Efficient Ensemble for Fine-tuning Language Models on Multiple Datasets

Dongyue Li[†]   **Ziniu Zhang**[†]   **Lu Wang**[‡]   **Hongyang R. Zhang**[†]

[†]Northeastern University, Boston, MA
[‡]University of Michigan, Ann Arbor, MI

Abstract

This paper develops an ensemble method for fine-tuning a language model to multiple datasets. Existing methods, such as quantized LoRA (QLoRA), are efficient when adapting to a single dataset. When training on multiple datasets of different tasks, a common setup in practice, it remains unclear how to design an efficient adaptation for fine-tuning language models. We propose to use an ensemble of multiple smaller adapters instead of a single adapter per task. We design an efficient algorithm that partitions n datasets into m groups, where m is typically much smaller than n in practice, and train one adapter for each group before taking a weighted combination to form the ensemble. The algorithm leverages a first-order approximation property of low-rank adaptation to quickly obtain the fine-tuning performances of dataset combinations since methods like LoRA stay close to the base model. Hence, we use the gradients of the base model to estimate its behavior during fine-tuning. Empirically, this approximation holds with less than 1% error on models with up to 34 billion parameters, leading to an estimation of true fine-tuning performances under 5% error while speeding up computation compared to base fine-tuning by 105 times. When applied to fine-tune Llama and GPT models on ten text classification tasks, our approach provides up to 10% higher average test accuracy over QLoRA, with only 9% more FLOPs. On a Llama model with 34 billion parameters, an ensemble of QLoRA increases test accuracy by 3% compared to QLoRA, with only 8% more FLOPs.

1 Introduction

Parameter-efficient fine-tuning has emerged as an efficient approach to adapting a large language model (LLM) to a downstream application. In practice, one often runs into a situation where the evaluation criterion involves a mix of many datasets or tasks. Thus, it would be desirable to have a fine-tuning method that can deliver robust performance in the presence of multiple training datasets. Directly applying a base fine-tuning method such as low-rank adaptation (Hu et al., 2021) or adapter-tuning (Houlsby et al., 2019) can lead to negative interference across tasks, as the optimal adapter weights for different datasets can conflict with each other (Wu et al., 2020; Yang et al., 2020). In this paper, we study the design of ensemble methods for parameter-efficient fine-tuning of language models in the presence of multiple datasets.

Besides being a common problem for evaluating LLMs (Hendrycks et al., 2021), other cases where the above problem arises include fine-tuning transformer models on human behavior data and cognitive psychology benchmarks (Coda-Forno et al., 2024), as well as reinforcement learning (Sodhani et al., 2021), and embodied reasoning such as task and motion planning.

One natural approach to the above problem would be to first pre-train a shared adapter on all the datasets and then fine-tune this adapter specifically on each dataset, akin to the paradigm of multi-task pretraining followed by supervised fine-tuning (namely MTL-FT in short) (Wang et al., 2019b; Liu et al., 2019). However, this would increase the computational overhead by a factor of two (one for pretraining and the other for task-specific fine-tuning), which would be costly when the dataset sizes are large. Directly using a parameter-efficient fine-tuning method such as LoRA (Hu et al., 2021) or QLoRA (Dettmers et al., 2023) can lead to negative transfer as the weights between the low-rank factors trained on one dataset can differ from another (See Section 2.1 for our experiments confirming this point).

A key technical challenge to ensure robust generalization across n datasets is to understand the relationship between the n datasets when they are

---

Email correspondence to {li.dongyu, zhang.zini, ho.zhang}@northeastern.edu and wangluxy@umich.edu.
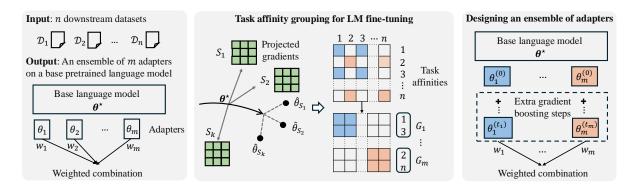
Figure 1: **Left**: We propose an ensemble method for fine-tuning language models on multiple datasets. Given n datasets and a base adaptation method such as LoRA, we design an ensemble of adapters that applies weighted averaging to their outputs, with minimal computation and memory overhead to the base method. **Middle:** We partition n datasets into m groups based on the task affinities scores. Our method first estimates fine-tuning performances on multiple dataset combinations, $S_1, S_2, \ldots, S_k$, by evaluating the gradients of the adapter weights at $\theta^\star$. For each subset S_i, we estimate the fine-tuned adapter weights $\hat{\theta}_{S_i}$ by solving a regression problem, using the projected gradients within S_i as features. This leads to an n by n task affinity matrix T, where $T_{i,j}$ is the affinity score computed from the estimates (see equation (1)). We then partition the n datasets into m groups with a clustering algorithm applied to T. In practice, m is usually much smaller than n. **Right:** We design an adapter ensemble by fine-tuning one adapter per group and further refining it with a few gradient-boosting steps. This overall procedure incurs little computational overhead, as we will describe in Table 1.

trained on top of an LLM. In this vein, one might also use influence functions (Koh and Liang, 2017), which would require scaling up influence computation to subsets of dataset combinations. Recent developments in data modeling (Ilyas et al., 2022) show that by randomly sampling subsets of data, one can estimate the influence of adding or removing one sample from the dataset upon the trained model efficiently (Park et al., 2023; Li et al., 2023). Inspired by this line of work, we propose an efficient estimation of model fine-tuning performances *without performing any fine-tuning*. We begin by noting that for fine-tuning transformer models such as Llama and GPT, the (low-rank) adapter weights typically stay very close to the base model ($\approx 0.2\%$ evaluated on a Llama model with 34 billion parameters). Thus, one can expand the loss using first-order Taylor's expansion around the initialization and then use the gradients measured at the base model to approximate model fine-tuning behavior. Empirically, we find that this approximation incurs less than 5% error for a variety of fine-tuning methods, including LoRA, adapter-tuning, quantized LoRA, and quantized adapter.

Provided with efficient estimation of fine-tuning performances on dataset combinations, we then design an ensemble of low-rank adapters. At a high level, given n datasets, our algorithm first partitions the n datasets into m similar groups (where m is typically much less than n). Concretely, we

achieve this based on a task affinity matrix that we estimated using the above first-order approximation property. Then, we fine-tune one adapter in each group, leading to m adapters in total. We further refine the adapters by applying a few gradient-boosting steps to reduce the loss of groups with the highest training losses. Importantly, except for computing the gradients at the base model, which we use during the estimation, the rest of this procedure can be done entirely on CPUs. Additionally, the memory overhead is now roughly m times the size of each adapter. See Figure 1 for an illustration of our approach.

We extensively validate our approach, showing its broad applicability in boosting a base fine-tuning method with little computational overhead. First, we show that our estimation method approximates fine-tuning losses within 5% relative error while reducing computation by $105\times$ compared to actual fine-tuning. Second, for fine-tuning Llama models across ten tasks from SuperGLUE, our approach improves QLoRA's average test accuracy by **10**%, incurring only 9% additional computation and 9 GB additional memory. Compared to two commonly used methods (Liu et al., 2019; Fifty et al., 2021), our method achieves similar accuracy with 45% less computation and memory. Further, we scale our approach to a federated learning setting with 500 datasets, observing qualitatively similar results. Finally, we analyze

Approach	Runtime	Memory
Base fine-tuning	T	A
Pre-train then fine-tune	$\approx 2T$	nA
Our ensemble method	$T + G$	MA

Table 1: A summary of the runtime and memory usage by our approach. T is the runtime of the base fine-tuning method; G is the cost of evaluating the gradients once for all the samples at the base model. In practice, T is usually much higher than G. A is the memory used by one adapter, including activation memory. M is the number of adapters in the ensemble.

the generalization of low-rank adapters by measuring their empirical generalization errors and their sharpness. We find that small-rank LoRA adapters consistently yield the lowest generalization errors across tasks. An ensemble of low-rank adapters further reduces generalization errors compared to one adapter of the same size. We provide the code implementation of this work at github.com/VirtuosoResearch/EnsembleLoRA.

2 Methods

We now describe our approach to designing an ensemble of adapters that applies to any base adaptation method. Our method quickly partitions n datasets into m groups. Then, we train one adapter for each group, with the option of adding a few additional adapters via gradient boosting. Finally, we take a weighted combination of all the adapters as the ensemble output.

2.1 Preliminaries

We present a case study by fine-tuning Llama-3-8B on SuperGLUE (Wang et al., 2019a). We collect ten datasets, covering five categories of sentence completion (COPA, H-SWAG, and Story Cloze datasets), natural language inference (CB and RTE), coreference resolution (WSC and Winogrande), question answering (BoolQ and MultiRC), and word sense disambiguation (WiC). We evaluate LoRA, adapter, QLoRA (Dettmers et al., 2023), and quantized adapter (QAdapter). In particular, QLoRA applies 4-bit quantization to the base model, with LoRA added to the quantized base model. We evaluate memory usage with a batch size of 4 with a sequence length of 512.

First, we compare (i) fine-tuning one model on each single task and (ii) fine-tuning one model on each pairwise combination of one task with the

other nine tasks. When using full fine-tuning, we observe that for 20 (out of 45 pairs), (ii) performs worse than (i). For LoRA and adapter, this increases to 27 and 24. For QLoRA and Qadapters, this further increases to 33 and 35, respectively.

Second, we show that an ensemble of adapters can improve task performance but can increase memory cost. For each method, we train one adapter on all tasks and fine-tune it on each individual task initialized from the pretrained adapter. Then, we apply a weighted combination to the outputs of all adapters to form the ensemble. With QLoRA, the ensemble boosts accuracy by 10.8% over a single QLoRA adapter. With QAdapter, it improves accuracy by 9.5%. However, this uses 4 times more memory since it combines the outputs from 10 adapters simultaneously.

The observations indicate that a single adapter underperforms full fine-tuning due to interference between different datasets. If we use a separate adapter for each dataset, the overhead is proportional to training n adapters. Thus, the key challenge of adapting an LLM to multiple datasets is to reduce this overhead in a computationally efficient way. We propose to achieve this by grouping similar datasets so that we can share one adapter for each group of datasets.

2.2 Task affinity grouping for LM fine-tuning

The first step of our approach is to partition the n given datasets into m disjoint subsets based on their affinities trained on top of a language model. We estimate affinity scores between dataset i and j for every $i, j \in \{1, 2, \ldots, n\}$ as follows. Let S be a subset of datasets containing i and j. Denote $f_i(S)$ as the fine-tuning performance on dataset i (e.g., validation loss) using all the datasets in S. Let the task affinity score between i and j be:

$$T_{i,j} = \frac{1}{n_{i,j}} \sum_{S_k : i \in S_k, j \in S_k} f_i(S_k), \qquad (1)$$

where $\{S_k\}$ are subsets sampled from $\{1, \ldots, n\}$ and $n_{i,j}$ is the number of random subsets that contain both i, j. This score is analogous to the feature importance scores in random forests.

Computing $T_{i,j}$ requires fine-tuning the model on multiple subset combinations. Instead, we design an algorithm to estimate $f_i(S)$, which can run on CPUs and deliver estimation results for fine-tuning billion-parameter models in seconds. The key idea is based on the observation that fine-tuned

adapters stay close to the base model. Thus, applying a first-order Taylor's expansion to the base model loss yields a negligible error, which allows us to approximate fine-tuning performance using gradients calculated on the base model.

We now measure the relative distance from the fine-tuned model weights to the base model, i.e., $\frac{\|X - \theta^\star\|_F}{\|\theta^\star_{\text{Full}}\|_F}$, where X is parameter of the fine-tuned adapter, $\theta^\star$ is the initialized parameter of the adapter, and $\theta^\star_{\text{Full}}$ is the parameter of the entire pretrained model. As shown in Table 2, we find that the relative distance remains less than 0.2% on average, across LoRA, adapter, QLoRA, and QAdapter on random data subsets.

Table 2: We report the distance between fine-tuned model weights relative to the base model. The results are averaged over 50 sampled task subsets of size 3.

	Llama-3-1B	Llama-3-3B	Llama-3-8B
LoRA	$0.16_{\pm0.04}\%$	$0.14_{\pm0.02}\%$	$0.12_{\pm0.02}\%$
QLoRA	$0.18_{\pm0.02}\%$	$0.16_{\pm0.03}\%$	$0.11_{\pm0.02}\%$
Adapter	$0.09_{\pm0.03}\%$	$0.05_{\pm0.01}\%$	$0.08_{\pm0.02}\%$
QAdapter	$0.11_{\pm0.03}\%$	$0.08_{\pm0.03}\%$	$0.07_{\pm0.01}\%$

A gradient-based estimation algorithm: Let the output of a neural network be given by $h_X(s, y)$, where s is an input such as a sentence and y is the ground-truth label. Suppose that $X \in \mathbb{R}^p$ is the trainable parameter in an adapter. The Taylor's expansion of $h_X(s, y)$ centered at $\theta^\star$ is given as:

$$h_X(s, y) \approx h_{\theta^\star}(s, y) + [\nabla_X h_{\theta^\star}(s, y)]^\top (X - \theta^\star) + \epsilon_s.$$

Next, we evaluate the magnitude of ϵ_s relative to $h_X(s, y)$ and report the results in Figure 2. We illustrate the error for six language models with up to 34 billion parameters. We find that for both Llama and GPT-J models, the approximation error is less than 1% in all cases. For QLoRA and QAdapter, the approximation error is less than 3%.

Since the approximation error is small, we next apply it to approximate the model loss. We will illustrate the case for log loss with binary classification, and the same idea applies to other types of loss functions. Let $y \in \{+1, -1\}$. Recall the log loss is $\log(1 + \exp(-y \cdot h_X(s, y)))$. Define $b_s = -y \cdot h_{\theta^\star}(s, y)$ and $g_s = \nabla h_{\theta^\star}(s, y)$. Using Taylor's expansion without the higher-order terms, we approximate $h_X(s, y)$ as:

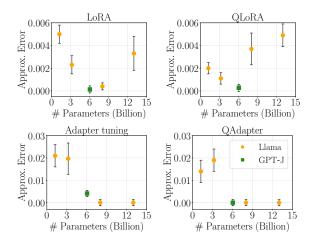$$\hat{\ell}(X) = \log\left(1 + \exp\left(b - y \cdot g^\top(X - \theta^\star)\right)\right).$$

Figure 2: We report the approximation error for Llama and GPT-J models with up to 34 billion parameters for LoRA, QLoRA, adapter, and QAdapter. We report the average and the standard deviation based on the results from 50 randomly sampled task subsets of size 3.

For a subset $S \subseteq \{1, 2, \ldots, n\}$ with training samples denoted as $\mathcal{D}_S$, we estimate fine-tuned adapter weights by minimizing averaged $\hat{\ell}(X)$ on $\mathcal{D}_S$:

$$\hat{\theta}_S \leftarrow \arg\min_{X \in \mathbb{R}^p} \frac{1}{n_S} \sum_{(s,y) \in \mathcal{D}_S} \hat{\ell}(X).$$

Using $\hat{\theta}_S$, we evaluate the estimated fine-tuning performance $\hat{f}_i(S)$ for every task $i \in S$. Then, we apply this estimation for k random subsets sampled from $[n]$ to compute $T_{i,j}$ for every i and j. This forms a $n \times n$ task affinity matrix T.

Given T, we apply a clustering algorithm to partition n datasets into m groups. The clustering objective maximizes the average density of scores within clusters, which is solved based on semi-definite programming relaxations. Additionally, we use a trace regularization term to determine the number of groups m. See details in Appendix A. Let $G_1, G_2, \ldots, G_m$ denote the resulting groups.

In summary, we only need to compute the gradients on all the training samples once at the base model. For each subset, the estimation solves a regression problem based on the gradients. A random projection is used to reduce the dimension of the gradients down to a few hundred, which provably preserves the Euclidean geometry between the gradient vectors.[1] The task affinity grouping procedure is presented in Algorithm 1.

---

[1] This can be achieved using the Johnson-Lindenstrauss Lemma. After projecting the dimension of the gradients down to several hundred, solving each logistic regression problem will only take a few seconds on a CPU.

Algorithm 1 Efficient task affinity grouping for language model fine-tuning

Input: n training/validation datasets
Require: Pretrained model $h_{\theta^\star}$, number of subsets k, and projection dimension d

1: $\{S_1, S_2, \ldots, S_k\} \leftarrow$ sample a list of subsets from $\{1, 2, \ldots, n\}$ with a fixed size
2: $P \leftarrow p$ by d Gaussian random matrix
3: **for** $(s, y) \in \mathcal{D}_{\{1,2,\ldots,n\}}$ **do**
4: $\quad \tilde{g}_s \leftarrow P^\top \nabla h_{\theta^\star}(s, y)$ ▷ project the gradient
5: $\quad b_s \leftarrow -y \cdot h_{\theta^\star}(s, y)$
6: **end for**
7: **for** $S \in \{S_1, S_2, \ldots, S_k\}$ **do**
8: $\quad \hat{X}_d \leftarrow$ solve a regression problem based on $(\tilde{g}_s, b_s, y)$ for all $(s, y) \in \mathcal{D}_S$
9: $\quad \hat{\theta}_S \leftarrow \theta^\star + P\hat{X}_d$ ▷ restore the dimension
10: $\quad \hat{f}_i(S) \leftarrow$ evaluate $h_{\hat{\theta}_S}$ on the validation set for each $i \in S$
11: **end for**
12: $T \leftarrow$ compute an n by n matrix T following equation (1)
13: $\{G_1, G_2, \ldots, G_m\} \leftarrow$ apply a clustering algorithm on T to obtain a partition of n tasks
14: Return $G_1, G_2, \ldots, G_m$

Algorithm 2 ENSEMBLELORA (Ensemble of Low-Rank Adapters for multiple datasets)

Input: n training/validation datasets
Require: Pretrained model $h_{\theta^\star}$, number of subsets k, projection dimension d, boosting steps b

1: $G_1, G_2, \ldots, G_m \leftarrow$ apply Algorithm 1 to the n input datasets
2: $\theta_1^{(0)}, \ldots, \theta_m^{(0)} \leftarrow$ Fine-tune one adapter for each group in $G_1, \ldots, G_m$
3: **for** $i = 1, \ldots, b$ **do** ▷ Boosting steps
4: $\quad G_{j_i} \leftarrow$ select the group with the highest training loss, where j_i is the group index
5: $\quad \theta_{j_i}^{(0)}, \ldots, \theta_{j_i}^{(t_{j_i})} \leftarrow$ get all the $t_{j_i} + 1$ adapters currently in G_{j_i}
6: $\quad \theta_{j_i}^{(t_{j_i}+1)} \leftarrow$ estimate an adapter to fit the negative gradients of samples in $\mathcal{D}_{G_{j_i}}$
7: **end for**
8: $w_1, w_2, \ldots, w_m \leftarrow$ train the weights for combining outputs of each group of adapters
9: Return $\left\{\theta_i^{(0)}, \ldots, \theta_i^{(t_i)}, w_i\right\}_{i=1}^m$

2.3 Designing an ensemble of adapters

The second step of our approach is to compute an adapter ensemble based on the m groups computed above. We first fine-tune one adapter on tasks in each group, yielding m adapters denoted as $\theta_1^{(0)}, \theta_2^{(0)}, \ldots, \theta_m^{(0)}$.

Next, we apply b gradient boosting steps to reduce the loss of groups with high training losses. At each step i, we choose a group G_{j_i} from $G_1, G_2, \ldots, G_m$ that has the largest training error, where j_i is the index of the group chosen at step i. Suppose there are t_{j_i} adapters for group G_{j_i}. We then fit a new adapter to predict $1 - p_s$ (the negative gradient of log loss), where p_s is the correct class prediction probability of the current model on a sample (s, y), by minimizing the mean squared loss on group G_{j_i}:

$$\min_X \frac{1}{n_{G_{j_i}}} \sum_{(s,y) \in \mathcal{D}_{G_{j_i}}} \left(1 - p_s - h_X(s, y)\right)^2.$$

We approximate the above loss based on the gradient estimation, leading to the following linear regression on group G_{j_i}:

$$\hat{\ell}(X) = \left(1 - p_s - h_{\theta^\star}(s, y) - g_s^\top(X - \theta^\star)\right)^2.$$

Let the minimizer of the above averaged over G_{j_i} be denoted as $\theta_{j_i}^{(t_{j_i}+1)}$. After obtaining this new adapter, we add it along with $\theta_{j_i}^{(0)}, \ldots, \theta_{j_i}^{(t_{j_i})}$ to form the ensemble for group G_i. The prediction for this group is based on adding all the outputs from the ensemble. In our ablation study of applying the boosting procedure on three groups of ten tasks, we found that the first boosting step alone reduces training error by 18%, leading to 0.4% average test accuracy improvement for tasks in one group.

The final ensemble contains a total of $M = m + b$ adapters. We then train the weights to form a weighted combination of the m groups. Along with the first step, we summarize the complete procedure in Algorithm 2.

3 Experiments

We evaluate our approach to answer the following questions. How accurate are the gradient-based estimates relative to the actual fine-tuning losses? What is the trade-off between performance, computation cost, and memory overhead between a base fine-tuning method and its ensemble?

As a summary of our empirical findings, we show that our approach estimates fine-tuning losses within **5%** error while using **105×** less computation than actual fine-tuning. For fine-tuning Llama models on SuperGLUE, ENSEMBLELORA boosts

the accuracy of QLoRA and QAdapter by up to **10%**, with 9% additional computation and 9 GB extra memory usage. Compared with pretraining followed by fine-tuning, our approach yields comparable performance, reducing computation and memory both by **45%**. Our approach can scale to 500 datasets and remains comparable to pretraining followed by fine-tuning, reducing computation by **90%** and memory by **91%**.

3.1 Experimental setup

Our approach applies to a wide range of parameter-efficient fine-tuning methods. For a representative evaluation, we focus on the results of using QLoRA, while the results of other methods are quantitatively similar and are deferred to Appendix 3. We fine-tune language models on ten NLP tasks from SuperGLUE. All the tasks are evaluated as classification tasks. The test accuracy is computed with the provided development set. We then split 10% of the training set as the validation set. The statistics of the datasets are summarized in Table 4. We use Llama-3.1-8B and CodeLlama-34B-Instruct as the base model. With a base fine-tuning protocol, we compare our approach against: Base fine-tuning, which trains a single adapter on the combined set of datasets; pretraining followed by fine-tuning (MTL-FT) (Liu et al., 2019), which performs multitask pretraining and then fine-tunes task-specific adapters from the pretrained adapter; task affinity grouping (TAG) (Fifty et al., 2021).

For each baseline, we assess the test accuracy averaged over all tasks. The error rate is defined as one minus the average test accuracy. We measure the computational cost as the total number of floating point operations (FLOPs) and the memory cost as the GPU during the inference phase of the model, tested on two Nvidia A6000 GPUs. In our approach, we sample $k = 200$ task subsets of size 3. We vary the number of groups m between 2 to 6 and the number of b between 1 to 4. We provide a complete list of other training parameters used by our approach in Appendix B.

3.2 Estimation results

We first evaluate the accuracy of Algorithm 1 in estimating fine-tuning losses. Since the first-order approximation incurs a negligible error in model outputs, we show that the estimated fine-tuning performance $\hat{f}_i(S)$ closely matches the actual performance $f_i(S)$. Using test accuracy as the metric, we measure the relative error between $\hat{f}_i(S)$ and

Table 3: We evaluate the error between estimated and true fine-tuning performances computed on 50 random task subsets. We report the speedup rate between the number of FLOPs used in our approach to full fine-tuning. d is the projection dimension in Algorithm 1.

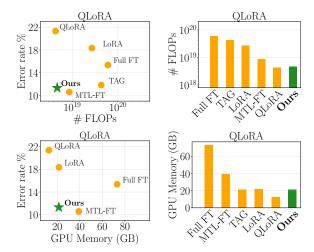| d	Llama-3-1B	Llama-3-3B	Llama-3-8B	Speedup
200	8.2%	8.1%	7.0%	105×
400	4.7%	4.8%	4.3%	105×
800	4.6%	4.4%	4.2%	105×

Figure 3: We compare error rate (one minus accuracy), computation cost, and memory usage across our approach and baselines when fine-tuning Llama-3-8B on ten NLP tasks. MTL-FT refers to first fine-tuning a shared LoRA on all the datasets, and then fine-tuning the low-rank adapter on each dataset, while Full FT refers to full fine-tuning of the entire model. Our approach boosts the test accuracy of QLoRA by 10% on average, only incurring 8% additional computation and 9 GB more memory. It performs on par with the best baseline with 45% less FLOPs.

$f_i(S)$ over 50 randomly sampled subsets of size 3. For the computation cost, we compare the number of FLOPs between actual full fine-tuning and that required in our estimation procedure. We experiment with Llama-3-1B, 3B, and 8B models, and vary the projected gradient dimension d between 200, 400, and 800.

Table 3 presents the results. Our approach achieves a relative error within 9%, using **105×** less computation than full fine-tuning. We observe that increasing d beyond 400 reduces the relative error to under **5%**, so we set $d = 400$ in our experiments. Additionally, we observe that our approach can achieve up to 0.6 correlation scores with the true fine-tuning performances.

In particular, our approach uses 6.5×10^{16}, 4.9×10^{16}, and 3.2×10^{16} FLOPS, when using

Table 4: We report the test accuracy (%) of our method, as compared with baselines. We also compute the average test accuracy across ten NLP datasets, along with the number of FLOPs and memory usage for fine-tuning Llama-3-8B using QLoRA. We run each experiment with three random seeds and report the standard deviations.

	BoolQ	CB	COPA	H-SWAG	MultiRC	Story Cloze	Winogrande	Average Accuracy	Number of FLOPs	GPU Memory
# Train	4,242	225	360	17,957	12,259	841	4,161			
# Validation	471	25	40	1,995	1,362	188	462			
# Test	3,270	56	100	10,042	4,848	1871	1267			
# classes	2	3	2	4	2	2	2			
Full FT	$87.6_{\pm0.9}$	$93.2_{\pm0.7}$	$92.0_{\pm1.0}$	$93.2_{\pm0.6}$	$86.4_{\pm0.2}$	$94.2_{\pm0.5}$	$75.1_{\pm0.2}$	$84.6_{\pm0.9}$	6.0×10^{19}	73.0GB
LoRA	$84.5_{\pm0.2}$	$87.7_{\pm0.8}$	$92.0_{\pm2.0}$	$93.4_{\pm0.3}$	$82.1_{\pm0.2}$	$92.9_{\pm0.3}$	$60.7_{\pm0.4}$	$81.6_{\pm0.8}$	2.7×10^{19}	21.5GB
QLoRA	$82.0_{\pm0.5}$	$83.7_{\pm0.7}$	$90.0_{\pm2.0}$	$92.3_{\pm0.3}$	$84.2_{\pm0.8}$	$91.5_{\pm0.7}$	$55.1_{\pm0.3}$	$78.6_{\pm1.2}$	4.3×10^{18}	12.6GB
MTL-FT	$89.9_{\pm0.2}$	$100_{\pm0.0}$	$95.0_{\pm0.5}$	$93.9_{\pm0.4}$	$89.6_{\pm0.1}$	$98.0_{\pm0.5}$	$84.4_{\pm0.3}$	$89.4_{\pm0.5}$	8.6×10^{18}	39.1GB
TAG	$88.5_{\pm0.4}$	$100_{\pm0.0}$	$94.0_{\pm0.5}$	$92.1_{\pm0.3}$	$89.1_{\pm0.7}$	$96.3_{\pm0.4}$	$80.1_{\pm0.8}$	$88.1_{\pm0.8}$	4.3×10^{19}	21.4GB
Ours	$89.9_{\pm0.9}$	$100_{\pm0.0}$	$94.0_{\pm1.0}$	$93.5_{\pm0.3}$	$89.1_{\pm0.3}$	$97.1_{\pm0.1}$	$82.0_{\pm0.5}$	$88.6_{\pm0.6}$	4.7×10^{18}	21.4GB

Llama-3-8B, 3B, and 1B models, respectively. Performing actual fine-tuning requires 6.8×10^{18}, $5.1 \times 10^{18}, 3.4 \times 10^{18}$, respectively. The speed-up ratio is computed between these two sets of results. Recall that our approach involves first evaluating the gradients of all training samples at the base model, and then solving logistic regression on the projected gradients for each subset. The gradient evaluation takes over 80% of the computation. The speed-up is calculated by the ratio between the runtime of actual fine-tuning relative to gradient evaluation, which remains consistent across the three models.

3.3 Experimental results

We illustrate the comparison of ENSEMBLELORA with baselines in Figure 3. We report the evaluation results in Table 4. Compared to QLoRA fine-tuning, our approach improves accuracy by **10%** with only 8% additional computation and 9 GB more memory. Compared to pretraining followed by fine-tuning and task grouping, it achieves comparable performance while reducing computation and memory both by 45%. Further, our approach improves over full fine-tuning by **4.0%**, with only 92% less computational cost and 52 GB less memory. Additionally, our approach also improves the base QAdapter fine-tuning method by **9%** accuracy, using 8% additional computation and 10 GB more memory. The quantitative results of using QAdapter are reported in Appendix B.

We also apply our approach to a CodeLlama-34B model using QLoRA as the base method. We observe an improved accuracy by **3.0%**, while incurring only 8% additional computation and 29 GB more memory. Compared to pretraining followed by fine-tuning, ENSEMBLELORA maintains comparable accuracy while reducing computation by

46% and memory from 66 GB to 28 GB.

Determining the ensemble size. We find that the test accuracy stabilizes after m reaches 3, so we report results using three clusters. We also find that a single boosting step suffices to reduce training error by 18%, resulting in a 0.4% increase in the average test accuracy across tasks. This leads to an ensemble of 4 adapters. In general, we choose m that maximizes the average density of task affinity scores within clusters, as it correlates with test accuracy in practice. Moreover, the boosting step b is determined by the point where the training loss no longer decreases, which can depend on the number of groups. For settings of fewer groups, a single step is often sufficient. Once task affinity is estimated, selecting these parameters takes only a few seconds, since no fine-tuning is involved.

Extension. We apply our approach to federated learning on a dataset with 500 tasks, where each speaking role in a play represents a distinct task (client). Data is split 80% for training and 20% for testing. We use QLoRA for fine-tuning, Llama-3-1B as the base model, and the Federated Averaging algorithm. We sample 2,000 subsets of size 10, varying the number of task groups m from 10 to 30 and boosting steps b from 1 to 10.

Our approach reduces test loss over QLoRA by over **9%**, while only using 8% additional computation and 15 GB more memory. Compared to pretraining followed by fine-tuning, ENSEMBLELORA gives comparable test losses while reducing computation by **90%** and memory by **91%**.

4 Generalization of Low-rank Adaptation

Next, we analyze the generalization behavior of low-rank adaptations and their ensembles. Our findings reveal that relatively small adapters often
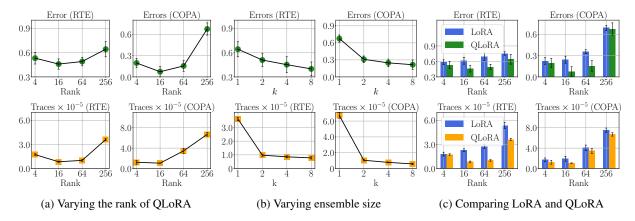
Figure 4: Illustrating the empirical generalization errors and sharpness measures with respect to QLoRA weights. (4a) Smaller adapters with a rank of 16 achieve the lowest generalization errors. Additionally, the Hessian trace values correlate with generalization errors, suggesting that smaller adapters tend to converge to flatter minima. (4b) An ensemble of k adapters leads to lower generalization errors and Hessian traces. Here, we fix the sum of the dimensions of k adapters to be equal to 256. (4c) QLoRA, which is trained on a quantized base model, yields lower generalization errors and Hessian trace values compared to LoRA.

achieve the best generalization performance. Additionally, ensembles of adapters yield lower generalization errors than individual adapters. We also observe that adapters trained on quantized models exhibit smaller generalization errors compared to those trained on full-precision models.

In more detail, we evaluate a sharpness measure based on the trace of the Hessian of the loss. Previous work has shown that this Hessian-based measure closely correlates with empirical generalization errors (Ju et al., 2022), which is derived from the PAC-Bayes generalization framework (Neyshabur et al., 2018). Specifically, we evaluate the trace of the loss's Hessian with regard to the adapter weights. Then, we compute the maximum of the trace over the training data points. A smaller trace value suggests a flatter loss landscape. We measure the empirical generalization error, defined as the difference between the test loss and the training loss of a fine-tuned model. Results for evaluating the largest eigenvalue of the loss Hessian as the sharpness measure are quantitatively similar.

For a base fine-tuning method, we vary the rank of adapter weights between 4, 16, 64, and 256 in fine-tuning Llama-3-8B. For adapter ensembles, we set a total rank of k adapters as 256. Then, we fine-tune k individual adapters with $256/k$ rank and ensemble them through weighted averaging. We vary k between 2, 4, and 8 (beyond which the ensemble exceeds the maximum memory limit). Additionally, we compare the generalization errors of fine-tuning adapters on the full-precision model (LoRA) and the 4-bit quantized model (QLoRA).

We fine-tune LoRA adapters on two NLP tasks, including RTE and COPA.

◇ Figure 4a shows the results of varying the rank of LoRA and QLoRA. We find that using a small-sized adapter with a rank of 16 typically achieves the smallest generalization error. Moreover, the Hessian-based sharpness measure correlates with the empirical generalization errors, suggesting that a smaller adapter tends to find a fatter minimum. We find that the results are consistent for other datasets and for adapter tuning, which are shown in Appendix B.5.

◇ Next, shown in Figure 4b, for ensembles of k adapters, we find that the ensemble of two adapters reduces the generalization errors by 44% and the sharpness measure by 74% on average. Increasing k further reduces the generalization errors. Similarly, the Hessian-based sharpness measure also correlates with generalization errors. This justifies the improved generalization performance of ensembles of adapters in our previous experiments.

◇ Lastly, shown in Figure 4c, we also notice that QLoRA, which is trained on the quantized model, yields lower generalization errors than those on the full precision language model. The generalization error and Hessian traces of QLoRA are 34% and 51% lower than LoRA on average.

5 Related Work

Parameter-efficient fine-tuning adapts LLMs by either fine-tuning a small subset of parameters or introducing new trainable parameters. Adapter tuning (Houlsby et al., 2019) inserts two feedfor-

ward layers per transformer layer, with one for down-projection and another for up-projection of the hidden representations. Prefix tuning (Li and Liang, 2021; Lester et al., 2021) involves prepending small, continuous task-specific soft prompts to input embeddings. LoRA (Hu et al., 2021) optimizes a low-rank decomposition of parameter updates with regard to pretrained model weights. Building on the individual tuning methods, Chen et al. (2023) propose an automatic parameter allocation algorithm to assign individual tuning methods to different parameter groups. Additionally, quantization can be used to reduce the memory costs of LoRA (Dettmers et al., 2023) by training on top of a 4-bit quantized LLM. LQ-LoRA (Guo et al., 2024) employs mixed-precision quantization to further reduce the memory cost, using integer linear programming to dynamically configure bit allocation. We refer readers to recent surveys for further references (Ding et al., 2023).

Several studies have explored training multiple adapters on top of LLMs. AdaMix (Wang et al., 2022) uses a mixture of adaptation modules per Transformer layer on a single dataset. AdaMix first trains the mixture using routing techniques from mixture-of-expert architectures (Fedus et al., 2022) and combines adapter weights through weight averaging (Wortsman et al., 2022). Similarly, there has been work on training a sparse mixture of soft prompts, using a gating mechanism to activate specific prompts per sample. Mini-ensemble of LoRA adapters (Ren et al., 2024) uses fewer trainable parameters while maintaining a higher rank of LoRA adapters to improve the performance of LoRA.

In the presence of multiple datasets, there has been work on fitting transformers that learns hypernetworks that generate adapters for every layer, which condition on task, adapter position, and layer within a transformer model. AdapterFusion (Pfeiffer et al., 2020) uses a group of adapters for multiple prediction tasks, by first learning task-specific adapters and then combining adapters with a cross-attention module to fuse the representations computed from each adapter.

Different notions of generalization in LLMs have been explored, including out-of-distribution generalization (Koh et al., 2021). There has been work on designing meta-learning algorithms for prompt tuning, enabling a single prompt to generalize across multiple datasets. Another relevant notion is cross-lingual generalization. Muennighoff et al. (2022) study prompted fine-tuning with mul-

tilingual instructions for zero-shot learning. Liu et al. (2023) propose scheduled unfreezing to mitigate catastrophic forgetting in multilingual adapter fine-tuning.

Supervision and representation strategies for language understanding have been well explored. Kozareva et al. (2011) enhance class-instance labeling using instance-instance graph propagation, showing that instance-level connections improve label coverage. Arora et al. (2016) explain the linear algebraic structures observed in word embeddings using a latent variable model that links partial mutual information with random walk dynamics. Karamanolakis et al. (2024) introduce a framework to combine rule extraction with expert feedback for efficient weak supervision. Min et al. (2023) survey pretrained language models, outlining fine-tuning, prompting, and generation paradigms along with their strengths and challenges.

A series of works have used gradient similarity to measure task similarity. Xia et al. (2024) propose selecting a subset of instruction data for targeted fine-tuning by computing gradient-based influence scores and selecting data with high cosine similarity to the target task. Park et al. (2023) introduce a scalable data attribution method that approximates model prediction losses on other data samples when adding or removing individual training samples. Li et al. (2024b,a) use gradients from a meta initialization to approximate fine-tuning loss and apply the estimates to select auxiliary tasks to jointly fine-tune with the target task. Inspired by these works, we conduct the first comprehensive study of linearization in parameter-efficient fine-tuning methods. We also propose an adapter ensemble that combines task clustering with gradient boosting.

6 Conclusion

This paper presents an ensemble method of low-rank adapters for adapting language models across multiple datasets. First, we develop an efficient task affinity grouping algorithm, with a first-order approximation for estimating task affinities and a clustering step to partition tasks into groups. Then, we construct an ensemble for groups of tasks, consisting of adapters fine-tuned on each group with additional boosting steps. Our method consistently improves fine-tuning performance with minimal computational overhead. Lastly, we analyze the sharpness measures of low-rank adapters.

Acknowledgments

Limitations

Our approach requires full access to model weights and gradients. Estimating fine-tuning performance without direct access to internal parameters remains an open question. This could involve exploring gradient-free optimization methods for closed-source models like GPT-4 and Gemini. Moreover, while our study focuses on multiple fixed datasets, other scenarios require dynamically adapting to new incoming tasks, such as in continual learning. Extending our work to actively manage a dynamic set of adapters could be a promising direction. Besides, our analysis of sharpness measures for generalization performance leaves open the question of their applicability to out-of-distribution generalization or adversarial robustness in NLP models, which may be worth considering in future work.

Broader Implications

This paper examines the problem of adapting language models to multiple datasets. While the use of language models may have potential societal consequences in the future, there are no specific concerns arising from our work. Due to the technical nature of this paper, there are no direct implications for potential negative impacts.

References

Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. 2016. A latent variable model approach to pmi-based word embeddings. *TACL*, 4:385–399.

Jiaao Chen, Aston Zhang, Xingjian Shi, Mu Li, Alex Smola, and Diyi Yang. 2023. Parameter-efficient fine-tuning design spaces. *ICLR*.

Julian Coda-Forno, Marcel Binz, Jane X Wang, and Eric Schulz. 2024. CogBench: a large language model walks into a psychology lab. In *ICML*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized llms. *NeurIPS*.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfeng Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*.

Chris Fifty, Ehsan Amid, Zhe Zhao, Tianhe Yu, Rohan Anil, and Chelsea Finn. 2021. Efficiently identifying task groupings for multi-task learning. *NeurIPS*.

Han Guo, Philip Greengard, Eric P Xing, and Yoon Kim. 2024. Lq-LoRA: Low-rank plus quantized matrix decomposition for efficient language model finetuning. *ICLR*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. *ICLR*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *ICML*.

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

Andrew Ilyas, Sung Min Park, Logan Engstrom, Guillaume Leclerc, and Aleksander Madry. 2022. Datamodels: Predicting predictions from training data. *ICML*.

Haotian Ju, Dongyue Li, and Hongyang R Zhang. 2022. Robust fine-tuning of deep neural networks with hessian-based generalization guarantees. In *ICML*.

Giannis Karamanolakis, Daniel Hsu, and Luis Gravano. 2024. Interactive machine teaching by labeling rules and instances. *TACL*, 12:1441–1459.

Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *ICML*.

Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, Tony Lee, Etienne David, Ian Stavness, Wei Guo, Berton Earnshaw, Imran Haque, Sara M Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea

Finn, and Percy Liang. 2021. WILDS: A benchmark of in-the-wild distribution shifts. In *ICML*.

Zornitsa Kozareva, Konstantin Voevodski, and Shanghua Teng. 2011. Class label enhancement via related instances. In *EMNLP*.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *EMNLP*.

Dongyue Li, Huy L Nguyen, and Hongyang R Zhang. 2023. Identification of negative transfers in multitask learning using surrogate models. *TMLR*.

Dongyue Li, Aneesh Sharma, and Hongyang R Zhang. 2024a. Scalable multitask learning using gradient-based estimation of task affinity. In *KDD*.

Dongyue Li, Ziniu Zhang, Lu Wang, and Hongyang R Zhang. 2024b. Scalable fine-tuning from multiple data sources: A first-order approximation approach. *Findings of EMNLP*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *ACL*.

Chen Cecilia Liu, Jonas Pfeiffer, Ivan Vulić, and Iryna Gurevych. 2023. Improving generalization of adapter-based cross-lingual transfer with scheduled unfreezing. *NAACL*.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019. Multi-task deep neural networks for natural language understanding. In *ACL*.

Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*, 56(2):1–40.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng-Xin Yong, Hailey Schoelkopf, Xiangru Tang, Dragomir Radev, Alham Fikri Aji, Khalid Almubarak, Samuel Albanie, Zaid Alyafeai, Albert Webson, Edward Raff, and Colin Raffel. 2022. Crosslingual generalization through multitask finetuning. *ACL*.

Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. 2018. A PAC-Bayesian approach to spectrally-normalized margin bounds for neural networks. *ICLR*.

Sung Min Park, Kristian Georgiev, Andrew Ilyas, Guillaume Leclerc, and Aleksander Madry. 2023. TRAK: Attributing model behavior at scale. *ICML*.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020. Adapterfusion: Non-destructive task composition for transfer learning. *EACL*.

Pengjie Ren, Chengshun Shi, Shiguang Wu, Mengqi Zhang, Zhaochun Ren, Maarten de Rijke, Zhumin Chen, and Jiahuan Pei. 2024. MELoRA: Mini-ensemble low-rank adapters for parameter-efficient fine-tuning. *ACL*.

Shagun Sodhani, Amy Zhang, and Joelle Pineau. 2021. Multi-task reinforcement learning with context-based representations. In *ICML*.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2019a. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. *NeurIPS*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019b. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *ICLR*.

Yaqing Wang, Subhabrata Mukherjee, Xiaodong Liu, Jing Gao, Ahmed Hassan Awadallah, and Jianfeng Gao. 2022. Adamix: Mixture-of-adapter for parameter-efficient tuning of large language models. *EMNLP*.

Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. 2022. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *ICML*.

Sen Wu, Hongyang R Zhang, and Christopher Ré. 2020. Understanding and improving information transfer in multi-task learning. In *ICLR*.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. 2024. LESS: Selecting influential data for targeted instruction tuning. In *ICML*.

Fan Yang, Hongyang R Zhang, Sen Wu, Christopher Ré, and Weijie J Su. 2020. Precise high-dimensional asymptotics for quantifying heterogeneous transfers. *arXiv preprint arXiv:2010.11750*.

A Omitted Details about Our Approach

Notations. We provide a list of mathematical notations for reference:

- S: A subset of $\{1, 2, \ldots, n\}$.

- $f_i(S)$: The performance of a model fine-tuned on tasks in S, evaluated on task i.

- $\theta^\star$: The vectorized model weights of the pre-trained initialization.

- $\hat{\theta}_S$: The vectorized model weights fine-tuned on a subset of tasks S.

- $h_{\theta^\star}(s, y)$: Model output given an input pair s, y.

- $\nabla h_X(s, y)$: Vectorized gradients of model output with respect to model weights X.

- $T_{i,j}$: The average performance of $f_i(S)$ over multiple subsets S that include task i, for every $i = 1, 2, \ldots, n$.

- j_i: The index of the group chosen at step i of the boosting procedure.

- G_{j_i}: The group of tasks chosen at step i of the boosting procedure.

- t_{j_i}: The number of adapters with in the group G_{j_i} fit in the gradient boosting procedure.

A.1 Clustering algorithms

We now describe a clustering algorithm to partition the n tasks into k disjoint subsets. Given an n by n task relevance score matrix T, we will find a clustering that maximizes the average density of all clusters. Concretely, let $C_1, \ldots, C_k$ be a disjoint partition of $[n]$. Let $v_1, \ldots, v_k$ be a 0-1 vector indicating whether a task is in one cluster. The average density of this clustering can be written as:

$$\frac{1}{k} \sum_{i=1}^{k} \frac{v_i^\top T v_i}{v_i^\top v_i}.$$

This integral objective is NP-hard to optimize in general.

We design a Semi-Definite Programming (SDP) relaxation and then round the SDP solution to a clustering. Let us denote the assignment variables as an $n \times k$ matrix V, such that each entry $V_{i,j}$ indicates whether a task i belongs to a cluster j, for every $i = 1, \ldots, n$, $j = 1, \ldots, k$. Moreover,

let the jth column of V, which is the characteristic vector of the j-th cluster, be denoted as v_j. Under this assignment, the sum of $V_{i,j}$ across any task i must be one, as we allow one task to be assigned in a single group. By contrast, the sum of $V_{i,j}$ across C_j is the number of tasks assigned to C_j, which is at least one.

Let e denote the all-ones vector. We state an integer program to maximize the average density of all k clusters as follows

$$\max_{V \in \mathbb{R}^{n \times k}} \quad \left\langle T, \frac{1}{k} \sum_{j=1}^{k} \frac{v_j v_j^\top}{v_j^\top v_j} \right\rangle$$

$$Ve = e, \sum_{i=1}^{n} V_{i,j} \geq 1, \text{ for } 1 \leq j \leq k$$

$$V_{i,j} \in \{0, 1\}, \text{ for } 1 \leq i \leq n, 1 \leq j \leq k.$$

Note that $v_i v_i^\top$ is a rank-one semidefinite matrix. Let us denote the sum of them (normalized by $v_i^\top v_i$) as the following new variable

$$X = \sum_{j=1}^{k} \frac{v_j v_j^\top}{v_j^\top v_j}.$$

X has rank k since it is the sum of k rank-1 matrices, and the v_i's are orthogonal to each other. Additionally, its trace is equal to k because the trace of $\frac{v_j v_j^\top}{v_j^\top v_j}$ is one for any j. Second, one can verify that the entries of every row of X sum up to one. Removing the 0-1 integer constraint and regularizing the number of clusters, we derive the problem as

$$\max_{X \in \mathbb{R}^{n \times n}} \quad \langle T, X \rangle + \lambda \operatorname{Tr}[X]$$

$$Xe = e$$

$$X \geq 0, X \succeq 0.$$

where λ is a hyperparameter for regularization. This leads to a convex program, which can be solved efficiently.

Given a solution of the SDP, denoted as $\hat{X}$, the last step is to round $\hat{X}$ into an integer solution. We set a threshold λ such that if $\hat{X}_{u,v} \geq \lambda$, tasks u and v are assigned to the same cluster. In the experiments, we set λ as c/n for a constant $c \geq 1$, since $\hat{X}_{u,v}$ should be $\frac{1}{|C_i|}$ when they are in the same cluster with $|C_i| < n$. Thus, the intra-cluster distance must always be at least λ with the assignment.

A.2 Boosting algorithms

In each step of our boosting procedure, it picks the group with the largest training error, denoted as G_{j_i}. Then, compute the negative gradient w.r.t. current model predictions for each sample in the group $(s, y) \in \mathcal{D}_{G_{j_i}}$:

$$-\frac{\partial \mathcal{L}}{\partial h(s, y)} = 1 - p,$$

where p is the current prediction probability for the correct class label of sample i. Then, we fit an adapter $\theta_{j_i}^{(t+1)}$ to the combined data set of $\mathcal{D}_{G_{j_i}}$, mapping the input sample s to the negative gradients $1 - p$ in regression. Update the current ensemble $h_{j_i}^{(t+1)}(s, y) = h_{j_i}^{(t)}(s, y) + \eta h_{\theta_{j_i}^{(t+1)}}(s, y)$.

After multiple steps, the algorithm returns the final ensemble for one group:

$$h(s, y) = h_{\theta_{j_i}^{(0)}}(s, y) + \eta \sum_{t=1}^{b} h_{\theta_{j_i}^{(t)}}(s, y).$$

We use η as 0.1. Note that we use one group as an example. We apply the boosting procedure to m groups in our approach.

Extension to AdaBoost: In a similar sense, we can apply AdaBoosting to boost adapters. At each iteration, the sample weights are modified, and a new adapter is trained on the reweighted training samples. Denote the error rate of $h^{(t)}$ at iteration t as $err^{(m)}$. Denote $\alpha^{(t)} = \log \frac{1-err^{(t)}}{err^{(t)}}$. The training samples that are misclassified by the classifier $h^{(t-1)}$ have their weights increased exponentially by $err^{(t)}$:

$$w_i^{(t)} \leftarrow w_i^{(t-1)} \exp\left(\alpha^{(t)} \mathbf{1}\big(y_i \neq h_{\theta^{(m)}}(s_i, y_i)\big)\right)$$

In our setup, weights can be assigned to separate tasks instead of single tasks. We use parameter-efficient fine-tuning to fit each model, such as LoRA and Adapters. At each iteration, a new model is fit on a reweighted version of the training dataset. Similarly, the first-order approximation can be applied to this case by solving a reweighted version of logistic regression on gradients. Therefore, we can also leverage Algorithm 1 to fit the adapter weights in each new model quickly.

A.3 Computation and memory costs

The computation cost of our approach involves:

- $O(n)$ gradient evaluation of all the tasks and solving logistic regression on M random subsets.

- Fine-tuning m adapters, each on a cluster of tasks. The combination of m task groups is n tasks.

- Applying b iterations of gradient boosting on the adapters, which computes $O(n)$ gradients of the existing model's outputs.

Thus, Algorithm 2 involves $O(n)$ computation in the number of forward and backward passes. This is comparable to training a single multitask model on all tasks. We report the exact computation cost of our approach in experiments.

As for the memory, Algorithm 2 uses the memory for maintaining $m + b$ adapter models and one base model.

B Omitted Experiments

B.1 List of models and datasets

We experiment with the following models: Llama-3.2-1B, Llama-3.2-3B, GPT-J-6B, Llama-3.1-8B, Llama-2-13B, and CodeLlama-34B-Instruct.

The datasets used in our experiments include SuperGLUE, HellaSwag, Story Cloze, WinoGrande and the Shakespeare dataset. We refer readers to their respective web pages for a detailed description of the dataset statistics. Due to computational constraints, we use 50% of the training set for training and sample 10% of it for validation. We report the test accuracy on the development set provided by the datasets.

B.2 Results for first-order approximation

We report the first-order approximation error for the outputs of fine-tuned language models by varying the fine-tuning distance from 0.05% to 0.25% across six pretrained language models in Table 5.

We report the correlation score between the estimated and true fine-tuning performances. We observe that our approach yields a correlation score up to 0.6, evaluated on the results of 50 randomly sampled subsets of size 3, across three Llama models of size up to 8 billion. The results are evaluated across d between $200, 400$, and 800.

B.3 Results on adapter tuning

Fine-tuning Llama-8B with QAdapter. In this section, we report additional evaluation results by

Table 5: We evaluate the first-order approximation errors within the fine-tuning region close to the model pre-trained initialization. We measure the fine-tuned distance as $\frac{\|X-\theta^\star\|_F}{\|\theta^\star_{\text{Full}}\|_F}$. We evaluate the RSS error between $h_X(s,y)$ and the first-order approximation using the pretrained model weights $\theta^\star$. To ensure the statistical significance of these results, we report the average over 50 random task subsets.

Distance/RSS	Llama-3.2-1B	Llama-3.2-3B	GPT-J-6B	Llama-3.1-8B	Llama-2-13B
	Full-precision pretrained model with LoRA fine-tuning				
0.05%	$9.1_{\pm3.1} \times 10^{-4}$	$6.7_{\pm3.3} \times 10^{-4}$	$4.5_{\pm0.8} \times 10^{-5}$	$3_{\pm0.3} \times 10^{-5}$	$2.5_{\pm1.1} \times 10^{-3}$
0.10%	$1.9_{\pm0.4} \times 10^{-3}$	$9.1_{\pm3.9} \times 10^{-4}$	$4.9_{\pm0.9} \times 10^{-5}$	$6_{\pm0.9} \times 10^{-5}$	$8.2_{\pm0.4} \times 10^{-3}$
0.15%	$3.4_{\pm1.2} \times 10^{-3}$	$1.6_{\pm0.9} \times 10^{-3}$	$5.2_{\pm1.3} \times 10^{-5}$	$3_{\pm0.6} \times 10^{-4}$	$3.1_{\pm0.0} \times 10^{-3}$
0.20%	$5.0_{\pm1.1} \times 10^{-3}$	$2.3_{\pm0.8} \times 10^{-3}$	$1.3_{\pm1.0} \times 10^{-4}$	$4_{\pm0.4} \times 10^{-4}$	$3.3_{\pm2.2} \times 10^{-3}$
0.25%	$5.4_{\pm1.0} \times 10^{-3}$	$5.2_{\pm2.0} \times 10^{-3}$	$1.4_{\pm0.6} \times 10^{-4}$	$5_{\pm0.4} \times 10^{-4}$	$2.3_{\pm1.2} \times 10^{-2}$
	4-bit quantized pretrained model with Quantized LoRA (QLoRA) fine-tuning				
0.05%	$1.7_{\pm0.5} \times 10^{-4}$	$2.0_{\pm0.9} \times 10^{-4}$	$1.0_{\pm0.4} \times 10^{-4}$	$1.2_{\pm1.1} \times 10^{-3}$	$2.9_{\pm0.3} \times 10^{-3}$
0.10%	$5.6_{\pm2.8} \times 10^{-4}$	$2.1_{\pm1.0} \times 10^{-4}$	$1.9_{\pm0.2} \times 10^{-4}$	$1.4_{\pm0.4} \times 10^{-3}$	$3.3_{\pm0.7} \times 10^{-3}$
0.15%	$1.0_{\pm0.5} \times 10^{-3}$	$6.2_{\pm1.9} \times 10^{-4}$	$2.5_{\pm2.3} \times 10^{-4}$	$5.3_{\pm4.2} \times 10^{-3}$	$4.6_{\pm0.4} \times 10^{-3}$
0.20%	$2.0_{\pm0.5} \times 10^{-3}$	$1.1_{\pm0.5} \times 10^{-3}$	$2.8_{\pm0.6} \times 10^{-4}$	$3.7_{\pm2.4} \times 10^{-3}$	$4.9_{\pm1.0} \times 10^{-3}$
0.25%	$2.2_{\pm0.9} \times 10^{-3}$	$1.7_{\pm0.7} \times 10^{-3}$	$4.3_{\pm2.5} \times 10^{-4}$	$1.3_{\pm0.9} \times 10^{-2}$	$6.8_{\pm0.7} \times 10^{-3}$
Distance/RSS	**Llama-3.2-1B**	**Llama-3.2-3B**	**GPT-J-6B**	**Llama-3-8B**	**Llama-2-13B**
	Full-precision pretrained model with adapter tuning				
0.05%	$8.9_{\pm3.0} \times 10^{-3}$	$1.4_{\pm0.1} \times 10^{-2}$	$3.5_{\pm0.8} \times 10^{-3}$	$1.3_{\pm0.2} \times 10^{-6}$	$7.7_{\pm1.1} \times 10^{-7}$
0.10%	$1.5_{\pm1.0} \times 10^{-2}$	$1.9_{\pm0.8} \times 10^{-2}$	$2.8_{\pm1.3} \times 10^{-3}$	$1.4_{\pm0.1} \times 10^{-6}$	$1.2_{\pm0.8} \times 10^{-6}$
0.15%	$2.1_{\pm1.1} \times 10^{-2}$	$1.8_{\pm1.0} \times 10^{-2}$	$3.4_{\pm0.8} \times 10^{-3}$	$1.9_{\pm0.4} \times 10^{-6}$	$1.4_{\pm0.1} \times 10^{-6}$
0.20%	$2.1_{\pm0.5} \times 10^{-2}$	$2.0_{\pm0.7} \times 10^{-2}$	$4.1_{\pm0.7} \times 10^{-3}$	$2.2_{\pm1.6} \times 10^{-6}$	$1.5_{\pm0.8} \times 10^{-6}$
0.25%	$7.0_{\pm1.1} \times 10^{-2}$	$7.5_{\pm6.4} \times 10^{-2}$	$6.9_{\pm6.7} \times 10^{-3}$	$2.9_{\pm1.9} \times 10^{-6}$	$1.5_{\pm0.7} \times 10^{-6}$
	4-bit quantized pretrained model with quantized adapter (QAdapter) tuning				
0.05%	$7.0_{\pm1.7} \times 10^{-3}$	$1.5_{\pm0.2} \times 10^{-2}$	$6.7_{\pm0.0} \times 10^{-7}$	$3.0_{\pm1.6} \times 10^{-6}$	$1.8_{\pm0.3} \times 10^{-6}$
0.10%	$1.1_{\pm0.9} \times 10^{-2}$	$1.7_{\pm0.8} \times 10^{-2}$	$7.3_{\pm0.8} \times 10^{-7}$	$3.1_{\pm0.8} \times 10^{-6}$	$2.1_{\pm1.4} \times 10^{-6}$
0.15%	$1.1_{\pm0.9} \times 10^{-2}$	$1.8_{\pm0.6} \times 10^{-2}$	$8.0_{\pm0.5} \times 10^{-7}$	$5.4_{\pm3.4} \times 10^{-6}$	$2.2_{\pm1.8} \times 10^{-6}$
0.20%	$1.4_{\pm0.5} \times 10^{-2}$	$1.9_{\pm0.5} \times 10^{-2}$	$8.8_{\pm1.1} \times 10^{-7}$	$6.1_{\pm6.4} \times 10^{-6}$	$1.8_{\pm0.4} \times 10^{-6}$
0.25%	$1.5_{\pm0.6} \times 10^{-2}$	$4.0_{\pm1.9} \times 10^{-2}$	$1.1_{\pm0.2} \times 10^{-6}$	$9.1_{\pm9.6} \times 10^{-6}$	$2.5_{\pm0.6} \times 10^{-6}$

using QAdapter as the base fine-tuning protocol in our ensemble method. Figure 5 presents the results of using QAdapter. Our approach improves QAdapter accuracy by **9%**, incurring 8% more computation and increasing memory from 9GB to 19GB. Compared to MTL-FT and task grouping, it achieves similar accuracy while reducing computation by 46% and memory from 43GB to 19GB. Against full fine-tuning, it lowers computation by 96% and memory by 74%, while improving test accuracy by **1%**.

Next, we report the full comparison in fine-tuning LLama-3-8B on ten NLP tasks using QAdapter and fine-tuning CodeLlama-34B-Instruct using QLoRA in Table 6.

For the hyperparameters, in all our experiments, we fine-tune models with AdamW with a learning rate $2e^{-5}$ for ten epochs. For LoRA, we set the LoRA rank to 4. For Adapters, we set the reduction ratio to 256. These are determined by a hyperparameter search via cross-validation.

B.4 Clustering based on gradient similarities

There has been a line of work on using gradient similarity to measure task similarity. In particular, Park et al. (2023) proposed a scalable data attribution method to quantify the influence of adding or removing one sample on the prediction of the other samples. Their approach also uses JL to reduce the dimension of the gradient features before solving each logistic regression. Xia et al. (2024) designed a method to select a small subset of instruction data for targeted instruction tuning on a target task. Their method computes gradient-based influence scores by comparing low-dimensional projected gradients from multiple model checkpoints in a warmup training phase. Then, they select data based on cosine similarity to target task gradients.

It should be worth emphasizing that our approach adds very little overhead relative to these gradient similarity-based methods, since we only need to compute the gradients at the base model

Table 6: We report the test accuracy (%) of our method, as compared with baselines. We also compute the average test accuracy across ten NLP tasks, along with the number of FLOPs and memory usage. We report the results of fine-tuning CodeLlama-34B-Instruct using QLoRA. We run each experiment with three random seeds and report the standard deviations.

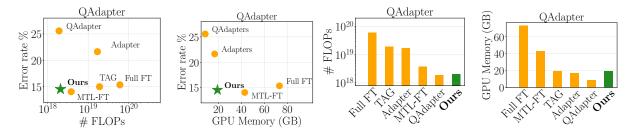|	BoolQ	CB	COPA	H-SWAG	MultiRC	RTE	Story Cloze	WiC	Winogrande	WSC	Average Accuracy	Number of FLOPs	GPU Memory
# Train	4,242	225	360	17,957	12,259	1,120	841	2,442	4,161	498			
# Valid	471	25	40	1,995	1,362	200	188	270	462	56			
# Test	3,270	56	100	10,042	4,848	277	1871	638	1267	104			
# classes	2	3	2	4	2	2	2	2	2	2			
QLoRA	$91.9_{\pm0.7}$	$96.2_{\pm0.6}$	$90.0_{\pm1.0}$	$90.7_{\pm0.9}$	$89.5_{\pm0.3}$	$84.6_{\pm0.4}$	$96.9_{\pm0.6}$	$74.2_{\pm0.5}$	$76.6_{\pm2.4}$	$75.5_{\pm1.2}$	$86.9_{\pm1.6}$	1.8×10^{19}	18.6GB
MTL-FT	$92.1_{\pm0.4}$	$100_{\pm0.0}$	$95.0_{\pm0.5}$	$96.1_{\pm0.9}$	$91.7_{\pm0.4}$	$90.7_{\pm0.4}$	$98.2_{\pm0.9}$	$78.9_{\pm0.4}$	$85.1_{\pm1.1}$	$82.7_{\pm1.5}$	$90.2_{\pm0.8}$	3.7×10^{19}	65.8GB
Ours	$92.1_{\pm0.4}$	$100_{\pm0.0}$	$94.0_{\pm1.5}$	$94.3_{\pm0.4}$	$90.7_{\pm0.4}$	$88.0_{\pm0.7}$	$98.1_{\pm0.8}$	$74.1_{\pm0.6}$	$84.3_{\pm1.5}$	$76.9_{\pm1.4}$	$89.9_{\pm0.9}$	1.9×10^{19}	47.8GB

Figure 5: This figure compares the error rate (one average minus test accuracy), computation cost, and GPU memory across our approach and baselines, for fine-tuning Llama-3-8B on ten NLP tasks with QAdapter.

once. It is an interesting question to further explore the use of gradient similarity as a measure. In our preliminary study, we find that by clustering the tasks with their gradients, the (average) test accuracy remains 0.8% lower than our method. The results are reported in Table 7.

In terms of how our work complements prior works, first, we have measured the linearization accuracy of parameter-efficient fine-tuning methods, including LoRA, Adapter tuning, quantized LoRA (QLoRA), and QAdapter, on a wide range of LLMs. Our empirical finding reinforces the belief that, locally, the output of large models can be accurately linearized, as these parameter-efficient fine-tuning methods all stay very close to the base pretrained model. Thus, the approximation error is less than 3% relative to model outputs. This has not been observed in prior works, including the work of Park et al. (2023). Second, we designed an ensemble method to use several low-rank adapters for multiple datasets as opposed to a single (high-rank) adapter for all datasets. This is complementary to the work of Xia et al. (2024). Third, we conducted extensive evaluations of our ensemble method in various multitask settings. This is again complementary to both of these prior works.

B.5 Sharpness measurements

We observe similar evaluation of generalization errors and sharpness measures on the BoolQ dataset.

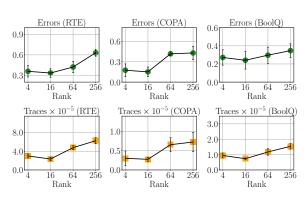We also notice consistent results of using QAdapter, which are reported in Figure 6.

Figure 6: We show the empirical generalization errors (termed Error in the figures) and estimated Hessian traces of the loss for fine-tuned adapters (using QAdapter on Llama-3-8B) across various hidden dimensions. Smaller adapters achieve the lowest generalization errors. Additionally, the Hessian trace values correlate with generalization errors, suggesting that smaller adapters tend to converge to flatter minima.

B.6 Measuring transfer effects

Consider fine-tuning a foundation model to predict n individual tasks. Given a fine-tuning protocol such as QLoRA, denoted by f, and a subset of tasks $S \subseteq \{1, 2, \ldots, n\}$, let $f_i(S)$ represent the loss value of the fine-tuned model evaluated on task i, for any $i \in S$. We define the positive transfer rate of f as follows. First, we say that tasks

Table 7: We compare our method, which uses gradients to estimate fine-tuning performances, with using gradient similarity for clustering tasks. We report the test accuracies in fine-tuning Llama-3-8B using QLoRA. We run each experiment with three random seeds and report the standard deviations.

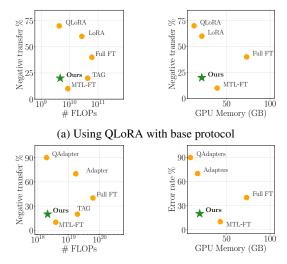|	BoolQ	CB	COPA	H-SWAG	MultiRC	RTE	Story Cloze	WiC	Winogrande	WSC
Clustering with gradient similarities	$88.0_{\pm0.7}$	$100_{\pm0.0}$	$92.0_{\pm1.0}$	$92.5_{\pm0.6}$	$89.1_{\pm0.3}$	$83.0_{\pm0.2}$	$96.5_{\pm0.9}$	$70.3_{\pm0.7}$	$81.5_{\pm0.9}$	$79.6_{\pm0.3}$
Clustering by task affinity scores (Ours)	$89.9_{\pm0.9}$	$100_{\pm0.0}$	$94.0_{\pm1.0}$	$93.5_{\pm0.3}$	$89.1_{\pm0.3}$	$85.7_{\pm0.3}$	$97.1_{\pm0.1}$	$69.9_{\pm0.9}$	$82.0_{\pm0.5}$	$79.8_{\pm2.2}$

S provide a positive transfer to task i if $f_i(S)$ is lower than the single-task loss value of $f_i\{i\}$. Otherwise, we say that the transfer is negative. Then, imagine a scenario where we examine a list of task subsets $S_1, \ldots, S_k$ (for instance, in the simplest case, m could be 1 and S_1 could include all the tasks as $S_1 = \{1, 2, \ldots, n\}$). With this notion of positive/negative transfer, we evaluate the positive transfer rate averaged over the k subsets as:

$$\frac{1}{k} \sum_{j=1}^{k} \frac{\left| \{i \in S_j : f_i(S_j) < f_i(\{i\})\} \right|}{|S_j|}.$$

With this definition in hand, next, we will examine the transfer rate of various fine-tuning methods, along with their memory usage. In a nutshell, we observe that there is an intricate trade-off between transfer and memory, as we will show below. Our overall goal is to design a boosting system that optimizes positive transfer on a base fine-tuning protocol with minimal computation and memory overhead.

Computational cost vs. transfer. We now report the memory usage of various fine-tuning algorithms. We evaluate the memory usage of an ensemble of adapters, which involves using multiple adapters and combining their outputs through weighted averaging. Our findings using Llama-3-8B as the base model (the results obtained with other base models are qualitatively similar) are the following. As expected, with full fine-tuning, memory size scales as 21.5 GB (size of Llama-8B) times m. With PEFT, this reduces to $21.5 + 2.8(m-1)$ GB for LoRA and $21.5 + 6.5(m-1)$ for adapter tuning. Applying 4-bit quantization to the base model further reduces the memory by 53% using LoRA and 56% using adapter tuning.

Next, we report the transfer rates from the above fine-tuning procedures. Recall that this depends on the specification of the subsets. We first consider $k = 1$ and $S_1 = \{1, 2, \ldots, n\}$. Interestingly, we note that the transfer rate decreases with (quantized) PEFT compared to full fine-tuning. For full fine-tuning, the positive transfer rate is

(a) Using QLoRA with base protocol

(b) Using QAdapter as base protocol

Figure 7: We illustrate the trade-off between transfer, computation cost, and GPU memory between our approach and baselines. ENSEMBLELORA achieves the best trade-off between task transfers and computation/memory costs. This experiment uses Llama-3-8B as the base model.

$6/10 = 60\%$. With LoRA/adapter, this decreases to 40%. For QLoRA and QAdapter, this further decreases to 30% and 10%.

Next, we consider $k = 45$ and let the list of subsets be $\{1, 2\}, \{1, 3\}, \ldots, \{n-1, n\}$, which includes all pairwise combinations. We find that for full fine-tuning, the positive transfer rate is 56%. For LoRA/adapter, this decreases to 38% and 48%, and 26% and 22% with quantization. In addition, we consider f by averaging the weights of two adapters, each trained on a single task. This further reduces positive transfer across tasks.

Our approach also achieves the best trade-off between the positive transfer rate and computation/memory cost, illustrated in Figure 7. Compared to QLoRA and QAdapter, our approach consistently boosts the positive transfer rates from 30% to 80%. Our approach achieves a comparable positive transfer rate as MTL-FT (90%) and also improves over the positive transfer rate of full model fine-tuning by 60%.

Next, we report the cosine similarity score be-

tween adapter weights to explain the decrease in transfer rates. We use the base model Llama-3-1B and evaluate fine-tuned QLoRA and QAdapter weights. Given two weight matrices from two models, we compute the score between their rank-r decompositions. For QLoRA, we apply the decomposition to the weight matrix, $\Delta W = BA$, with the same rank as A, and average the scores over layers. We measure the similarity between models fine-tuned on random subsets S and the single-task fine-tuned models, varying the size of S from 2 to 10. We also compute the scores using weight averaging as f. As illustrated in Figure 8, for both fine-tuning and weight averaging, the similarity score becomes lower as the size of S increases.
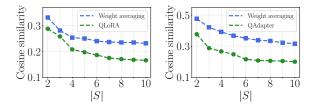

Figure 8: We measure the average similarity between models fine-tuned on random subsets S and single-task models. We observe that the similarity score becomes lower as the size of S increases.