LoRA Learns Less and Forgets Less

Anonymous authors
Paper under double-blind review

Abstract

Low-Rank Adaptation (LoRA) is a widely-used parameter-efficient finetuning method for large language models. LoRA saves memory by training only low rank perturbations to selected weight matrices. In this work, we compare the performance of LoRA and full finetuning on two target domains, programming and mathematics. We consider both the instruction finetuning (≈100K prompt-response pairs) and continued pretraining (≈10B unstructured tokens) data regimes. Our results show that, in most settings, LoRA substantially underperforms full finetuning. Nevertheless, LoRA exhibits a desirable form of regularization: it better maintains the base model's performance on tasks outside the target domain. We show that LoRA provides stronger regularization compared to common techniques such as weight decay and dropout; it also helps maintain more diverse generations. We show that full finetuning learns perturbations with a rank that is 10-100X greater than typical LoRA configurations, possibly explaining some of the reported gaps. We conclude by proposing best practices for finetuning with LoRA.

1 Introduction

Finetuning large language models (LLMs) with billions of weights requires a non-trivial amount of GPU memory. Parameter-efficient finetuning methods reduce the memory footprint during training by freezing a pretrained LLM and only training a small number of additional parameters, often called adapters. Low-Rank Adaptation (LoRA; Hu et al. (2021)) trains adapters that are low-rank perturbations to selected weight matrices.

Since its introduction, LoRA has been touted as a strict efficiency improvement that does not compromise accuracy on the new, target domain (Hu et al., 2021; Dettmers et al., 2024; Raschka, 2023; Zhao et al., 2024b). However, only a handful of studies benchmark LoRA against full finetuning for LLMs with billions of parameters, (Ivison et al., 2023; Zhuo et al., 2024; Dettmers et al., 2024), reporting mixed results. Some of these studies rely on older models (e.g. RoBERTa), or coarse evaluation benchmarks (such as GLUE or ROUGE) that are less relevant for contemporary LLMs. In contrast, more sensitive domain-specific evaluations, e.g., code, reveal cases where LoRA is inferior to full finetuning (Ivison et al., 2023; Zhuo et al., 2024). **Here we ask: under which conditions does LoRA approximate full finetuning accuracy on challenging target domains, such as code and math?**

By training fewer parameters, LoRA is assumed to provide a form of regularization that constrains the finetuned model's behavior to remain close to that of the base model (Sun et al., 2023; Du et al., 2024). **We also ask: Does LoRA act as a regularizer that mitigates "forgetting" of the source domain?**

In this study, we rigorously compare LoRA and full finetuning for Llama-2 7B (and in some cases, 13B) models across two challenging target domains, code and mathematics. Within each domain, we explore two training regimes. The first is *instruction finetuning*, the common scenario for LoRA involving question-answer datasets with tens to hundreds of millions of tokens. Here, we use Magicoder-Evol-Instruct-110K (Wei et al., 2023) and MetaMathQA (Yu et al., 2023). The second regime is *continued pretraining*, a less common application for LoRA which involves training on billions of unlabeled tokens; here we use the StarCoder-Python (Li et al., 2023) and OpenWebMath (Paster et al., 2023) datasets (Table 1).

We evaluate target-domain performance (henceforth, *learning*) via challenging coding and math benchmarks (HumanEval; Chen et al. (2021), and GSM8K; Cobbe et al. (2021)). We evaluate source-domain *forgetting* performance on language understanding, world knowledge, and common-sense reasoning tasks (Zellers et al., 2019; Sakaguchi et al., 2019; Clark et al., 2018).

We find that for code, LoRA substantially underperforms full finetuning, whereas for math, LoRA closes more of the gap (Sec. 4.1), while requiring longer training. Despite this performance gap, we show that LoRA better maintains source-domain performance compared to full finetuning (Sec. 4.2). Furthermore, we characterize the tradeoff between performance on the target versus source domain (learning versus forgetting). For a given model size and dataset, we find that LoRA and full finetuning form a similar learning-forgetting tradeoff curve: LoRA's that learn more generally forget as much as full finetuning, though we find cases (for code) where LoRA can learn comparably but forgets less (Sec. 4.3).

We then show that LoRA – even with a less restrictive rank – provides stronger regularization when compared to classic regularization methods such as dropout (Srivastava et al., 2014), and weight decay (Goodfellow et al., 2016). We also show that LoRA provides regularization at the output level: we analyze the generated solutions to HumanEval problems and find that while full finetuning collapses to a limited set of solutions, LoRA maintains a diversity of solutions more similar to the base model (Sun et al., 2023; Du et al., 2024).

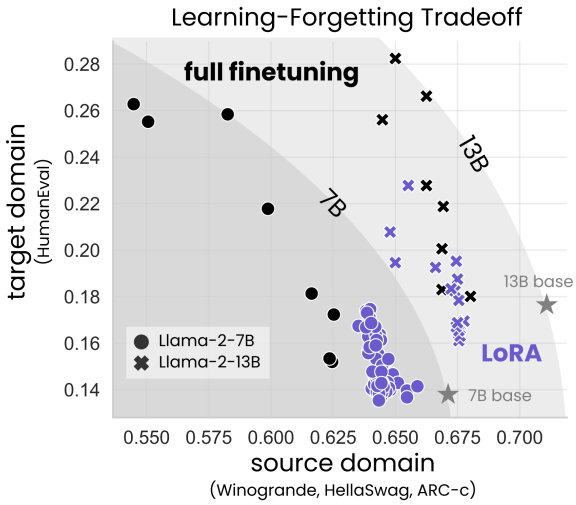


Figure 1: **Learning vs. forgetting tradeoff curves for Llama-2-7B and Llama-2-13B trained on Starcoder-Python**. Gray regions are hypothetical Pareto frontiers for performance on the source domain and the code target domain.

Why does LoRA underperform full finetuning? LoRA was originally motivated in part by the hypothesis that finetuning results in low-rank perturbations to the base model's weight matrix (Li et al., 2018; Aghajanyan et al., 2020; Hu et al., 2021). However, the tasks explored by these works are relatively easy for modern LLMs, and certainly easier than the coding and math domains studied here. Thus, we perform a singular value decomposition to show that full finetuning barely changes the spectrum of the base model's weight matrices, and yet the difference between the two (i.e. the perturbation) is high rank. The rank of the perturbation grows as training progresses, with ranks 10-100$\times$ higher than typical LoRA configurations (Figure 7).

We conclude by proposing best practices for training models with LoRA. We find that LoRA is especially sensitive to learning rates, and that the performance is affected mostly by the choice of target modules and to a smaller extent by rank.

To summarize, we contribute the following results:

- Full finetuning is more accurate and sample-efficient than LoRA in code and math (Sec.4.1).
- LoRA forgets less of the source domain, providing a form of regularization (Sec. 4.2 and 4.3).
- LoRA's regularization is stronger compared to common regularization techniques; it also helps maintaining the diversity of generations (Sec. 4.4).
- Full finetuning finds high rank weight perturbations (Sec. 4.5).
- Compared to full finetuning, LoRA is more sensitive to hyperparameters, namely learning rate, target modules, and rank (in decreasing order; Sec. 4.6) .

Dataset	Domain	# Used Tokens	Experiment
StarCoder-Python (Li et al., 2023)	Code	20B	CPT
OpenWebMath (Paster et al., 2023)	Math	8.6B	CPT
Magicoder-Evol-Instruct-110K (Wei et al., 2023)	Code	72.3M	IFT
MetaMathQA (Yu et al., 2023)	Math	103M	IFT

Table 1: Datasets and token counts for math and code experiments

2 Background

LoRA involves freezing a pretrained weight matrix $W_{\text{pretrained}} \in \mathbb{R}^{d\times k}$, and learning only a low-rank perturbation to it, denoted here as Δ, as follows:

$$W_{\text{finetuned}} = W_{\text{pretrained}} + \Delta$$
$$\Delta = AB, \quad A \in \mathbb{R}^{d\times r}, B \in \mathbb{R}^{r\times k}$$
$$A_0 \sim \mathcal{N}(0,1), \quad B_0 = 0.$$

The user chooses which $W_{\text{pretrained}}$ to adapt ("target modules") and the rank $r << d, k$. By doing so, only $d \times r + r \times k$ parameters are trained instead of $d \times k$, which reduces the memory and FLOPS required for computing the gradient. As an example, applying a $r = 16$ LoRA to a 7B weight matrix with $d = k = 4096$ trains $< 1\%$ of the original parameter count. Sec. D.2 lays out the approximate memory savings by LoRA.

LoRA can additionally enable efficient multi-tenant serving by sharing a single base model across users with personalized adapters (Sheng et al., 2023).

Here, we simplify the combinatorial search over target modules, by grouping modules into three classes: the "Attention" matrices- $\{W_q^{(l)}, W_k^{(l)}, W_v^{(l)}, W_o^{(l)}\}_{l=1}^L$, the Multi Layer Perceptron ("MLP") matrices - $\{W_{\text{up}}^{(l)}, W_{\text{down}}^{(l)}\}_{l=1}^L$, and "All" – their union, for L layers.[1] Note that while in the original study LoRA was only applied to the attention matrices (Hu et al., 2021), it has become the standard to target additional modules (Raschka, 2023; Dettmers et al., 2024).

3 Experimental Setup

We train on code and math datasets that have been shown to increase downstream performance. We motivate the training datasets and evaluation benchmarks below.

3.1 Datasets for Continued Pretraining (CPT) and Instruction Finetuning (IFT)

Coding CPT - Starcoder-Python This dataset (Li et al., 2023) consists of permissively licensed repositories from GitHub, including Git commits, in 80+ programming languages. We chose the Python subset and sub-sampled it to 20B tokens.

Math CPT - OpenWebMath We trained on a subset of up to 8.59B out of 14.7B tokens. The dataset (Paster et al., 2023) includes mathematical web pages from Common Crawl, correctly formatted to preserve mathematical content such as LaTeX equations.[2] We note that this dataset contains a considerable amount of full English sentences. [3]

[1]The Llama-2 architecture uses the SwiGLU (Shazeer, 2020) architecture for the MLP module of the attention block, which consists of a Gated Linear Unit that uses Swish activation function (Ramachandran et al., 2017)). SwiGLU has a gating matrix W_{gate} in addition to W_{up} and W_{down}, which we did not modify in our LoRA experiments.

[2]`https://huggingface.co/datasets/open-web-math/open-web-math`

[3]Out of a random selection of 100K examples, a regex search shows that 75% of the examples contain LaTex. The data is classified as 99.7% English and "overwhelmingly English" by the langdetect and fasttext tools.

Coding IFT - Magicoder-Evol-Instruct-110k This dataset (Wei et al., 2023) contains 72.97M tokens of programming questions and answers. It reproduces the "Evol-Instruct' dataset of WizardCoder (Luo et al., 2023): an LLM (GPT-4) is iteratively prompted to increase the difficulty of a set of question-answer pairs (from Code Alpaca; Chaudhary (2023)).

Math IFT - MetaMathQA This dataset (Yu et al., 2023) was built by bootstrapping mathematical word problems from the *training* sets of GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) by rewriting the questions with variations using GPT-3.5. This dataset contains 395K question-answer pairs and roughly 103M tokens.[4]

We quantify learning and forgetting via benchmarks reported on the Open LLM Leaderboard[5] for state of the art open-source LLMs such as Llama (Touvron et al., 2023).

3.2 Measuring Learning with Coding and Math Benchmarks (*target domain* evaluation)

Coding - HumanEval This benchmark (Chen et al., 2021)) contains 164 problems that involve generating of a Python program given a docstring and a function signature. A generation is considered correct if it passes all supplied unit tests. We use Code Generation LM Evaluation Harness (Ben Allal et al., 2022), configured to output 50 generations per problem, sampling with softmax temperature=0.2, and calculating "pass@1".

Math - GSM8K This benchmark (Cobbe et al., 2021) includes a collection of 8.5K grade-school math word problems . We evaluate on the test split of GSM8K (1,319 samples) as implemented in LM Evaluation Harness (Gao et al., 2023), with default generation parameters (temperature=0, five few-shot, pass@1).

3.3 Forgetting Metrics (*source domain* evaluation)

HellaSwag This benchmark (Zellers et al., 2019) includes 70K problems, each describing an event with multiple possible continuations. The task is to pick the most plausible continuation, which requires making inferences about nuanced everyday situations.

WinoGrande This benchmark (Sakaguchi et al., 2019) also assesses commonsense reasoning. It includes 44K problems with sentences that require ambiguous pronoun resolution.

ARC-Challenge This benchmark (Clark et al., 2018) consists of 7,787 grade-school level, multiple-choice science questions, testing capabilities in complex reasoning and understanding of scientific concepts.

4 Results

4.1 LoRA underperforms full finetuning in programming and math tasks

We compare LoRA and full finetuning after performing an exhaustive learning rate sweep for each method, which we found to be crucial (Dettmers et al., 2024). We include learning rate sweep results in Figure 8.

We perform a sample-efficiency analysis – i.e., compute the learning metrics as a function of training samples seen – for both LoRA and full finetuning. For IFT, we train separate models for $1, 2, 4, 8, 16$ epochs. For CPT, we manipulate the number of unique tokens ($0.25, 0.5, 1, 2, 4, 8, 16, 20$ billion), using individual learning rate cooldown schedules. We train six LoRA models for each condition (3 target modules ["Attention", "MLP", and "All"] $\times$ 2 ranks $[16, 256]$).

In Fig. 2, we summarize the performance of the six LoRA models with their minimum, average, and maximum performance for each training duration (in purple), and compare them to full finetuning (solid black lines) and the base model (dashed horizontal line); the results are further broken down per LoRA configuration in Fig. S2. We first note that for both domains, IFT leads to more significant improvements compared to CPT,

[4] https://huggingface.co/datasets/meta-math/MetaMathQA
[5] https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

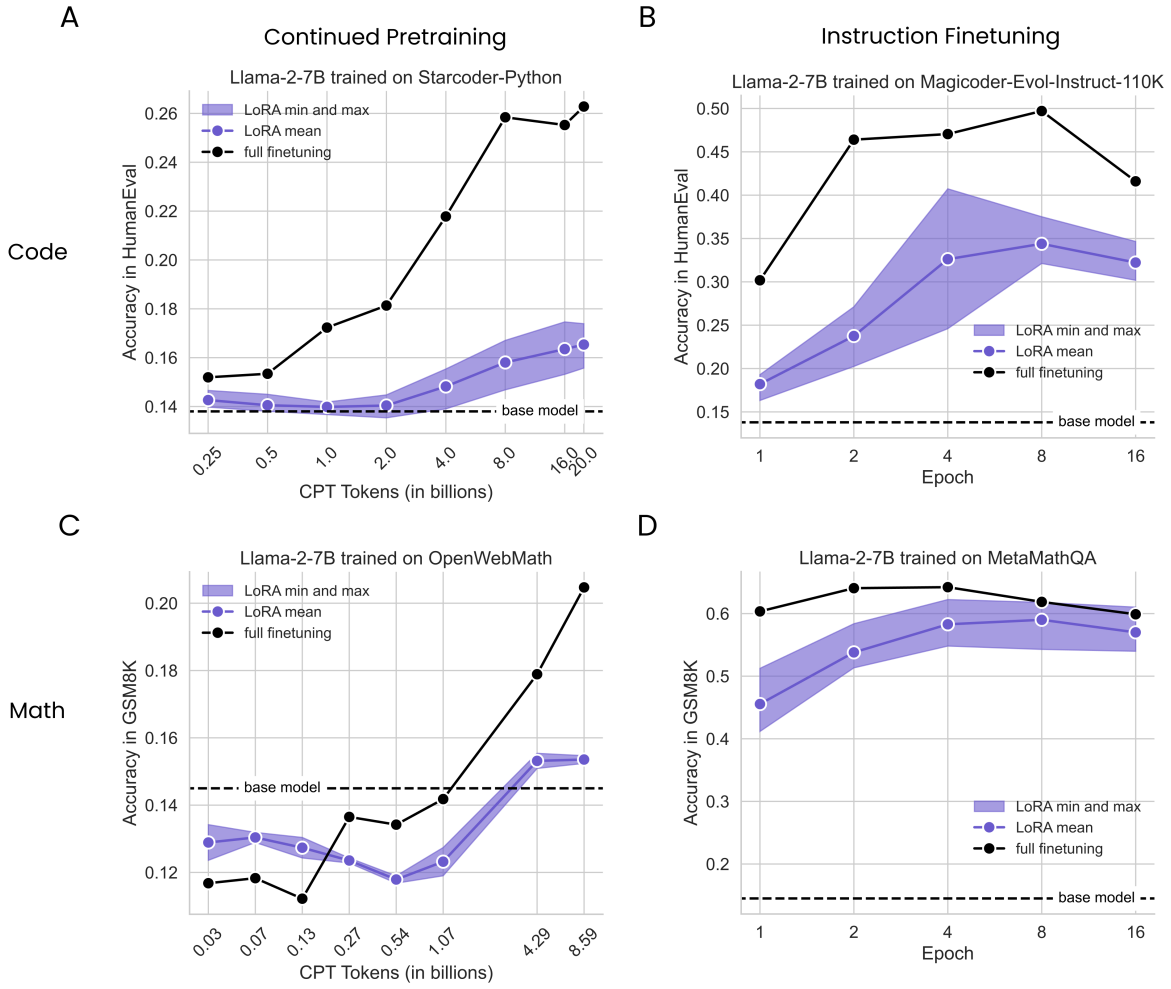


Figure 2: **LoRA underperforms full finetuning in programming and in math.** We use six LoRA configurations per condition and plot their mean, minimum, and maximum values in purple. (A) Starcoder-Python, (B) Magicoder-Evol-Instruct-110K, (C) OpenWebMath, (D) MetaMathQA. Note that 16 epochs are ≈1.16B and ≈1.6B tokens, for Magicoder-Evol-Instruct-110K and MetaMathQA.

which is expected because IFT's problems are more similar to the evaluation problems (e.g., for code, IFT achieves maximum HumanEval of 0.50 VS 0.26 for CPT).

For **Code CPT** (Fig. 2A), we identify a substantial gap between full finetuning and LoRA that grows with more data. The overall best LoRA peaks at 16B tokens (rank=256, "All") with HumanEval=0.175, roughly matching full finetuning with 1B tokens (HumanEval=0.172). Full finetuning reaches its peak HumanEval of 0.263 at 20B tokens. For **Code IFT** (Fig. 2B), the best performing LoRA ($r = 256$, "All") achieves HumanEval=0.407 at epoch 4, meaningfully underperforming full finetuning at epoch 2 (HumanEval=0.464) and at its peak HumanEval score of 0.497 at 8 epochs. **Math CPT** (Fig. 2C) results show that training for 1B tokens or fewer degrades GSM8K results below baseline (GSM8K=0.145). Improvements appear with more data, where the best LoRA (rank=256, "All") achieves GSM8K=0.187 at 8.6 billion tokens, underperforming full finetuning at 4.3 billion tokens (GSM8K=0.191) and at 8.6 billion tokens (GSM8K=0.230). LoRA closes most of the gap with full finetuning in the **Math IFT** (Fig. 2D) dataset. However, LoRA still remains less sample efficient. LoRA ($r = 256$, "All") peaks at 4 epochs (GSM8K=0.622) while full finetuning achieves

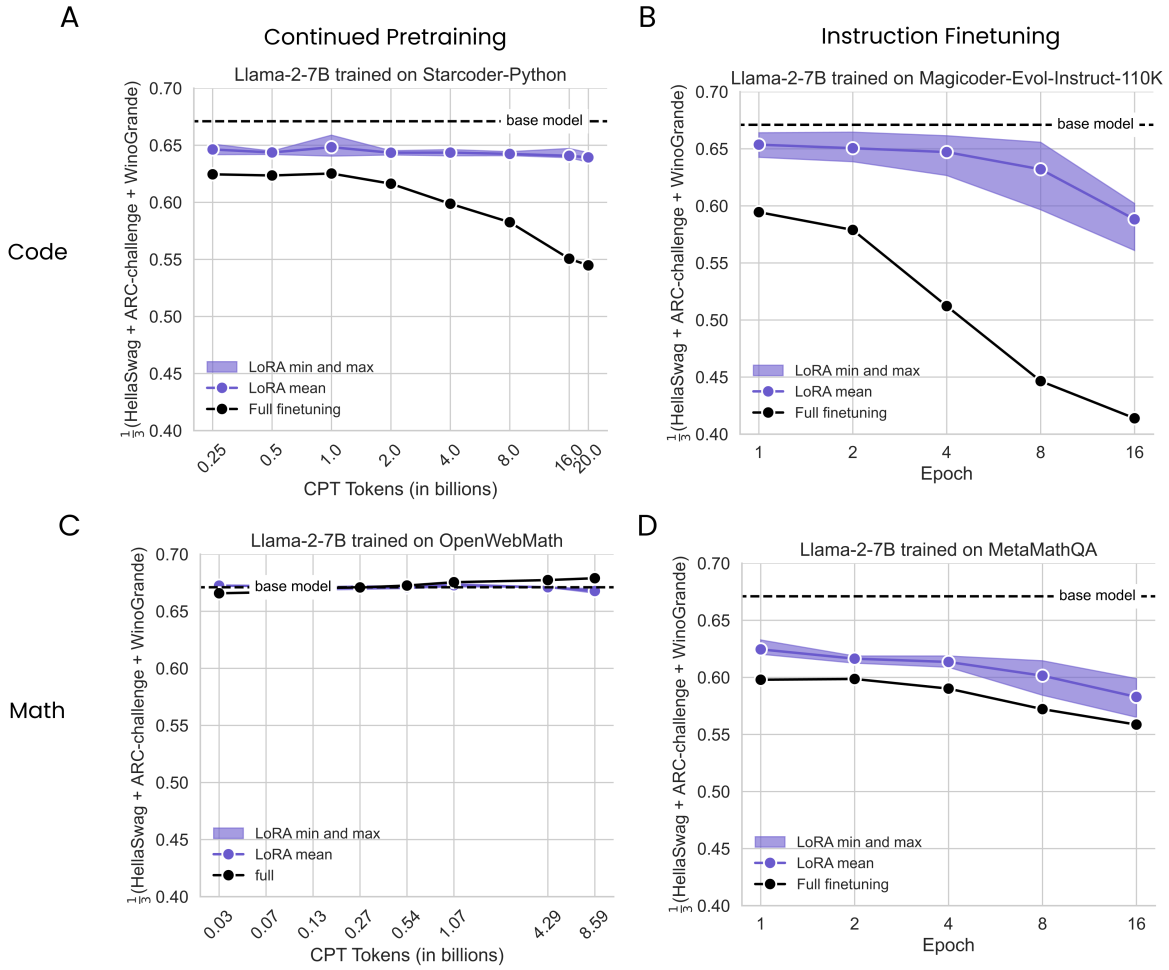


Figure 3: **LoRA forgets less than full finetuning**. We plot the average of HellaSwag, ARC-Challenge and Winogrande for Llama-2-7B trained trained on: (A) StarCoder-Python (B) Magicoder-Evol-Instruct-110k (C) OpenWebMath (D) MetaMathQA.

GSM8K=0.640 at 2 epochs and peaks at 4 epochs, with GSM8K=0.642.[6] Both methods substantially exceed the base model. We hypothesize that the milder gaps here correspond to a smaller domain shift between the math problems and the pretraining data, different from the larger shifts in code.

In summary, across LoRA configurations and training durations, it still appears to underperform full finetuning. These effects are more pronounced for programming than math. For both domains, instruction finetuning leads to larger accuracy gains than continued pretraining.

4.2 LoRA forgets less than full finetuning

We define forgetting as the degradation in the average of HellaSwag, ARC-challenge, and WinoGrande benchmarks, and investigate its extent as a function of data in Fig. 3.

Overall, we observe that (1) IFT induces more forgetting than than CPT, (2) programming induces more forgetting than math, and (3) forgetting tends to increase with data. Most importantly, LoRA forgets less

[6]We note that the original MetaMath paper reports a maximum accuracy of 0.665 when (fully) finetuning Llama-2-7B on the MetaMathQA dataset. We attribute this to small differences in hyperparameters; they trained on 3 epochs with a batch size of 128 using the AdamW optimizer, a learning rate of 2e-5, a learning rate warmup of 3%.

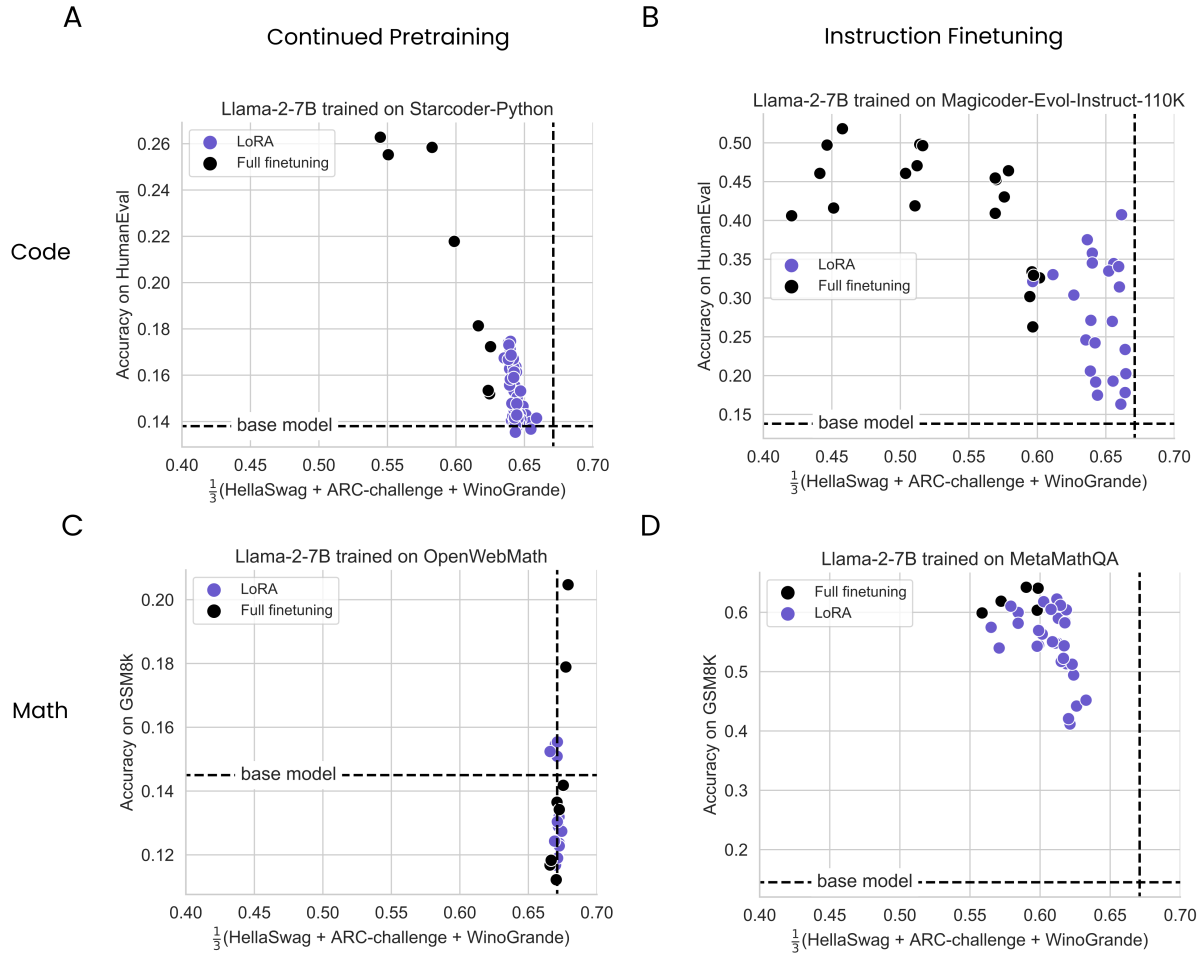


Figure 4: **LoRA vs. Full Finetuning trade-off for LLaMA-2-7B.** Relative to full finetuning, LoRA learns less (lower values on the y-axis) and forgets less (higher values on the x-axis). Dots represent individual models trained for various epochs. For LoRA models, each configuration is shown as a separate dot. In panel B, we scatter four additional full finetuning models with non-zero attention dropout and weight decay, showing epochs 1,2,4 and 8. Same data as Figures 2, 3 and S5.

than full finetuning, and as in 4.1, the effects are more pronounced for the programming domain. In code CPT, LoRA's forgetting curve is roughly constant, whereas full finetuning degrades with more data (the forgetting metric at peak HumanEval: Full finetuning=0.54 at 20B tokens, LoRA=0.64 at 16B tokens). In programming IFT, both methods degrade when trained for more epochs, and at their peak performance (4 and 8 epochs), LoRA scores 0.63 and full finetuning scores 0.45. For math, there are no clear trends on the OpenWebMath CPT dataset, except that both LoRA and full finetuning exhibit no forgetting. This is likely due to the fact that the OpenWebMath dataset is dominated by English sentences, unlike the StarCoder-Python dataset which is majority Python code (see 3.1 for details). In math IFT, LoRA again forgets less than full finetuning (0.63 versus 0.57, repectively, at epoch 4).

4.3 The Learning-Forgetting Tradeoff

It is trivial that models that change less when finetuned to a new target domain, will forget less of the source domain. The nontrivial question is: do LoRA and full finetuning differ in how they tradeoff learning and forgetting? Can LoRA achieve similar target domain performance but with diminished forgetting?

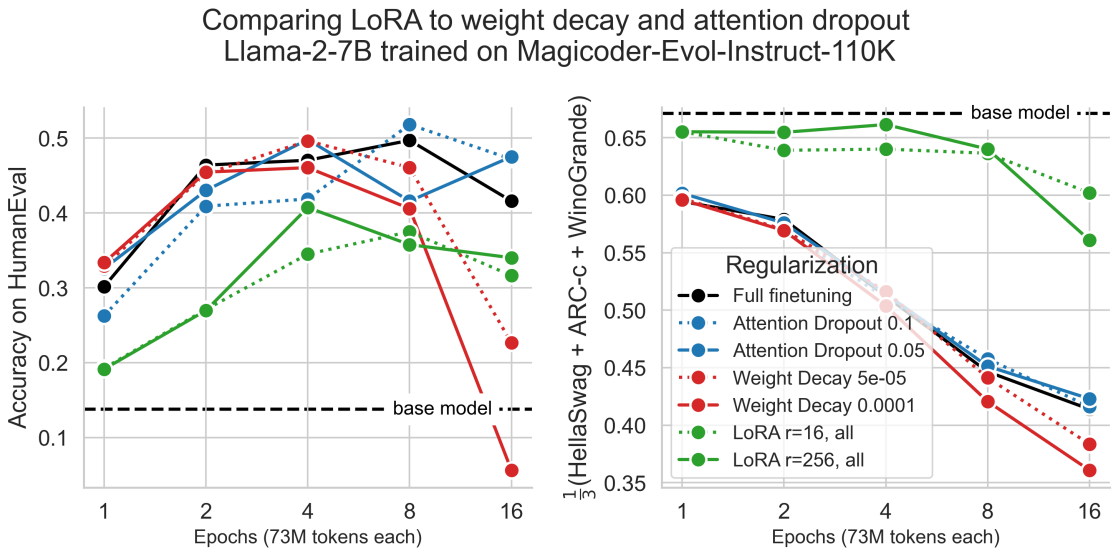


Figure 5: **LoRA provides stronger regularization compared to attention dropout and weight decay.** LoRA finetuning (green) leads to less learning (as measured by accuracy on HumanEval, left) and less forgetting (as measured by HellaSwag, ARC and WinoGrande, right).

We form a learning-forgetting Pareto curve by plotting the aggregate forgetting metric vs. the learning metric (GSM8K and HumanEval), with different models (trained for different durations) scattered as points in this space (Fig. 4). LoRA and full finetuning seem to occupy the same Pareto curve, with the LoRA models on the lower right – learning less and forgetting less. However, we are able find cases, especially for code IFT, where for a comparable level of target-domain performance, LoRA exhibits higher source-domain performance, presenting a better tradeoff. In supplementary Fig. S5 we show the raw evaluation scores for each model. In Fig. S3 we scatter the Llama-2-13B results in the same plot as Llama-2-7B for Code CPT.

4.4 LoRA's regularization properties

Here, we define regularization (loosely) as a training mechanism that keeps the finetuned LLM similar to the base LLM. We first analyze similarity in the learning-forgetting tradeoff and then in the generated text.

LoRA is a stronger regularizer compared to weight decay and attention dropout. We use Llama-2-7B models trained on Magicoder-Evol-Instruct-110K dataset, and compare LoRA ($r = 16, 256$, "All") to weight decay (Goodfellow et al., 2016) with values $5e^{-5}, 1e^{-4}$ and attention dropout (Srivastava et al., 2014) with values $0.05, 0.1$. We find that LoRA provides stronger regularization: it learns less and forgets less.

LoRA helps maintain diversity of token generations. We again use Llama-2-7B models trained on Magicoder-Evol-Instruct-110K dataset to scrutinize the generated strings during HumanEval. We calculate the unique number of output strings out of 50 generations (for base model, full finetuning, and LoRA) serving as a coarse proxy for predictive diversity. In Figure 6 we separately show the results for correct and incorrect answers. As in the reinforcement learning from human feedback literature (Du et al., 2024; Sun et al., 2023), we find that full finetuning results in fewer unique generations ("distribution collapse") compared to the base model, for both pass and fail generations. We find that LoRA provides a compromise between the two, at the level of generations. The above works also suggest that LoRA could even substitute a common regularization term that keeps the probabilities of the generated text similar between the finetuned and base model.

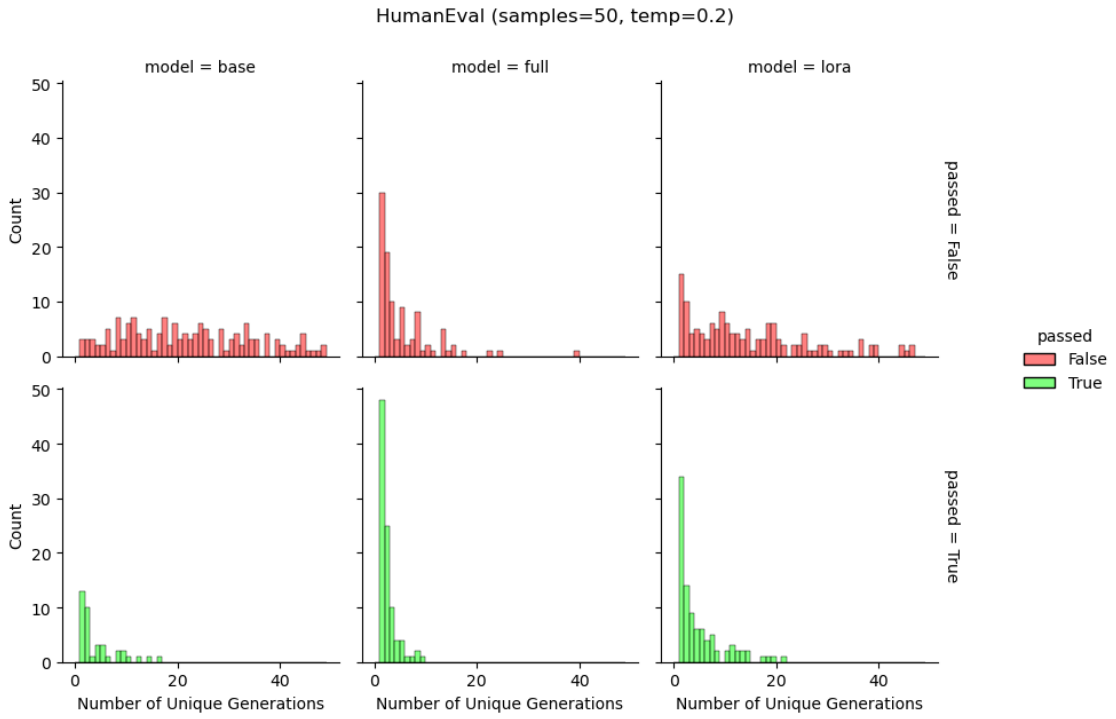


Figure 6: **LoRA maintains output token diversity relative to full finetuning.**

4.5 Full finetuning on code and math does not learn low-rank perturbations

In this section, we seek to study whether we should expect low-rank training to be a good approximation to full finetuning, and if so, what is the necessary rank. Recall that full finetuning can be written as $W_{\text{finetuned}} = W_{\text{pretrained}} + \Delta$; here we compute the Singular Value Decomposition of all three terms in the equation. We focus on continued pretraining for code, where there are drastic differences between LoRA and full finetuning. We analyze checkpoints obtained at 0.25, 0.5, 1, 2, 4, 8, 16, and 20 billion training tokens.

First, in Figure S6 we present results for the W_q projection at layer 26 of Llama-2-7B (with dimensions 4096×4096). We show that the spectrum of the finetuned weight matrix is very similar to that of the base weight matrix, both decaying slowly and requiring keeping $\approx 50\%$ of singular vectors ($\approx 2000/4096$) to explain 90% of the variance in the weight matrix. Critically, the difference Δ also has a similar spectrum to the finetuned and base weight matrices (up to a multiplicative scaling). We suggest that there is nothing extraordinary about the full finetuning spectra; similar spectra can be achieved by adding low-magnitude Gaussian i.i.d noise to a weight matrix (Fig. S7).

Next, we ask when during training does the perturbation become high rank, and whether it meaningfully varies between module types and layers. We estimate the rank needed to explain 90% of the variance in the matrix. The results appear in Figure 7. We find that: (1) The earliest checkpoint at 0.25B CPT tokens exhibits Δ matrices with a rank that is 10-100X larger than typical LoRA ranks; (2) the rank of Δ increases when trained on more data; (3) MLP modules have higher ranks compared to attention modules; (4) first and last layers seem to be lower rank compared to middle layers.

4.6 Practical takeaways for optimally configuring LoRA

Though optimizing LoRA hyperparameters does not close the gaps with full finetuning, some hyperparamater choices are substantially more effective than others, as we highlight below.

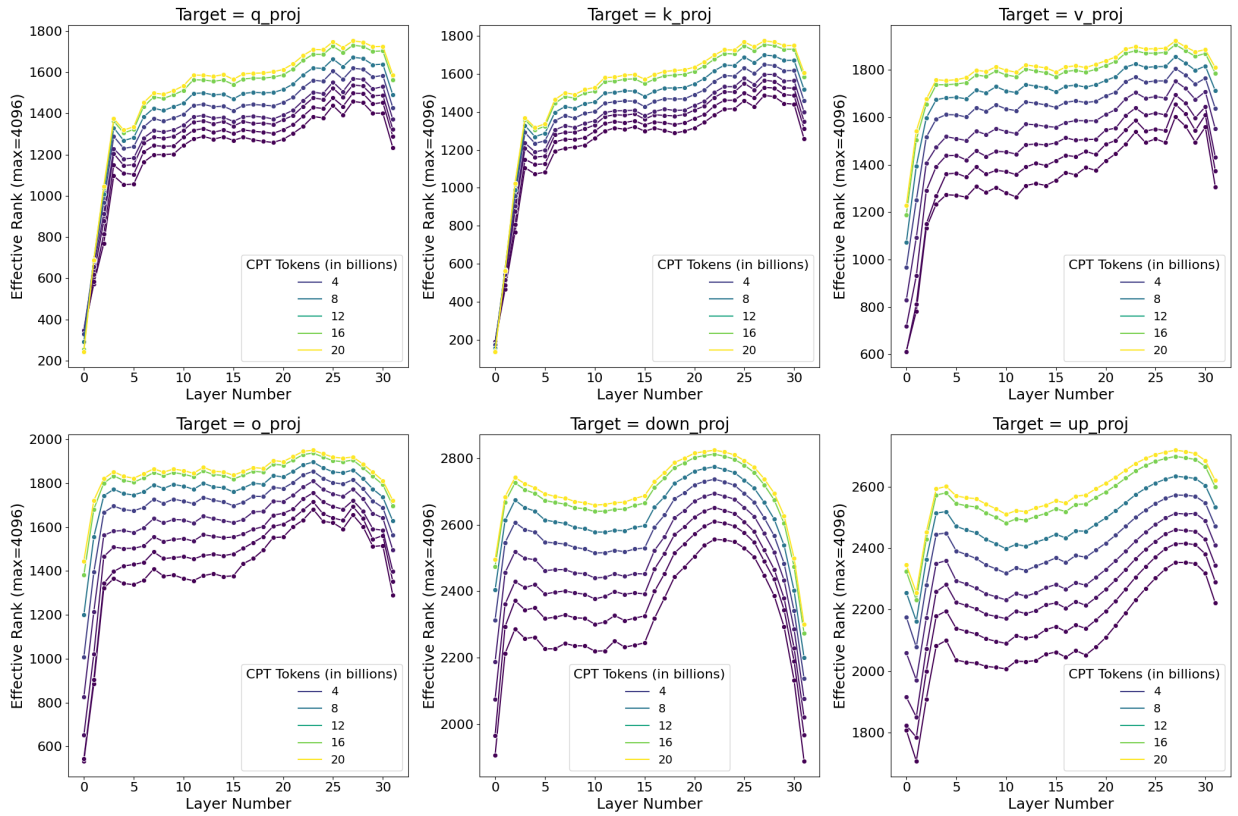


Figure 7: **Dynamics of rank for Llama-2-7b trained on the Starcoder (CPT) data**. In each panel, the x-axis denotes layer number and the y-axis denotes rank needed to explain at least 90% of the variance (maximal dimensionality is 4096). Colors denote CPT tokens, with lighter colors trained for longer.

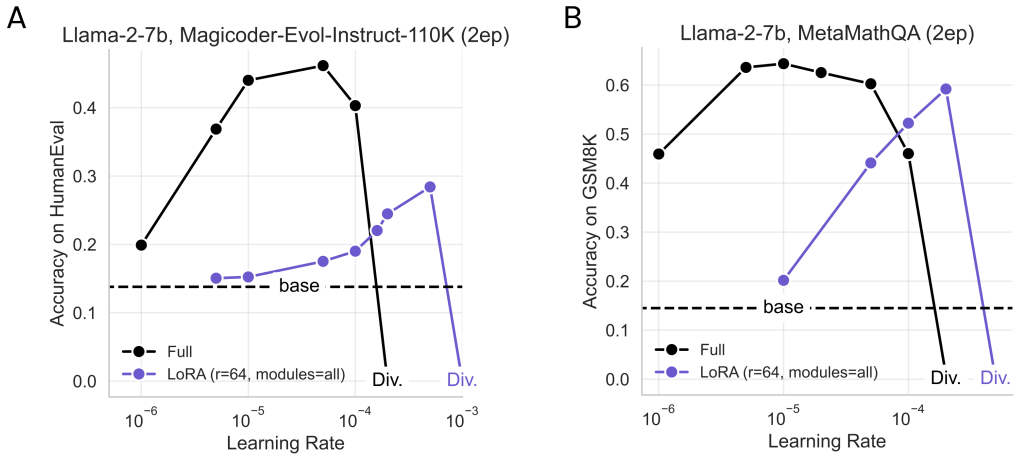
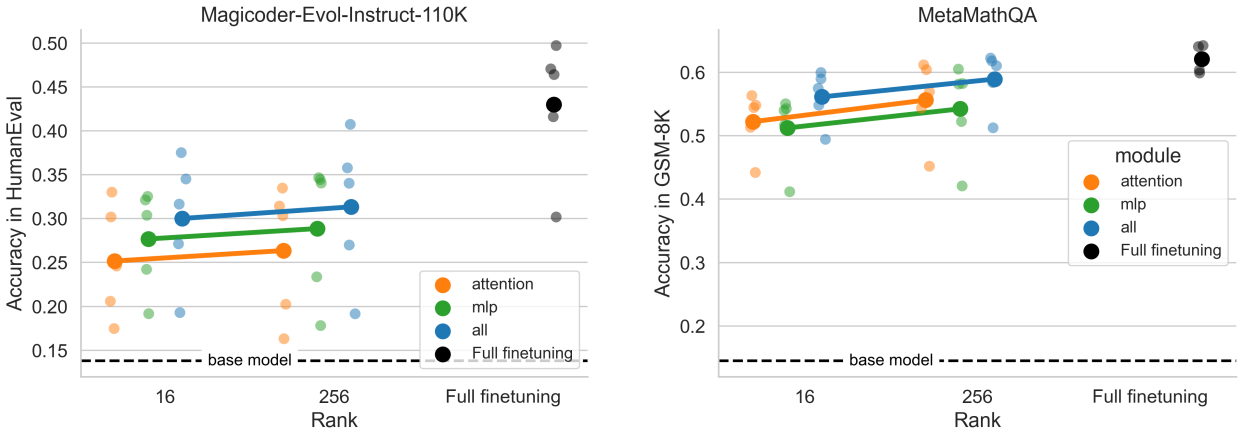


Figure 8: **LoRA is more sensitive to learning rates compared to full finetuning**. LLaMA-2-7B models (A) trained on *Magicoder-Evol-Instruct-110k* (Wei et al., 2023) and evaluated on HumanEval, (B) trained on *MetaMathQA* (Yu et al., 2023) and evaluated on GSM8K. Experiments here are performed with LionW; see Fig. S1 for a comparion to AdamW.



(a) LLaMA-2-7B trained on Magicoder-Evol-Instruct-110K for up to 16 epochs.

(b) LLaMA-2-7B trained on MetaMathQA for up to 16 epochs.

Figure 9: The improvement from targeting more modules (shown as different colors) is more substantial than from increasing the rank by 16× (x-axis). Different points indicate different training durations.

4.6.1 LoRA is highly sensitive to learning rates

We analyze learning rate sensitivity for Llama-2-7B, trained for two epochs on the code and math IFT datasets, and followed by HumanEval and GSM8K evaluation, respectively. For LoRA, we set $r = 64$, and target modules="All". Fig. 8 shows that LoRA improves monotonically with learning rate up to a value at which training diverges, with best learning rates of $5e^{-4}$ for code and $2e^{-4}$ for math. On both datasets, these best LoRA learning rates are underperformed by four alternative full finetuning learning rates. The best full finetuning learning rates are $5e^{-5}$ and $1e^{-5}$, respectively, an order of magnitude smaller than LoRA. For LoRA, we cannot find alternative learning rates that achieve at least 90% of the best learning rate's performance. For full finetuning, there are two viable alternative learning rates for code and three for math.

4.6.2 Choice of target modules matters more than rank

With the best learning rates at hand, in Fig. 9, we proceed to analyze the effect of rank ($r = 16, 256$) and target modules. We find that "All" > "MLP" > "Attention" and that though the effects of rank are more subtle, $r = 256 > r = 16$. We therefore conclude that targeting "All" modules with a relatively low rank (e.g., $r = 16$) provides a good tradeoff between performance and accuracy.

All in all, we recommend using LoRA for IFT and not CPT; identifying the highest learning rate that enables stable training; targeting "All" modules and choosing rank according to memory constraints, with 16 being a good choice; exploring training for at least four epochs.

5 Related Work

Extensions to LoRA LoRA has inspired many variants and extensions that are more memory-efficient or performant, such as QLoRA (Dettmers et al., 2024), VeRA (Kopiczko et al., 2023), DoRA (Liu et al., 2024), Chain of LoRA (Xia et al., 2024), and GaLoRe, (Zhao et al., 2024a), as well as efficient inference techniques building on LoRA, such as S-LoRA (Sheng et al., 2023).

Benchmarking LoRA vs. Full Finetuning The original LoRA study by Hu et al. (2021) reported that LoRA matched full finetuning performance for RoBERTa (Liu et al., 2019) on GLUE (Wang et al., 2018), and GPT-2 on E2E NLG Challenge (Novikova et al., 2017), and GPT-3 on WikiSQL (Zhong et al., 2017), MNLI (Williams et al., 2017), and SAMSum (Gliwa et al., 2019). Many subsequent studies follow this template and report encoder model performance on tasks in GLUE such as SST-2 (Socher et al., 2013) and MNLI

(Williams et al., 2017). Models such as RoBERTa are less than 340M parameters, however, and classification tasks such as MNLI are quite trivial for modern billion-parameter LLMs such as Llama-2-7B. Despite LoRA's popularity, only a few studies have rigorously compared LoRA to full finetuning in this setting and with challenging domains such as code and math. Dettmers et al. (2024) for example found that QLoRA matched full finetuning MMLU (Hendrycks et al., 2020) performance when finetuning Llama-1 7B, 13B, 33B and 65B on the Alpaca (Taori et al., 2023) and FLAN (Chung et al., 2024) datasets. Ivison et al. (2023) on the other hand found that QLoRA did not perform as well as full finetuning for Llama-2-7B, 13B and 70B models trained on the Tülü-2 dataset when evaluated across a suite of tasks including MMLU, GSM8K, AlpacaEval (which uses LLM-as-a-judge; (Dubois et al., 2024)) and HumanEval. One recent notable study is Astraios, which found that LoRA performed worse than full finetuning on 8 datasets and across 4 model sizes (up to 16 billion parameters), on 5 representative code tasks (Zhuo et al., 2024). Our study corroborates these results.

The conclusions have also been mixed with regards to the practical details surrounding LoRA target modules and rank: Raschka (2023) and Dettmers et al. (2024) show that optimized LoRA configurations perform as well as full finetuning, and that performance is governed by choice of target modules but *not* rank. In contrast, Liu et al. (2024) shows that LoRA *is* sensitive to ranks. It is likely that some of these discrepancies are due to differences in finetuning datasets and evaluations.

Learning-Forgetting tradeoffs Continual learning on a new target domain often comes at the expense of performance in the source domain (Lesort et al., 2020; Wang et al., 2024). A relevant example is that code-finetuned LLMs lose some of their capabilities in language understanding and commonsense reasoning (Li et al., 2023; Roziere et al., 2023; Wei et al., 2023). A common approach to mitigate forgetting involves "replaying" source-domain data during continual learning, which can be done by storing the data in a memory buffer, or generating it on the fly (Lesort et al., 2022; Scialom et al., 2022; Sun et al., 2019).

6 Discussion

Does the difference between LoRA and full finetuning decrease with model size? Studies in the past have hinted at a relationship between the effectiveness of finetuning and model size (Aghajanyan et al., 2020; Hu et al., 2021; Zhuo et al., 2024). While recent studies have successfully applied LoRA to 70B parameter models (Ivison et al., 2023; Yu et al., 2023), we leave a rigorous study of these intriguing scaling properties to future work.

Limitations of the spectral analysis. The observation that full finetuning tends to find high rank solutions does not rule out the possibility of low-rank solutions; rather, it shows that they are not typically found. An alternative interpretation is that the rank needed to reconstruct the weight matrix is higher than the rank needed for a downstream task.

Why does LoRA perform well on math and not code? One hypothesis is that math datasets involve a smaller domain shift; they include a larger percentage of English and lead to decreased forgetting. The second hypothesis is that the GSM8K evaluation is too easy and does not capture the new college-level math learned in finetuning.

7 Conclusion

This work sheds light on the downstream performance of contemporary LLMs (with 7 and 13 billion parameters) trained with LoRA. Different from most prior work, we use domain-specific datasets in code and math, associated with sensitive evaluation metrics. We show that LoRA underperforms full finetuning in both domains. We also show that LoRA keeps the finetuned model's behavior close to that of the base model, with diminished source-domain forgetting and more diverse generations at inference time. We investigate LoRA's regularization properties, and show that full finetuning finds weight perturbations are far from being low-rank. We conclude by analyzing LoRA's increased sensitivity to hyperparameters and highlighting best practices.

References

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`, 2022.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018. URL `https://api.semanticscholar.org/CorpusID:3922816`.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

Yuqing Du, Alexander Havrilla, Sainbayar Sukhbaatar, Pieter Abbeel, and Roberta Raileanu. A study on improving reasoning in language models. In *I Can't Believe It's Not Better Workshop: Failure Modes in the Age of Foundation Models*, 2024. URL `https://openreview.net/forum?id=tCZFmDyPFm`.

Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL `https://zenodo.org/records/10256836`.

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.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 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.

Hamish Ivison, Yizhong Wang, Valentina Pyatkin, Nathan Lambert, Matthew Peters, Pradeep Dasigi, Joel Jang, David Wadden, Noah A Smith, Iz Beltagy, et al. Camels in a changing climate: Enhancing lm adaptation with tulu 2. *arXiv preprint arXiv:2311.10702*, 2023.

Weisen Jiang, Han Shi, Longhui Yu, Zhengying Liu, Yu Zhang, Zhenguo Li, and James Kwok. Forward-backward reasoning in large language models for mathematical verification. 2023.

Dawid Jan Kopiczko, Tijmen Blankevoort, and Yuki Markus Asano. Vera: Vector-based random matrix adaptation. *arXiv preprint arXiv:2310.11454*, 2023.

Timothée Lesort, Vincenzo Lomonaco, Andrei Stoian, Davide Maltoni, David Filliat, and Natalia Díaz-Rodríguez. Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges. *Information fusion*, 58:52–68, 2020.

Timothée Lesort, Oleksiy Ostapenko, Diganta Misra, Md Rifat Arefin, Pau Rodríguez, Laurent Charlin, and Irina Rish. Challenging common assumptions about catastrophic forgetting. *arXiv preprint arXiv:2207.04543*, 2022.

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. *arXiv preprint arXiv:2402.09353*, 2024.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The e2e dataset: New challenges for end-to-end generation. *arXiv preprint arXiv:1706.09254*, 2017.

Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. Openwebmath: An open dataset of high-quality mathematical web text. *arXiv preprint arXiv:2310.06786*, 2023.

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

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Sebastian Raschka. Practical tips for finetuning llms using lora (low-rank adaptation), 2023. URL https://magazine.sebastianraschka.com/p/practical-tips-for-finetuning-llms#%C2%A7enable-lora-for-more-layers.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019.

Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan. Fine-tuned language models are continual learners. *arXiv preprint arXiv:2205.12393*, 2022.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2023.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1): 1929–1958, 2014.

Fan-Keng Sun, Cheng-Hao Ho, and Hung-Yi Lee. Lamol: Language modeling for lifelong language learning. *arXiv preprint arXiv:1909.03329*, 2019.

Simeng Sun, Dhawal Gupta, and Mohit Iyyer. Exploring the impact of low-rank adaptation on the performance, efficiency, and regularization of rlhf, 2023.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application, 2024.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. *arXiv preprint arXiv:2212.09561*, 2022.

Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2017.

Wenhan Xia, Chengwei Qin, and Elad Hazan. Chain of lora: Efficient fine-tuning of language models via residual learning. *arXiv preprint arXiv:2401.04151*, 2024.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024a.

Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. Lora land: 310 fine-tuned llms that rival gpt-4, a technical report. *arXiv preprint arXiv:2405.00732*, 2024b.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. Astraios: Parameter-efficient instruction tuning code large language models. *arXiv preprint arXiv:2401.00788*, 2024.

Appendix

A Experimental Setup

Code CPT.

- Optimizer: Decoupled LionW ($\beta = 0.9, 0.95$, weight decay= $1e-6$)
- Learning rate = $1e-5$ for full finetuning and LoRA after a learning rate sweep.
- LR Scheduler: Inverse square root with warmup $t_{warmup} = 500$ batches, $t_{scale} = 500$ batches, $t_{cooldown} = 5200$ batches $\alpha_{fdecay} = 1.0$ $\alpha_{fcooldown} = 0.0$
- We train the model once for 20B tokens. Then perform individual cooldowns as follows. We set a target max training duration, and define the last 20% of max training duration as the cooldown period. We retrain from the latest available checkpoint prior to the cooldown period.
- LoRA configuration: $\alpha = 32$, dropout=0.05.
- maximum sequence length = 4096.

Math CPT.

- Optimizer: Decoupled LionW ($\beta = 0.9, 0.95$, weight decay= $1e-6$).
- Learning rate = $4e-5$ for LoRA and $5e-6$ for full finetuning.
- We ran individual runs with the Cosine with Warmup scheduler ($\alpha_f = 0, t_{warmup} = 100$ batches), without learning rate cooldowns
- LoRA configuration $\alpha = 2r$, dropout=0.05.
- maximum sequence length = 4096

Code IFT.

- Optimizer: Decoupled LionW ($\beta = 0.9, 0.95$, weight decay= 0)
- LR scheduler: Cosine with Warmup scheduler with $\alpha_f = 0.01$ and $t_{warmup} = 0.1$ of the total duration.
- LoRA configuration: $\alpha = 32$, dropout=0.05.
- maximum sequence length = 4096

Math IFT. Same as code IFT, except that

- maximum sequence length = 1024

We compared the two optimizers by training for two epochs of Magicoder-Evol-Instruct-110K using different learning rates. We found that Decoupled LionW outperformed DecoupledAdamW on HumanEval for both LoRA and full finetuning, and across learning rates, as seen in Fig. S1.

B Learning rate searches

For IFT we find that LoRA LRs should be an order of magnitude higher. For the longer CPT, these effects are more subtle.

B.1 Learning rate sensitivity analysis across optimizers

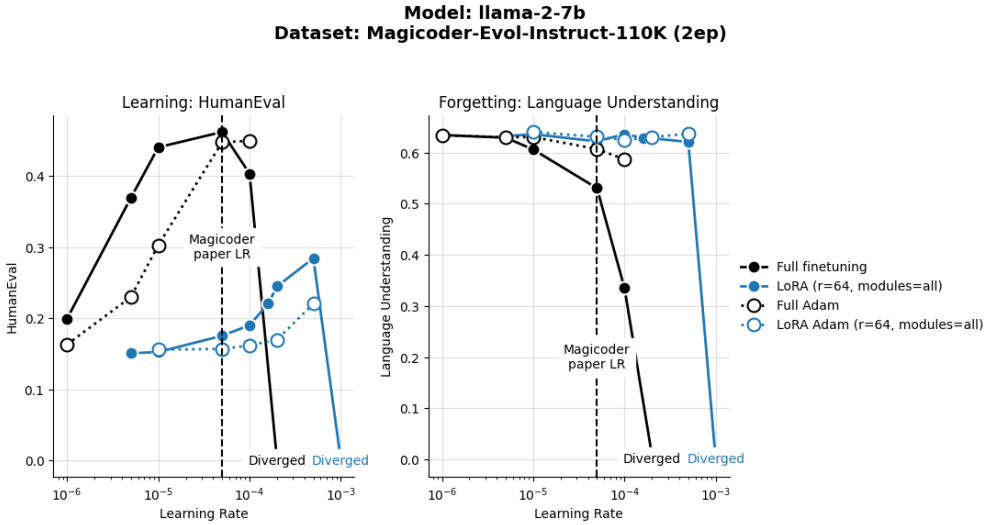


Figure S1: Comparing LionW to AdamW across learning rates for two epochs of the Megicoder-Evol-Instruct-110K dataset. Left: HumanEval; right: Average of "Language Understanding" benchmarks. Both methods peak at the learning rate used in the original paper (Wei et al., 2023)

.

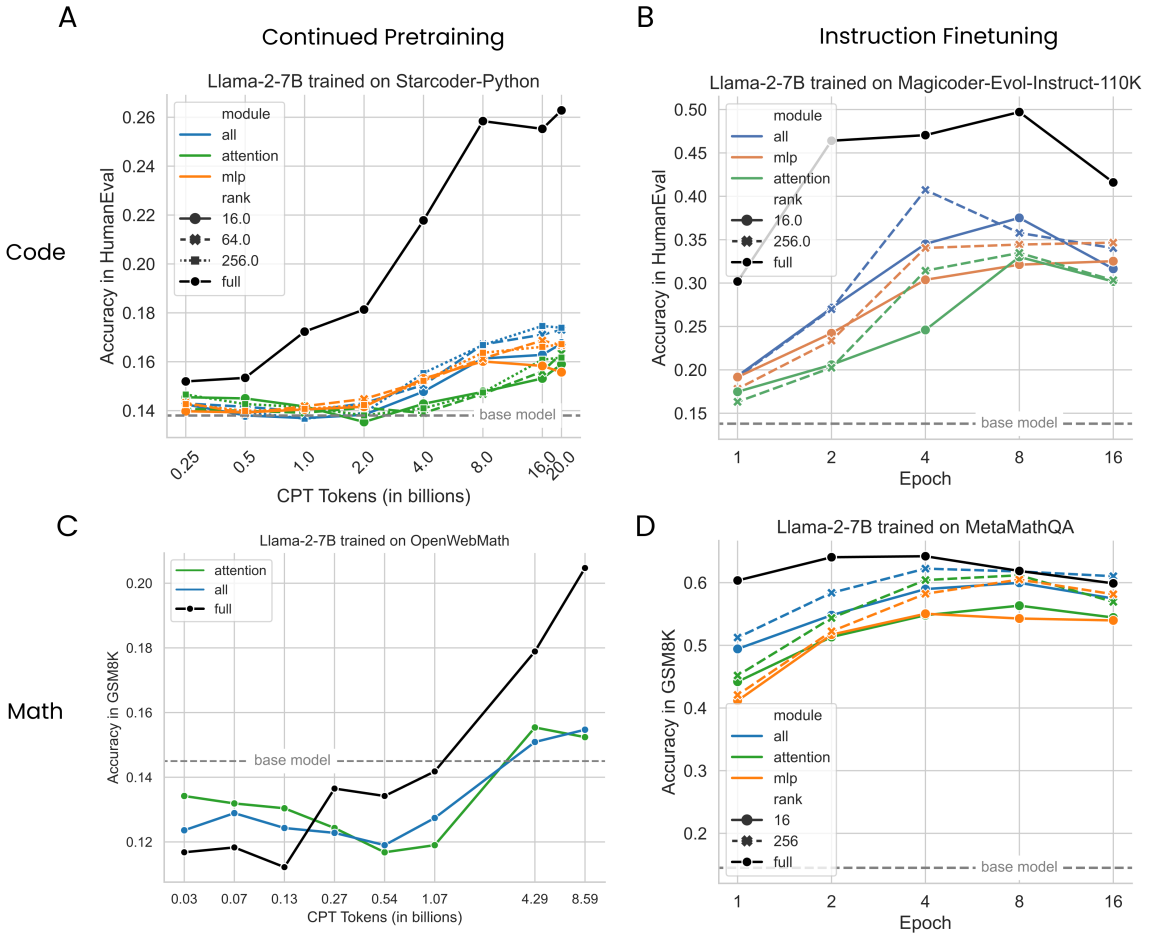


Figure S2: Sample-efficiency curves matching Fig. 2, with all individual LoRA configurations.

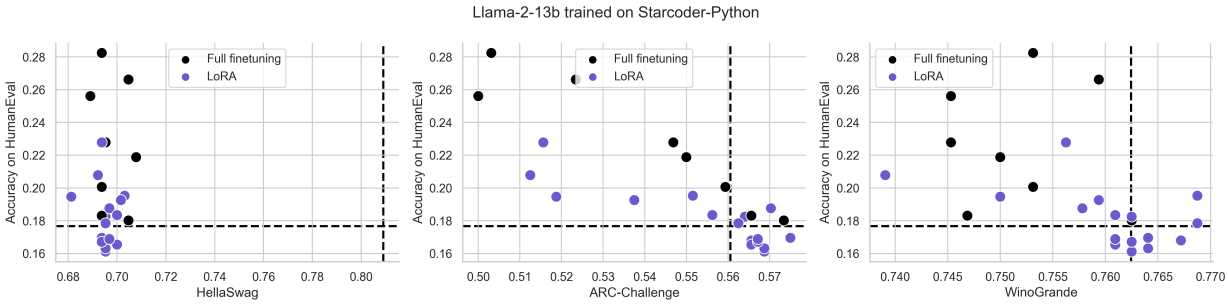
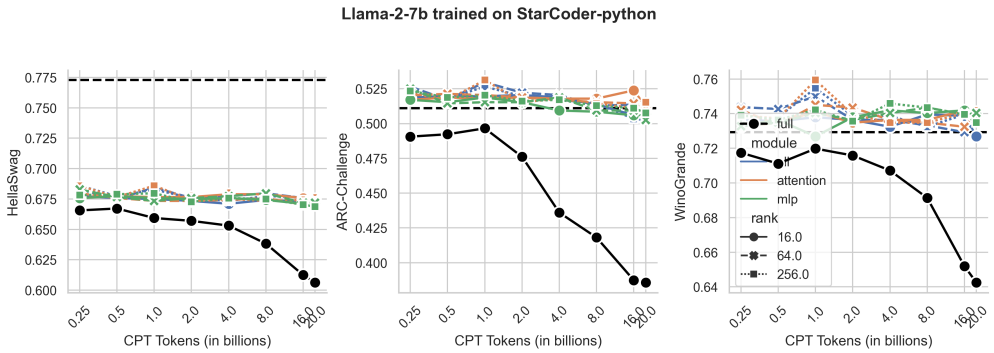
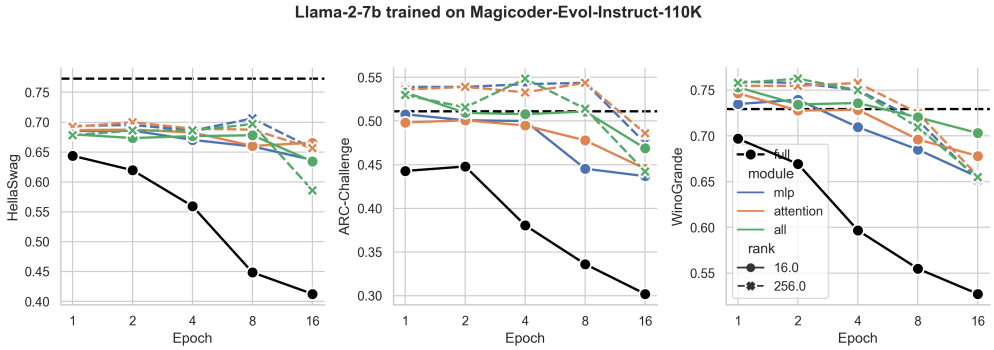


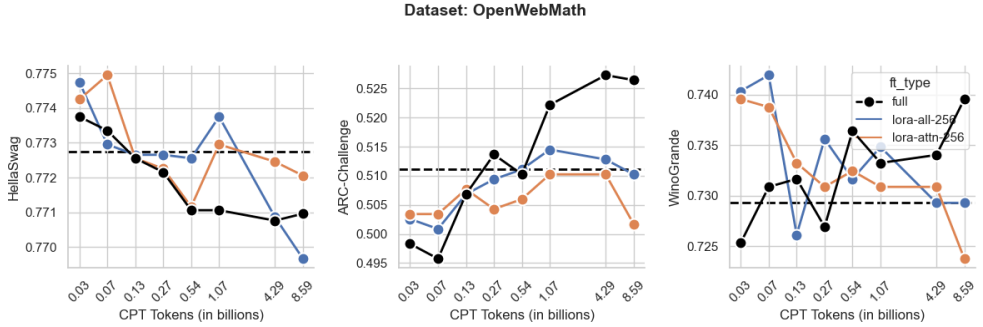
Figure S3: Pareto curves for continued pretraining of Llama-2-13B on up to 20B tokens of the Starcoder-Python (Code CPT).



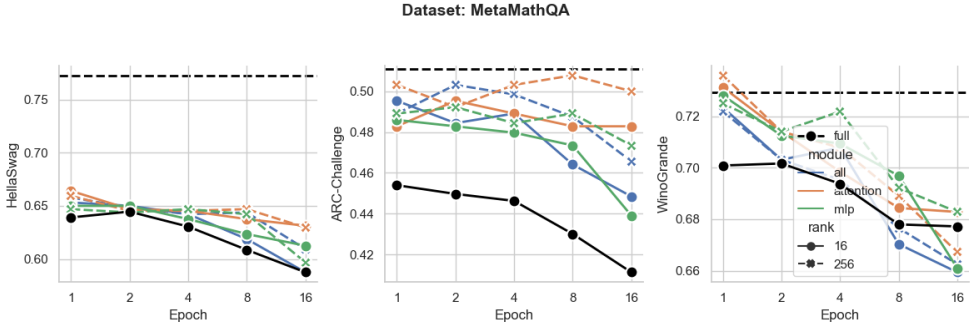
(a) **Code CPT**: Individual forgetting plots for Llama-2-7B on Starcoder-Python.



(b) **Code IFT**: Individual forgetting plots for Llama-2-7B on Magicoder-Evol-Instruct-110K.

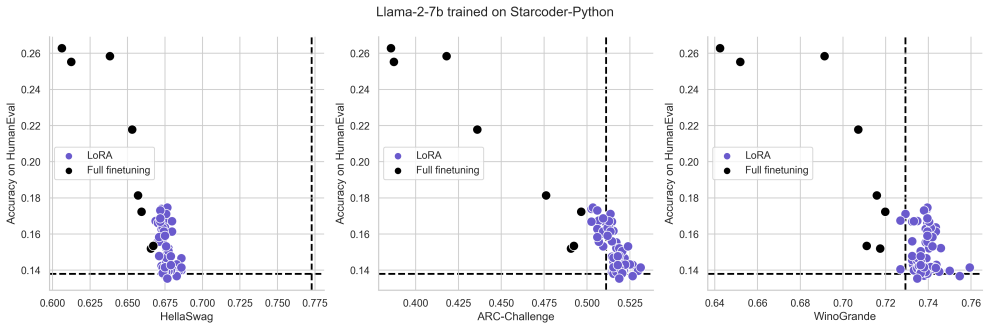


(c) **Math CPT**: Individual forgetting plots for Llama-2-7B on OpenWebMath.

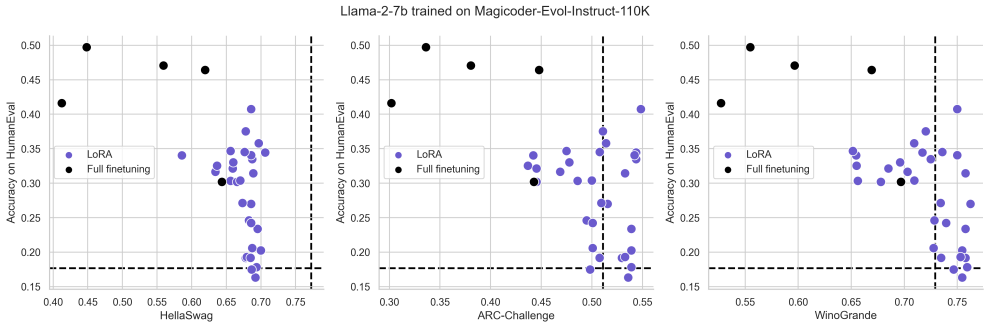


(d) **Math IFT**: Individual forgetting plots for Llama-2-7B on MetaMathQA.

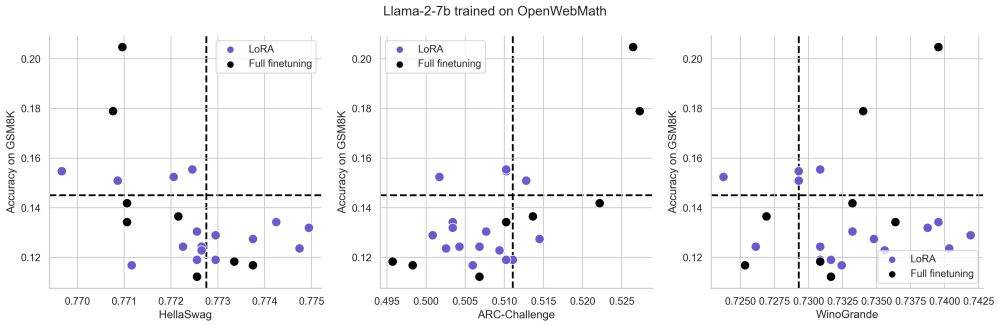
Figure S4: Same data as Fig. 3



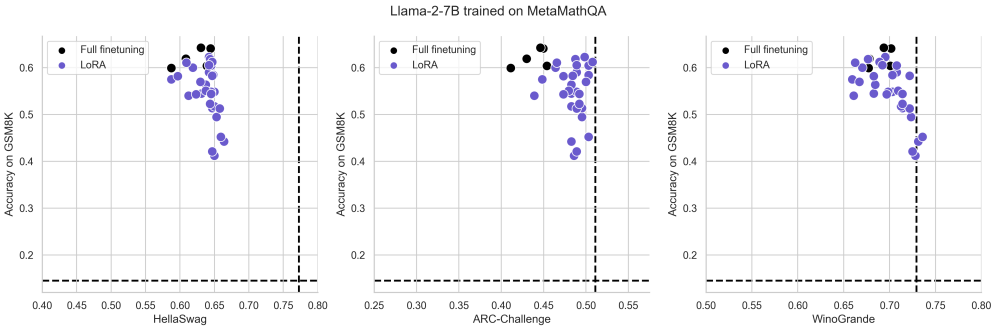
(a) **Code CPT**: Individual Pareto curves for Llama-2-7B on Starcoder-Python.



(b) **Code IFT**: Individual Pareto curves for Llama-2-7B on Magicoder-Evol-Instruct-110K.



(c) **Math CPT**: Individual Pareto curves for Llama-2-7B on OpenWebMath.



(d) **Math IFT**: Individual Pareto curves for Llama-2-7B on MetaMathQA.

Figure S5: Same data as in Fig. 4 plotted for individual tasks HellaSwag, ARC-Challenge and WinoGrande

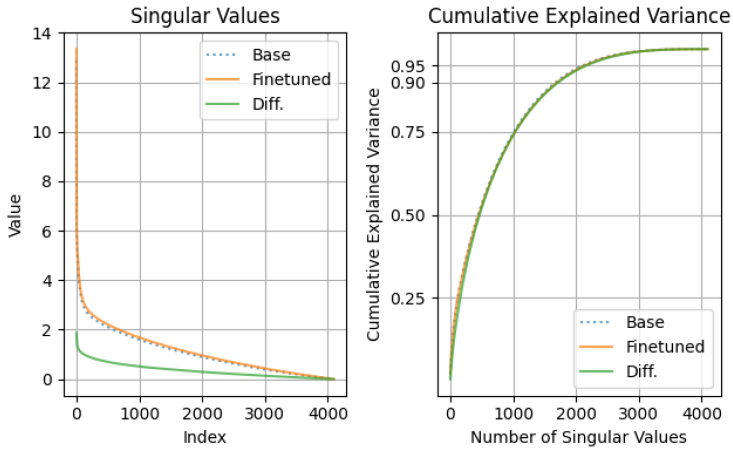
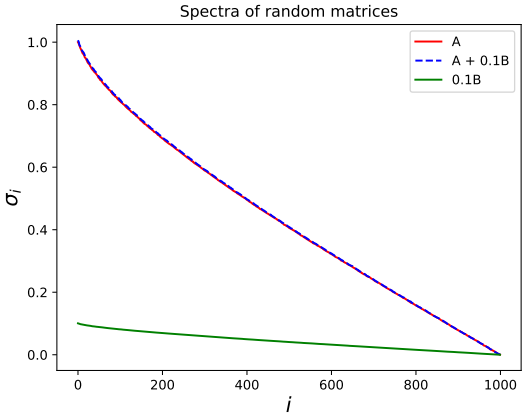
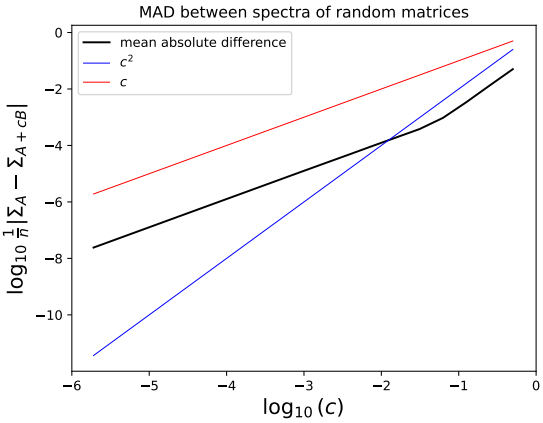


Figure S6: SVD analysis for 4096×4096 matrix W_q at layer 26. Left: singular values for base weights, finetuned weights, and their difference. Right: cumulative explained variance. Notice that for all three matrices, a rank > 1500 is needed to explain 90% of the variance.



(a) Spectrum of A, $A + cB$ and cB for $c = 0.1$.



(b) Mean absolute difference between spectra of A and $A + cB$ for various c.

Figure S7: Analyzing the sprectra of the sum of two 1000×1000 Gaussian i.i.d matrices. A and B are 1000×1000 random matrices with i.i.d. standard normal Gaussian entries.

C Training Datasets

C.1 MetaMathQA (Math IFT)

The MetaMathQA dataset (Yu et al. (2023), `https://huggingface.co/datasets/meta-math/MetaMathQA`) contains 395,000 samples that are bootsrapped from the GSM (Cobbe et al., 2021) and Math (Hendrycks et al., 2021) training sets. These samples are augmented by GPT-3.5 using the following methods:

- Answer Augmentation (155k samples, Yu et al. (2023)): this method proposed by the MetaMathQA authors generates multiple reasoning paths for a given mathetical question and filters for generated reasoning paths that contain the correct final answer.
- Rephrasing (130k samples, (Yu et al., 2023)): this method proposed by the MetaMathQA authors uses GPT-3.5 to rephrase questions. They check for the correctness of rephrased questions by using few-shot Chain of Thought prompting to compare reasoning chains and proposed answers with ground truth answers.

Both Self-Verification (Weng et al., 2022) and FOBAR (Jiang et al., 2023) fall under the category of "backward reasoning," where the question starts with a given condition and requires reasoning backwards to solve for an unknown variable. In order to generate new mathematical questions, a numerical value in the original question is masked as a variable X, and the question is rephrased accordingly.

- Self-Verification (55k samples, Weng et al. (2022)): the question is rephrased into a declarative statement followed by a question about the masked variable X.
- FOBAR (55k samples, Jiang et al. (2023)): this approach is similar to Self-Verification but directly appends the answer to the question using the template "If we know the answer to the above question is $\hat{A}_c$, what is the value of unknown variable x?"

MetaMathQA samples are organized by 4 columns: `type`, `original_question`, `query` and `response`.

We include two full examples below:

Example C.1: MetaMathQA

`type`: `GSM_Rephrased`

`query`: Scott wants to prepare and freeze a large portion of ratatouille. He purchases 5 pounds of eggplants and 4 pounds of zucchini at a rate of $2.00 per pound from the farmers' market. Additionally, he needs 4 pounds of tomatoes that cost $3.50 per pound, 3 pounds of onions priced at $1.00 per pound, and a pound of basil sold for $2.50 per half pound. If this quantity yields 4 quarts, what is the cost per quart?

`original_question`: Scott wants to make and freeze a large batch of ratatouille. At the farmers' market he buys 5 pounds of eggplants and 4 pounds of zucchini at $2.00 a pound. He needs 4 pounds of tomatoes that are $3.50 a pound. The onions are $1.00 a pound and he needs 3 pounds. Then he needs a pound of basil which is sold for $2.50 per half pound. If this yields 4 quarts, how much does each quart cost?

`response`: The cost of the eggplants is 5 pounds x $2.00 per pound = $10.00. The cost of the zucchini is 4 pounds x $2.00 per pound = $8.00. The cost of the tomatoes is 4 pounds x $3.50 per pound = $14.00. The cost of the onions is 3 pounds x $1.00 per pound = $3.00. The cost of the basil is 1 pound x ($2.50 per half pound x 2) = $5.00. The total cost for all the ingredients is $10.00 + $8.00 + $14.00 + $3.00 + $5.00 = $40.00. Since this quantity yields 4 quarts, the cost per quart is $40.00 / 4 quarts = $10.00 per quart. #### 10 The answer is: 10

Example C.2: MetaMathQA

type: `MATH_SV`
`query`: If a school bus leaves school with X students on board, and one-half of the students get off the bus at each of the first three stops, The number of students remaining on the bus after the third stop is 6. What is the value of unknown variable X?

`original_question`: If a school bus leaves school with 48 students on board, and one-half of the students get off the bus at each of the first three stops, how many students remain on the bus after the third stop?

`response`: To solve this problem, we need to determine the value of x, which represents the initial number of students on the bus. Let's break down the information given: Number of students remaining on the bus after the third stop: 6 Number of students getting off the bus at each stop: 1/2 Number of stops: 3 We can set up the equation as follows: Number of students on the bus * (1/2)^ 3 = Number of students remaining on the bus after the third stop x * (1/2)^ 3 = 6 Let's simplify and solve for x: x * (1/8) = 6 To isolate x, we multiply both sides of the equation by 8: x * (1/8) * 8 = 6 * 8 x = 48 The value of x is 48. The answer is: 48

C.2 Magicoder-Evol-Instruct-110k (Code IFT)

Example C.3: Magicoder-Evol-Instruct-110k

`instruction`: Please amend the subsequent Python script so that it includes a 'while' loop rather than the existing 'for' loop, which iterates through the items of an integer list. The script currently has a bug where it attempts to print an object that is outside the bounds of the list. Fix this error and modify the script to use 'while' instead of 'for' loop. Ensure your script correctly handles empty lists.

```
‘‘‘python
# Establish an integer list
arr = [1, 2, 3, 4]
# Determine the length of the list
n = len(arr)
# Traverse the list and output each individual element
for i in range(n+1): print(arr[i]) ‘‘‘
```

`response`:

```
‘‘‘python # Establish an integer list arr = [1, 2, 3, 4]
# Determine the length of the list
n = len(arr)
# Initialize index at 0 i = 0
# Traverse the list and output each individual element
while i < n: print(arr[i]) i += 1
‘‘‘
```

In the given code, it tries to access ‘arr[n]‘ which is out of bounds as python list indexing starts from 0 and ends at n-1 for a list of length n. Also, I used ‘while‘ loop as per your request. By initializing ‘i=0‘ before the loop and ‘i += 1‘ within the loop, we can iterate over the list. This iteration continues until ‘i‘ is less than ‘n‘, the length of the list. The ‘while‘ loop will not proceed if the list is empty so it correctly handles empty lists.

C.3 Starcoder Python (Code CPT)

Example C.4: Starcoder-Python

```
```python
"""
function of invoking Gitee API
"""

import base64
import logging
import requests
from flask import current_app
from requests import exceptions
logger = logging.getLogger(__name__)

ORG_URL = "\href{https://gitee.com/api/v5/orgs}{https://gitee.com/api/v5/orgs}"

REPO_URL = "\href{https://gitee.com/api/v5/repos}{https://gitee.com/api/v5/repos}"

def get_request(url, params):

    """

    get request

    """

    logger.debug("Get request, connect url: %s", url)

    try:

        response = requests.get(url,params=params)

        return True, response

    except exceptions.ConnectionError as err:

        logger.error(err)

        return False, 'connection error'

    except IOError as err:

        logger.error(err)

        return False, 'IO error'
```

more functions truncated...
```

C.4 OpenWebMath (Math CPT)

Example C.5: OpenWebMath

`url:` `http://math.stackexchange.com/questions/222974/probability-of-getting-2-aces-2-kings-and-1-queen-in-a-five-card-poker-hand-pa`

`text:` # Probability of getting 2 Aces, 2 Kings and 1 Queen in a five card poker hand (Part II) So I reworked my formula in method 1 after getting help with my original question - Probability of getting 2 Aces, 2 Kings and 1 Queen in a five card poker hand. But I am still getting results that differ...although they are much much closer than before, but I must still be making a mistake somewhere in method 1. Anyone know what it is? Method 1 $P(2A \cap 2K \cap 1Q) = P(Q|2A \cap 2K)P(2A|2K)P(2K)$ $$= \frac{1}{12}\frac{{4 \choose 2}{46 \choose 1}}{50 \choose 3}\frac{{4 \choose 2}{48 \choose 3}}{52 \choose 5}$$ $$= \frac{(6)(17296)(6)(46)}{(2598960)(19600)(12)}$$ $$= 4.685642 * 10
– Jean-Sébastien Oct 28 '12 at 19:56 But there already answers to my original question so those answers would not make sense now that I am using a new formula for method 1.
– sonicboom Oct 28 '12 at 20:03 Conditional probability arguments can be delicate. Given that there are exactly two Kings, what's the 46 doing? That allows the possibility of more Kings.
– André Nicolas Oct 28 '12 at 20:26 The 46 is because have already taken two kings from the pack leaving us with 50. And now we have chosen 2 aces and we have to pick the other 1 card from the 50 remaining cards less the 4 aces?
– sonicboom Oct 28 '12 at 20:42 show 1 more comment $$\frac{1}{11}\frac{{4 \choose 2}{44 \choose 1}}{48 \choose 3}\frac{{4 \choose 2}{48 \choose 3}}{52 \choose 5}$$ If you wrote this as $$\frac{{4 \choose 2}{48 \choose 3}}{52 \choose 5}\frac{{4 \choose 2}{44 \choose 1}}{48 \choose 3}\frac{{4 \choose 1}{40 \choose 0}}{44 \choose 1}$$ it might be more obvious why they are the same.

`date:` `2014-03-07 11:01:44`

D Theoretical Memory Efficiency Gains with LoRA for Single and Multi-GPU Settings

Modern systems for training neural networks store and operate on the following objects (following the conventions in Rajbhandari et al. (2020)). Most memory requirements relate to *model states*, which include:

- parameter weights
- gradients
- higher order optimization quantities such as optimizer momentum and variance in the Adam optimizer, and the momentum in the Lion optimizer

The remaining memory requirements come from the *residual states*:

- activations (which depend on batch size and maximum sample sequence length)
- temporary buffers for intermediate quantities in the forward and backward pass.

which will require more memory when increasing the batch size and maximum sequence lengths.

LoRA offers memory savings with respect to the *model states*. The next two sections describe these memory savings in the single GPU and multi-GPU setting with examples loosely inspired by Rajbhandari et al. (2020).

The data stored at single precision includes:

- a "master copy" of the tuned parameter weights

- the gradient
- all optimizer states (both momentum and variance for Adam, and just momentum for Lion)

For simplicity, we do not consider mixed-precision training, which involves storing critical data at single precision (fp32; 4 bytes per number) while performing some computations at half precision (fp16 or bfloat16; 2 bytes per number).

D.1 Training on a Single GPU

In the single GPU setup, the difference in memory requirements between LoRA and full finetuning is particularly drastic when using the Adam optimizer (Hu et al., 2021; Rajbhandari et al., 2020).

Storing the master weights in fp32 requires 4 bytes per parameter, while storing the gradient in fp32 requires 4 bytes *per tuned parameter*. In order to maintain the optimizer state in fp32 for Adam, 8 bytes per tuned parameter are required; 4 bytes for the momentum term, and 4 bytes for the variance term. Let Ψ be the number of model parameters. Therefore, in the Adam full finetuning setting of a $\Psi = 7B$ parameter model, the total memory requirements are at least roughly $4 \times \Psi + 4 \times \Psi + 8 \times \Psi = 112$ GB.

The Lion optimizer only uses a momentum term in the gradient calculation, and the variance term in Adam therefore disappears. In the Lion full finetuning setting of a $\Psi = 7B$ parameter model, the total memory requirements are therefore roughly $4 \times \Psi + 4 \times \Psi + 4 \times \Psi = 84$ GB.

LoRA, on the other hand, does not calculate the gradients or maintain optimizer states (momentum and variance terms) *for most of the parameters*. Therefore the amount of memory used for these terms is drastically reduced.

A LoRA setting with Adam that only tunes matrices that are 1% of the total parameter count (e.g. $\Psi = 7B$ base model with 70M additional parameters used by LoRA) requires roughly $4 \times \Psi(1 + 0.01) + 4 \times \Psi \times 0.01 + 8 \times \Psi \times 0.01 = 29.12$ GB of memory. Theoretically this can be reduced further to $2 \times \Psi + 16 \times \Psi \times 0.01 = 15.12$ GB *if the non-tuned parameter weights are stored in bfloat16*. We use this assumption for the subsequent examples.

Note again that these numbers do not take into consideration sample batch size or sequence length, which affect the memory requirements of the activations.

D.2 Training on a Multiple with Fully Sharded Data Parallelism

Past approaches for training LLMs across multiple GPUs include model parallelism, where different layers of the LLM are stored on different GPUs. However this requires high communication overhead and has very poor throughput (Rajbhandari et al., 2020). Fully Sharded Data Parallelism (FSDP) shards the parameters, the gradient, and the optimizer states across GPUs. This incredibly efficient and actually is competitive with the memory savings offered by LoRA in certain settings.

FSDP sharding the parameter and optimizer states across N devices results in less memory usage relative to LoRA. LoRA on the other hand enables training on GPUs with far less memory and also emanes training without needing as many GPUs to shard across.

For example, in the Adam full finetuning setting of a $\Psi = 7B$ parameter model on 8 GPUs with FSDP, the total memory requirement for *each* GPU is roughly $(4 \times \Psi + 4 \times \Psi + 8 \times \Psi)/8 = 14$ GB. This reduces further to 3.5 GB for FSDP with 32 GPUs (see Table S1).

The LoRA with Adam setup on 8 GPUs (where $\Psi = 7B$ base model and there are 70M additional LoRA parameters) requires roughly $(2 \times \Psi + 16 \times \Psi \times 0.01)/8 = 1.89$ GB of memory per GPU. With 32 GPUs this decreases further to 0.4725 GB.

Standard industry level GPUs have on-device memory between 16 GB (e.g. V100s) and 80 GB (e.g. A100s and H100s). As Table S1 demonstrates, the per-GPU memory requirements for training a 7B parameter model decrease drastically as the number of GPUs increase. The memory requirements for training a 7B

7B Training	1 GPU	8 GPUs	16 GPUs	32 GPUs	64 GPUs
Adam	112 GB	14 GB	7 GB	3.5 GB	1.75 GB
Adam + LoRA	15.12 GB	1.89 GB	0.945 GB	0.4725 GB	0.236 GB
Lion	84 GB	10.5 GB	5.25 GB	2.625 GB	1.3125 GB
Lion + LoRA	14.84 GB	1.855 GB	0.9275 GB	0.464 GB	0.232 GB

Table S1: **Theoretical memory required to store the model and optimizer state during training for a 7B parameter model.** Note that the numbers exclude memory needed to store activations. FSDP sharding the parameter and optimizer states across N devices results in less memory usage relative to LoRA. LoRA on the other hand enables training on GPUs with far less memory and also emanes training without needing as many GPUs to shard across.

70B Training	1 GPU	8 GPUs	16 GPUs	32 GPUs	64 GPUs
Adam	1.12 TB	140 GB	70 GB	35 GB	17.5 GB
Adam + LoRA	151.2 GB	18.9 GB	9.45 GB	4.725 GB	2.36 GB
Lion	840 GB	105 GB	52.5 GB	26.25 GB	13.125 GB
Lion + LoRA	148.4 GB	18.55 GB	9.275 GB	4.64 GB	2.32 GB

Table S2: **Theoretical memory required to store the model and optimizer state during training for a 70B parameter model.**

model with Adam + LoRA on a single GPU are 15.12 GB, but the same per-GPU memory requirement for training a 7B model with Adam but *without* LoRA on 8 GPUs is 14 GB. In this 8 GPU scenario, the efficiency gains from LoRA disappear.

Table S2 applies similar calculations to a 70B parameter model. Finetuning such a large model on 8 GPUs is *only* possible using a technique like LoRA; where Adam requires 140 GB per GPU, Adam+LoRA requires 18.9 GB per GPU. The efficiency gains of LoRA relative to FSDP therefore depend on the model size and GPU availability/cost considerations.

We do the same analysis for a 405B parameter model to highlight how LoRA is beneficial as model size scales (Table S3).

405B Training	1	8	16	32	64	128	256
Adam	6480	810	405	202.5	101.25	50.625	25.3
Adam + LoRA	874.8	109.35	54.65	27.34	13.67	6.83	3.42
Lion	4860	607.5	303.75	151.875	75.94	37.97	18.98
Lion + LoRA	858.6	107.325	53.66	26.83	13.42	6.71	3.35

Table S3: **Theoretical memory required to store the model and optimizer state during training for a 405B parameter model.** Units are in GB