [8, 32] on a single GPU, which is a common case for fine-tuning on low-end devices. The selected fine-tuning task is Recognizing Textual Entailment (RTE), part of the GLUE benchmark [51]. The sparsity ratio for PREBACKRAZOR and BACKRAZOR is set to [80%, 90%].

Table 5. Performance of different fine-tuning schemes for RTE from GLUE Benchmark with BERT base.

Batch Size	Pruning Ratio	Method	Memory Peak Usage	Finish Time	Accuracy(%)
32	-	FULL-FT	16625MB	2min50s	70.40
	90%	BACKRAZOR	11865MB	6min26s	66.43
		(ours) PREBR	11871MB	6min24s	65.70
	80%	BACKRAZOR	12888MB	6min41s	65.70
		(ours) PREBR	13006MB	6min19s	66.07
8	-	FULL-FT	5575MB	4min28s	70.04
	90%	BACKRAZOR	4734MB	16min59s	66.43
		(ours) PREBR	4747MB	16min05s	69.68
	80%	BACKRAZOR	4812MB	17min39s	69.31
		(ours) PREBR	4840MB	16min36s	**72.92**

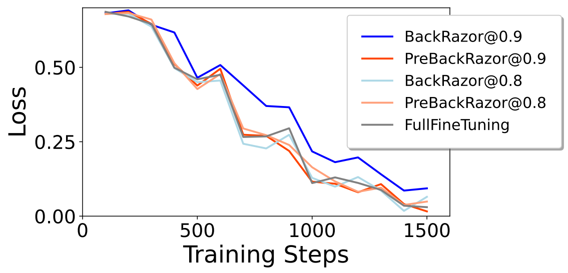


Figure 4. Training loss during fine-tuning BERT-base for RTE.

Performance. We provide a comparison Tab. 5 in which we compare fine-tuning methods, including fine-tuning the full model, element-wise BACKRAZOR, and column-wise PREBACKRAZOR. Both pruning methods successfully reduce memory usage compared to the Full-FT approach. For instance, at the pruning ratio of 90% and batch size of 32, BACKRAZOR and PREBR show similar memory usage significantly lower than Full-FT, yielding a saved memory of 28.6% (4.8GB). The wall clock time for both pruning methods is higher compared to Full-FT. However, our proposed PREBACKRAZOR tends to have a slightly shorter time than BACKRAZOR in all cases, up to 9.7%× speedup. For $bs = 8$ which is a common case for fine-tuning large language models (LLM) on consumer-level devices, the introduction of element-wise self-sparsified activation pruning methods leads to an increased time cost of up to 5×, which is not justified by the benefits obtained (12% memory reduction). PREBACKRAZOR demonstrates a slight improvement in convergence speed and reduction in loss compared to PREBACKRAZOR as shown in Fig. 4, highlighting its enhanced efficiency in fine-tuning, and outperforms not only BACKRAZOR but also Full-FT at an 80% pruning ratio, achieving the highest accuracy of 72.92%. Overall, while both pruning methods effectively reduce memory usage over Full-FT, PREBACKRAZOR tends to maintain or achieve slightly higher accuracy and speedup than BACKRAZOR, demonstrates a better balance between memory efficiency, completion time, and accuracy, particularly at lower batch sizes and higher pruning ratios.

6. Conclusion

In this work, we reveal the inefficiency of the existing element-wise self-sparsified activation pruning method. Based on this finding, we develop a computation and memory-efficient activation pruning framework called PREBACKRAZOR for jointly pruning activations and gradients. We have demonstrated that integrating PREBACKRAZOR with efficient attention kernels can significantly enhance the accuracy of existing methods. In addition, we also theoretically prove that our method has the same convergence rate as BACKRAZOR. However, while PREBACKRAZOR has exhibited performance improvements in ViT and BERT fine-tuning, its applicability to larger foundation models warrants further investigation. Additionally, theoretical proofs and analyses for its effectiveness in such models require continued research efforts.

Acknowledgements

This work is supported by STI 2030—Major Projects (No. 2021ZD0201405), NSFC No. 62222117, the Fundamental Research Funds for the Central Universities under contract WK3490000005.

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017. 2

[2] Arash Ardakani, Carlo Condo, and Warren J Gross. Activation pruning of deep convolutional neural networks. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1325–1329. IEEE, 2017. 3

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002. 5

[4] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018. 2

[5] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. 1

[6] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Freezeout: Accelerate training by progressively freezing layers. *arXiv preprint arXiv:1706.04983*, 2017. 3

[7] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce activations, not trainable parameters for efficient on-device learning. *arXiv preprint arXiv:2007.11622*, 2020. 1, 2

[8] Ayan Chakrabarti and Benjamin Moseley. Backprop with approximate activations for memory-efficient network training. *Advances in Neural Information Processing Systems*, 32, 2019. 1, 3

[9] Congliang Chen, Li Shen, Haozhi Huang, and Wei Liu. Quantized adam with error feedback. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 12(5):1–26, 2021. 2

[10] Congliang Chen, Li Shen, Fangyu Zou, and Wei Liu. Towards practical adam: Non-convexity, convergence theory, and mini-batch acceleration. *Journal of Machine Learning Research*, 23(229):1–47, 2022. 3

[11] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, pages 1803–1813. PMLR, 2021. 3

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. 2

[13] Feng Cheng, Mingze Xu, Yuanjun Xiong, Hao Chen, Xinyu Li, Wei Li, and Wei Xia. Stochastic backpropagation: A memory efficient strategy for training video models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8301–8310, 2022. 2

[14] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021. 4

[15] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 48–50. IEEE, 2021. 7

[16] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. 2

[17] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. 2

[18] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023. 2

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 1

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1

[21] Chao Fang, Wei Sun, Aojun Zhou, and Zhongfeng Wang. Efficient n: M sparse dnn training using algorithm, architecture, and dataflow co-design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023. 3, 4

[22] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 3

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1): 6869–6898, 2017. 1

[24] Ziyu Jiang, Xuxi Chen, Xueqin Huang, Xianzhi Du, Denny Zhou, and Zhangyang Wang. Back razor: Memory-efficient transfer learning by self-sparsified backpropagation. *Advances in Neural Information Processing Systems*, 35: 29248–29261, 2022. 1, 3, 6

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3

[26] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 2, 6

[27] Bingrui Li, Jianfei Chen, and Jun Zhu. Memory efficient optimizers with 4-bit states. *arXiv preprint arXiv:2309.01507*, 2023. 2

[28] Tianjian Li, Haoran Xu, Philipp Koehn, and Kenton Murray. Efficiently harnessing parameter importance for better training. 2023. 3

[29] Junjie Liu, Zhe Xu, Runbin Shi, Ray CC Cheung, and Hayden KH So. Dynamic sparse training: Find efficient sparse

network from scratch with trainable masked layers. *arXiv preprint arXiv:2005.06870*, 2020. 3

[30] Rui Liu and Barzan Mozafari. Communication-efficient distributed learning for large batch optimization. In *International Conference on Machine Learning*, pages 13925–13946. PMLR, 2022. 2, 3, 4, 5, 6

[31] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. 1

[32] Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. Adalomo: Low-memory optimization with adaptive learning rate. *arXiv preprint arXiv:2310.10195*, 2023. 2

[33] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv:2306.09782*, 2023. 2

[34] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *arXiv preprint arXiv:2305.17333*, 2023. 2

[35] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. 2

[36] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *International Conference on Machine Learning*, pages 4646–4655. PMLR, 2019. 3

[37] Zizheng Pan, Peng Chen, Haoyu He, Jing Liu, Jianfei Cai, and Bohan Zhuang. Mesa: A memory-saving training framework for transformers. *arXiv preprint arXiv:2111.11124*, 2021. 1, 3

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 6

[39] Jeff Pool. Accelerating sparsity in the nvidia ampere architecture. *GTC 2020*, 2020. 4

[40] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. *Advances in neural information processing systems*, 30, 2017. 3

[41] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020. 1, 2

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021. 2

[43] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018. 2

[44] Li Shen, Yan Sun, Zhiyuan Yu, Liang Ding, Xinmei Tian, and Dacheng Tao. On efficient training of large-scale deep learning models: A literature review. *arXiv preprint arXiv:2304.03589*, 2023. 1

[45] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. *Advances in Neural Information Processing Systems*, 31, 2018. 2

[46] Hao Sun, Li Shen, Qihuang Zhong, Liang Ding, Shixiang Chen, Jingwei Sun, Jing Li, Guangzhong Sun, and Dacheng Tao. Adasam: Boosting sharpness-aware minimization with adaptive learning rate and momentum for training deep neural networks. *Neural Networks*, 169:506–519, 2024. 3

[47] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023. 2

[48] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *International Conference on Machine Learning*, pages 3299–3308. PMLR, 2017. 3

[49] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. 2, 3

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 1

[51] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018. 8

[52] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020. 3

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019. 8

[54] Qing Yang, Wei Wen, Zuoguan Wang, and Hai Li. Joint regularization on activations and weights for efficient neural network pruning. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 790–797. IEEE, 2019. 3

[55] Yuxin Zhang, Yiting Luo, Mingbao Lin, Yunshan Zhong, Jingjing Xie, Fei Chao, and Rongrong Ji. Bi-directional masks for efficient n: M sparse training. *arXiv preprint arXiv:2302.06058*, 2023. 3

[56] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of adam and rmsprop. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pages 11127–11135, 2019. 3