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

Anonymous Author(s)
Affiliation
Address
email

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 and general 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 employed on back-propagation. Such asymmetric pruning avoids affecting the 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 a similar convergence rate as vanilla SGD. Extensive experiments on both Convolutional Neural Networks and Vision Transformers show that Back Razor could yield up to 97% sparsity, saving 9.2x memory usage, without losing accuracy. The code is included in supplementary materials.

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 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

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Bitfit also fails to compress the activation to memory limitation (as shown in 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.

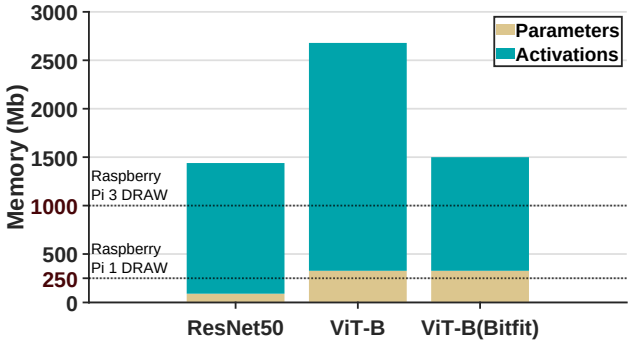


Figure 1: Training memory usage and on-device memory limitation illustration. The training memory is calculated under a batch size of 16. ViT-B (Bitfit) denotes the ViT-B model tuned with Bitfit.

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 fine-tuning method called Back Razor that prunes the activation during back-propagation. Our proposed method can be easily applied to various neural networks, including popular convolutional neural networks and transformers. To the best of our knowledge, this is the first work that studies pruning activation stored for back-propagation.
- We provide a theoretical analysis of our self-sparsified back-propagation method for Convolutional Neural Networks . 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 convolutional neural networks and Transformer for multiple datasets and show that the proposed method can achieve up to 97% sparsity with saving $9.2\times$ memory saving without losing accuracy. By further verifying on GPU, the proposed method achieves $2.7\times$ on-device memory efficiency for ViT.

2 Related Works

2.1 Pruning

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

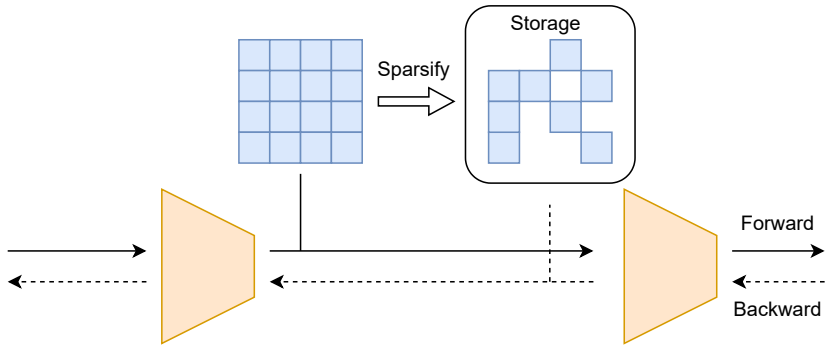


Figure 2: Pipeline of the proposed Back Razor.

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. In this work, we perform a simple yet meaningful twist: only prune the activations used for back-propagation.

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 [16, 17, 18, 19]. In contrast to the pre-training that is conducted on large clusters, transfer learning always happens on personal computing devices with limited memory constrains. To fit the large model to small devices, a common choice is to fine-tune the last several layers of the model [20, 16, 21, 22]. 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, 23]. However, it can easily lead to memory outages. On the other hand, some parameter efficient transfer learning [24, 25, 26, 27] works only requires 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.

2.3 Propagation Activation Compression

Few previous works have explored the approximate activation techniques. Most of them focus on the quantization methods [2, 3]. [4] also explored the JPEG compression. Following works improve the quantization via fine-grained group division [28, 29]. Recently, activation compression is extended to Vision transformer [30] and Graph Neural Networks [31]. However, to the best of our knowledge, none of them have explored the compressing via sparsifying, which is an orthogonal direction with existing methods.

3 Method

3.1 Activation is the bottleneck of memory

The forward of neural networks can be express as following

$$\begin{aligned} f(\boldsymbol{x};\boldsymbol{\theta}) &= f_l\left(f_{l-1}\left(\ldots f_0\left(\boldsymbol{x};\boldsymbol{\theta}_0\right)\ldots;\boldsymbol{\theta}_{l-1}\right);\boldsymbol{\theta}_l\right) \\ L &= \mathcal{L}(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \end{aligned} \tag{1}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ is the input and label, respectively. L is the loss between prediction $f(\boldsymbol{x};\boldsymbol{\theta})$ and label $\boldsymbol{y}$ calculated by loss function $\mathcal{L}$. f_i and $\boldsymbol{\theta}_i$ 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 $\boldsymbol{z}_i = f_i(\boldsymbol{z}_{i-1}, \boldsymbol{\theta}_i)$ and $\boldsymbol{z}_1 = f_1(\boldsymbol{x}; \boldsymbol{\theta}_1)$. When conducting back-propagation, $\frac{\partial L}{\partial \boldsymbol{z}_i}$ and $\frac{\partial L}{\partial \boldsymbol{\theta}_i}$ are required to be calculated for each activation.

Taking linear layer with $\boldsymbol{z}_i = W_i \boldsymbol{z}_{i-1} + b_i$ ($\boldsymbol{\theta}_i = [W_i, b_i]$) as an example, the gradient required to be calculated are [1]:

$$\frac{\partial L}{\partial \boldsymbol{z}_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} \frac{\partial \boldsymbol{z}_{i+1}}{\partial \boldsymbol{z}_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} W_i^\top, \quad \frac{\partial L}{\partial W_i} = \boldsymbol{z}_i^\top \frac{\partial L}{\partial \boldsymbol{z}_{i+1}}, \quad \frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} \tag{2}$$

As shown in Equation 2, the activation $\boldsymbol{z}_i$ 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 $\boldsymbol{z}_i$. 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 [32]) in transformer would involve the activation for computing $\frac{\partial L}{\partial \boldsymbol{z}_i}$, which cannot be avoided if previous layers requires updating. To tackle this challenge, we design a more general method: directly sparsifying the activation for backpropagation.

3.2 Back-propagation activation self-sparsification

In contrast to the activation sparsification [5, 6] 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

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

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

$$\frac{\partial L}{\partial \boldsymbol{z}_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} h_i^{(z)}(\tilde{\boldsymbol{z}}_i, \boldsymbol{\theta}_i), \quad \frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} h_i^{(\theta)}(\tilde{\boldsymbol{z}}_i, \boldsymbol{\theta}_i), \tag{4}$$

where $\tilde{\boldsymbol{z}}_i$ denotes the pruned activation $\boldsymbol{z}_i$. 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 $\tilde{\boldsymbol{z}}_i$. With the above backward, there is no need for saving the dense activation $\boldsymbol{z}_i$. Instead, we store a sparse version $\tilde{\boldsymbol{z}}_i$ 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 [33]. In practice, we would prune all the activations to a fixed sparsity of λ 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.

When saving the sparse matrix, we save it 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.

Algorithm 1 Backward and Update with Sparse Activation

Sample $\boldsymbol{x}, y$ from $\mathcal{D}$
Initialize $\boldsymbol{z}_0 = \boldsymbol{x}$
for $i = 1$ to l **do** ▷ Forward pass.
  $\boldsymbol{z}_i = f_i(\boldsymbol{z}_{i-1}, \theta_i)$
  $\tilde{\boldsymbol{z}}_{i-1} \leftarrow$ TopK_Prune$(\boldsymbol{z}_{i-1})$ ▷ Sparsify $\boldsymbol{z}_{i-1}$ and store it.
end for
Calculate the training loss $L = \mathcal{L}(\boldsymbol{z}_l, y)$
Initialize $\boldsymbol{s}_l \leftarrow \frac{\partial L}{\partial \boldsymbol{z}_l}$
for $i = l-1$ to 1 **do** ▷ Backward pass.
  Calculate $\frac{\partial L}{\partial \boldsymbol{z}_i}$: $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_{i+1} h_i^{(z)}(\tilde{\boldsymbol{z}}_i, \boldsymbol{\theta}_i)$ ▷ Calculate the gradient for the next activation
  Calculate $\frac{\partial L}{\partial \boldsymbol{\theta}_i}$: $\boldsymbol{s}_{i+1} h_i^{(\theta)}(\tilde{\boldsymbol{z}}_i, \boldsymbol{\theta}_i)$ and update $\boldsymbol{\theta}_i$.
end for

The bitmap has the same shape as the origin sparse tensor, but it is much smaller since it is in bool type. The memory cost of this format is $\frac{n}{8} + \lambda n c_{\text{type}}$ byte, where c_{type} is the count of bytes for the employed data type.

For the previous activation pruning method, the error that comes from pruning could gradually accumulate layer by layer in the forward, causing the deviation from the pre-trained model and leading to performance degradation. In contrast, the proposed Back Razor ensures the correctness of the forward. Moreover, in the backward progress, the pruning error of many layers would not accumulate: the activations of those layers are only used for computing the gradient of itself (e.g. linear layer, batch/layer normalization layer). It's also worth noting that the proposed Back Razor saves the same amount of memory as activation pruning under the same pruning ratio.

Comparison with recent parameter-efficient transfer methods. Recent studies reveal that fine-tuning can be parameter efficient. For example, diffPrune [24] points out that the fine-tuning performance can match fully fine-tuning via only modifying 0.5% of the weights. However, it employs Gumbel-Softmax [34] to make the masked parameters differentiable, which means the dense activations are required for updating the weight. Therefore, it cannot yield memory efficiency. On the other hand, other works also reveal that fine-tuning part of the model can yield competitive performance [25, 26, 27, 35]. Though it can lead to memory saving, we empirically verified that Back Razor can yield a higher saving ratio.

3.3 Theoretical Results

Now we study the convergence of Back Razor. We simplify the notion and define $\mathcal{L}$ to be the training loss of a convolutional neural network that takes the model's weights as the input. $\boldsymbol{\theta}_t = \{W_i\}_{i=1}^L$ as the flattened vector of the model's parameters at step t, and $\boldsymbol{g}_t$ be the gradient calculated with all the samples in the training dataset. Furthermore, we denote the *stochastic* gradient by $\tilde{\boldsymbol{g}}_t$, which is the gradient calculated with one sample from the training set. We make several classical assumptions [36]:

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

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

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

$$\mathbb{E}[\tilde{\boldsymbol{g}}_t] = \boldsymbol{g}_t, \qquad \mathbb{E}[(\tilde{\boldsymbol{g}}_t - \boldsymbol{g}_t)_j^2] \leq \sigma_j^2, \forall j \in [n],$$

where n is the number of parameters of models and σ_j^2 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 A. We first prove the convergence for linear networks.

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

In linear networks, $\frac{\partial L}{\partial \boldsymbol{z}_i}$ 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, $\tilde{\boldsymbol{A}}_t$ is exclusively constructed by activations' gradients, and not changed after activation pruning.

We call $\tilde{\boldsymbol{A}}_t$ 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*

$$\mathbb{E}[\frac{1}{T}\sum_{t=1}^{T}\|\boldsymbol{g}_t\|^2] \leq \frac{2(\mathcal{L}(\boldsymbol{\theta}_0) - L^*)}{[1-(p+\alpha\beta)K_a]\alpha T} + \frac{(p+\alpha\beta)K_a}{1-(p+\alpha\beta)K_a}\sigma^2, \tag{5}$$

where the T is the number of iterations, $\beta = \|\boldsymbol{\beta}\|_\infty$, $\boldsymbol{\theta}_0$ 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. The gradient variance can be reduced by larger batch sizes: when using a batch size of B, the variance is reduced by B.

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 more specifically, we 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]:

$$\frac{\partial L}{\partial \boldsymbol{z}_i} = W_i^\top * \frac{\partial L}{\partial \boldsymbol{z}_{i+1}}, \quad \frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \boldsymbol{z}_{i+1}} * \boldsymbol{z}_i, \tag{6}$$

which suggests the gradient on $\boldsymbol{z}_i$ will not be affected if we use the pruned activation $\tilde{\boldsymbol{z}}_i$ to obtain the derivatives on $\boldsymbol{\theta}_i$ 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 structure. The self-attentions inside the Transformer do not meet the Lipchitz condition [37], therefore we do not provide theoretical analysis for them. However we empirically observe that Back Razor is also applicable on Vision Transformer [38] with decent performance (refer to Section 4.3)

4 Experiment

4.1 Setting

Dataset Following the common practice [24, 39, 40], we employ supervised pretrain of Imagenet [41]. Specifically, we employ ImageNet1K and Imagenet22K for convolutional neural networks and ViT, respectively. For downstream fine-tuning, we consider four datasets: CIFAR10, CIFAR100 [42], Aircraft [43] and Pets [44]. For all datasets, we employ resolution of 224×224 following [1].

Platform Our experiments are implemented with Pytorch [45] and conducted on 1080 Ti or V100 GPUs.

Training Memory Measuring We by default report the theoretical memory usage following [1]. We also report the memory usage measured on the GPU at Section 4.3.

Table 1: The comparison between the proposed 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 [35, 46]. 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.

Method	Train Memory	Pets	Aircraft	CIFAR10	CIFAR100	Flowers	Cars	CUB	Food
FT-Last	31MB	91.3	44.9	85.9	68.8	90.1	50.9	73.3	68.7
FT-Norm+Last	192MB	92.2	68.1	94.8	80.2	94.3	77.9	76.3	77.0
FT-Full	366MB	93.0±0.05	88.2±0.24	97.1±0.02	84.1±0.07	96.8	91.2	81.3	83.8
TinyTL [1]	37MB	91.8	75.4	95.9	81.4	95.5	85.0	77.1	79.7
TinyTL@320 [1]	65MB	92.9	82.3	96.1	81.5	96.8	88.8	81.0	82.9
Back Razor@90% (ours)	42MB	92.9±0.11	87.6±0.16	96.9±0.07	83.3±0.05	97.1	89.8	81.3	82.1
Back Razor@95% (ours)	41MB	93.2	85.7	96.8	82.7	-	-	-	-
Back Razor@97% (ours)	40MB	93.0	85.2	96.6	82.6	-	-	-	-
Back Razor@99% (ours)	39MB	92.5	80.7	96.1	80.2	-	-	-	-

4.2 Back Razor with Convolutional Neural Networks

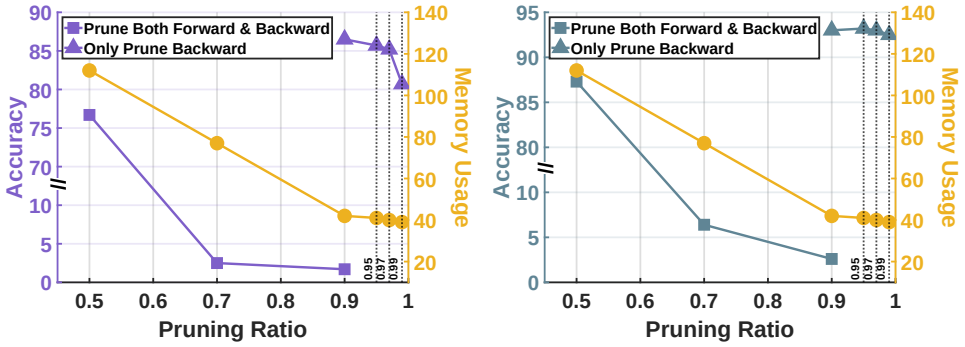
Setting We choose ProxylessNAS-Mobile [47] as the backbone following TinyTL [1]. It is worth noting that we keep the origin setting of ProxylessNAS-Mobile instead of modifying the architecture as in TinyTL given Back Razor is applicable for any architecture. We also follow the most of training settings of TinyTL for fair comparison: The fine-tuning epochs and batch size is set as 50 epochs and 8, respectively. The model is optimized with adam [48] optimizer and cosine learning lr schedule [49]. The initial learning rate is tuned for each dataset. We freeze all the parameters of Batch Normalization layers [50] 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 In Table 1, we compare the proposed Back Razor with previous transfer learning methods on CNN. 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 method [1]: TinyTL and TinyTL@320 (TinyTL@320 employs a larger resolution image of 320).

Compared with FT-Full, the proposed 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%, 17.5%, 2.1%, 3.1%] worse than the proposed Back Razor@90% in [Pets, Aircraft, CIFAR10, CIFAR100], respectively. Though FT-Last does yield slightly less training memory (32MB v.s. 42MB), the performance largely degrades: the accuracy is lower by [1.6%, 42.7%, 11.0%, 14.5%] than the proposed Back Razor@90% in [Pets, Aircraft, CIFAR10, CIFAR100], respectively.

Compared with the previous State-of-The-Art method TinyTL, the proposed 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%] at [Pets, Aircraft, CIFAR10, CIFAR100], respectively. When compared with TinyTL@320, which employs a larger resolution, the proposed Back Razor@90% saves 1.5 times of the memory while still yielding an improvement of [0.0%, 5.3%, 0.8%, 1.8%] at [Pets, Aircraft, CIFAR10, CIFAR100], respectively.

It's also worth noting that the proposed 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%], 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).



(a) Aircraft [43]

(b) Pet [44]

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 consumbtion is the same under the same pruning ratio.

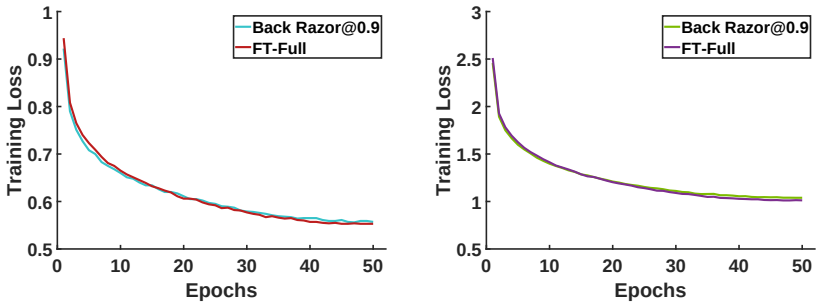
Can we prune at forward pass We further compare the performance difference between activation pruning (prune both forward and backward) with the proposed Back Razor (only prune backward) in Figure 3. We implement the activation pruning via directly applying the mask of the Back Razor to the forward pass. It can be observed that the model can easily collapse (the model can hardly converge and only have accuracy below 5%) for activation pruning even with a mild pruning ratio of 70%. In contrast, for the proposed 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 the proposed Back Razor is even faster. And in the last stage of the training, the FT-Full is only marginally higher than Back Razor.

4.3 Back Razor with Vision Transformer

Setting In this section, we study the proposed Back Razor on ViT [38]. The patch size is by default set as 16. The finetuning employ the standard SGD optimizer with cosine learning rate decay. 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 [38, 51, 30].

Comparison In Table 2, we compare the proposed Back Razor with previous transfer learning methods on ViT. It includes the FT-Last and FT-Full with the same definition of Section 4.3. Moreover, we further compare the parameter efficient training method Bitfit [25] and the backpropagation quantization method Mesa [25].



(a) CIFAR10 [42]

(b) CIFAR100 [42]

Figure 4: The training loss convergence speed comparison between the proposed Back Razor@0.9 and FT-Full. The convergence speed of them is comparable.

Table 2: The comparison between the proposed 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.

Method	Train Memory	Pets	Aircraft	CIFAR10	CIFAR100	Flowers	Cars	CUB	Food
FT-Last	525MB	91.6	-	96.8	86.5	-	-	-	-
FT-Full	19235MB	94.1	78.8	99.0	93.1	99.5	85.5	85.5	90.3
Bitfit [25]	9460MB	94.0	-	98.8	92.3	-	-	-	-
Mesa [30]	5442MB	93.8	-	98.9	92.8	-	-	-	-
Back Razor@80% (ours)	4565MB	93.8	77.3	98.9	92.9	99.4	84.0	86.3	90.5
Back Razor@90% (ours)	3912MB	93.3	-	98.9	91.8	-	-	-	-
Back Razor@95% (ours)	3496MB	92.4	-	98.8	90.0	-	-	-	-
Back Razor@80% + Mesa (ours)	2937MB	93.5	-	99.0	92.7	-	-	-	-

Compared with FT-full, the proposed Back Razor@80% yields comparable performance across the datasets while achieving $3.9\times$ memory saving. Though FT-Last employs smaller train memory than the proposed method, the accuracy drops by [2.2%, 2.1%, 6.4%] in [Pets, CIFAR10, CIFAR100], respectively, compared to the proposed Back Razor@80%.

The Bitfit achieves a competitive performance via tuning less than 0.1% of the parameters. However, it can only reduce the memory saving of $2.0\times$ memory given it requires saving some large activations in full (e.g. attention map). In contrast, the proposed Back Razor can reduce the size of any activations. While Mesa has similar memory usage and performance compared with the proposed BackRazor, the method of mesa is orthogonal with the proposed Back Razor and by combining them together, we can achieve $7.5\times$ memory saving with comparable performance to FT-Full.

Table 3: Comparison of the on-device memory usage on GPU under the batch size of 128 (test in CIFAR100). We skip Bitfit here as it cannot yield hardware memory efficiency on GPU with Pytorch.

Method	On-device Memory
FT-Last	1449MB
FT-Full	22631MB
Mesa [30]	10425MB
Back Razor@80%	8501MB
Back Razor@90%	7604MB
Back Razor@95%	7189MB
Back Razor@80% + Mesa	6640MB

When employing a larger pruning ratio in ViT, the memory usage can be more efficient and the performance would not incur large degradation. Especially, for CIFAR10, the performance only drop by 0.2% compared to FT-Full at a pruning ratio of 95%, demonstrating the proposed Back Razor can achieve an aggressive pruning ratio in ViT.

Hardware memory efficiency Table 3 illustrates the hardware memory usage for different methods. When testing on the modern computing device GPU, the Back Razor@80% can yield $2.7\times$ memory efficiency. By combining with Mesa, its efficiency can achieve $3.4\times$. The theoretical memory usage aligns with real memory usage. As the theoretical memory usage is not achieved, there is still optimizing space for the implementation.

5 Conclusion

In this work, we propose Back Razor, a memory-efficient transfer learning method, which is also the first asymmetric pruning 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.

Limitation and future work In this work, we mainly study the transfer classification task. Our future works targets extending Back Razor to various downstream tasks, including semantic segmentation, dense prediction, and language understanding.

References

[1] 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.

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

[3] 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.

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

[5] 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, 2017.

[6] Qing Yang, Wei Wen, Zuoguan Wang, Yiran Chen, and Hai Li. Integral pruning on activations and weights for efficient neural networks, 2019.

[7] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[8] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[9] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

[10] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[11] Haobo Sun, Yingxia Shao, Jiawei Jiang, Bin Cui, Kai Lei, Yu Xu, and Jiang Wang. Sparse gradient compression for distributed sgd. In *International Conference on Database Systems for Advanced Applications*, pages 139–155. Springer, 2019.

[12] 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.

[13] Bingzhen Wei, Xu Sun, Xuancheng Ren, and Jingjing Xu. Minimal effort back propagation for convolutional neural networks. *arXiv preprint arXiv:1709.05804*, 2017.

[14] Zhiyuan Zhang, Pengcheng Yang, Xuancheng Ren, Qi Su, and Xu Sun. Memorized sparse backpropagation. *Neurocomputing*, 415:397–407, 2020.

[15] Xucheng Ye, Pengcheng Dai, Junyu Luo, Xin Guo, Yingjie Qi, Jianlei Yang, and Yiran Chen. Accelerating cnn training by pruning activation gradients. In *European Conference on Computer Vision*, pages 322–338. Springer, 2020.

[16] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[17] 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.

[18] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[19] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.

[20] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[21] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alex G Hauptmann. Devnet: A deep event network for multimedia event detection and evidence recounting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2568–2577, 2015.

[22] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[23] Hugo Touvron, Matthieu Cord, Alaaeldin El-Nouby, Jakob Verbeek, and Hervé Jégou. Three things everyone should know about vision transformers. *arXiv preprint arXiv:2203.09795*, 2022.

[24] Demi Guo, Alexander M Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. *arXiv preprint arXiv:2012.07463*, 2020.

[25] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199*, 2021.

[26] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. *arXiv preprint arXiv:2106.04647*, 2021.

[27] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586*, 2021.

[28] Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. Don't waste your bits! Squeeze activations and gradients for deep neural networks via TinyScript. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3304–3314. PMLR, 13–18 Jul 2020.

[29] R David Evans and Tor Aamodt. Ac-gc: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems*, 34, 2021.

[30] 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.

[31] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2021.

[32] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[33] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[34] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[35] Pramod Kaushik Mudrakarta, Mark Sandler, Andrey Zhmoginov, and Andrew Howard. K for the price of 1: Parameter efficient multi-task and transfer learning. In *International Conference on Learning Representations*, 2019.

[36] Léon Bottou et al. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[37] Hyunjik Kim, George Papamakarios, and Andriy Mnih. The lipschitz constant of self-attention. In *International Conference on Machine Learning*, pages 5562–5571. PMLR, 2021.

[38] 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.

[39] Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2661–2671, 2019.

[40] Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge Belongie. Large scale fine-grained categorization and domain-specific transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4109–4118, 2018.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

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

[43] Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013.

[44] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.

[45] 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.

[46] Jonathan Frankle, David J Schwab, and Ari S Morcos. Training batchnorm and only batchnorm: On the expressive power of random features in cnns. *arXiv preprint arXiv:2003.00152*, 2020.

[47] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

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

[49] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[50] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[51] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pages 10347–10357. PMLR, 2021.

[52] 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.

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]