TOWARDS GREEN AI IN FINE-TUNING LARGE LANGUAGE MODELS VIA ADAPTIVE BACKPROPAGATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Fine-tuning is the most effective way of adapting pre-trained large language models to downstream applications. With the increasing eagerness for LLM-powered applications and model personalization, fine-tuning has been performed pervasively and intensively by researchers and developers. Such high usage of computing resources significantly consumes energy, produces large carbon footprints, and accelerates climate change. For mitigation, reducing FLOPs directly correlates with energy consumption. Existing schemes focus on improving memory efficiency but don't optimize the FLOPs of backpropagation, which leads to limited performance. Instead, to effectively reduce FLOPs, we envision that the selection of trainable portions should be constrained by their backpropagation costs. In this paper, we propose GreenTrainer, which only consumes user-defined FLOPs to fine-tune LLMs with minimum accuracy loss. Experiment results show that GreenTrainer can save up to 64% training FLOPs compared to full fine-tuning without noticeable accuracy loss. Compared to existing schemes, GreenTrainer can achieve up to 4% accuracy improvement with on-par FLOPs reduction.

1 INTRODUCTION

Large language models (LLMs), being pre-trained on large-scale text data, have been used as foundational tools in generative AI for natural language generations. The most effective way of adapting LLMs to downstream applications, such as personal chat bots and podcast summarizers, is to fine-tune a generic LLM using the specific application data (Devlin et al., 2018). Intuitively, fine-tuning is less computationally expensive than pre-training due to the smaller amount of training data, but it may result in significantly high energy consumption and carbon footprint when being intensively performed worldwide and hence bring large environmental impact. In particular, enabled by the democratization of open-sourced LLMs (Candel et al., 2023) and convenient APIs of operating these LLMs (Ott et al., 2019; Wolf et al., 2019), even non-expert individuals can easily fine-tune LLMs using a few lines of codes, either for performance enhancement or model personalization (Scialom et al., 2022). For example, when a LLaMA-13B model (Touvron et al., 2023) is fine-tuned by 10k users using A100-80GB GPUs, such fine-tuning consumes 6.9× more GPU hours than pre-training a GPT-3 model (Brown et al., 2020) with 175B parameters. The amount of energy being consumed by such fine-tuning, correspondingly, is comparable to those consumed by small towns or even some underdeveloped countries, and the amount of emitted carbon dioxide is equivalent to 500× of that produced by a New York-San Francisco round-trip flight (aii, 2023).

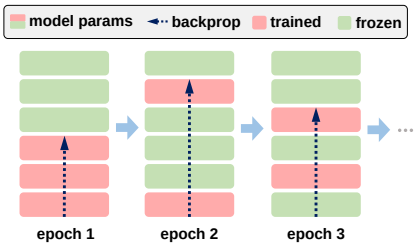


Figure 1: GreenTrainer adaptively selects the trainable portion

Mitigating such environmental impact towards Green AI directly correlates to reducing the number of floating operations (FLOPs) of fine-tuning, as FLOPs is a fundamental measure that represents the amount of computational operations and hence energy consumption in training (Schwartz et al., 2020). Most existing techniques, however, are limited to optimizing LLM fine-tuning for lower memory consumption rather than FLOPs reduction (Malladi et al., 2023; Liao et al., 2023). Some other methods reduce the amount of computations by only fine-tuning specific types of model parameters such as bias (Zaken et al., 2021), LayerNorm and output layer weights (Lu et al., 2021), but they significantly impair the model's expressivity and are only applicable to simple non-generative learning tasks. Instead, researchers suggested keeping the original model parameters frozen but injecting additional trainable parameters either to the input (Lester et al., 2021; Liu et al., 2022) or internal layers (Li & Liang, 2021; Hu et al., 2023). Recent LoRA-based methods (Hu et al., 2021; Zhang et al., 2023) further reduce the overhead of computing weight updates for these in-

jected parameters via low-rank approximation. These methods can achieve comparable accuracy on generative tasks with full fine-tuning. However, they still need to compute the activation gradients through the whole model, and their FLOPs reduction is hence limited to computations of weight updates, which are only 25%-33% of the total training FLOPs.

Besides computing weight updates, FLOPs in training are also produced in i) forward propagation and ii) backward propagation of activation gradients. Since complete forward propagation is essential to calculate the training loss, we envision that the key to effective FLOPs reduction is to take the backpropagation cost of activation gradients, which is at least 33% of the total training FLOPs, into account and selectively involve only the most appropriate model structures in backpropagation. The major challenge, however, is that such selective training will nearly always bring model accuracy loss. Our basic idea to minimize the accuracy loss is to adapt such selection in backpropagation to a flexible objective of FLOPs reduction, which is determined by the carbon footprint in energy supply for LLM fine-tuning. For example, when such carbon footprint is low due to more insertion of renewable energy, a lower objective of FLOPs reduction can be used to retain more model structures to training and hence retain the training accuracy. On the other hand, high carbon footprint in energy production could lead to a higher objective of FLOPs reduction for better embracing Green AI.

Based on this idea of adaptive backpropagation, in this paper we present *GreenTrainer*, a new training technique for efficient LLM fine-tuning with the minimum accuracy loss. As shown in Figure 1, given an objective of FLOPs reduction, GreenTrainer adaptively selects the most appropriate trainable portions at run-time, based on evaluation of different neural network (NN) tensors' importance in training. Such importance evaluation is difficult because NN tensors do not directly associate with any input data variables or intermediate features, and most attribution techniques (Sundararajan et al., 2017; Hesse et al., 2021) that evaluate feature importance are hence not applicable. Traditional approaches based on weight magnitudes (Li et al., 2016), random perturbations (Breiman, 2001), and gating functions (Guo et al., 2020), on the other hand, are either inaccurate or computationally expensive for LLMs. Instead, our approach is to follow a similar rationale with current attribution techniques that measures the importance of an input data variable as the accumulation of relevant gradients, to evaluate tensor importance as the cumulative gradient changes of its weight updates in training. In this way, we ensure that selected tensors will make the maximum contribution to reducing the training loss.

Another challenge is how to precisely profile the training FLOPs of different tensor selections. Due to the interdependency between different tensors, their total FLOPs in training is usually not equal to the summation of their individual training FLOPs. Such interdependency is determined by the backpropagation characteristics of the specific NN operators connected to each tensor, but existing FLOPs models cannot link NN operators to tensors based on the computing flow of backpropagation. To tackle this challenge, we build a new FLOPs model that incorporates the relations between tensors and NN operations into profiling of training FLOPs. Based on this model, we develop a dynamic programming (DP) algorithm that can find the optimal tensor selection from an exponential number of possibilities (e.g., 2^{515} for 515 tensors in OPT-2.7B model (Zhang et al., 2022)), with negligible computing overhead.

To our best knowledge, GreenTrainer is the first work that adapts the training FLOPs reduction in LLM fine-tuning to the need of Green AI, via adaptive control of backpropagation cost in training. Our detailed contributions are as follows:

Tensor FLOPs Profiling (Section 3.1). We build new FLOPs models that allow precise profiling of tensor selection's training FLOPs for transformer-based LLMs.

Efficient Tensor Importance Evaluation (Section 3.2). We developed a runtime evaluation process of tensor importance with low memory cost, high accuracy, and high adaptability.

Efficient DP for Tensor Selection (Section 3.3). Our lightweight DP algorithm can decide the optimal selection of tensors at runtime that maximizes the training loss reduction.

To evaluate GreenTrainer, we did extensive experiments with three open-sourced LLMs, namely OPT (Zhang et al., 2022), BLOOMZ (Muennighoff et al., 2022) and FLAN-T5 (Chung et al., 2022), on text generation datasets including SciTLDR (Cachola et al., 2020) and DialogSum (Chen et al., 2021). Our experiment results show that GreenTrainer can save up to 64% training FLOPs compared to full fine-tuning, without noticeable accuracy loss. Compared to existing schemes such as

Prefix Tuning (Li & Liang, 2021) and LoRA (Hu et al., 2021), GreenTrainer can improve the model accuracy by 4%, with the same amount of FLOPs reduction.

2 BACKGROUND & MOTIVATION

2.1 TRANSFORMER ARCHITECTURES FOR TEXT GENERATION

LLMs are stacked by transformer blocks (Vaswani et al., 2017). Each transformer block mainly contains a Multi-Head Attention (MHA) layer, LayerNorms (Ba et al., 2016), and a Feed-Forward Network (FFN) with two dense layers. Given an input sequence $X \in \mathbb{R}^{n\times d}$ with n tokens, MHA projects all the tokens into (Q, K, V) space h times independently, using h suites of trainable projectors $(W_Q^{(i)}, W_K^{(i)}, W_V^{(i)})_{i=1,...,h}$. Each projection $f_i : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times \frac{d}{h}}$ is defined as:

$$Q_i, K_i, V_i = XW_Q^{(i)}, XW_K^{(i)}, XW_V^{(i)} \tag{1}$$

The output (Q_i, K_i, V_i) then performs attention mechanisms to produce O_i by weighting V_i with the attention scores between Q_i and K_i. The MHA's final output is obtained by concatenating each O_i, following a linear projection $g : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$ with a trainable projector W_o:

$$O_i = \text{Softmax}\left(Q_i K_i^\top / \sqrt{d/h}\right) V_i \qquad \text{MHA}_{\text{out}} = \text{Concat}(O_1, O_2, ..., O_h)W_o \tag{2}$$

Due to the auto-regressive nature, LLMs can only generate a single target token in each forward pass, which is inefficient in training. Instead, LLMs adopt teacher-forcing (Lamb et al., 2016) to generate the entire target sequence in a single forward pass. Specifically, causal masks are applied to MHA's attention scores, so that each target token can be predicted from the label tokens at previous positions. In this way, LLMs can be trained in a standard way like any feed-forward models.

Table 1: Fine-tuning different substructures of OPT-2.7B and FLAN-T5-3B LLMs on DialogSum dataset (ROUGE-1 score on the test set is used as accuracy metric)

Trainable substructure	**OPT-2.7B**		**FLAN-T5-3B**	
	FLOPs ($\times 10^{15}$)	**Acc. (%)**	**FLOPs** ($\times 10^{15}$)	**Acc. (%)**
All params	262.0	23.6	135.7	46.5
Last 2 layers	181.6 (31%↓)	20.8	46.1 (66%↓)	39.2
Decoder prefix	174.7 (33%↓)	13.4	55.3 (60%↓)	37.6
(W_Q, W_V)	174.7 (33%↓)	23.8	90.5 (33%↓)	44.7

2.2 OPPORTUNITIES FOR ON-DEMAND BACKPROPAGATION

By stacking a sufficient number of large transformer blocks, pre-trained LLMs are able to capture general language patterns and world knowledge. However, when fine-tuned for a specific downstream task, the LLMs can be over-parameterized because not all learned knowledge can be useful. Freezing some parameters or updating them less frequently could have little impact on final accuracy while saving the computation of backpropagation. Based on this intuition, existing work provides several options for fine-tuning LLM substructures. However, their selections are not optimized for both accuracy and FLOPs reduction. As shown in Table 1, fine-tuning the last 2 layers or the decoder prefix can achieve up to 60% FLOPs reduction, but suffers great accuracy loss. On the other hand, fine-tuning the linear projectors (W_Q, W_V) in each transformer block can achieve on-par accuracy with full fine-tuning, but can only save at most 33% FLOPs due to propagating activation gradients through all the transformer blocks. To better guide the substructure selection for GreenTrainer, we derive a tensor importance metric that evaluates how fine-tuning each tensor contributes to the training quality. By knowing each substructure's importance, GreenTrainer is able to maximize training quality with desired FLOPs reduction.

2.3 FLOPS MODEL OF BACKPROPAGATION

GreenTrainer should have a general FLOPs model to compute backpropagation FLOPs, as the cost constraint, of any selected trainable portions. In general, the computation of backpropagation can be

decomposed into two parts using the chain rule. As shown in Figure 2, when training a 4-layer dense NN without bias, each layer requires to compute the activation gradient $\mathrm{dy_i}$ - the loss L's gradient w.r.t the activation y_i, and the weight update $\mathrm{dw_i}$ - the loss gradient w.r.t weight W_i:

$$\mathrm{dy_i} = \frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial y_{i+1}} W_i^\top = \mathrm{dy_{i+1}} W_i^\top \qquad \mathrm{dw_i} = \frac{\partial L}{\partial W_i} = y_i^\top \frac{\partial L}{\partial y_{i+1}} = y_i^\top \mathrm{dy_{i+1}} \tag{3}$$

In this way, $(\mathrm{dy_i}, \mathrm{dw_i})$ can be computed from the upstream $(\mathrm{dy_{i+1}}, \mathrm{dw_{i+1}})$. Both $\mathrm{dy_i}$ and $\mathrm{dw_i}$ take considerable computation and either cannot be omitted. Based on the above computation dependency, we can derive the FLOPs of training selected layers. For example, if only Layer 2 is trainable and other layers are frozen, the total FLOPs for the backpropagation is the FLOPs of computing $\mathrm{dy_3}$ and $\mathrm{dy_4}$ plus the FLOPs of computing $\mathrm{dw_2}$. Due to the generality of the chain rule, such FLOPs derivation is also applicable to any other type of layers.

Based on this rationale, GreenTrainer constructs FLOPs models for LLM modules, including MHA and FFN. However, the layer-level time model is coarse-grained and can lead to inaccurate selection. Some important parameters may be unselected because many others within the same layer are unimportant. In GreenTrainer, we push the selection granularity to the tensor level, which can be well-supported by tensorized NN libraries (e.g., TensorFlow (Abadi, 2016) and PyTorch (Paszke et al., 2019)). Although the weight-level selection is more fine-grained, it also requires fine-grained indexing and leads to high overhead.

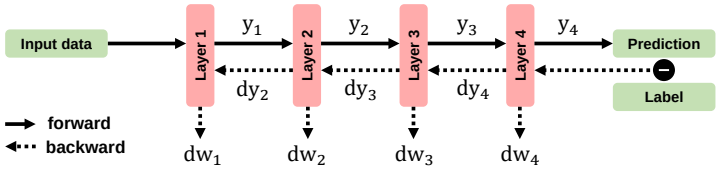


Figure 2: Backpropagation of a 4-layer dense NN

3 GreenTrainer Method

Problem formulation. To reduce the FLOPs of fine-tuning, an intuitive formulation is to minimize the training FLOPs while achieving the desired objective of fine-tuning quality (e.g., model loss and accuracy). However, it is hard to determine an appropriate accuracy objective in advance. Setting the objective higher than the model's capability would lead to no FLOPs reduction. Instead, GreenTrainer aims to maximize the loss reduction while achieving the desired FLOPs reduction:

$$\max \Delta_{loss}(\boldsymbol{m}) \quad \text{s.t. } T_{selective}(\boldsymbol{m}) \leq \rho T_{full} \tag{4}$$

where $\boldsymbol{m}$ is a binary selector to be solved for tensor selection. $\boldsymbol{m}$ parametrizes both Δ_{loss}: the loss reduction and $T_{selective}$: the per-batch FLOPs of training the selected tensors. $T_{selective}$ is constrained to be lower than a user-controlled ratio (ρ) of the per-batch full fine-tuning FLOPs (T_{full}). For example, $\rho = 0.5$ means that the fine-tuning FLOPs should be reduced to 50% of that in full fine-tuning. In practice, users can either set ρ to be constant or schedule it at runtime.

To clearly identify each tensor's contribution during fine-tuning, we model $\Delta_{loss}(\boldsymbol{m})$ as the aggregated importance of the selected tensors. $T_{selective}(\boldsymbol{m})$ can also be refined by plugging the FLOPs model of backpropagation mentioned in Section 2.3. Eventually, Eq 4 can be rewritten as:

$$\max \ \boldsymbol{m} \cdot \boldsymbol{I} \quad \text{s.t. } T_{fp} + \boldsymbol{m} \cdot \boldsymbol{t}_{dw} + \sigma(\boldsymbol{m}) \cdot \boldsymbol{t}_{dy} \leq \rho T_{full} \tag{5}$$

where T_{fp} indicates the per-batch FLOPs of the forward pass, and each pair in $(\boldsymbol{t}_{dy}, \boldsymbol{t}_{dw})$ corresponds to the FLOPs pair of computing $(\mathrm{dy}, \mathrm{dw})$ for each tensor. Based on the FLOPs model of backpropagation, $\sigma(\cdot)$ transforms the binary selector $\boldsymbol{m}$ to incorporate t_{dy} of all tensors along the backward pass into total fine-tuning FLOPs. For example, if $\boldsymbol{m} = [0, 0, 1, 0, 1, 0, 0]$, then $\sigma(\boldsymbol{m}) = [0, 0, 1, 1, 1, 1, 1]$.

To ground the above formulation and solve $\boldsymbol{m}$, GreenTrainer consists of three key components: (i) tensor FLOPs profiling, which pre-computes the FLOPs of $(\mathrm{dy_i}, \mathrm{dw_i})$ for every NN tensor; (ii) tensor importance evaluation, which quantifies the contribution of updating each NN tensor to the training quality at runtime; (iii) tensor selector, which grounds the tensor selection problem using tensor FLOPs and importance scores, and provides solutions via dynamic programming at runtime.

3.1 TENSOR FLOPS PROFILING

Standard NN profilers can measure the execution FLOPs or time of each NN operator (e.g., matrix multiplication and convolution). However, such measurements are limited to the operation level and have no correspondence to NN tensors that participate in these operations. In other words, when a set of selected tensors is being trained, the executed FLOPs will not equal the summation of the backpropagtion FLOPs of individual tensors. To address this limitation, as shown in Figure 3, we first convert the original layer-based NN structure into a tensor-level computing graph, which retains the execution order of all tensors' involvement in training. Then we extract the related backpropagation operators, and derive each tensor's FLOPs (t_{dy}, t_{dw}) by matching and aggregating the FLOPs of these NN operators. We categorize such matching and aggregation rules by the type of LLM layers where tensors are located.

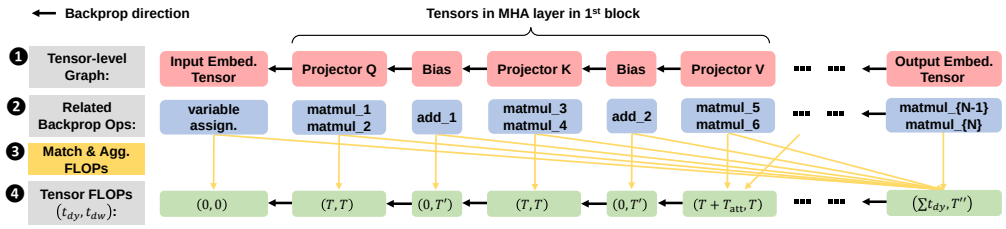


Figure 3: Workflow of tensor FLOPs profiling

Input & output embedding layers. The input embedding layer contains a trainable embedding tensor that maps each raw token into a dense representation through efficient lookup operations. Given the activation gradient $\mathrm{dy_{i+1}}$ from upstream layers, deriving the update $\mathrm{dw_i}$ of this embedding tensor only involves variable assignment without any heavy computations $\rightarrow t_{dw} \approx 0$. Specifically, if a raw token is mapped to k^{th} vector in the embedding tensor during the forward pass, then during backpropagation, $\mathrm{dy_{i+1}}$ from the upstream will be only assigned to k^{th} row of $\mathrm{dw_i}$.

$$\mathrm{dw_i}[s] = \mathrm{dy_{i+1}} \text{ if } s = k, \text{ else } \mathbf{0} \tag{6}$$

Since the input layer doesn't propagate activation gradients, we can also conclude that $t_{dy} = 0$.

Reversely, the output embedding layer projects each token back to the probability space by multiplying its trainable tensor with the token vector. Intuitively, its (t_{dy}, t_{dw}) can be derived the same as the dense layer by Eq. (3) in Section 2.3. However, in most LLMs, the output embedding layer shares the trainable tensor with the input embedding layer. This implies if the output embedding is trained, the input embedding will also be tied to training. To reflect this behavior, all the t_{dy} from the LLM's output up to the input embedding layer should be accumulated to the t_{dy} of the output embedding tensor, while its t_{dw} remains unchanged.

Multi-Head Attention (MHA) layer. As described in Section 2.2, MHA contains multiple linear projectors as trainable tensors, of which the (t_{dy}, t_{dw}) can be derived the same as the dense layer. In practice, some LLMs, such as OPT, also include bias as another type of trainable tensor after projection. Based on the chain rule, the backpropagation of bias is computed as:

$$\mathrm{dy_i} = \mathrm{dy_{i+1}} \qquad\qquad \mathrm{dw_i} = \mathbf{1}^\top \mathrm{dy_{i+1}} \tag{7}$$

which indicates $t_{dy} = 0$ since $\mathrm{dy_i}$ is identically passed from $\mathrm{dy_{i+1}}$. t_{dw} can be derived as the FLOPs of adding up the elements along all the dimensions except the last dimension of $\mathrm{dy_{i+1}}$.

The attention mechanism in Eq. 2 is backpropagated prior to the projectors. If any of the projectors are trained, attention's propagation FLOPs must be counted. To ensure this behavior, we accumulate such FLOPs to the Projector tensor W_V's t_{dy}.

LayerNorm. Given a token, LayerNorm first normalizes its features and then uses two trainable tensors γ and β to element-wise multiply with and add to the token, respectively. The multiplication and addition are similar to those in the dense layer, so its (t_{dy}, t_{dw}) can be derived without special handling. However, the backpropagation FLOPs of the normalization operators should be accumulated to the previous tensor's t_{dy}. That means if any tensors in previous layers are trained, the FLOPs of propagating the normalization operators should be also included.

Feed-Forward Network (FFN). In the FFN, there is a nonlinear activation function between two dense layers. Following the same method for LayerNorm, we accumulate the FLOPs of propagating through this activation function to the bias tensor's t_{dy} in the first dense layer.

3.2 Tensor Importance Evaluation

Gradient-based importance metric. Since NN training iteratively updates the NN weights to minimize the loss function, an intuitive approach to evaluating the importance of a weight update in a given iteration is to undo this update and check how the training loss value increases back:

$$\Delta L = L(w) - L(w + \Delta w) \tag{8}$$

so that higher ΔL means this update is more important and the corresponding weight should be selected for fine-tuning. However, repeatedly applying this approach to every NN weight is expensive due to the large number of weights. Instead, our approach is to leverage the backpropagation to efficiently approximate the importance of all the NN weights in one shot. Specifically, we compute the importance score of each weight by smoothing the undo operation and computing the loss gradients with respect to the updates corresponding to all the weights. Letting the multiplicative $\boldsymbol{c} \in [0, 1]^M$ denote the continuous undo operation for all the M weights, we can compute the loss gradient with respect to $\boldsymbol{c}$ as

$$-\frac{\partial L(\boldsymbol{w} + \boldsymbol{c} \odot \Delta \boldsymbol{w})}{\partial \boldsymbol{c}} = -\Delta \boldsymbol{w} \odot \frac{\partial L(\boldsymbol{u})}{\partial \boldsymbol{u}}\bigg|_{\boldsymbol{u}=\boldsymbol{w}+\boldsymbol{c}\odot\Delta\boldsymbol{w}} \tag{9}$$

where $\odot$ denotes element-wise multiplication. By setting $\boldsymbol{c} = \boldsymbol{0}$, Eq. (9) becomes an importance vector where each element corresponds to NN weights. Since the loss gradient is parametrized by all the NN weights, the computed weight importance implicitly incorporates the impact of weight dependencies. The importance of a tensor k, then, is a summation of all its weights' importance. We further scale all the tensor importance by the maximum amplitude to improve numerical stability.

$$I_k = -\sum_i \Delta w_i^{(k)} \frac{\partial L}{\partial w_i^{(k)}} \qquad\qquad \widehat{I}_k = I_k / \max(|I_1|, |I_2|, ...) \tag{10}$$

Intuitively, if the training adopts naive gradient descent algorithms, each weight update Δw_i should be equivalent to $-\alpha \partial L / \partial w_i$ so that Eq. (10) can be simplified, where α is the learning rate. However, modern NN training usually includes schedulers (e.g., cosine decay) and advanced optimizers with momentum and runtime scaling, such as Adam (Kingma & Ba, 2014) and AdamW (Loshchilov & Hutter, 2017). A general expression of the weight update is

$$\Delta w_i^{(k)} = -\mathbf{Scheduler}(\alpha) \cdot \mathbf{Optimizer}\left(\partial L / \partial w_i^{(k)}\right) \tag{11}$$

In practice, most existing NN libraries don't allow retrieving the runtime status of the scheduler and optimizer for Eq. (11) calculation. Instead, we run a probing iteration to obtain the weight update by $\Delta \boldsymbol{w} = \boldsymbol{w}_{t+1} - \boldsymbol{w}_t$. On the other hand, to prevent this probing iteration from affecting the training dynamics, we cache the original weights $\boldsymbol{w}_t$ before probing, and restore them afterward.

Reducing memory usage. Our importance evaluation involves caching original model weights, which significantly increases the GPU memory consumption. Especially for LLMs with tens of gigabytes of disk size, duplicating the model weights on the GPU can lead to out-of-memory. Instead, we suggest offloading the weights and gradients in Eq. 10 to the main memory and letting the CPU run the final aggregation. Although the CPU is slower and the data transmission between the GPU and CPU brings extra latency, we observe that even in per-epoch frequency, such overhead is negligible compared to the time span of LLMs training.

Based on our problem formulation 5, under an objective FLOPs constraint, some tensors in early layers will anyway not be selected for training due to the high cost of propagating activation gradients. We can exclude evaluating these tensors' importance and save a significant amount of computation and GPU memory. Specifically, the backpropagation during importance evaluation can be early stopped at a certain tensor k, such that the cumulative FLOPs just exceeds the objective constraint:

$$\sum_{i=k-1,\ldots,N} t_{dy}^{(i)} < \rho T_{full} \leq \sum_{i=k,\ldots,N} t_{dy}^{(i)} \tag{12}$$

where $t_{dy}^{(i)}$ is the FLOPs from profiling in Section 3.1. By applying such early-stopping, we observe up to 10%-20% of GPU memory saving compared to full importance evaluation.

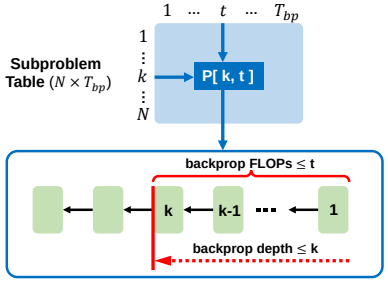
3.3 TENSOR SELECTION

Since Eq. 5 is a nonlinear integer programming problem and hence NP-hard, we will solve it in pseudo-polynomial time by dynamic programming (DP). Specifically, we decompose the whole problem into many subproblems which are constrained by different depths of backpropagation. These subproblems can be sequentially solved from the easiest one with the smallest depth, by using their recurrence relations. Since we only focus on backpropagation FLOPs, we can equivalently convert the original objective ρ w.r.t the (forward + backpropagation) FLOPs into the one only w.r.t the backpropagation FLOPs as ρ_{bp}. The equivalent constraint and ρ_{bp} are:

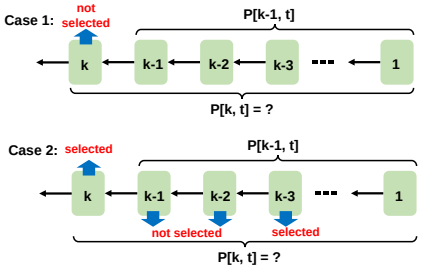
$$\boldsymbol{m} \cdot \boldsymbol{t}_{dw} + \sigma(\boldsymbol{m}) \cdot \boldsymbol{t}_{dy} \leq \rho_{bp} T_{bp} \qquad \rho_{bp} = (1 + T_{fp}/T_{bp})\rho - T_{fp}/T_{bp} \tag{13}$$

where T_{bp} is the full backpropagation FLOPs. Although $T_{fp}/T_{bp} \approx 1/2$ generally holds for many NN models (e.g., MLPs), we still explicitly compute T_{fp}/T_{bp} for a given LLM to ensure preciseness.

Subproblem definition. As shown in Figure 4(a), we define each subproblem $P[k, t]$ as to maximize the cumulative importance of selected tensors when 1) selection is among the top[1] k tensors and 2) backpropagation FLOPs is at most t. DP starts by solving the smallest subproblem $P[k = 1, t = 1]$ and gradually solves larger subproblems based on the results of smaller subproblems. The key challenge is how to find the recurrence relation of these subproblems.



(a) Subproblem definition



(b) Finding recurrence relations

Figure 4: Solving the selection problem by DP

Recurrence relations of subproblems. The recurrence relation between subproblem $P[k, t]$ and $P[k-1, t]$ depends on whether we further select the top tensor k from the solution of $P[k-1, t]$. As shown in Figure 4(b): **Case 1:** If the top k tensor is not selected, $P[k, t]$ will fall back to $P[k-1, t]$ since the importance of tensor selection cannot be further increased. **Case 2:** If the top k tensor is selected, then two portions of FLOPs will be surely included in the solution of $P[k, t]$, no matter which other tensors are selected: 1) the time to update tensor k; 2) the FLOPs to pass activation gradients from the closest selected tensor k_c, such as tensor $k-3$ as shown in Figure 4(b), to tensor k. This implies that $P[k, t]$ can fall back to a previously solved subproblem $P[k-k_c, t-\Delta t]$ where:

$$\Delta t = t_{dw}^{(k)} + \sum\nolimits_{j=k_c}^{k-1} t_{dy}^{(j)}. \tag{14}$$

Since k_c is unknown in advance, we backtrace the previously solved subproblems and explore all the possibilities of k_c, by reducing the depth of backpropagation from k.

As a result, the optimal solution to $P[k, t]$ is the one with higher cumulative importance of selected tensors between Case 1 and 2. Based on this recurrence relation, we can solve all the subproblems by sequentially traversing the subproblem space. The time complexity of solving each subproblem is $O(N)$ due to backtracing in Case 2, and the overall time complexity of DP algorithm is $O(N^2 T_{bp})$.

Reducing computational cost. Due to the large value of FLOPs, there could be billions of subproblems in practical scenarios. To reduce the computational cost of DP, we reduce the subproblem space by skipping two types of subproblems: 1) **invalid ones**, whose FLOPs constraint t exceeds the desired constraint ($\rho_{bp} T_{bp}$); 2) **redundant ones**, whose FLOPs to pass activation gradients to the maximally allowed depth (k) exceeds t. Doing so on OPT model with $\rho_{bp} = 50\%$ can reduce the number of subproblems by 5.5× without affecting optimality.

To further reduce the number of subproblems, we scale tensors' FLOPs (t_{dw}, t_{dy}) by multiplying a factor of Z:

$$\widetilde{t_{dw}} = \lfloor t_{dw} \cdot Z \rfloor, \quad \widetilde{t_{dy}} = \lfloor t_{dy} \cdot Z \rfloor, \tag{15}$$

[1] We consider the tensor that is closest to the NN output as the topmost.

where $Z = \frac{T_q}{T_{bp}}$ and the backropagation FLOPs is reduced to a resolution $T_q < T_{bp}$. The overall time complexity of DP is then reduced to $O(N^2 T_q)$. Such reduced resolution could increase the ambiguity in DP and affect optimality. To avoid this issue, we will run DP with different resolutions and find the best resolution that balances optimality and computational cost. On the other hand, since the time complexity of DP increases quadratically with N, its computing cost could still be high when being applied to extremely big LLMs (e.g., GPT-3). In these cases, we could further leverage the existing parallelization techniques (e.g., multithreading Tan et al. (2008)) and hardware accelerators (e.g., GPUs) to speed up DP.

4 EXPERIMENTS

We implement GreenTrainer in PyTorch and conduct our experiments on a Lambda Cloud instance with one Nvidia H100 80GB GPU and 24 vCPUs. In our evaluation, we choose multiple recently open-sourced decoder-only LLMs: OPT (Zhang et al., 2022) and BLOOMZ (Muennighoff et al., 2022) and the encoder-decoder LLM: FLAN-T5 (Chung et al., 2022). All the model structures and pre-trained weights are downloaded from Hugging Face (Wolf et al., 2019) model hub. The number of parameters ranges from 350M to 6.7B depending on specific model variants. In evaluation, we conduct experiments on two abstractive summarization datasets:

• **SciTLDR** (Cachola et al., 2020) is a dataset of 5.4K summaries over 3.2K papers. It contains both author-written and expert-derived TLDRs, where the latter are collected using a novel annotation protocol that produces high-quality summaries while minimizing the annotation burden. We let LLMs summarize the paper abstracts.

• **DialogSum** (Chen et al., 2021) is a large-scale dialogue summarization dataset, consisting of 13,460 dialogues with corresponding manually labeled summaries and topics. It has been demonstrated more challenging than other summarization datasets, such as SAMSum (Gliwa et al., 2019), CNN/Daily (Nallapati et al., 2016) at a similar scale.

For OPT and BLOOMZ, we follow GPT2-like (Radford et al., 2019) prompt structures: "[source seq.] TL;DR: " for summarization tasks to preprocess all the input data. For FLAN-T5, we adopt the prompt structure: "summarize: [source seq.]" which is used in the original T5 pre-training. We truncate the source sequences so that every preprocessed input sequence doesn't exceed 512 tokens before being fed to the model. On the test data, we use the beam search size of 4, and set the maximum number of the generated tokens to 64 for SciTLDR and 128 for DialogSum.

In particular, we don't consider non-generative tasks, such as sentimental classification, entailment classification, and extractive QA. Not only because they are not application venues for today's LLMs, but also because they are likely to be over-killed by LLMs which results in exaggerated performance gain over the baseline. On SciTLDR and DialogSum, we compare the performance of GreenTrainer (GT) with the following four baselines:

• **Full Fine-Tuning (Full FT)** fine-tunes all the LLM parameters and is the most common way for downstream task adaptation. Intuitively, it should bring the best training quality and GreenTrainer should try to align with its accuracy.

• **Fine-Tuning Top2 (FT-Top2)** only fine-tunes the last two layers of the LLM, which typically include the embedding layer and a LayerNorm. The input and output embedding layers are tied for OPT and BLOOMZ, but are not tied for FLAN-T5. This naive baseline only fine-tunes the smallest portion of parameters and is used to identify whether the dataset is trivial to the LLM.

• **Prefix Tuning (Prefix-T)** (Li & Liang, 2021) is a popular method for efficient fine-tuning of LLMs. It inserts trainable prefixes into each block's input sequence while freezing the model parameters. For encoder-decoder LLMs, the trainable prefixes are only inserted into the decoder blocks.

• **LoRA** (Hu et al., 2021) is another state-of-the-art method for efficient fine-tuning and becomes the most successful in the LLM era. It uses low-rank matrix decomposition to reduce the computation cost. Following the original paper, we apply LoRA to query and value projectors.

In all experiments, we use a batch size of 4 and fine-tune the model for 5 epochs. We adopt AdamW optimizer (Loshchilov & Hutter, 2017) at a learning rate of 2×10^{-5} with linear schedule and weight decay of 10^{-2}. We report ROUGE scores (%R1/R2/RL) (Lin, 2004) on test datasets as the accuracy metric, training Peta-FLOPs (PFLOPs), and wall-clock time for each run.

Table 2: Training Cost & Accuracy

# Model & Method	SciTLDR			DialogSum		
	PFLOPs	Time (h)	R1/R2/RL	PFLOPs	Time (h)	R1/R2/RL
OPT-2.7B						
Full FT	41.8	0.92	32.9/14.9/27.1	262.0	5.5	23.6/9.5/18.8
FT-Top2	29.0 (31%↓)	0.61 (34%↓)	9.1/4.0/7.6	181.6 (31%↓)	3.8 (31%↓)	20.8/7.9/17.5
Prefix-T	27.9 (33%↓)	0.58 (37%↓)	7.6/0.4/6.1	174.7 (33%↓)	3.7 (33%↓)	13.4/3.3/10.9
LoRA	27.9 (33%↓)	0.59 (36%↓)	28.2/12.1/21.0	174.7 (33%↓)	3.6 (35%↓)	23.8/9.5/18.8
GT-0.5	20.8 (50%↓)	0.46 (50%↓)	30.5/13.1/25.2	130.1 (50%↓)	2.7 (51%↓)	21.4/8.2/17.6
GT-0.7	29.2 (30%↓)	0.68 (26%↓)	33.1/15.2/27.6	182.7 (30%↓)	4.0 (27%↓)	26.8/11.0/21.6
BLOOMZ-3B						
Full FT	47.2	1.0	28.3/12.1/22.5	294.8	6.5	26.1/10.6/21.0
FT-Top2	36.5 (23%↓)	0.75 (25%↓)	23.7/8.8/18.8	227.9 (23%↓)	4.6 (29%↓)	22.1/8.5/17.8
Prefix-T	31.5 (33%↓)	0.68 (34%↓)	6.5/2.2/5.5	196.5 (33%↓)	4.2 (35%↓)	29.6/9.4/24.9
LoRA	31.5 (33%↓)	0.69 (33%↓)	27.4/11.7/21.8	196.5 (33%↓)	4.3 (34%↓)	35.4/14.3/28.6
GT-0.5	23.4 (51%↓)	0.51 (50%↓)	26.7/10.7/21.2	146.4 (50%↓)	3.1 (52%↓)	24.9/9.5/20.0
GT-0.7	32.3 (32%↓)	0.74 (28%↓)	28.0/12.2/22.4	204.7 (31%↓)	4.3 (34%↓)	36.8/14.7/29.4
FLAN-T5-3B						
Full FT	21.7	0.64	37.1/18.5/31.7	135.7	4.0	46.5/20.8/38.5
FT-Top2	7.3 (66%↓)	0.21 (67%↓)	36.5/18.4/31.5	46.1 (66%↓)	1.4 (65%↓)	39.2/16.7/32.9
Prefix-T	8.0 (63%↓)	0.23 (64%↓)	36.0/18.2/31.0	55.3 (60%↓)	1.7 (57%↓)	37.6/16.4/32.1
LoRA	14.4 (33%↓)	0.41 (36%↓)	36.6/18.5/31.5	90.5 (33%↓)	2.5 (38%↓)	44.7/19.8/37.1
GT-0.34	7.5 (65%↓)	0.23 (64%↓)	36.4/18.4/31.7	53.5 (61%↓)	1.4 (65%↓)	42.7/18.3/35.1
GT-0.4	10.0 (54%↓)	0.38 (41%↓)	36.7/18.5/31.5	62.5 (54%↓)	2.3 (43%↓)	46.0/20.7/38.1
GT-0.5	12.4 (43%↓)	0.44 (31%↓)	36.3/17.7/30.9	77.6 (43%↓)	2.6 (35%↓)	46.2/20.7/38.1

4.1 Training Cost & Accuracy

We first compare GreenTrainer with other baseline schemes on 3B models and both datasets. We set the GreenTrainer's FLOPs objective to be (0.5, 0.7) for OPT and BLOOMZ, (0.34, 0.4, 0.5) for FLAN-T5, for pair-wise comparison with baselines at different levels of the training cost. As shown in Table 2, for OPT-2.7B, GT-0.5 can save 50% training FLOPs and wall-clock time with at most 2% accuracy loss on both datasets, and GT-0.7 can even achieve 0.2%-3% higher ROUGE scores than Full FT. We hypothesize that GT mitigates the overfitting by only fine-tuning the top important tensors. However, insufficient trainable parameters can also lead to underfitting, as FT-Top2 has significantly lower ROUGE scores than all other schemes, which indicates the fine-tuning task is non-trivial for OPT-2.7B. Compared to LoRA and Prefix Tuning, GT-0.7 achieves at least 2% higher accuracy under the same training FLOPs.

Similarly, for BLOOMZ-3B, GT-0.5 can save 50% training FLOPs and wall-clock time with $< 2\%$ accuracy loss. Compared to Full FT, GT-0.7 achieves the same ROUGE scores on SciTLDR, and 4% to 10% higher on DialogSum. Under the same training FLOPs, GT-0.7 has 0.4%-1.4% higher ROUGE scores than the best baseline LoRA. The datasets are non-trivial for BLOOMZ since the naive baseline FT-Top2 still loses accuracy significantly. Note that Prefix tuning performs much worse than any other baselines on SciTLDR. We suspect the inserted trainable prefixes break the original prompt structure and confuse the model learning on the scientific corpus.

For FLAN-T5-3B, we observe that FT-Top2 achieves similar fine-tuning qualities to Full FT, with significant FLOPs reduction, which indicates that SciTLDR is trivial for FLAN-T5. This is because FLAN-T5 has been instruction-fine-tuned after pre-training, and can potentially have better zero-shot adaptability. In that case, GT-0.34 can achieve the same training FLOPs and ROUGE scores by not selecting most of the tensors. On the other hand, FT-Top2 loses accuracy significantly on DialogSum. GT-0.4 reduces 54% training FLOPs and 43% wall-clock time without noticeable accuracy loss to Full FT. GT-0.4 also outperforms LoRA by 1% on ROUGE scores and reduces 11% more training FLOPs. Compared to Prefix tuning, GT-0.34 achieves 2%-5% higher ROUGE scores while reducing the same amount of training FLOPs.

Table 3: Impact of FLOPs Objective (OPT-2.7B)

Method	SciTLDR			DialogSum		
	PFLOPs	**Time (h)**	**R1/R2/RL**	**PFLOPs**	**Time (h)**	**R1/R2/RL**
Full FT	41.8	0.92	32.9/14.9/27.1	262.0	5.5	23.6/9.5/18.8
LoRA	27.9 (33%↓)	0.59 (36%↓)	28.2/12.1/21.0	174.7 (33%↓)	3.6 (35%↓)	23.8/9.5/18.8
GT-0.36	14.9 (64%↓)	0.32 (65%↓)	4.1/1.7/3.6	92.9 (65%↓)	1.9 (65%↓)	15.7/5.0/13.8
GT-0.4	16.6 (60%↓)	0.36 (61%↓)	28.6/11.6/23.5	103.4 (61%↓)	2.2 (60%↓)	17.9/6.3/15.4
GT-0.5	20.8 (50%↓)	0.46 (50%↓)	30.5/13.1/25.2	130.1 (50%↓)	2.7 (51%↓)	21.4/8.2/17.6
GT-0.6	25.0 (40%↓)	0.56 (39%↓)	33.4/15.3/27.8	156.6 (40%↓)	3.3 (40%↓)	24.0/9.7/19.2
GT-0.7	29.2 (30%↓)	0.68 (26%↓)	33.1/15.2/27.6	182.7 (30%↓)	4.0 (27%↓)	26.8/11.0/21.6
GT-0.8	33.4 (20%↓)	0.77 (16%↓)	33.1/15.5/27.6	209.6 (20%↓)	4.4 (20%↓)	23.9/9.9/19.1

4.2 Impact of FLOPs Objective

To better understand GreenTrainer's cost-accuracy efficiency, we fine-tune OPT-2.7B and vary the FLOPs objective in a more fine-grained manner. We mainly compare GT with LoRA since it performs the best among all the baselines. As shown in Table 3, overall GT achieves significantly better cost-accuracy efficiency than the baselines. On SciTLDR, by increasing the FLOPs objective, GT-0.4 first outperforms LoRA, and it achieves 2% higher ROUGE scores and saves 25% more FLOPs and wall-clock time. Further, GT-0.6 is the first to outperform Full FT on ROUGE scores and saves 40% of training FLOPs and 39% of wall-clock time. Similarly, on DialogSum, GT-0.6 first outperforms both Full FT and LoRA on ROUGE scores. It saves 40% of training FLOPs from Full FT and saves 7% more than LoRA.

Since GreenTrainer targets saving fine-tuning computation, it's impractical for users to enumerate all the possible FLOPs objectives to find the best trade-off. Based on the above empirical study, we suggest users set 0.6 as a generally good FLOPs objective, which can bring 40% training cost reduction without a noticeable risk of accuracy loss.

Table 4: Efficacy of Tensor Importance Metrics (OPT-2.7B)

Method	SciTLDR			DialogSum		
	PFLOPs	**Time (h)**	**R1/R2/RL**	**PFLOPs**	**Time (h)**	**R1/R2/RL**
Full FT	41.8	0.92	32.9/14.9/27.1	262.0	5.5	23.6/9.5/18.8
GT-0.7 (Δw)	29.4 (30%↓)	0.68 (26%↓)	32.7/15.2/27.2	183.8 (30%↓)	4.0 (27%↓)	24.9/10.2/19.7
GT-0.7 ($\frac{\partial L}{\partial w}$)	29.4 (30%↓)	0.67 (27%↓)	32.8/15.1/27.2	184.0 (30%↓)	4.0 (27%↓)	25.0/10.2/20.0
GT-0.7 ($\Delta w \frac{\partial L}{\partial w}$)	29.2 (30%↓)	0.68 (26%↓)	33.1/15.2/27.6	182.7 (30%↓)	4.0 (27%↓)	26.8/11.0/21.6

4.3 Efficacy of Tensor Importance Metrics

The fine-tuning quality of GreenTrainer is ensured by the effectiveness of tensor importance evaluation. We compare our metric ($\Delta w \frac{\partial L}{\partial w}$) to the magnitude-based metric (Δw) (Lee et al., 2020) and the gradients-only metric ($\frac{\partial L}{\partial w}$) (Aji & Heafield, 2017) with OPT-2.7B at the FLOPs objective of 0.7. As shown in Table 4, with the same training cost reduction, $\Delta w \frac{\partial L}{\partial w}$ achieves the highest accuracy and outperforms Full FT by 1%-3% on ROUGE scores. This is because magnitude-based metrics ignore the dependencies of weight updates, and gradient-only metrics only contain the direction information about tensor importance and cannot reflect the intensity of importance. Inaccurate importance measurements will in turn lead to the inefficient selection of trainable tensors.

4.4 Impact of Model Complexity

A specific type of LLM may contain several variants with different parameter sizes. To study GreenTrainer's performance with different model complexities, we run OPT model with parameter sizes from 350M to 6.7B at the FLOPs objective of (0.4, 0.5, 0.7). As shown in Table 5, even on small models (OPT-350M), GT-0.5 can save 17%-21% more training FLOPs than LoRA while achieving the 2%-4% higher accuracy on SciTDR and the same accuracy on DialogSum. For OPT-1.3B

and OPT-2.7B, GT-0.5 outperforms LoRA and GT-0.7 outperforms Full FT on SciTLDR. On DialogSum, GT-0.7 performs similarly compared to LoRA. Since Full FT of OPT-6.7B causes out-of-memory on the single H100 80GB GPU, we only compare GT with LoRA. On SciTLDR, GT-0.4 can save 27% training FLOPs while achieving the same training quality as LoRA.

Table 5: Impact of Model Complexity (OPT)

# Params & Method	SciTLDR			DialogSum		
	PFLOPs	Time (h)	R1/R2/RL	PFLOPs	Time (h)	R1/R2/RL
350M						
Full FT	5.4	0.15	30.9/13.9/25.7	33.8	0.92	23.2/9.0/18.5
LoRA	3.6 (33%↓)	0.10 (33%↓)	25.9/10.8/20.3	22.5 (33%↓)	0.65 (29%↓)	21.5/7.7/17.3
GT-0.4	2.1 (61%↓)	0.06 (60%↓)	27.7/12.2/23.4	13.3 (61%↓)	0.36 (61%↓)	17.3/5.8/14.6
GT-0.5	2.7 (50%↓)	0.08 (47%↓)	29.9/13.2/24.9	16.7 (51%↓)	0.45 (51%↓)	21.3/7.8/17.3
GT-0.7	3.8 (30%↓)	0.12 (20%↓)	30.6/13.5/25.0	23.6 (30%↓)	0.66 (28%↓)	24.2/9.3/19.3
1.3B						
Full FT	20.8	0.46	32.1/14.3/26.4	130.8	2.9	25.4/10.3/20.2
LoRA	13.9 (33%↓)	0.31 (33%↓)	28.1/11.9/22.0	87.2 (33%↓)	1.9 (34%↓)	24.6/9.9/19.4
GT-0.4	8.2 (61%↓)	0.18 (61%↓)	28.9/11.9/23.8	51.4 (61%↓)	1.1 (62%↓)	16.9/5.7/14.6
GT-0.5	10.3 (50%↓)	0.23 (50%↓)	30.0/12.7/24.5	64.2 (51%↓)	1.4 (51%↓)	20.1/7.4/16.7
GT-0.7	14.5 (30%↓)	0.34 (26%↓)	31.2/14.2/25.8	90.8 (30%↓)	2.0 (31%↓)	24.4/9.7/19.4
2.7B						
Full FT	41.8	0.92	32.9/14.9/27.1	262.0	5.5	23.6/9.5/18.8
LoRA	27.9 (33%↓)	0.59 (36%↓)	28.2/12.1/21.0	174.7 (33%↓)	3.6 (35%↓)	23.8/9.5/18.8
GT-0.4	16.6 (60%↓)	0.36 (61%↓)	28.6/11.6/23.5	103.4 (61%↓)	2.2 (60%↓)	17.9/6.3/15.4
GT-0.5	20.8 (50%↓)	0.46 (50%↓)	30.5/13.1/25.2	130.1 (50%↓)	2.7 (51%↓)	21.4/8.2/17.6
GT-0.7	29.2(30%↓)	0.68 (26%↓)	33.1/15.2/27.6	182.7 (30%↓)	4.0 (27%↓)	26.8/11.0/21.6
6.7B						
Full FT	103.9	-	-	649.9	-	-
LoRA	69.3 (33%↓)	1.3	28.4/12.3/22.7	433.3 (33%↓)	8.1	24.9/10.2/19.4
GT-0.4	41.2 (60%↓)	0.9	28.9/11.8/23.4	257.9 (60%↓)	5.2	19.7/7.0/16.3

5 Conclusion & Broader Impact

In this paper, we present GreenTrainer, a new fine-tuning technique for LLMs, that allows efficient selection of trainable parameters via elastic backpropagation, so as to achieve high training quality while reducing computation cost significantly. GreenTrainer can save up to 64% training FLOPs compared to full fine-tuning without noticeable accuracy loss. Compared to existing schemes Prefix Tuning and LoRA, GreenTrainer can achieve up to 4% accuracy improvement at the same level of FLOPs saving.

Although we target LLM fine-tuning in this paper, the rationale of GreenTrainer's elastic backpropagation can also be extended to large generative models in other fields, such as Stable Diffusion (Rombach et al., 2022) for image generation and PaLM-E (Driess et al., 2023) for motion planning of multimodal embodied agents. We leave these extensions as our future work.

References

2023 ai index report. `https://aiindex.stanford.edu/report/`, 2023.

Martín Abadi. Tensorflow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 1–1, 2016.

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

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Isabel Cachola, Kyle Lo, Arman Cohan, and Daniel S Weld. Tldr: Extreme summarization of scientific documents. *arXiv preprint arXiv:2004.15011*, 2020.

Arno Candel, Jon McKinney, Philipp Singer, Pascal Pfeiffer, Maximilian Jeblick, Prithvi Prabhu, Jeff Gambera, Mark Landry, Shivam Bansal, Ryan Chesler, et al. h2ogpt: Democratizing large language models. *arXiv preprint arXiv:2306.08161*, 2023.

Yulong Chen, Yang Liu, Liang Chen, and Yue Zhang. Dialogsum: A real-life scenario dialogue summarization dataset. *arXiv preprint arXiv:2105.06762*, 2021.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

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.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. *arXiv preprint arXiv:1911.12237*, 2019.

Yunhui Guo, Yandong Li, Liqiang Wang, and Tajana Rosing. Adafilter: Adaptive filter fine-tuning for deep transfer learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4060–4066, 2020.

Robin Hesse, Simone Schaub-Meyer, and Stefan Roth. Fast axiomatic attribution for neural networks. *Advances in Neural Information Processing Systems*, 34:19513–19524, 2021.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023.

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

Alex M Lamb, Anirudh Goyal ALIAS PARTH GOYAL, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks. *Advances in neural information processing systems*, 29, 2016.

Jaeho Lee, Sejun Park, Sangwoo Mo, Sungsoo Ahn, and Jinwoo Shin. Layer-adaptive sparsity for the magnitude-based pruning. *arXiv preprint arXiv:2010.07611*, 2020.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

Baohao Liao, Shaomu Tan, and Christof Monz. Make your pre-trained model reversible: From parameter to memory efficient fine-tuning. *arXiv preprint arXiv:2306.00477*, 2023.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W04-1013.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 61–68, 2022.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Kevin Lu, Aditya Grover, Pieter Abbeel, and Igor Mordatch. Pretrained transformers as universal computation engines. *arXiv preprint arXiv:2103.05247*, 1, 2021.

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.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng-Xin Yong, Hailey Schoelkopf, et al. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*, 2022.

Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*, 2016.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

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.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, 63(12):54–63, 2020.

Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan. Fine-tuned language models are continual learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 6107–6122, 2022.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pp. 3319–3328. PMLR, 2017.

Guangming Tan, Ninghui Sun, and Guang R Gao. Improving performance of dynamic programming via parallelism and locality on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):261–274, 2008.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

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.

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.

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.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*, 2023.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.