Back Razor: Memory-Efficient Transfer Learning by Self-Sparsified Backpropagation

Ziyu Jiang†, Xuxi Chen‡, Xueqin Huang†, Xianzhi Du‡, Denny Zhou§, Zhangyang Wang‡
†Texas A&M University  ‡University of Texas at Austin  §Google
{jiangziyu, xueq13}@tamu.edu, {xxchen, atlaswang}@utexas.edu
{xianzhi, dennyzhou}@google.com

Abstract
Transfer learning from the model trained on large datasets to customized downstream tasks has been widely used as the pre-trained model can greatly boost the generalizability. However, the increasing sizes of pre-trained models also lead to a prohibitively large memory footprints for downstream transferring, making them unaffordable for personal devices. Previous work recognizes the bottleneck of the footprint to be the activation, and hence proposes various solutions such as injecting specific lite modules. In this work, we present a novel memory-efficient transfer framework called Back Razor, that can be plug-and-play applied to any pre-trained network without changing its architecture. The key idea of Back Razor is asymmetric sparsifying: pruning the activation stored for back-propagation, while keeping the forward activation dense. It is based on the observation that the stored activation, that dominates the memory footprint, is only needed for back-propagation. Such asymmetric pruning avoids affecting the precision of forward computation, thus making more aggressive pruning possible. Furthermore, we conduct the theoretical analysis for the convergence rate of Back Razor, showing that under mild conditions, our method retains the similar convergence rate as vanilla SGD. Extensive transfer learning experiments on both Convolutional Neural Networks and Vision Transformers with classification, dense prediction, and language modeling tasks show that Back Razor could yield up to 97% sparsity, saving 9.2x memory usage, without losing accuracy. The code is available at: https://github.com/VITA-Group/BackRazor_Neurips22.

1 Introduction
Edge devices equipped with deep learning applications are becoming ubiquitous. However, only deploying pre-trained models is not sufficient: these edge devices may keep collecting novel data that general pre-trained models have never seen before, and fine-tuning on them would notably improve the model’s performance. A typical approach is to upload these newly collected data to cloud servers for fine-tuning, and download the trained model from the cloud to local devices. However, such a way has potential privacy issues due to data sensitivity and also imposes undesirable pressures on the Internet bandwidth, brought by transmitting both the collected data and trained models. These challenges motivate researchers to explore on-device learning methods, i.e., how to fine-tune pre-trained models on memory-limited devices.

One of the main challenges of fine-tuning on edge devices is the memory constraints. Figure 1 exemplifies the memory usage of training some popular networks: ResNet50 with a small batch size of 16 can easily exceed the memory limitation of onboard devices. The newly emerged Vision

*Equal contribution

Transformer (ViT) would incur even more memory consumption. Recent studies show that the activations stored for back-propagation take up a large proportion of memory cost during training. Therefore, popular model compression techniques such as weight pruning and quantization are not highly efficient in reducing training memory footprints. Parameter efficient training method like Bitfit also fails to compress the activation to memory limitation (see Figure 1). Gradient checkpointing can reduce the training memory by storing only a subset of activations, but requires more FLOPs at back-propagation due to the re-computation of discarded activations. Using a smaller batch size or half-precision for training can also lower the memory cost, but they incur a decrease in the model’s accuracy and training speed. [1] successfully avoided saving activation via freezing the weights for convolutional neural networks and only learning a newly added the lite residual module, thus no need to store the intermediate activations. However, it comes at a certain price of the transfer performance.

A new stream of works studied how to directly compress the activations. They mainly leverage the quantization technique that discretizes the activation tensors so these tensors can be represented by efficient data formats and require less memory [2, 3, 4]. However, the activation pruning techniques have been less explored. [5] applied random activation masks on each layer in both forward and backward passes to reduce latency. [6] integrated weight pruning and activation pruning by learning importance scores on neuron’s outputs. Moreover, they adopt the symmetric scheme that prunes the activation at both the forward and backward processes. Such a pruning scheme undermines the precision of models’ outputs, leading to significantly worse performance when the pruning ratio is high.

In this work, we present a general activation compression method called Back Razor. Inspired by the observations that the activations are only stored for back-propagation, we adopt the asymmetric pruning scheme that only prunes the activation during the backward process. Specifically, after activation is used by a subsequent layer, our method replaces it with a pruned activation which carries a lower memory cost. During the backward phase, the gradients are calculated with the approximated activation, but only with a small error between those with the exact activation.

Our contributions are summarized as following:

- We propose a memory-efficient transfer learning method called Back Razor that prunes the activation for back-propagation. Our proposed method can be easily applied to various backbones, including convolutional neural networks (CNNs) and vision transformers (ViTs).
- We provide a theoretical analysis of our self-sparsified back-propagation method for CNNs. Specifically, we discover that under mild assumptions, our method can share a similar convergence rate as vanilla SGD.
- We extensively verify the proposed Back Razor on both CNNs and ViTs for multiple datasets and show that the proposed method can achieve up to 97% sparsity with saving 9.2× memory saving without losing accuracy. By further verifying on GPU, the proposed method achieves 2.7× on-device memory efficiency for ViTs.

2 Related Works

2.1 Pruning and Sparse Training

Pruning can be operated at different levels, e.g., pruning on weight, weight gradients, activations, and activation gradients. Pruning on weights are most widely studied since it can accelerate the inference
phase [7, 8, 9]: pruning on weight gradients can help reduce communication cost especially in distributed training [10, 11]; pruning on activation gradients [12, 13, 14, 15] or activation itself have been less addressed since they are mostly dynamic during training. [5] randomly generated static sparse masks and applied them on each layer in both forward and backward passes. [6] proposed integral pruning that prunes both activation and weights by consecutively learning and applying two masks on weights and activations. However, these works are adopting the symmetric pruning scheme, i.e., pruning the activation at both the forward and the backward process. Pruning the activations at each layer brings errors, and they will accumulate as the forwarding proceeds. Backdrop [16] improves the generalizability of large batch gradient descent via randomly masking parts of the backward gradient propagation. SWAT [17] empirically explores sparsifying both weights and activations for training CNNs from scratch, and the authors also discovered that pruning activations only for backpropagation can effectively preserve accuracy better. While our idea shares similarity to SWAT, Back Razor is dedicated to transfer learning and is applicable to general backbones (both CNNs and ViTs), and we also supply the theoretical analysis for such self-sparsified backpropagation for the first time.

2.2 Memory Efficient Transfer Learning

Transfer Learning: From a model pre-trained on large datasets to small datasets has been widely used for many applications [18, 19, 20, 21]. In contrast to the pre-training that is always conducted on large clusters, transfer learning may happen on personal computing devices with limited memory constraints. To fit the large model to small devices, a common choice is to fine-tune the last several layers of the model [22, 18, 23, 24]. However, it can be a sub-optimal choice, especially for the downstream tasks with large distribution differences. In contrast, fine-tuning the whole can yield better accuracy in most cases [1, 25]. However, it can easily lead to memory outages. On the other hand, recent parameter efficient transfer learning algorithms [26, 27, 28, 29] only require tuning less than 1% of the parameters. But this parameter efficiency only brings a mild memory efficiency. The most relevant prior work is TinyTL [1], it recognizes the main bottleneck of memory is in activation and introduces a new trainable module that only requires down-sampled activation. However, it still focuses on exact gradient calculation while we employ approximate gradient calculation, which enables compressing activation without changing the model architecture. A few other works have also explored the approximate activation techniques. Most of them focus on activation quantization [2, 3, 30, 31]. Recently, activation compression is extended to ViTs [32] and Graph Neural Networks [33].

3 Method

3.1 Activation is the bottleneck of memory

The forward of neural networks can be express as following

\[
f(x; \theta) = f_l(f_{l-1}(\ldots f_0(x; \theta_0)\ldots; \theta_{l-1}); \theta_l)
\]

\[L = \mathcal{L}(f(x; \theta), y)\]  

(1)
where $x$ and $y$ is the input and label, respectively. $L$ is the loss between prediction $f(x; \theta)$ and label $y$ calculated by loss function $\mathcal{L}$. $f_i$ and $\theta_i$ denote the function and parameters of $i$th layer, respectively. The network is composed of $l$ layers. We further denote the output of $i$th layer (a.k.a activation) as $z_i = f_i(z_{i-1}, \theta_i)$ and $z_1 = f_1(x; \theta_1).$ When conducting back-propagation, $\frac{\partial L}{\partial z_i}$ and $\frac{\partial L}{\partial \theta_i}$ are required to be calculated for each activation.

Taking linear layer with $z_i = z_{i-1} W_i + b_i (\theta_i = [W_i, b_i])$ as an example, the gradients required to be calculated are [1]:

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} W_i^\top, \quad \frac{\partial L}{\partial W_i} = z_i^\top \frac{\partial L}{\partial z_{i+1}}, \quad \frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_{i+1}}$$

As shown in Equation 2, the activation $z_i$ is employed for computing one of the gradients. This creates the need of saving the activation of the forward process. Otherwise, extra FLOPs are required for re-computing the activation. The size of activation is always larger than the weight as weight is always shared across different patches or tokens. Also, the size of activation can increase linearly with the batch size. Moreover, in the transformer, the length of input tokens can also lead to a quadratic increase of the memory.

**Discussion about freezing weight.** TinyTL [1] pointed out that we can avoid saving activation via freezing the weight $W_i$ as other terms do not require the participation of $z_i.$ This is true for most architectures of convolutional neural networks. However, the emergence of ViT breaks this assumption. Many operations (e.g. self-attention, Softmax, GeLU [34]) in transformer would involve the activation for computing $\frac{\partial L}{\partial z_i},$ which cannot be avoided if previous layers requires updating. Therefore we adopt the more general method: directly sparsifying the activation for backpropagation.

### 3.2 Back-propagation activation self-sparsification

In contrast to the activation sparsification [5] that prunes the activation of both forward and backward, our proposed Back Razor leverages the back-propagation activation sparsification: it only prunes the activation that is used for backward while keeping the forward activation dense as shown in Figure 2. The normal backward process can be formally defined as

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} h_i^{(z)}(z_i, \theta_i), \quad \frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial \theta_i} = \frac{\partial L}{\partial z_{i+1}} h_i^{(\theta)}(z_i, \theta_i)$$

where $h_i^{(z)}(z_i, \theta_i) := \frac{\partial z_{i+1}}{\partial z_i}$ and $h_i^{(\theta)}(z_i, \theta_i) := \frac{\partial z_{i+1}}{\partial \theta_i},$ representing the derivatives of $z_{i+1}$ with respect to $z_i$ and $\theta_i$, respectively. For Back Razor, the backward process can be expressed as

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} h_i^{(z)}(\hat{z}_i, \theta_i), \quad \frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial z_{i+1}} h_i^{(\theta)}(\hat{z}_i, \theta_i),$$

where $\hat{z}_i$ denotes the pruned activation $z_i.$ We inherit the mathematical forms behind $h_i^{(z)}$ and $h_i^{(\theta)}$ from the normal backward to preserve the gradient flow, but calculate the gradients on weights and activations with the pruned layerwise activation $\hat{z}_i.$ With the above backward, there is no need for saving the dense activation $z_i.$ Instead, we store a sparse version $\hat{z}_i$ that is more memory efficient.

The detailed optimization algorithm with the proposed Back Razor can be found in Algorithm 1. In the forward pass, we would compute with dense activation while only saving its sparse version. In the backward, we employ the sparse activation for computing the gradient.

We use a simple pruning method for backward activation pruning: prune the smallest magnitude activations. The smallest magnitude values are usually playing less significant roles than those with larger magnitude. This pruning method is also widely used for many pruning works [35]. In practice, we would prune all the activations to a fixed sparsity of $\lambda$ by setting all the values under $k$th smallest value as zero, where $k = \lambda n, n$ is the total number of values for the activation. As the ranking can be slow for large activations, instead of ranking the whole activation, we rank the activation of each sample and use the sample-wise threshold for pruning. This largely shortens the ranking time. Also, we noted that the thresholds across different samples are very close, which means it is functionally similar to global ranking.

We save the sparse matrix into two parts: a bitmap that indicates the position of the non-zero elements and a smaller dense matrix that contains the values of the non-zero elements. The bitmap has the
The objective

**Assumption 1.** Let \( \theta \) be the gradient of the objective \( \mathcal{L} \) at point \( \theta \), then for all \( x \) and \( y \) we assume there exists a non-negative constant vector \( \beta \) that

\[
|\mathcal{L}(y) - [\mathcal{L}(x) + g(x)(y - x)\top]| \leq \frac{1}{2} \sum_j \beta_j (y - x)_j^2
\]

**Assumption 2.** For each iteration \( t \), the stochastic gradient \( \tilde{g}_t \) satisfies the following conditions:

\[
\mathbb{E}[\tilde{g}_t] = g_t, \quad \mathbb{E}[(\tilde{g}_t - g_t)_j^2] \leq \sigma^2_j, \forall j \in [n],
\]

where \( n \) is the number of parameters of models and \( \sigma^2_j \) is a constant representing the variance for coordinate \( j \).

**Assumption 3.** The objective \( \mathcal{L} \) is bounded below by \( L^* \).

These assumptions are discussed in Appendix. We first prove the convergence for linear networks.

**Lemma 1.** For linear neural networks, the gradient of parameters can decomposed by \( \tilde{g}_t = \tilde{z}_t A_t\top \), and approximated gradient after activation pruning can decomposed by \( \tilde{g}'_t = \tilde{z}'_t A_t\top \), where \( \tilde{z}_t \) is the exact activation and \( \tilde{z}'_t \) is the pruned activations.

\begin{algorithm}
\caption{Backward and Update with Sparse Activation}
\begin{algorithmic}
\State Sample \( x, y \) from \( \mathcal{D} \)
\State Initialize \( z_0 = x \)
\For {\( i = 1 \) to \( l \)} \Comment{Forward pass.}
\State \( z_i = f_i(z_{i-1}, \theta_i) \)
\State \( \tilde{z}_{i-1} \leftarrow \text{TopK}_\text{Prune}(z_{i-1}) \) \Comment{Sparsify \( z_{i-1} \) and store it.}
\EndFor
\State Calculate the training loss \( L = \mathcal{L}(z_t, y) \)
\State Initialize \( s_t \leftarrow \frac{\partial L}{\partial z_t} \)
\For {\( i = l - 1 \) to \( 1 \)} \Comment{Backward pass.}
\State Calculate \( \frac{\partial L}{\partial z_i} : s_i \leftarrow s_{i+1} h_i^\top(\tilde{z}_i, \theta_i) \) \Comment{Calculate the gradient for the next activation}
\State Calculate \( \frac{\partial L}{\partial \theta_i} : s_{i+1} h_i(\tilde{z}_i, \theta_i) \) and update \( \theta_i \).
\EndFor
\end{algorithmic}
\end{algorithm}
In linear networks, $\frac{\partial L}{\partial z_i}$ only depends on the model’s weights after the $i$th layer, so pruning the activation stored for backward does not change the its gradient. Therefore in Lemma 1, $\tilde{A}_i$ is exclusively constructed by activations’ gradients, and not changed after activation pruning.

We call $\tilde{A}_t$ the transformation matrix for activations at step $t$.

**Theorem 1** (Convergence). If Assumption 1-3 hold, $(p + \alpha \beta)K_a < 1$ where $p$ is the pruning ratio and $K_a$ is the squared condition number of the transformation matrix, then

$$
\mathbb{E}\left[ \frac{1}{T} \sum_{t=1}^{T} \|g_t\|^2 \right] \leq \frac{2(L(\theta_0) - L^*)}{1 - (p + \alpha \beta)K_a} \alpha T + \frac{(p + \alpha \beta)K_a}{1 - (p + \alpha \beta)K_a} \sigma^2,
$$

where the $T$ is the number of iterations, $\beta = \|\beta\|_\infty$, $\theta_0$ is the initial weights and $\sigma^2 = \sum_{j=1}^{n} \sigma_j^2$.

The Eqn. 5 consists of two terms: the first term will converge to zero as the number of iterations $T$ goes to infinity. The second term will not vanish, but its value can be controlled by using larger batch size. Intuitively, it proves that our algorithm reach to the neighborhood of a stationary point, and the radius of the neighborhood is bounded by the gradient variance.

**Remark.** The key factor for theoretical convergence is $(p + \alpha \beta)K_a$. $\alpha \beta$ can be reduced by using smaller learning rate. The theoretical convergence will not be guaranteed when the transformation matrix is ill-conditioned.

We next discuss the convergence for non-linear networks, and focus on analyzing convolutional neural networks with a common CONV-BN-ReLU structure [15]. The activation pruning happens after the convolutional layer, where the gradient on weights and activations can be calculated as [15]:

$$
\frac{\partial L}{\partial z_i} = W_i^T \ast \frac{\partial L}{\partial z_{i+1}}, \quad \frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial z_{i+1}} \ast z_i,
$$

which suggests the gradient on $z_i$ will not be affected if we use the pruned activation $\tilde{z}_i$ to obtain the derivatives on $\theta_i$ when back-warding through the convolutional layer. The activations are neither stored nor pruned for BN and ReLU operations, so the Lemma 1 holds for convolutional neural networks. So under the same assumptions, we achieve similar convergence situation for CNNs. We present more details in Appendix.

We lastly discuss the Transformer structures. Unfortunately, the self-attention module does not meet the Lipchitz condition [39], therefore the above theoretical analysis will not apply out-of-the-box to them. However we empirically observe that Back Razor is also applicable on ViTs [40] with decent performance (refer to Section 4.3). A more rigorous theoretical framework customized for transformers is left as our future work.

## 4 Experiment

### 4.1 Settings

Following the common practice [26, 41, 42], we employ supervised pre-trained models on ImageNet as the starting point of transfer learning. Specifically, we employ ImageNet-1K and Imagenet-22K for CNNs and ViTs, respectively. For downstream fine-tuning, we consider eight datasets: Pets [43], Aircraft [44], CIFAR10, CIFAR100 [45], Flowers [46], Cars [47], CUB [48], and Food [49]. The default image resolution is $224 \times 224$ following [1].

Our experiments are implemented with Pytorch [50] and conducted on 1080 Ti or V100 GPUs. To measure the training memory, by default we report the theoretical memory usage following [1]. We also report the actual memory usage measured on the GPU at Section 4.3.

### 4.2 Back Razor with Convolutional Neural Networks

We choose ProxylessNAS-Mobile [52] as the CNN backbone following TinyTL [1]. It is worth noting that we keep the origin setting of ProxylessNAS-Mobile instead of modifying the architecture as in TinyTL given Back Razor is applicable for any architecture. We also follow the most of training settings of TinyTL for fair comparison: The fine-tuning epochs and batch size is set as 50 epochs.
Table 1: The comparison between Back Razor with the previous methods on different datasets with CNN. All the reported results are the top1 accuracy (%). FT-Last denotes fine-tuning the linear classifier (the last fully connected layer). FT-Norm+Last denotes fine-tuning the batch normalization layers plus the linear classifier [37,51]. TinyTL@320 denotes training and inference with a larger resolution of 320. The Back Razor@90% denotes the Back Razor with a sparsity ratio of 90%. The memory footprint of the training at a batch size of 8 (test in CIFAR100) is also reported in the second column. Some experiments are conducted five times with different random seeds. The mean and variance are reported.

<table>
<thead>
<tr>
<th>Method</th>
<th>Train Memory</th>
<th>Pets</th>
<th>Aircraft</th>
<th>CIFAR10</th>
<th>CIFAR100</th>
<th>Flowers</th>
<th>Cars</th>
<th>CUB</th>
<th>Food</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT-Last</td>
<td>31MB</td>
<td>91.3</td>
<td>44.9</td>
<td>85.9</td>
<td>68.8</td>
<td>90.1</td>
<td>50.9</td>
<td>73.3</td>
<td>68.7</td>
</tr>
<tr>
<td>FT-Norm+Last</td>
<td>192MB</td>
<td>92.2</td>
<td>68.1</td>
<td>94.8</td>
<td>80.2</td>
<td>94.3</td>
<td>77.9</td>
<td>76.3</td>
<td>77.0</td>
</tr>
<tr>
<td>FT-Full</td>
<td>366MB</td>
<td>93%</td>
<td>88.2</td>
<td>97.1</td>
<td>84.1</td>
<td>97.0</td>
<td>91.0</td>
<td>89.0</td>
<td>83.8</td>
</tr>
<tr>
<td>TinyTL</td>
<td>37MB</td>
<td>91.8</td>
<td>75.4</td>
<td>95.9</td>
<td>81.4</td>
<td>95.5</td>
<td>85.0</td>
<td>77.1</td>
<td>79.7</td>
</tr>
<tr>
<td>TinyTL@320</td>
<td>65MB</td>
<td>92.9</td>
<td>82.3</td>
<td>96.1</td>
<td>81.5</td>
<td>96.8</td>
<td>88.8</td>
<td>81.0</td>
<td>82.9</td>
</tr>
<tr>
<td>Back Razor@90%</td>
<td>42MB</td>
<td>92%</td>
<td>87.6</td>
<td>96.9</td>
<td>83.3</td>
<td>96.9</td>
<td>85.1</td>
<td>81.0</td>
<td>83.2</td>
</tr>
<tr>
<td>Back Razor@95%</td>
<td>41MB</td>
<td>93.2</td>
<td>85.7</td>
<td>96.8</td>
<td>82.7</td>
<td>96.7</td>
<td>87.2</td>
<td>79.5</td>
<td>83.1</td>
</tr>
<tr>
<td>Back Razor@97%</td>
<td>40MB</td>
<td>93.0</td>
<td>85.2</td>
<td>96.6</td>
<td>82.6</td>
<td>95.9</td>
<td>88.1</td>
<td>78.8</td>
<td>82.3</td>
</tr>
<tr>
<td>Back Razor@99%</td>
<td>39MB</td>
<td>92.5</td>
<td>80.7</td>
<td>96.1</td>
<td>80.2</td>
<td>95.0</td>
<td>85.6</td>
<td>76.9</td>
<td>80.4</td>
</tr>
</tbody>
</table>

and 8, respectively. The model is optimized with adam [53] optimizer and cosine learning learning rate schedule [54]. The initial learning rate is tuned for each dataset. We freeze all the parameters of Batch Normalization layers [55] as they occupy more memory than convolutional layers while having much fewer parameters to update. By freezing them, we could free the occupied memory with less influence on the fine-tuning performance. We also use the origin activation for ReLU6, which is memory efficient (equal to the memory cost of a bitmap with the same shape). The Back Razor is then applied to convolutional layers and the fully-connected classification head.

**Comparison Results:** In Table 1 we compare Back Razor with previous transfer learning methods on CNNs. Specifically, it includes i) FT-Last: fine-tuning the last fully connected layer of the network, which is also referred as linear classification head. ii) FT-Norm+Last: fine-tuning the batch normalization layers plus the classification head iii) FT-Full: fine-tuning the full model. Also, we include two different variants of the previous state-of-the-art (SOTA) method [1]: TinyTL and TinyTL@320 (TinyTL@320 employs a larger resolution image of 320).

Compared with FT-Full, Back Razor@90% use 8.7x times smaller memory while achieving comparable performance. The variance for BackRazor@90% with respect to different random seeds is also comparable with FT-Full and they are both small. In contrast, FT-Norm+Last takes 4.6x time more memory while being [0.7%, 19.5%, 2.1%, 3.1%, 2.6%, 12.2%, 4.7%, 6.2%] worse than Back Razor@90% in [Pets, Aircraft, CIFAR10, CIFAR100, Flowers, Cars, CUB, Food], respectively. Though FT-Last does yield slightly less training memory (31MB v.s. 42MB), the performance largely degrades: the accuracy is lower by [1.6%, 42.7%, 11.0%, 14.5%, 6.8%, 39.2%, 7.7%, 14.5%] than Back Razor@90% in [Pets, Aircraft, CIFAR10, CIFAR100, Flowers, Cars, CUB, Food], respectively.

Compared with the previous State-of-The-Art method TinyTL, Back Razor@90% is comparable in terms of the training memory usage (37MB v.s 42MB). Meanwhile, it yields higher accuracy of [1.1%, 12.2%, 1.0%, 1.9%, 1.4%, 5.1%, 3.9%, 3.5%] at [Pets, Aircraft, CIFAR10, CIFAR100, Flowers, Cars, CUB, Food], respectively. When compared with TinyTL@320, which employs a larger resolution, Back Razor@90% saves 1.5 times of the memory while still yielding an improvement of [0.0%, 5.3%, 0.8%, 1.8%, 0.1%, 1.3%, 0.0%, 0.3%] at [Pets, Aircraft, CIFAR10, CIFAR100, Flowers, Cars, CUB, Food], respectively.

It’s also worth noting that Back Razor method can even achieve more aggressive sparsity. For instance, the performance of Back Razor is [92.9%, 93.2%, 93.0%, 92.5%] for pruning ratio of [90%, 95%, 97%, 99%] at Pets, respectively. The performance maintains the same level even with the pruning ratio of 97% and only marginally drops at the pruning ratio of 99%. A similar trend is also observed in other datasets. However, the high pruning ratio does not bring further memory reduction since the main bottleneck of memory is no longer the accumulated activations. Instead, it is the model parameters (of 11.6MB) and one large forward activation (of 24.5MB).

**Can we prune at forward pass?** We further compare the performance difference between activation pruning (prune both forward and backward) with Back Razor (only prune backward) in Figure 3. We implement the activation pruning via directly applying the mask of the Back Razor to the forward pass. It can be observed that the model can easily collapse (the model can hardly converge and only
Figure 3: Comparison between pruning both forward and backward with only pruning backward under different pruning ratios on (a) Aircraft. (b) Pet. For both figures, the y axis in the left is the top 1 accuracy (%) while the y axis in the right is the memory usage, and x-axis is the pruning ratio. The line is coupled with y axis that is with the same color. The yellow line in both figures denotes the memory usage in different pruning ratios. Note that two methods share the same line as their memory consumption is the same under the same pruning ratio.

have accuracy below 5%) for activation pruning even with a mild pruning ratio of 70%. In contrast, for Back Razor, the performance can be at a high level even with an aggressive pruning ratio of 99% (the performance is even higher than activation prune at a sparsity of 50%). This demonstrates that the backward activation can be more sparse than that of the forward.

Convergence speed in practice: As shown in Figure 4, the convergence speed in transfer experiments is comparable with each other. In the early stage of the training, the convergence speed of Back Razor is even faster. Even the last stage, FT-Full is only marginally better than Back Razor.

Comparison with SWAT: We further compare with SWAT [17] which leverages the similar insight on the sparsity of backward activation. Although [17] was originally developed for training from scratch, here we apply it to the identical fair setting of transfer learning. As shown in Table 2 while similar memory footprint is employed for Back Razor@90% and SWAT@80%, the proposed BackRazor can yield a higher performance by 0.6% and 2.6% on CIFAR10 and CIFAR100, respectively.

4.3 Back Razor with Vision Transformer

In this section, we study Back Razor on ViT [40]. The patch size is by default set as 16. We employ the standard SGD optimizer with cosine learning rate decay for finetuning. The training steps are fixed as 20k and the initial learning rate is tuned for each dataset. We employ a larger batch size of 128 following the common practice for accelerating training [40, 56, 32].

Figure 4: The training loss convergence speed comparison between Back Razor@0.9 and FT-Full. The convergence speed of them is comparable.
Table 2: Comparison between BackRazor and SWAT [17] on CIFAR10 and CIFAR100 with Resnet-18. The memory footprint of training at a batch size of 128 is reported in the second column. We reproduce SWAT with the official code under fine-tuning setting for fair comparison.

<table>
<thead>
<tr>
<th>Method</th>
<th>Train Memory</th>
<th>CIFAR10</th>
<th>CIFAR100</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT-Full</td>
<td>164.4MB</td>
<td>96.5</td>
<td>82.0</td>
</tr>
<tr>
<td>SWAT@80%</td>
<td>42.7MB</td>
<td>95.8</td>
<td>79.6</td>
</tr>
<tr>
<td>SWAT@90%</td>
<td>31.7MB</td>
<td>95.3</td>
<td>77.7</td>
</tr>
<tr>
<td>Back Razor@90%</td>
<td>41.7MB</td>
<td>96.4</td>
<td>82.2</td>
</tr>
</tbody>
</table>

Table 3: The comparison between Back Razor with the previous methods on different datasets with ViT-B/16. All the reported results are the top1 accuracy (%). The memory footprint of the training at a batch size of 128 (compute in CIFAR100) is reported in the second column. Back Razor@80% + Mesa denotes combining the BackRazor with Mesa.

<table>
<thead>
<tr>
<th>Method</th>
<th>Train Memory</th>
<th>Pets</th>
<th>Aircraft</th>
<th>CIFAR10</th>
<th>CIFAR100</th>
<th>Flowers</th>
<th>Cars</th>
<th>CUB</th>
<th>Food</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT-Last</td>
<td>525MB</td>
<td>91.6</td>
<td>44.4</td>
<td>96.8</td>
<td>86.5</td>
<td>99.3</td>
<td>57.4</td>
<td>85.6</td>
<td>86.3</td>
</tr>
<tr>
<td>FT-Full</td>
<td>1923MB</td>
<td>94.1</td>
<td>78.8</td>
<td>99.0</td>
<td>93.1</td>
<td>99.5</td>
<td>85.5</td>
<td>85.5</td>
<td>90.3</td>
</tr>
<tr>
<td>Bitfit [27]</td>
<td>9460MB</td>
<td>94.0</td>
<td>74.2</td>
<td>98.8</td>
<td>92.3</td>
<td>99.5</td>
<td>80.6</td>
<td>85.7</td>
<td>89.7</td>
</tr>
<tr>
<td>Mesa [32]</td>
<td>5442MB</td>
<td>93.8</td>
<td>77.2</td>
<td>98.9</td>
<td>92.8</td>
<td>99.4</td>
<td>83.3</td>
<td>85.8</td>
<td>90.5</td>
</tr>
<tr>
<td>Back Razor@80%</td>
<td>4565MB</td>
<td>93.8</td>
<td>77.3</td>
<td>98.9</td>
<td>92.9</td>
<td>99.4</td>
<td>84.0</td>
<td>86.5</td>
<td>90.5</td>
</tr>
<tr>
<td>Back Razor@90%</td>
<td>3912MB</td>
<td>93.3</td>
<td>75.7</td>
<td>98.9</td>
<td>91.8</td>
<td>99.4</td>
<td>83.4</td>
<td>86.6</td>
<td>89.8</td>
</tr>
<tr>
<td>Back Razor@95%</td>
<td>3496MB</td>
<td>92.4</td>
<td>74.3</td>
<td>98.8</td>
<td>90.0</td>
<td>99.4</td>
<td>81.5</td>
<td>86.1</td>
<td>88.7</td>
</tr>
<tr>
<td>Back Razor@80% + Mesa (ours)</td>
<td>2937MB</td>
<td>93.5</td>
<td>76.9</td>
<td>99.0</td>
<td>92.7</td>
<td>99.4</td>
<td>84.1</td>
<td>86.5</td>
<td>90.5</td>
</tr>
</tbody>
</table>

Comparison Results: In Table 3 we compare Back Razor with previous transfer learning methods on ViT, including FT-Last and FT-Full same as defined in Section 4.3. We further compare two parameter efficient transfer learning method from the literature: Bitfit [27], and the SOTA ViT transfer learning method, Mesa [27] that is based on backpropagation quantization.

Table 4: Comparison of the on-device memory usage on GPU under the batch size of 128 (tested in CIFAR100). We do not report Bitfit here as it cannot yield hardware memory efficiency on GPU with Pytorch.

<table>
<thead>
<tr>
<th>Method</th>
<th>On-device Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT-Last</td>
<td>1449MB</td>
</tr>
<tr>
<td>FT-Full</td>
<td>22631MB</td>
</tr>
<tr>
<td>Mesa [32]</td>
<td>10425MB</td>
</tr>
<tr>
<td>Back Razor@80%</td>
<td>8501MB</td>
</tr>
<tr>
<td>Back Razor@90%</td>
<td>7604MB</td>
</tr>
<tr>
<td>Back Razor@95%</td>
<td>7189MB</td>
</tr>
<tr>
<td>Back Razor@80% + Mesa (ours)</td>
<td>6640MB</td>
</tr>
</tbody>
</table>

Compared with FT-full, Back Razor@80% yields comparable performance across the datasets while achieving 3.9× memory saving. Though FT-Last employs smaller train memory than Back Razor, the accuracy drops by [2.2%, 32.9%, 2.1%, 6.4%, 0.1%, 26.6%, 0.9%, 4.2%] in [Pets, Aircraft, CIFAR10, CIFAR100, Flowers, Cars, CUB, Food], respectively, compared to Back Razor@80%.

Bitfit achieves competitive performance via tuning less than 0.1% of the parameters. However, it can only reduce the memory saving of 2.0× memory given it requires saving some large activations in full (e.g. attention map). In contrast, Back Razor can reduce the size of any activations. While Mesa has similar memory usage and performance compared with Back Razor, we note that its methodology is orthogonal with Back Razor. By combining Meta and Back Razor together, we can achieve 7.5× memory saving with comparable performance to FT-Full.

When employing a larger pruning ratio for BackRazor in ViTs, more memory footprint savings can be achieved at little performance degradation. Especially, for CIFAR10, the performance of BackRazor only drops by 0.2% compared to FT-Full at a pruning ratio of 95%.

4.4 Back Razor with More Tasks

In this section, we further exploring applying the proposed BackRazor to more downstream tasks.
Table 5: BackRazor for Pascal VOC segmentation task with DeepLabV3-MobileNet \cite{57,58}. The memory footprint of the training at a batch size of 8 is reported in the second row.

<table>
<thead>
<tr>
<th>Pruning ratio</th>
<th>0% (FT-Full)</th>
<th>10%</th>
<th>20%</th>
<th>50%</th>
<th>70%</th>
<th>90%</th>
<th>95%</th>
<th>97%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (MB)</td>
<td>8864</td>
<td>8295</td>
<td>7467</td>
<td>4983</td>
<td>3327</td>
<td>1671</td>
<td>1257</td>
<td>1092</td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>70.9</td>
<td>71.0</td>
<td>70.9</td>
<td>70.6</td>
<td>70.5</td>
<td>70.3</td>
<td>69.4</td>
<td>68.5</td>
</tr>
</tbody>
</table>

Table 6: BackRazor for RTE from GLUE Benchmark \cite{59} with BERT base \cite{19}. The theoretical and on-device memory footprints of training at a batch size of 8 are reported.

<table>
<thead>
<tr>
<th>Method</th>
<th>Theoretical Memory (MB)</th>
<th>Actual GPU Memory (MB)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT-Full</td>
<td>4174</td>
<td>7002</td>
<td>70.4</td>
</tr>
<tr>
<td>Back Razor@90%</td>
<td>2187</td>
<td>4858</td>
<td>69.7</td>
</tr>
</tbody>
</table>

Semantic Segmentation: As shown in Table 5 when applying to semantic segmentation tasks, BackRazor can yield $5.3 \times$ memory efficiency at a pruning ratio of 90% with only a marginal accuracy drop of 0.6% compared to the fully fine-tuning baseline.

Language modeling: As demonstrated in Table 6 when applying Back Razor on the language model, Back Razor can achieve a significant memory efficiency improvement at a pruning ratio of 90% with only a marginal accuracy drop of 0.6%.

5 Conclusion

In this work, we propose Back Razor, a memory-efficient transfer learning method, which is also the first asymmetric sparse transfer method that only sparsifies the back-propagation activations while keeping the forward. Through extensive experiments in both Convolutional Neural Networks and Vision Transformer across multiple datasets, we demonstrated the proposed Back Razor can achieve state-of-the-art memory efficiency with an aggressive pruning ratio. Moreover, it can be combined with the propagation quantization method and yield better memory efficiency. As limitations, Back Razor has only been tested on a limited range of tasks, and its theoretical underpinning is so far restricted to CNN backbones despite its empirical success in ViTs, calling for continued efforts.

Acknowledgment

Z. Wang is in part supported by the National Science Foundation under Grant IIS-2212176, and a Google TensorFlow Model Garden Award.

References


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A] The paper itself comes with a license.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
      We do not curate/release new assets.
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]