ADDAX: MEMORY-EFFICIENT FINE-TUNING OF LANGUAGE MODELS WITH A COMBINATION OF FORWARD-BACKWARD AND FORWARD-ONLY PASSES

Zeman Li **Xinwei Zhang** **Meisam Razaviyayn**
University of Southern California
{zemanli,xinweiz,razaviya}@usc.edu

ABSTRACT

Fine-tuning language models (LMs) with first-order optimizers often demands excessive memory, limiting accessibility, while zeroth-order optimizers use less memory, but suffer from slow convergence depending on model size. We introduce a novel method named Addax that integrates the recently introduced Memory-Efficient Zeroth-order Optimizer of Malladi et al. (2023) with Stochastic Gradient Descent (SGD). Addax obtains zeroth-order and first-order gradient estimates and optimally combines them as the descent direction in each step. The first-order updates are performed "in-place" to further save memory. Theoretically, we establish the convergence of Addax under mild assumptions, demonstrating less restrictive hyper-parameters and independence from model size. Our extensive experiments with diverse LMs and tasks show that Addax consistently outperforms zero-shot and MeZO in terms of accuracy. Moreover, Addax surpasses the performance of standard fine-tuning approaches, such as SGD and Adam, in specific scenarios with significantly less memory requirement.

1 INTRODUCTION

Fine-tuning pre-trained language models (LMs) is crucial for diverse natural language processing tasks, including text classification and sentiment analysis (Devlin et al., 2019), as well as their use in different domains (Gururangan et al., 2020). However, standard fine-tuning with Adam optimizer demands excessive memory usage due to gradient and/or optimizer state storage, presenting a challenge as LMs grow in scale (Brown et al., 2020; OpenAI, 2023). For instance, fine-tuning a 13-billion-parameter model like OPT (Zhang et al., 2022) in mixed precision requires over 316 GB of memory, hindering accessibility for researchers and practitioners with limited resources and specialized hardware. This memory burden restricts innovation and experimentation.

Recently, various memory-efficient methods for fine-tuning Large Language Models (LLMs) have been proposed. In-context learning (ICL) utilizes a single *inference pass*, incorporating label examples in its context for prediction (Brown et al., 2020). Despite its limited success, ICL's performance is constrained by the model's context size and is shown to be less effective than traditional Adam fine-tuning for medium-sized LMs (Brown et al., 2020). Parameter-Efficient Fine-Tuning (PEFT) selectively tunes a fraction of the network while freezing the rest of the parameters, and significantly reduces the *parameters* needed for fine-tuning (Hu et al., 2022; Li & Liang, 2021; Lester et al., 2021). Despite its efficiency, fine-tuning LMs with PEFT still requires more memory than model inference. For example, fine-tuning OPT-13B with Adam with a batch size of 8 requires $4\times$A100 GPUs (316GB total), whereas utilizing PEFT decreases this to $2\times$A100 GPUs (158GB total) with a batch size of 16 (Brown et al., 2020). Nonetheless, this memory requirement is still $6\times$ greater than what is needed for model inference, which is around 25GB.

Memory-Efficient Zeroth-order Optimizer (MeZO) proposed by Malladi et al. (2023) generates gradient estimators solely through forward passes with minimal memory overhead. Unlike classical zeroth-order optimization method ZO-SGD (Spall, 1992), MeZO allows in-place perturbation of model parameters to avoid storing the perturbation vector. Despite having a memory footprint equivalent to the inference phase, MeZO exhibits slower convergence compared to widely used

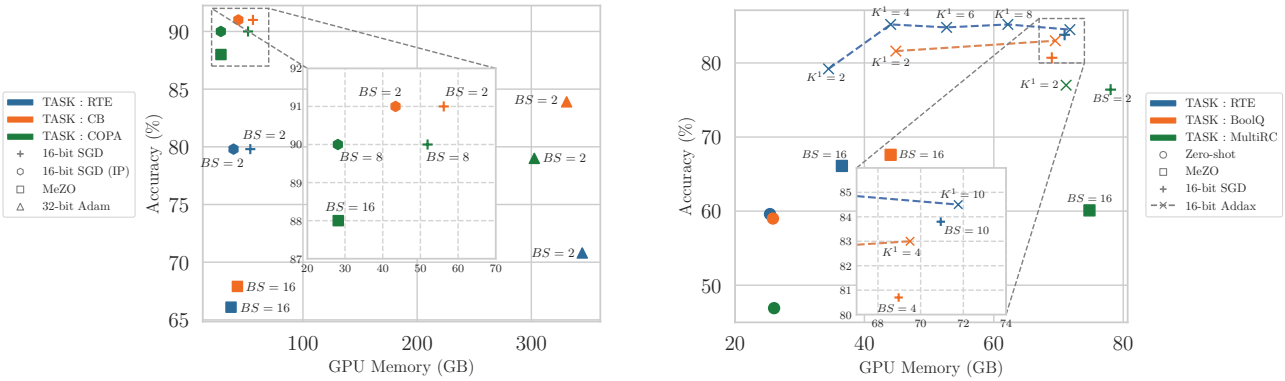


Figure 1: **Left**: Fine-tuning OPT-13B using SGD with in-place (IP) gradient updates and small batch sizes (BS) can outperform MeZO and Adam, while maintaining similar memory consumption to MeZO. **Right**: Fine-tuning OPT-13B using Addax with different first-order batch size K^1 outperforms zero-shot, MeZO and SGD. The figure reports the memory consumption and accuracy on three different datasets with zeroth-order batch size K^0 fixed at 12 for Addax.

first-order fine-tuning methods like Adam and AdamW (Kingma & Ba, 2015; Loshchilov & Hutter, 2019), both in theory and practice. Furthermore, the final performance of models fine-tuned with MeZO fails to match the ones fine-tuned with first-order methods.

Although Adam and AdamW have achieved great success in training deep learning models, the requirement to store optimizer states significantly burdens memory consumption. In contrast, SGD is simpler and more memory-efficient, and our experimental results demonstrate that fine-tuning OPT-13B using SGD with 16-bit floating-point calculations can outperform configurations that utilize Adam with 32-bit floating-point calculations (See Figure 1). The success of using SGD in fine-tuning LLMs may be attributed to the smoothness of the parameter space in LLMs and the favorable conditions already established by the loss function (Hao et al., 2019). While SGD significantly reduces memory consumption compared with Adam by eliminating the use of optimizer states, it is still non-comparable to the memory usage of MeZO. This is because the straightforward implementation of SGD necessitates additional storage for activations used for calculating the first-order gradient in backward propagation, as well as the gradients themselves.

To further reduce the memory footprint for SGD, several studies have explored the utilization of *in-place (IP) gradient update* during backward propagation (Zhao et al., 2024; Lv et al., 2023). Instead of separating the backward propagation and weight update steps, which requires storing the gradients for all layers, they combine the two steps by updating the weights in each layer as soon as the gradients are calculated. IP update does not require storage of the gradients of all layers and, thus, significantly reduces the memory requirement, making it comparable to MeZO for certain tasks. Our experiments, as illustrated in Figure 1, demonstrate that fine-tuning LLMs with SGD employing in-place gradient updates and small batch sizes can attain memory consumption levels comparable to those achieved when fine-tuning with MeZO. Therefore, in certain low-memory settings where MeZO can be applied, first-order gradient updates can also be feasible for small batch sizes. Given the availability of both first- and zeroth-order updates, we can ask the following natural question:

Question: Can we develop an optimizer for fine-tuning language models (LMs) that requires significantly less memory than standard first-order methods but still enjoys relatively fast convergence and produces high-quality fine-tuned models?

In this work, we answer this question by proposing Addax (***ADD**ition of gr**A**dient estimates through memory-efficient e**X**ecution*), a method that has the benefits of both worlds: **i)** being memory efficient algorithm, **ii)** having fast convergence speed and **iii)** achieving the best performance across different fine-tuning methods. Specifically, our contributions are:

1. Addax integrates MeZO with SGD to enhance MeZO's convergence rate and the resulting final model performance, while still keeping its memory consumption significantly smaller than that of standard first-order optimizers.

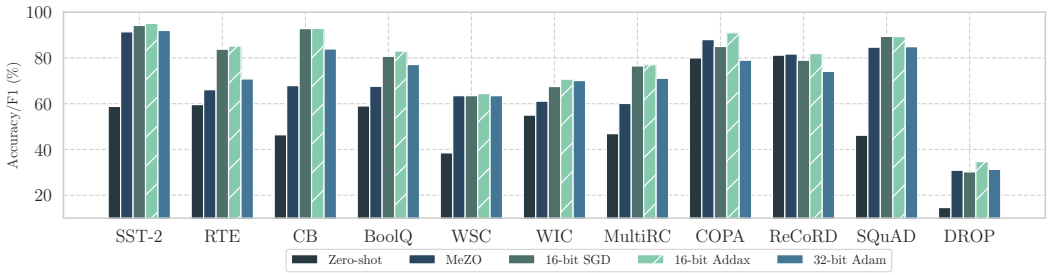


Figure 2: Accuracy/F-1 score resulted from fine-tuning OPT-13B model with zero-shot, MeZO, SGD, Addax, and Adam. Addax consistently outperforms other methods with GPU memory consumption comparable to that of MeZO (See Figure 3). The exact numbers are provided in Table 1.

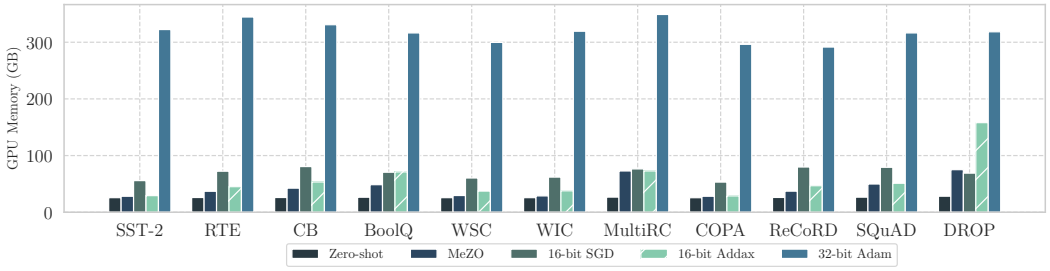


Figure 3: GPU memory requirement of fine-tuning OPT-13B with different methods on various tasks. The exact numbers are provided in Table 1.

2. We establish convergence of Addax under a set of mild assumptions. We show that Addax achieves $\mathcal{O}(1/\sqrt{T})$ convergence rate for non-convex smooth problems and the hyper-parameters for Addax are less restrictive than the Zeroth-order algorithms. Moreover, Addax's convergence rate is *independent* of the model size.
3. Our experiments include a broad range of model architectures (e.g. masked LM and autoregressive LM), scales (ranging from 350M to 13B parameter models), and tasks (e.g., classification, multiple-choice questions, and content generation). In experiments with RoBERTa-large, Addax consistently outperforms the performance of both zero-shot and MeZO across all tasks, and even outperforms the standard fine-tuning methods in several tasks (See Figure 4). With OPT-13B, Addax outperforms competing methods in 10 distinct tasks, while consuming up to $11\times$ less memory than standard fine-tuning and requiring only 1.04 to $2.1\times$ more memory than MeZO (See Figure 2 and Figure 3).
4. Further investigation reveals that Addax offers a versatile trade-off between resource availability and performance. In scenarios where memory is severely constrained, Addax is capable of being as memory efficient as MeZO by using small first-order batch sizes and using the rest for zeroth-order gradient estimates, thereby conserving memory of activations. Our experiments have shown that, under the same memory constraint as running SGD, Addax can outperform SGD with similar memory consumption in specific scenarios. (See Figure 1).

Notations. We are interested in the optimization of a smooth (possibly non-convex) loss function $\mathcal{L}$ with parameter $\boldsymbol{\theta} \in \mathbb{R}^d$. In other words, we are interested in solving

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^d} \mathcal{L}(\boldsymbol{\theta}) := \mathbb{E}_{x \in \mathcal{D}}[\ell(\boldsymbol{\theta}; x)], \tag{1}$$

where $\mathcal{D}$ denotes the data distribution and $x \in \mathcal{D}$ denotes the samples. Throughout the paper, we mark the values related to zeroth- and first-order gradient with $(\cdot)^0$, $(\cdot)^1$, respectively, and denote the iteration and coordinate indices as $(\cdot)_t$, $(\cdot)_i$, where $t \in \{0, \ldots, T\}$, $i \in \{1, \ldots, d\}$, respectively.

2 Addax: Algorithm Design

Addax uses both first- and zeroth-order update rules for fine-tuning. It is known that first-order update rules lead to faster algorithms (than zeroth-order ones), but they require more memory. To exploit and balance the benefits of both algorithms, Addax first draws a random batch $\mathcal{B}^0$ (with

Algorithm 1 Addax: ADDition of grAdient estimates through memory-efficient eXecution

Input: $\boldsymbol{\theta}, T, \mathcal{L}, K^0, K^1$, learning rates $\{\eta_t\}$, perturbation scale ϵ, weight parameter $\alpha \in [0, 1]$
for $t \in \{0, 1, \cdots, T-1\}$ **do**
  Randomly draw mini-batches $\mathcal{B}^0, \mathcal{B}^1$ uniformly from $\mathcal{D}$ with K^0, K^1 samples.
  $(g^0, s) \leftarrow$ **ZerothGrad**$(\boldsymbol{\theta}, \mathcal{L}, \mathcal{B}^0, \epsilon)$ (Algorithm 2) *# Estimate zeroth-order gradient*
  $\boldsymbol{g}^1 \leftarrow \frac{1}{K^1} \sum_{x \in \mathcal{B}^1} \nabla \mathcal{L}(\boldsymbol{\theta}, x)$ *# Estimate first-order gradient*
  Reset random number generator with seed s
  for $i \in \{1, \ldots, d\}$ **do**
    $z \sim \mathcal{N}(0, 1)$
    $\theta_i \leftarrow \theta_i - \eta_t \left(\alpha z g^0 + (1-\alpha) g_i^1\right)$ *# Update model parameters*
Output: $\boldsymbol{\theta}$

$|\mathcal{B}^0| = K^0$) of data and a random search direction $\mathbf{z} \in \mathbb{R}^d$. Then, it uses the drawn samples to obtain a stochastic zeroth-order estimate of the directional derivative of the objective function in the direction $\mathbf{z}$ at the point $\boldsymbol{\theta}$ based on (Spall, 1992; Malladi et al., 2023):

$$g^0 = \frac{1}{K^0} \sum_{x \in \mathcal{B}^0} \frac{\ell(\boldsymbol{\theta} + \epsilon \boldsymbol{z}; x) - \ell(\boldsymbol{\theta} - \epsilon \boldsymbol{z}; x)}{2\epsilon},$$

where ϵ is some small constant. Then, it draws a random batch $\mathcal{B}^1$ (with $|\mathcal{B}^1| = K^1$) of data and computes $\mathbf{g}^1 = \frac{1}{K^1} \sum_{x \in \mathcal{B}^1} \nabla \ell(\boldsymbol{\theta}; x)$. Finally, it updates the parameters of the model by:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \left(\alpha \mathbf{z} g^0 + (1-\alpha) \mathbf{g}^1\right),$$

where η is the step-size and $\alpha \in [0, 1]$ is a mixing constant for combining the two gradient estimates. We present the key steps of Addax in Algorithm 1, and leave the detailed discussions to Appendix A.

The choice of mini-batch sizes K^0, K^1 controls the memory usage of Addax. In the presence of devices with larger available memory, we can choose larger values of K^1, and when having less memory, we can reduce K^1 (and possibly increase K^0). The parameter α controls the balance between the zeroth- and first-order methods. When α is close to 1, the algorithm behaves more similar to ZO-SGD. However, when α is close to 0, Addax behaves similar to SGD. Specifically, SGD is an extreme case of Addax when $\alpha = 0$, and MeZO is another extreme case when $\alpha = 1$.

3 Addax: Theoretical analysis

This section presents our theoretical analysis for Algorithm 1. We present the informal statement of the convergence result here and relegate the formal statement and the proof to Appendix E.

Theorem 3.1 (Informal)**.** *Assume that the loss $\mathcal{L}$ is Lipschitz smooth and the stochastic gradients are unbiased and have bounded variance, then by running Algorithm 1 with $\eta = \mathcal{O}(T^{-1/2})$ and $\epsilon = \mathcal{O}(d^{-1/2} T^{-1/4})$), the output of the algorithm satisfies*

$$\mathbb{E}_t[\|\nabla \mathcal{L}(\boldsymbol{\theta}_t)\|^2] = \mathcal{O}\left(\frac{1}{\sqrt{T}} \cdot \sqrt{\frac{(1-\alpha)^2}{K^1} + \frac{\alpha^2 d}{K^0}}\right).$$

Note that α balances the importance of the zeroth- and first-order gradient in Algorithm 1, and we can optimize it to achieve an optimal convergence rate. It can be shown that the optimal α is given by $\alpha_\star = \frac{K^0}{K^0 + dK^1}$, leading to the convergence rate $\mathcal{O}\left(\sqrt{\frac{d}{T(K^0 + dK^1)}}\right)$. Moreover, compared with existing ZO algorithms, the condition for Algorithm 1 is less restrictive. For example, ZO-SGD requires smaller stepsizes $\epsilon = \mathcal{O}(d^{-1} T^{-1/2})$ and $\eta = \mathcal{O}(1/\sqrt{dT})$. In contrast, our stepsizes η and ϵ can be much larger, i.e., when $\frac{K^0}{K^1} \ll d$, the convergence rate of Algorithm 1 further reduces to $\mathcal{O}(1/\sqrt{TK^1})$, *independent* of the model size d. Our experiment verifies our reasoning (See Appendix C.4 for hyper-parameters details).

4 Experiments

In this section, we compare the performance of Addax with several baselines, including i) *zero-shot*, ii) *MeZO* (Malladi et al., 2023), iii) *SGD*, and iv) *Adam* (Kingma & Ba, 2015), where *zero-shot*

Table 1: Experiments on OPT-13B (with 1000 examples). Addax outperforms zero-shot, MeZO, SGD and Adam across the board on 10 tasks. For the accuracy of MeZO and 32-bit Adam, we report the results from Malladi et al. (2023).

Metrics	Task / Task type	**SST-2**	**RTE**	**CB**	**BoolQ**	**WSC**	**WIC**	**MultiRC**	**COPA**	**ReCoRD**	**SQuAD**	**DROP**
		—classification—							—multiple choice—		—generation—	
Accuracy/F1 (%)	Zero-shot	58.8	59.6	46.4	59.0	38.5	55.0	46.9	80.0	81.2	46.2	14.6
	32-bit Adam	92.0	70.8	83.9	77.1	63.5	70.1	71.1	79.0	74.1	84.9	31.3
	MeZO	91.4	66.1	67.9	67.6	63.5	61.1	60.1	88.0	81.7	84.7	30.9
	16-bit SGD	94.2	83.8	92.8	80.7	63.5	67.5	76.5	85.0	79.0	**89.4**	30.2
	16-bit Addax	**95.1**	**85.2**	**92.9**	**83.0**	**64.4**	**70.7**	**77.0**	**91.0**	**81.9**	89.3	**34.7**
Batch Size	32-bit Adam	—8—										
	MeZO	—16—										
	16-bit SGD	16	10	8	4	16	16	2	16	10	4	2
(K^1, K^0)	16-bit Addax	—(4, 12)—				(10, 12)	(8, 12)	(2, 12)	—(8, 12)—		(2, 12)	(2, 6)
Memory (GB)	32-bit Adam	322.4	344.5	330.9	316.4	299.7	319.4	349.0	296.3	291.5	316.4	318.4
	MeZO	28.2	37.0	42.5	48.6	29.5	28.8	72.8	28.2	37.2	49.8	75.1
	16-bit SGD	55.8	72.4	80.6	70.6	60.4	62.0	76.4	53.2	79.8	79.3	69.0
	16-bit Addax	29.3	45.1	53.4	71.2	37.3	37.8	72.9	28.8	46.8	51.1	158.1

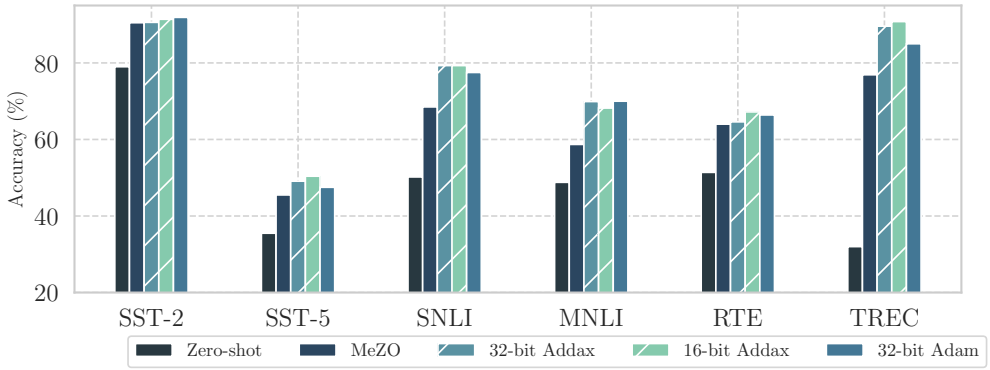


Figure 4: Experiments on RoBERTa-large: 16/32-bit Addax outperform zero-shot and MeZO across all tasks and outperform Adam in four out of six tasks. Detailed numbers can be found in Table 6.

evaluates on the pre-trained models without any fine-tuning. We implement Addax in both 16-bit (FP16) and 32-bit (FP32) data types.

Experiment Settings: We conduct two sets of experiments: firstly, fine-tuning the masked LM RoBERTa-large of (Liu et al., 2019) (350M) using zero-shot, MeZO, 32-bit Adam, and 16/32-bit Addax. Secondly, fine-tuning OPT-13B with different algorithms to assess their performance and memory usage. We also explore the impact of hyper-parameters α and $\frac{K^1}{K^0+K^1}$ on Addax's performance, detailed in Appendix D.2. Further details can be found in Appendix C.

Empirical Observation: *Addax outperforms other baseline methods while using substantially less memory than Adam*. For RoBERTa-large experiments, 16/32-bit Addax outperforms zero-shot and MeZO across six different tasks and surpasses Adam in four out of six tasks (Figure 4). In the experiments with OPT-13B model in Figure 2, 16-bit Addax outperforms zero-shot, MeZO, 16-bit SGD and Adam across 10 distinct tasks with moderate memory consumption. For example, Addax consumes up to $11\times$ less memory than standard fine-tuning and requires only 1.04 to $2.1\times$ more memory than MeZO (Figure 3). The batch size details for different algorithms can be found in Table 1. Notably, fine-tuning OPT-13B using Addax with a smaller first-order batch size K^1 surpasses the performance of SGD with larger batch sizes. This suggests that the zeroth-order gradient estimate in Addax provides stability (and regularization of the gradient) when K^1 is small and effectively reduces memory usage. For additional experimental results, refer to Appendix D.

5 Conclusion

This paper introduces Addax, a memory-efficient fine-tuning method for Large Language Models (LLMs). By leveraging both first- and zeroth-order stochastic gradient estimates, Addax demonstrates improved memory efficiency without sacrificing convergence speed or model performance, as validated by our extensive experiments across various models, tasks, and datasets.

REFERENCES

Stephen H Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V Nayak, Abheesht Sharma, Taewoon Kim, M Saiful Bari, Thibault Fevry, et al. Promptsource: An integrated development environment and repository for natural language prompts. *arXiv preprint arXiv:2202.01279*, 2022.

Roy Bar-Haim, Ido Dagan, Bill Dolan, Lisa Ferro, and Danilo Giampiccolo. The second pascal recognising textual entailment challenge. *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, 01 2006.

Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. *TAC*, 7:8, 2009.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 632–642, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1075. URL https://aclanthology.org/D15-1075.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Proceedings of the First International Conference on Machine Learning Challenges: Evaluating Predictive Uncertainty Visual Object Classification, and Recognizing Textual Entailment*, MLCW'05, pp. 177190, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540334270. doi: 10.1007/11736790_9. URL https://doi.org/10.1007/11736790_9.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Marie-Catherine De Marneffe, Mandy Simons, and Judith Tonhauser. The commitmentbank: Investigating projection in naturally occurring discourse. In *proceedings of Sinn und Bedeutung*, volume 23, pp. 107–124, 2019.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Long and Short Papers)*, 1:4171–4186, 2019.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

inc Fairscale. Fairscale: A general purpose modular pytorch library for high performance and large scale training, 2021.

Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*, 2020.

Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and Bill Dolan. The third pascal recognizing textual entailment challenge. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, RTE '07, pp. 19, USA, 2007. Association for Computational Linguistics.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8342–8360, 2020.

Insu Han, Rajesh Jayaram, Amin Karbasi, Vahab Mirrokni, David P Woodruff, and Amir Zandieh. Hyperattention: Long-context attention in near-linear time. *arXiv preprint arXiv:2310.05869*, 2023.

Yaru Hao, Li Dong, Furu Wei, and Ke Xu. Visualizing and understanding the effectiveness of bert. *arXiv preprint arXiv:1908.05620*, 2019.

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. In *In International Conference on Learning Representations*, 2022.

Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. Looking beyond the surface: A challenge set for reading comprehension over multiple sentences. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 252–262, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, 2021.

Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*, 2012.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, 2021.

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.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv:2306.09782*, 2023.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

OpenAI. Gpt-4 technical report. *arXiv: 2303.08774*, 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Mohammad Taher Pilehvar and Jose Camacho-Collados. Wic: the word-in-context dataset for evaluating context-sensitive meaning representations. *arXiv preprint arXiv:1808.09121*, 2018.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI Spring Symposium Series*, 2011.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew 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, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL `https://aclanthology.org/D13-1170`.

J.C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, 1992. doi: 10.1109/9.119632.

Ellen M Voorhees and Dawn M Tice. Building a question answering test collection. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 200–207, 2000.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems*, 32, 2019.

Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1101. URL `https://aclanthology.org/N18-1101`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Amir Zandieh, Insu Han, Majid Daliri, and Amin Karbasi. Kdeformer: Accelerating transformers via kernel density estimation. *arXiv preprint arXiv:2302.02451*, 2023.

Sheng Zhang, Xiaodong Liu, Jingjing Liu, Jianfeng Gao, Kevin Duh, and Benjamin Van Durme. Record: Bridging the gap between human and machine commonsense reading comprehension. *arXiv preprint arXiv:1810.12885*, 2018.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *arXiv: 2205.01068*, 2022.

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*, 2024.

A More discussion on Addax

Algorithm 1 outlines the detailed steps of Addax. In Step 4, the zeroth-order gradient estimator g^0 and random seed s are obtained using the samples $\mathcal{B}^0$ with batch size K_1, which are drawn uniformly from the total dataset $\mathcal{D}$. Similarly, Step 5 gets the first-order gradients $\boldsymbol{g}^1$ from samples $\mathcal{B}^1$ through backward propagation.

Algorithm 2 ZerothGrad (Malladi et al., 2023)

1: **Input:** parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, loss $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$, samples $\mathcal{B}$, perturbation scale ϵ.
2: Generate random seed s.
3: $\boldsymbol{\theta} \leftarrow$ **PertubParameters**$(\boldsymbol{\theta}, \epsilon, s)$
4: $\ell_+ \leftarrow \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$
5: $\boldsymbol{\theta} \leftarrow$ **PertubParameters**$(\boldsymbol{\theta}, -2\epsilon, s)$
6: $\ell_- \leftarrow \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$
7: $\boldsymbol{\theta} \leftarrow$ **PertubParameters**$(\boldsymbol{\theta}, \epsilon, s)$
8: $g \leftarrow (\ell_+ - \ell_-)/(2\epsilon)$
9: **Output:** g, s

Algorithm 3 PertubParameters

1: **Input:** parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, perturbation scale ϵ, random seed s.
2: Reset random number generator with seed s
3: **for** $i \in \{1, \ldots, d\}$ **do**
4: $\quad z \sim \mathcal{N}(0, 1)$
5: $\quad \theta_i \leftarrow \theta_i + \epsilon z$
6: **Output:** $\boldsymbol{\theta}$

A major step in Algorithm 1 is the computation of zeroth-order directional derivative g_0, done in Step 4, which is the subroutine call of Algorithm 2. Algorithm 2 is also used in MeZO. The directional derivative is obtained through the classical ZO gradient estimate SPSA (Definition A.1). To get the zeroth-order gradient estimate, Algorithm 2 requires the evaluation of the loss function $\mathcal{L}$ through two forward passes at points $\boldsymbol{\theta} + \epsilon \boldsymbol{z}$ and $\boldsymbol{\theta} - \epsilon \boldsymbol{z}$. The vanilla SPSA algorithm costs the twice of the memory of inference because of the need to store the $\boldsymbol{z} \in \mathbb{R}^d$. Algorithm 2 minimizes this overhead by generating a random seed s and resetting the random number generator each time model parameters are perturbed (see Step 3-7 in Algorithm 2). This approach guarantees that Algorithm 3 maintains a consistent direction for the random vector $\boldsymbol{z}$ across iterations. Employing this in-place operation results in Algorithm 2 having memory consumption comparable to that of inference.

Definition A.1 (Simultaneous Perturbation Stochastic Approximation (SPSA) (Spall, 1992))**.** Given a loss function $\mathcal{L}$ parameterized by $\boldsymbol{\theta} \in \mathbb{R}^d$, perturbation scale ϵ, and random direction $\boldsymbol{z} \in \mathbb{R}^d$, SPSA estimates the gradient on minibatch $\mathcal{B}$ as

$$\widehat{\nabla}\mathcal{L}(\boldsymbol{\theta}; \mathcal{B}) = \frac{\mathcal{L}(\boldsymbol{\theta} + \epsilon \boldsymbol{z}; \mathcal{B}) - \mathcal{L}(\boldsymbol{\theta} - \epsilon \boldsymbol{z}; \mathcal{B})}{2\epsilon} \boldsymbol{z}$$

Steps 7- 9 in Algorithm 1 leads to the main update rule of Addax. We use the same idea as in Malladi et al. (2023), where the seed s is stored instead of the random vector $\boldsymbol{z}$. The random generator is reset before updating the components (see Step 6 in Algorithm 1). This ensures that the random vector $\boldsymbol{z}$ maintains a consistent direction in the Algorithm 3 across each iteration. For each component θ_i in $\boldsymbol{\theta}$ where i ranges from 1 to d, the process begins by generating a random direction $z \sim \mathcal{N}(0, 1)$ in Step 8. Subsequently, each θ_i is updated using the weight combination of zeroth-order and first-order gradients, specifically $\left(\alpha z g^0 + (1 - \alpha) g_i^1\right)$, multiplied by the learning rate η_t. When iteration t reaches T, Addax outputs the final model parameters $\boldsymbol{\theta}$.

B ADDAX WITH IN-PLACE UPDATES

In this section, we provide the algorithm description for Addax with in-place gradient update (Algorithm 4). The technique of in-place gradient updates during backward propagation, as referenced in our approach, has been previously used in Zhao et al. (2024); Lv et al. (2023). In the modern deep learning training frameworks, such as PyTorch (Paszke et al., 2019)[1], store the gradient tensor for computing optimizer states and update the model weights after all layers of gradients are computed. This approach is not problematic for models with a small number of parameters; however, fine-tuning a large model like 13 billion parameters requires significant memory because the gradient tensor has the same size as the number of model parameters. For example, as for OPT-13B model, each parameter needs 2 bytes or 4 bytes for gradient storage, totaling 26 GB or 52 GB of memory,

[1] https://pytorch.org/

Algorithm 4 Addax: In-place Updates

1: **Input:** $\boldsymbol{\theta}, T, \mathcal{L}, K^0, K^1$, model $f(\cdot)$ with L layers, learning rates $\{\eta_t\}$, perturbation scale ϵ, weight parameter $\alpha \in [0, 1]$
2: **for** $t \in \{0, 1, \cdots, T-1\}$ **do**
3: Randomly draw mini-batches $\mathcal{B}^0$ uniformly from $\mathcal{D}$ with K^0 samples.
4: $(g^0, s) \leftarrow$ **ZerothGrad**$(\boldsymbol{\theta}, \mathcal{L}, \mathcal{B}^0, \epsilon)$ (Algorithm 2) *# Estimate zeroth-order gradient*
5: Randomly draw mini-batches $\mathcal{B}^1 = (\boldsymbol{x}, \boldsymbol{y})$ uniformly from $\mathcal{D}$ with K^1 samples.
6: $\hat{\boldsymbol{y}} \leftarrow f(\boldsymbol{x}, \boldsymbol{\theta})$ *# Forward pass*
7: $\ell \leftarrow \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$
8: **for** $l = L, \ldots, 1$ **do**
9: $\boldsymbol{\theta_l} \leftarrow [\theta_i$ for $\theta_i \in$ layer $l]$ *# Backward propagation*
10: $\boldsymbol{g_l} \leftarrow \frac{\partial \ell}{\partial \boldsymbol{\theta_l}}$
11: $\boldsymbol{\theta_l} \leftarrow \boldsymbol{\theta_l} - \eta_t(1-\alpha)\boldsymbol{g_l}$ *# Update model parameters*
12: $\boldsymbol{g_l} \leftarrow$ None *# Clear gradients*
13: Reset random number generator with seed s
14: **for** $i \in \{1, \ldots, d\}$ **do**
15: $z \sim \mathcal{N}(0, 1)$
16: $\theta_i \leftarrow \theta_i - \eta_t \alpha z g^0$ *# Update model parameters*
17: **Output:** $\boldsymbol{\theta}$

respectively. Because Addax does not require any optimizer states, such memory overhead can be avoided by combining the computation of first-order gradient estimates and parameter updates into a single step. As described in Algorithm 4 lines 8-12, we sequentially iterate over the Lth layer to the 1st layer, compute the gradient $\boldsymbol{g}_l$ (line 10), and perform in-place update to $\boldsymbol{\theta}_l$ (line 11). Right after that, we free the memory for gradient $\boldsymbol{g}_l$ (line 12). The loss computation, update of zeroth-order gradient g_0 remains the same as Algorithm 4. It is important to note that Algorithm 4 and Algorithm 1 differ only by the implementation, while the update rules remain unchanged. Therefore the final outcome of each optimization step is exactly the same. Unless otherwise noted, all experiments using Addax are implemented using in-place update as described in Algorithm 4.

C Experiment setup

C.1 Datasets

Our setup mainly follows Malladi et al. (2023). We employ the same datasets utilized in Malladi et al. (2023) and apply the same data processing procedure and settings for validation and training.

For the RoBERTa-large model, we utilize the following datasets: SST-2 (Socher et al., 2013), SST-5 (Socher et al., 2013), SNLI (Bowman et al., 2015), MNLI (Williams et al., 2018), RTE (Dagan et al., 2005; Bar-Haim et al., 2006; Giampiccolo et al., 2007; Bentivogli et al., 2009), and TREC (Voorhees & Tice, 2000). The test set is limited to 1,000 examples for both training and validation purposes. In our few-shot learning experiments, we set $k = 16$, where k represents the number of examples per class for training and validation.

For the OPT experiments, we employ the SuperGLUE dataset (Wang et al., 2019), comprising BoolQ (Clark et al., 2019), CB (De Marneffe et al., 2019), COPA (Roemmele et al., 2011), MultiRC (Khashabi et al., 2018), ReCoRD (Zhang et al., 2018), RTE (Dagan et al., 2005; Bar-Haim et al., 2006; Giampiccolo et al., 2007; Bentivogli et al., 2009), WIC (Pilehvar & Camacho-Collados, 2018), and WSC (Levesque et al., 2012). Following the approach in Malladi et al. (2023), we also include SST-2 (Socher et al., 2013) for development purposes, along with two question answering (QA) datasets: SQuAD (Rajpurkar et al., 2016) and DROP (Dua et al., 2019). For each dataset, we randomly select $1,000$ examples for training, 500 examples for validation, and $1,000$ examples for the testing.

C.2 Prompts

To ensure a fair comparison, we employ the same prompts as those used in Malladi et al. (2023), which were initially adapted from Gao et al. (2020), GPT-3 (Brown et al., 2020), and Prompt-

Source (Bach et al., 2022). Table 2 presents the prompts employed in our RoBERTa-large experiments, while Table 3 details the prompts utilized for the OPT experiments.

Table 2: The prompts for each datasets we used in our RoBERTa-large experiments, identical to those used in Malladi et al. (2023). There are three different task types: sentiment classification (sentiment cls.), topic classification (topic cls.) and natural language inference (NLI). C is the number of classes for each datasets. The label words can be filled in the `[MASK]` token of the prompt template. <S1> and <S2> are the first and second (if any) input sentence.

Dataset	C	**Type**	**Prompt**	**Label words**
SST-2	2	sentiment cls.	<S1> It was `[MASK]`.	{great, terrible}
SST-5	5	sentiment cls.	<S1> It was `[MASK]`.	{great, good, okay, bad, terrible}
TREC	6	topic cls.	`[MASK]` : <S1>	{Description, Expression, Entity, Human, Location, Number}
MNLI	3	NLI	<S1> ? `[MASK]`, <S2>	{Yes, Maybe, No}
SNLI	3	NLI	<S1> ? `[MASK]`, <S2>	{Yes, Maybe, No}
RTE	2	NLI	<S1> ? `[MASK]`, <S2>	{Yes, No}

Table 3: The prompts for each datasets we used in our OPT-13B experiments, identical to those used in Malladi et al. (2023). There are three types of tasks: classfication (cls.), multiple-choice (mch.), and question answering (QA). <text> is the input from the dataset and blue text are the label words. We follow the same practice in Malladi et al. (2023): for inference task, we incorporate different candidates into the prompt, compute the average log-likelihood for each, and select the candidate with the highest score. For question answering (QA) tasks, answers are produced through greedy decoding

Dataset	**Type**	**Prompt**
SST-2	cls.	<text> It was terrible/great
RTE	cls.	<premise> Does this mean that "<hypothesis>" is true? Yes or No? Yes/No
CB	cls.	Suppose <premise> Can we infer that "<hypothesis>"? Yes, No, or Maybe? Yes/No/Maybe
BoolQ	cls.	<passage> <question>? Yes/No
WSC	cls.	<text> In the previous sentence, does the pronoun "<span2>" refer to <span1>? Yes or No? Yes/No
WIC	cls.	Does the word "<word>" have the same meaning in these two sentences? Yes, No? <sent1> <sent2> Yes/No
MultiRC	cls.	<paragraph> Question: <question> I found this answer "<answer>". Is that correct? Yes or No? Yes/No
COPA	mch.	<premise> so/because <candidate>
ReCoRD	mch.	<passage> <query>.replace("@placeholder", <candidate>)
SQuAD	QA	Title: <title> Context: <context> Question: <question> Answer:
DROP	QA	Passage: <context> Question: <question> Answer:

C.3 IMPLEMENTATION

For the experiments involving both RoBERTa and OPT-13B, we adhere to a consistent fine-tuning paradigm in Malladi et al. (2023).

For the RoBERTa-large experiment, we evaluate Addax in two separate computational precision settings: one using 16-bit floating-point calculations (FP16), referred to as 16-bit Addax, and the other using 32-bit floating-point calculations (FP32), denoted as 32-bit Addax for clarity. For all RoBERTa-large experiments, MeZO and Adam are loaded in FP32 setting.

For the OPT-13B experiment, Addax is evaluated solely in the FP16 setting, as it is not feasible to operate Addax in FP32 on a single A100 (80G) GPU. Additionally, we also evaluate SGD in the FP16 setting and Adam in the FP32 setting. We adhere to the same naming convention for SGD and Adam, respectively referring to them as 16-bit SGD for the FP16 setting and 32-bit Adam for the FP32 setting.

We do not employ advanced quantization techniques such as `LLM.int8()` (Dettmers et al., 2022) and QLoRA (Dettmers et al., 2023), nor do we integrate Addax with with Parameter-Efficient Fine-Tuning methods (PEFT) (Hu et al., 2022; Li & Liang, 2021; Lester et al., 2021). For model inference, we utilize the standard PyTorch (Paszke et al., 2019) implementation of transformer. We do not use the memory-efficient approaches such as FlashAttention (Dao et al., 2022), KDEformer (Zandieh et al., 2023), and HyperAttention Han et al. (2023). Although the combination between Addax and these methods remains unexplored, we posit that their combination could significantly enhance Addax by further diminishing memory demands and augmenting performance. We leave the exploration of Addax's potential with various memory-efficient methods to future works.

C.4 HYPER-PARAMETERS

We present the hyper-parameters for all experiments conducted with RoBERTa-large in Table 4 and those for OPT-13B in Table 5. It is important to note that for both models, the hyper-parameters grid utilized for MeZO and Adam adheres to the specifications set forth in Malladi et al. (2023).

For RoBERTa-large experiments, both Addax and MeZO employ a constant learning rate schedule, while Adam uses linear scheduling. For the training process, Addax and Adam are set for $1,000$ steps, while MeZO extends to $100,000$ steps. We check validation performance every 50 training steps and save the best validation checkpoint for testing.

For OPT-13B experiments, Addax, SGD, and MeZO similarly adopt a constant learning rate schedule, with Adam maintaining its linear scheduling. Here, Adam is configured for 5 epochs, whereas Addax and SGD are set for $1,000$ steps, and MeZO for $20,000$ steps. We check validation performance every 50 training steps and save the best validation checkpoint for testing.

Addax can achieve a larger learning rate η than MeZO, resulting in a faster convergence speed. For the RoBERTa-large experiments, Addax uses the learning rate η of $\{1\mathrm{e}-5, 5\mathrm{e}-5, 1\mathrm{e}-4\}$, while MeZO uses the learning rate η of $\{1\mathrm{e}-7, 1\mathrm{e}-6, 1\mathrm{e}-5\}$. For the OPT-13B experiments, we fix the learning rate η of Addax to $1\mathrm{e}-4$, while MeZO uses significantly smaller learning rate η of $\{1\mathrm{e}-6, 1\mathrm{e}-7\}$.

Given the sensitivity of algorithms to hyper-parameter selection and our adherence to the identical settings detailed in Malladi et al. (2023), we report the accuracy of MeZO and Adam from their published results, applicable to both RoBERTa-large and OPT-13B experiments.

C.5 MEMORY PROFILING

In our memory profiling, we conform to the methodologies previously established in Malladi et al. (2023). Our implementation utilizes the default configuration provided by the `transformers` (Wolf et al., 2020) package. We do not turn on any advanced memory optimization techique such as gradient checkpointing. For multi-GPU backpropagation, we utilize the Fully Sharded Data Parallel (FSDP) (Fairscale, 2021) by PyTorch (Paszke et al., 2019). We use the Nvidia's `nvidia-smi` command to monitor the GPU memory usage. We report the maximum GPU memory consumption observed throughout all experiments.

Table 4: The hyper-parameter grids used for RoBERTa-large experiments. Addax and MeZO use a constant learning rate schedule, and Adam uses linear scheduling. Addax and Adam use 1K steps and MeZO use 100K steps. We check validation performance every 50 training steps and save the best for testing.

Experiment	Hyper-parameters	Values
16-bit/32-bit Addax	$K^0 + K^1$	64
	$\frac{K^1}{K^0+K^1}$	$\{0.1, 0.2, 0.3, 0.4, 0.5\}$
	Learning Rate η	$\{1\text{e}-5, 5\text{e}-5, 1\text{e}-4\}$
	ϵ	$1\text{e}-3$
	α	$\{3\text{e}-4, 1\text{e}-3, 3\text{e}-3, 4\text{e}-3,$ $5\text{e}-3, 7\text{e}-3, 1\text{e}-2, 1\text{e}-1\}$
MeZO	Batch size	64
	Learning Rate η	$\{1\text{e}-7, 1\text{e}-6, 1\text{e}-5\}$
	ϵ	$1\text{e}-3$
32-bit Adam	Batch size	$\{2, 4, 8\}$
	Learning Rate η	$\{1\text{e}-5, 3\text{e}-5, 5\text{e}-5\}$

Table 5: The hyper-parameter grids used for OPT-13B experiments. Addax, SGD, and MeZO use a constant learning rate schedule, and Adam uses linear scheduling. Adam uses 5 epochs. Addax, and SGD use 1K steps and MeZO 20K steps. We check validation performance every 50 training steps and save the best for testing.

Experiment	Hyper-parameters	Values
16-bit Addax	K^1	$\{2, 4, 6, 8, 10, 12, 14, 16\}$
	K^0	12
	Learning Rate η	$1\text{e}-4$
	ϵ	$1\text{e}-3$
	α	$\{1\text{e}-3, 3\text{e}-3, 5\text{e}-3, 7\text{e}-3, 9\text{e}-3\}$
MeZO	Batch size	16
	Learning Rate η	$\{1\text{e}-6, 1\text{e}-7\}$
	ϵ	$1\text{e}-3$
16-bit SGD	Batch size	$\{2, 4, 6, 8, 10, 12, 14, 16\}$
	Learning Rate η	$\{5\text{e}-3, 1\text{e}-2, 5\text{e}-2\}$
32-bit Adam	Batch size	8
	Learning Rate η	$\{1\text{e}-5, 5\text{e}-5, 8\text{e}-5\}$

D Additional experiment results

D.1 RoBERTa-large experiments main results

Table 6 reports the detailed numbers of the accuracy on the RoBERTa-large model across different fine-tuning methods that is shown in Figure 4. For the accuracy of MeZO and 32-bit Adam, we directly report the results from Malladi et al. (2023).

D.2 Investigation on the hyper-parameters of Addax

In this section, we explore the interplay between hyper-parameters, specifically reporting on the accuracy of both 32-bit and 16-bit Addax across various tasks utilizing the RoBERTa-large model in different combinations of hyper-parameters. We include the combinations of α and $\frac{K^1}{K^0+K^1}$ for 32-bit and 16-bit Addax in Figure 5 and Figure 6. Generally, it is observed that an increase in the ratio $\frac{K^1}{K^0+K^1}$ correlates with improved accuracy across tasks for both 16-bit and 32-bit Addax configurations, as evidence by the top row of the heatmaps for each task in Figure 5 and Figure 6.

Table 6: Experiments on RoBERTa-large (350M parameters). 16-bit Addax and 32-bit Addax outperform zero-shot and MeZO across the board on 6 tasks, while surpass Adam in four out six tasks. All experiements use prompts (Appendix C.2). For the accuracy of MeZO and 32-bit Adam, we report the results from Malladi et al. (2023).

Task Type	**SST-2** —sentiment—	**SST-5**	**SNLI** natural language inference	**MNLI**	**RTE**	**TREC** –topic–
Zero-shot	79.0	35.5	50.2	48.8	51.4	32.0
Samples per classes: $k = 16$						
MeZO	90.5	45.5	68.5	58.7	64.0	76.9
32-bit Addax	90.6	49.1	79.3	**69.9**	64.6	89.6
16-bit Addax	**91.4**	**50.4**	**79.3**	68.2	**67.2**	**90.8**
32-bit Adam	91.9	47.5	77.5	70.0	66.4	85.0

We did not identify a consistent trend for α across different tasks for both 16-bit and 32-bit Addax, suggesting that the optimal α could be task-specific.

D.3 OPT-13B EXPERIMENTS MAIN RESULTS

Table 1 reports the detailed numbers of the accuracy on the OPT-13B model across different fine-tuning methods that is shown in Figure 2. Details on batch size for different algorithms are also available in Table 1. For the accuracy of MeZO and 32-bit Adam, we report the results from Malladi et al. (2023). We also report GPU memory consumption across tasks and different fine-tuning methods for the OPT-13B model in Figure 3, with exact number reported in Table 1. See Appendix C.5 for memory profiling details.

D.4 THE EFFECT OF DIFFERENT K^1 ON THE PERFORMANCE OF 16-BIT ADDAX ON OPT-13B

This section examines the effects of adjusting first-order batch sizes, K^1, on the performance of 16-bit Addax with the OPT-13B model. We vary K^1 within the set $\{2, 4, 6, 8, 10, 12, 14, 16\}$ while maintaining $K^0 = 12$. The results are summarized in Table 7. We evaluate the maximum accuracy 16-bit Addax can attain on different tasks using a single A100 GPU (80GB) with the OPT-13B model. Notably, fine-tuning the OPT-13B model with Addax and a small first-order batch size K^1 results in high accuracy. This indicates that the zeroth-order gradient estimate in Addax offers stability and acts as gradient regularization when K^1 is small, thereby efficiently decreasing memory consumption. In scenarios where memory constraints are not a concern, Addax can utilize a greater number of first-order samples to improve performance. This reveals that Addax could offer a versatile trade-off between resource availability and performance (See Figure 1).

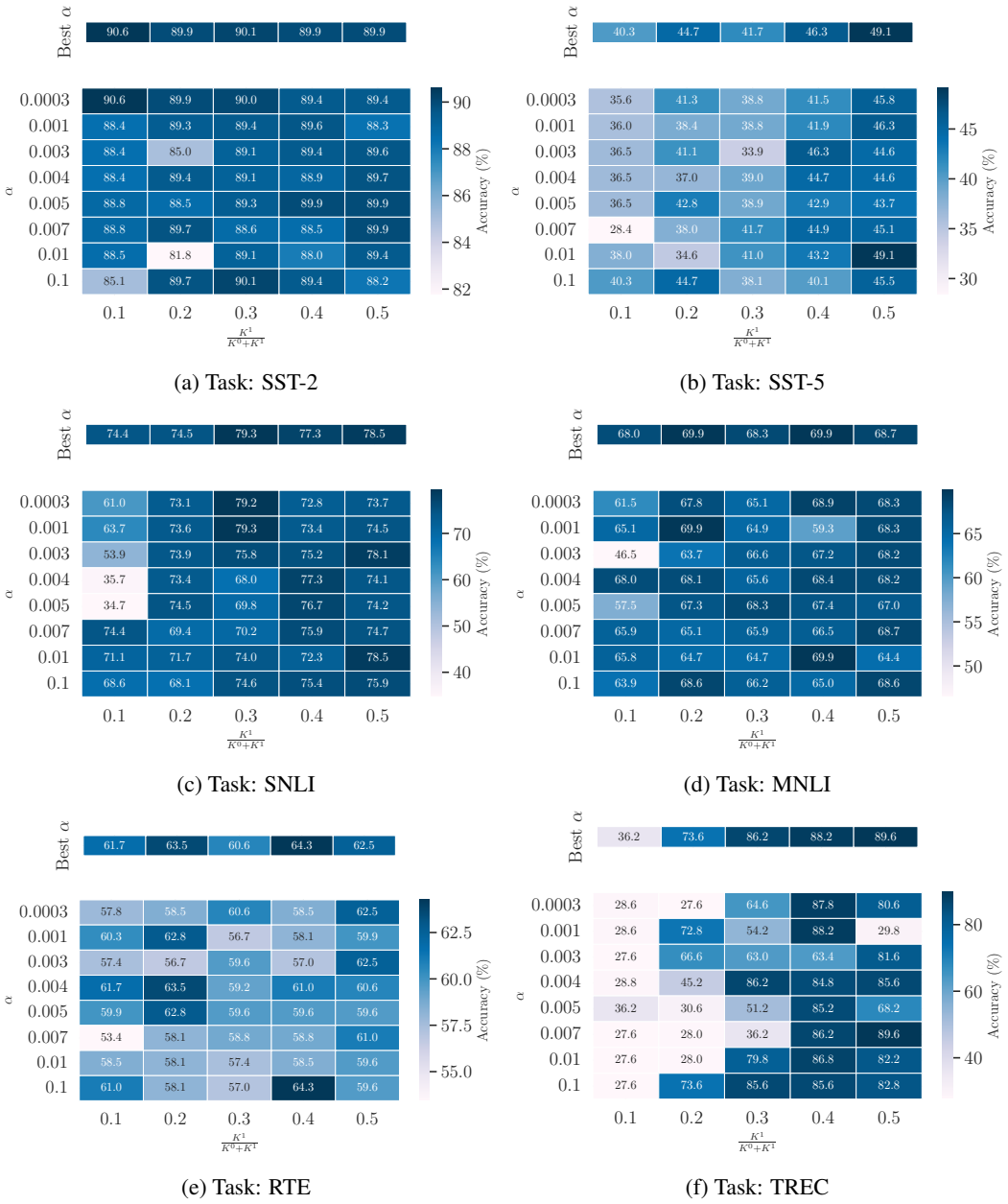
D.5 THE EFFECT OF DIFFERENT BATCH SIZE ON THE PERFORMANCE OF 16-BIT SGD ON OPT-13B

In this section, we demonstrate that, in general, larger batch sizes result in higher GPU memory consumption as well as improved accuracy for fine-tuning the OPT-13B model with 16-bit SGD. We evaluate the best accuracy that 16-bit SGD can achieve across different tasks under the constraints of a single A100 GPU (80GB) on the OPT-13B model. We searched three different learning rates $\eta = \{5\text{e}-3, 1\text{e}-2, 5\text{e}-2\}$ and evaluate the performance across different tasks using the different batch size of $\{2, 4, 6, 8, 10, 12, 14, 16\}$. The results are presented in Table 8. It is observed that for certain tasks, fine-tuning experiments with larger batch sizes encounter CUDA out of memory errors.

D.6 CONVERGENCE SPEED OF DIFFERENT TUNING METHODS ON THE OPT-13B MODEL

In this section, we demonstrate that 16-bit Addax reaches a convergence speed comparable to 16-bit SGD, despite SGD using $4\times$ more first-order samples for backward propagation. Meanwhile, Addax

[2]We fix $K^0 = 6$ and use two A100 GPUs for this dataset.



(a) Task: SST-2 (b) Task: SST-5

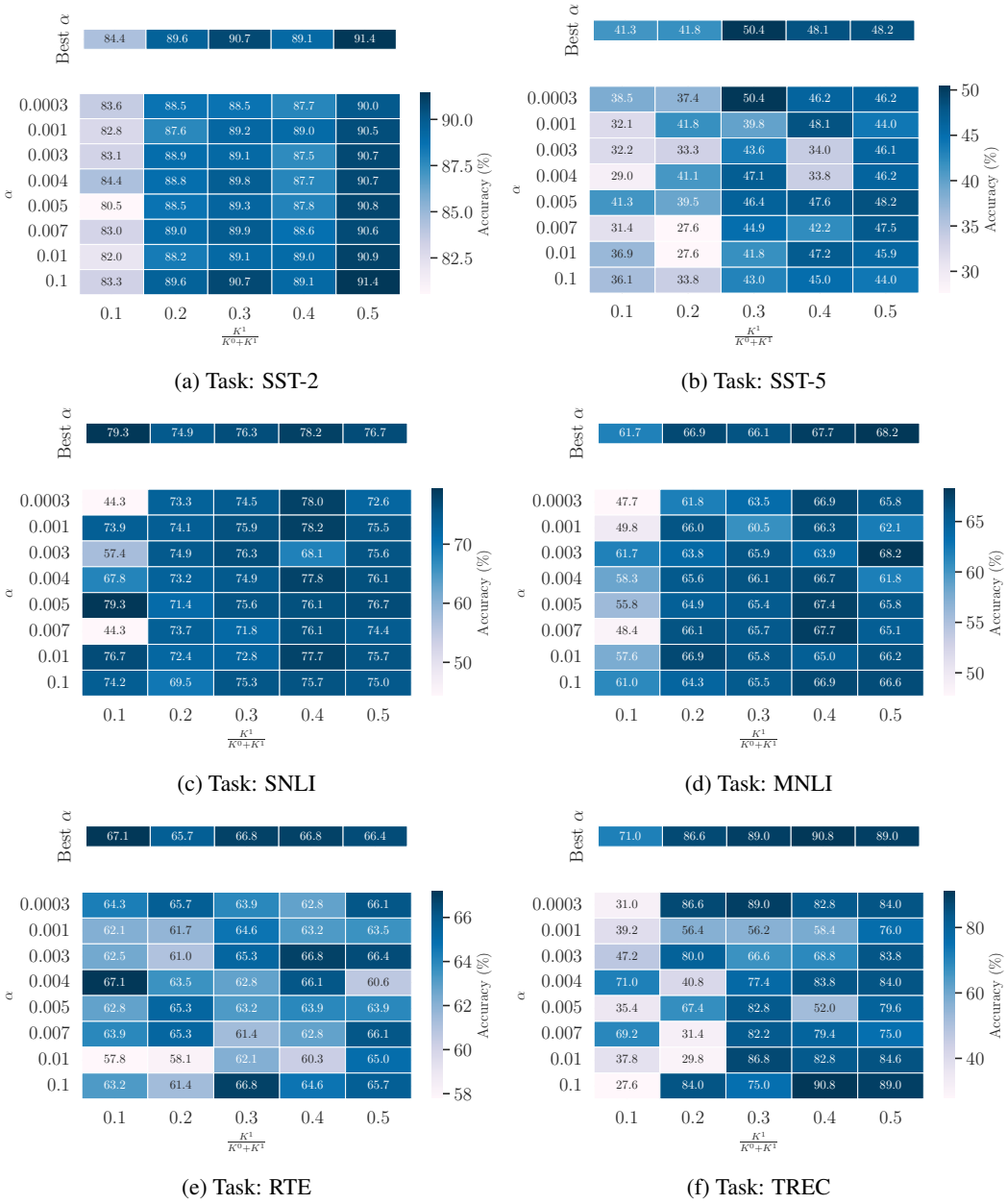
(c) Task: SNLI (d) Task: MNLI

(e) Task: RTE (f) Task: TREC

Figure 5: The accuracy (%) of the 32-bit Addax across different tasks on the RoBERTa-large model, with variable combinations of α and $\frac{K^1}{K^1+K^0}$.

requires only 1.04 to 2.1× more memory compared to MeZO. The comparison of convergence speeds across the three methods is illustrated in Figure 7. For MeZO and SGD, the batch size is set to 16, while for Addax, we configure (K^1, K^0) as $(4, 12)$. The learning rate for Addax is set at $\eta = 1\text{e}-4$. For SGD, the learning rates are $\eta = \{5\text{e}-3, 1\text{e}-2, 5\text{e}-2\}$. For MeZO, we utilize learning rates of $\eta = \{1\text{e}-6, 1\text{e}-7\}$. We select the hyper-parameters that yield the best validation accuracy across three methods. We utilize a single A100 GPU (80GB total) for running both Addax and MeZO, whereas SGD requires two A100 GPUs (160GB total). MeZO requires significantly more steps (20K steps) to converge compared to Addax and SGD (1K steps). Addax with smaller first-order batch size $K^1 = 4$ achieves a convergence speed similar to SGD with a batch size of 16, despite requiring significantly less memory.

[2]Running 16-bit SGD in task BoolQ with batch size of 16 will encounter CUDA out of memory error. We report the training loss of 16-bit SGD with batch size of 4 here.



(a) Task: SST-2 (b) Task: SST-5

(c) Task: SNLI (d) Task: MNLI

(e) Task: RTE (f) Task: TREC

Figure 6: The accuracy (%) of the 16-bit Addax across different tasks on the RoBERTa-large model, with variable combinations of α and $\frac{K^1}{K^1+K^0}$.

E PROOF OF THEOREM 3.1

Assumption E.1. $\mathcal{L}(\boldsymbol{\theta};x)$ is L-Lipschitz smooth, i.e.,

$$\|\nabla\mathcal{L}(\boldsymbol{\theta};x)-\nabla\mathcal{L}(\boldsymbol{\theta}';x)\|\leq L\left\|\boldsymbol{\theta}-\boldsymbol{\theta}'\right\|,\forall\boldsymbol{\theta},\boldsymbol{\theta}'\in\mathbb{R}^d,x\in\mathcal{D}.$$

Assumption E.2. The stochastic gradient is unbiased and has bounded variance, i.e.,

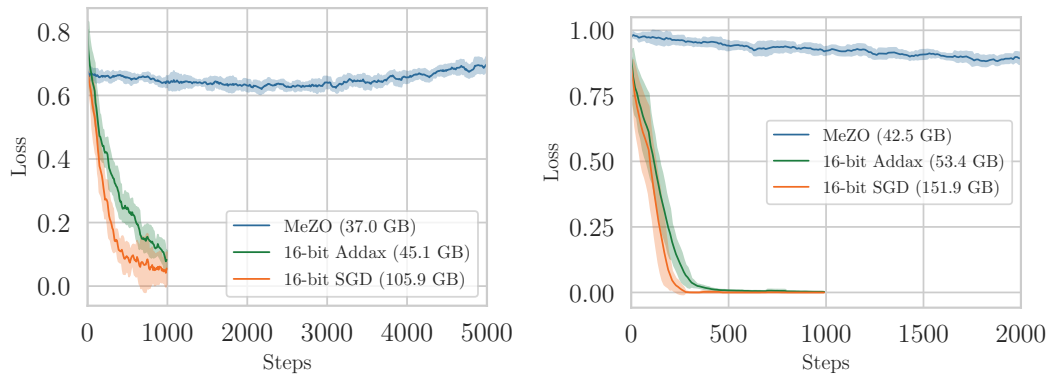
$$\mathbb{E}_x[\nabla\mathcal{L}(\boldsymbol{\theta};x)]=\nabla\mathcal{L}(\boldsymbol{\theta}),\mathbb{E}_x[\|\mathcal{L}(\boldsymbol{\theta};x)-\mathcal{L}(\boldsymbol{\theta})\|^2]\leq\sigma^2,\forall\boldsymbol{\theta}\in\mathbb{R}^d.$$

Lemma E.3 (Gao et al., 2014, Lemma 4.1 (b))**.** *Suppose Assumption E.1 holds, then the expected gradient estimated with SPSA is a biased estimation of $\nabla\mathcal{L}(\boldsymbol{\theta})$ and satisfies*

$$\left\|\mathbb{E}_{\mathcal{B}}[\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\mathcal{B})]-\nabla\mathcal{L}(\boldsymbol{\theta})\right\|^2\leq\frac{\epsilon^2L^2d^2}{4}.$$

Table 7: Fine-tuning results of 16-bit Addax using one A100 (80GB) with different K^1 on different tasks with the OPT-13B model. We fix K^0 of 16-bit Addax at 12. * means the fine-tuning task encounters CUDA out of memory error.

Task	Metric	Values							
	K^1	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
SST-2	GPU Memory (MB)	28617	29341	30985	32493	34207	36175	38799	40645
	Accuracy (%)	93.4	95.1	94.5	94.6	94.9	94.6	94.5	95.0
RTE	GPU Memory (MB)	35285	45117	53969	63663	73481	*	*	*
	Accuracy (%)	79.2	85.2	84.8	85.2	84.5	*	*	*
CB	GPU Memory (MB)	37757	53407	69671	79521	*	*	*	*
	Accuracy (%)	89.3	92.9	92.9	89.3	*	*	*	*
BoolQ	GPU Memory (MB)	45937	71171	*	*	*	*	*	*
	Accuracy (%)	81.6	83.0	*	*	*	*	*	*
WSC	GPU Memory (MB)	29391	30251	32415	34919	37269	38737	41009	43085
	Accuracy (%)	63.5	63.5	63.5	63.5	64.4	63.5	63.5	55.5
WIC	GPU Memory (MB)	28885	30253	32493	34789	37929	40551	42739	46871
	Accuracy (%)	67.4	66.5	68.7	70.7	68.3	69.0	70.5	68.0
MultiRC	GPU Memory (MB)	72885	*	*	*	*	*	*	*
	Accuracy (%)	77.0	*	*	*	*	*	*	*
COPA	GPU Memory (MB)	28411	28615	28597	28849	28811	28835	29003	28803
	Accuracy (%)	89.0	90.0	90.0	91.0	91.0	90.0	90.0	91.0
ReCoRD	GPU Memory (MB)	35823	46777	57567	70199	79527	*	*	*
	Accuracy (%)	81.1	81.9	81.7	82.1	81.5	*	*	*
SQuAD	GPU Memory (MB)	51131	78981	*	*	*	*	*	*
	F1 (%)	89.3	89.0	*	*	*	*	*	*
DROP[2]	GPU Memory (MB)	158180	*	*	*	*	*	*	*
	F1 (%)	34.7	*	*	*	*	*	*	*



(a) Task: RTE (b) Task: CB

Figure 7: Convergence speed of three fine-tuning methods (16-bit Addax, MeZO, and 16-bit SGD) on two fine-tuning datasets with the OPT-13B model. We set the batch size to 16 for MeZO and SGD and fix $(K^1, K^0) = (4, 12)$ for Addax. We utilize a single A100 GPU (80GB total) for running both Addax and MeZO, whereas SGD requires two A100 GPUs (160 GB total) to run with $BS = 16$. MeZO requires significantly more steps to converge compared to Addax and SGD. Addax with $4\times$ less first-order samples achieves a convergence speed similar to SGD, despite requiring significantly less memory.

Lemma E.4. *Suppose Assumption E.1 holds, then the variance of the gradient estimated with SPSA satisfies*

$$\mathrm{Var}(\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\mathcal{B})) = \mathbb{E}_{\mathcal{B}}\left[\left\|\mathbb{E}_{\mathcal{B}}[\hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\mathcal{B})] - \hat{\nabla}\mathcal{L}(\boldsymbol{\theta};\mathcal{B})\right\|^2\right] \leq \frac{d}{K}\sigma^2.$$

Table 8: Fine-tuning results of 16-bit SGD using one A100 (80GB) with different batch size on different tasks with the OPT-13B model. We can see that in general SGD achieves better performance with more memory and larger batch size. * means the fine-tuning task encounters CUDA out of memory error.

Task	Metric	Batch Size							
	Batch Size (BS)	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
SST-2	GPU Memory (MB)	51427	52113	52859	53115	53703	54671	54409	55771
	Accuracy (%)	93.5	93.5	94.8	94.6	94.4	94.7	94.2	94.2
RTE	GPU Memory (MB)	53901	62799	69189	79055	72389	*	*	*
	Accuracy (%)	79.8	80.8	81.2	84.5	83.8	*	*	*
CB	GPU Memory (MB)	56239	69359	77005	80595	*	*	*	*
	Accuracy (%)	91.0	89.3	92.8	92.8	*	*	*	*
BoolQ	GPU Memory (MB)	65387	70625	*	*	*	*	*	*
	Accuracy (%)	78.8	80.7	*	*	*	*	*	*
WSC	GPU Memory (MB)	52093	53391	54497	57211	58833	60719	62515	60413
	Accuracy (%)	63.4	63.4	63.5	63.5	63.5	65.4	63.5	63.5
WIC	GPU Memory (MB)	52297	53979	55689	58585	57185	58599	60513	61943
	Accuracy (%)	60.5	64.9	67.5	63.3	66.6	68.5	68.0	67.5
MultiRC	GPU Memory (MB)	79929	*	*	*	*	*	*	*
	Accuracy (%)	76.4	*	*	*	*	*	*	*
COPA	GPU Memory (MB)	51081	51367	51407	51885	52219	52615	52873	53157
	Accuracy (%)	86.0	79.0	79.0	90.0	81.0	89.0	90.0	85.0
ReCoRD	GPU Memory (MB)	55539	60063	69169	77007	79821	*	*	*
	Accuracy (%)	80.6	81.4	79.3	78.5	79.0	*	*	*
SQuAD	GPU Memory (MB)	60499	79275	*	*	*	*	*	*
	F1 (%)	87.7	89.4	*	*	*	*	*	*
DROP	GPU Memory (MB)	68964	*	*	*	*	*	*	*
	F1 (%)	30.2	*	*	*	*	*	*	*

Theorem E.5. *Under Assumptions E.1, E.2, by running Algorithm 1 for T iterations with $\epsilon \leq$, $\eta_t \leq \eta \leq, \forall t$, the output satisfies*

$$\begin{aligned}\mathbb{E}[\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\|^2] \leq & \frac{4(\mathcal{L}(\boldsymbol{\theta}_0)-\mathcal{L}_\star)}{\eta T(2-\alpha)} \\ & + \frac{\alpha(1+\alpha-\alpha^2/2)\epsilon^2 L^2 d^2}{2(2-\alpha)} + \frac{4\eta L}{(2-\alpha)}\left(\frac{(1-\alpha)^2}{2K^1}+\frac{\alpha^2 d}{2K^0}\right)\sigma^2.\end{aligned} \tag{2}$$

Proof: By Assumption E.1:

$$\begin{aligned}\mathbb{E}_t[\mathcal{L}(\boldsymbol{\theta}_{t+1})] \leq & \mathcal{L}(\boldsymbol{\theta}_t) + \mathbb{E}_t[\langle\nabla\mathcal{L}(\boldsymbol{\theta}_t), \boldsymbol{\theta}_{t+1}-\boldsymbol{\theta}_t\rangle] + \frac{L}{2}\mathbb{E}_t[\|\boldsymbol{\theta}_{t+1}-\boldsymbol{\theta}_t\|^2] \\ \overset{(a)}{=} & \mathcal{L}(\boldsymbol{\theta}_t) - \eta_t\left\langle\nabla\mathcal{L}(\boldsymbol{\theta}_t), (1-\alpha)\nabla\mathcal{L}(\boldsymbol{\theta}_t) + \alpha\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\rangle \\ & + \frac{L\eta_t^2}{2}\left\|(1-\alpha)\nabla\mathcal{L}(\boldsymbol{\theta}_t) + \alpha\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \\ & + \frac{L\eta_t^2(1-\alpha)^2}{2}\mathbb{E}_{\mathcal{B}^1}[\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t) - \nabla\mathcal{L}(\boldsymbol{\theta}_t;\mathcal{B}^1)\right\|^2] + \frac{L\eta_t^2\alpha^2}{2}\mathrm{Var}(\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)) \\ \overset{(b)}{\leq} & \mathcal{L}(\boldsymbol{\theta}_t) - (1-\alpha)\eta_t\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\|^2 - \alpha\eta_t\left\langle\nabla\mathcal{L}(\boldsymbol{\theta}_t), \mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\rangle \\ & + \frac{L\eta_t^2}{2}\left\|(1-\alpha)\nabla\mathcal{L}(\boldsymbol{\theta}_t) + \alpha\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \\ & + \frac{L\eta_t^2(1-\alpha)^2}{2K^1}\sigma^2 + \frac{L\eta_t^2\alpha^2 d}{2K^0}\sigma^2,\end{aligned} \tag{3}$$

where (a) substitutes the update of $\boldsymbol{\theta}$ and takes expectations to $\boldsymbol{g}^0, \boldsymbol{g}^1$;$(b)$ follows from the Lemma E.4. The third term on the Right-Hand-Side (RHS) can be further bounded as

$$\begin{aligned}
&-\alpha\eta_t \left\langle \nabla\mathcal{L}(\boldsymbol{\theta}_t), \mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)] \right\rangle \\
&\overset{(a)}{=} -\frac{\alpha\eta_t}{2}\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 - \frac{\alpha\eta_t}{2}\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 + \frac{\alpha\eta_t}{2}\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t) - \mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \\
&\overset{(b)}{\leq} -\frac{\alpha\eta_t}{2}\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 - \frac{\alpha\eta_t}{2}\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 + \frac{\alpha\eta_t\epsilon^2L^2d^2}{8},
\end{aligned} \tag{4}$$

where (a) uses the fact that $\|\mathbf{u}+\mathbf{v}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 + 2\langle\mathbf{u},\mathbf{v}\rangle$; (b) applies Lemma E.3 to the last term. The fourth term on the RHS of equation 3 can be bounded as

$$\begin{aligned}
&\frac{L\eta_t^2}{2}\left\|(1-\alpha)\nabla\mathcal{L}(\boldsymbol{\theta}_t) + \alpha\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \\
&= \frac{L\eta_t^2}{2}\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t) + \alpha\left(\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)] - \nabla\mathcal{L}(\boldsymbol{\theta}_t)\right)\right\|^2 \\
&\overset{(a)}{\leq} L\eta_t^2\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 + \alpha^2L\eta_t^2\left\|\left(\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)] - \nabla\mathcal{L}(\boldsymbol{\theta}_t)\right)\right\|^2 \\
&\overset{(b)}{\leq} L\eta_t^2\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 + \frac{\alpha^2\eta_t^2\epsilon^2L^3d^2}{4},
\end{aligned} \tag{5}$$

where (a) applies Cauchy-Schwarz inequality; (b) applies Lemma E.3 to the last term. Substitute equation 4, equation 5 back to equation 3, we have

$$\begin{aligned}
\mathbb{E}_t[\mathcal{L}(\boldsymbol{\theta}_{t+1})] \leq \mathcal{L}(\boldsymbol{\theta}_t) - (1 - \frac{\alpha}{2} - L\eta_t)\eta_t\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 - \frac{\alpha\eta_t}{2}\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \\
+ \frac{\alpha\eta_t\epsilon^2L^2d^2(1+2\alpha\eta_tL)}{8} + \frac{L\eta_t^2(1-\alpha)^2}{2K^1}\sigma^2 + \frac{L\eta_t^2\alpha^2}{2}\sigma^0.
\end{aligned} \tag{6}$$

Choose $\eta_t \leq \frac{2-\alpha}{4L}$, we have $1 - \frac{\alpha}{2} - L\eta_t \geq \frac{2-\alpha}{4} > 0$, $1 + 2\alpha\eta_tL \leq 1 + \alpha - \frac{\alpha^2}{2}$ and

$$\begin{aligned}
\frac{(2-\alpha)\eta_t}{4}\left\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\right\|^2 + \frac{\alpha\eta_t}{2}\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2 \leq \mathcal{L}(\boldsymbol{\theta}_t) - \mathbb{E}_t[\mathcal{L}(\boldsymbol{\theta}_{t+1})] \\
+ \frac{\alpha\eta_t\epsilon^2L^2d^2(1+\alpha-\alpha^2/2)}{8} + \frac{L\eta_t^2(1-\alpha)^2}{2K^1}\sigma^2 + \frac{L\eta_t^2\alpha^2}{2}\sigma^0.
\end{aligned} \tag{7}$$

Sum from $t = 0$ to T, we have

$$\begin{aligned}
\sum_{t=0}^{T}\left(\frac{(2-\alpha)\eta_t}{4}\mathbb{E}[\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\|^2] + \frac{\alpha\eta_t}{2}\mathbb{E}\left[\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2\right]\right) \leq \mathcal{L}(\boldsymbol{\theta}_0) - \mathbb{E}[\mathcal{L}(\boldsymbol{\theta}_{T+1})] \\
+ \sum_{t=0}^{T}\eta_t\cdot\frac{\alpha(1+\alpha-\alpha^2/2)\epsilon^2L^2d^2}{8} + \sum_{t=0}^{T}\eta_t^2\cdot\left(\frac{L(1-\alpha)^2}{2K^1}\sigma^2 + \frac{L\alpha^2d}{2K^0}\sigma^2\right).
\end{aligned} \tag{8}$$

Choose $\eta_t = \eta \leq \frac{2-\alpha}{4L}, \forall t$, and divide both side by $\frac{(2-\alpha)\eta T}{4}$, we have

$$\begin{aligned}
\mathbb{E}[\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\|^2] + \frac{2\alpha}{2-\alpha}\mathbb{E}\left[\left\|\mathbb{E}_{\mathcal{B}^0}[\hat{\nabla}\mathcal{L}(\theta_t;\mathcal{B}^0)]\right\|^2\right] \leq \frac{4(\mathcal{L}(\boldsymbol{\theta}_0) - \mathcal{L}_\star)}{\eta T(2-\alpha)} \\
+ \frac{\alpha(1+\alpha-\alpha^2/2)\epsilon^2L^2d^2}{2(2-\alpha)} + \frac{4\eta L}{(2-\alpha)}\left(\frac{(1-\alpha)^2}{2K^1} + \frac{\alpha^2d}{2K^0}\right)\sigma^2,
\end{aligned} \tag{9}$$

which completes the proof.

Corollary E.6. *By choosing* $\eta = \min\left\{\frac{2-\alpha}{4L}, \sqrt{\frac{2(\mathcal{L}(\boldsymbol{\theta}_0)-\mathcal{L}_\star)}{TL\sigma^2\left(\frac{(1-\alpha)^2}{K^1}+\frac{\alpha^2d}{K^0}\right)}}\right\}$ *and*

$$\epsilon \leq \left(\frac{2\mathcal{L}(\boldsymbol{\theta}_0) - \mathcal{L}_\star)\sigma^2\left((1-\alpha)^2/K^1 + \alpha^2d/K^0\right)}{T}\right)^{1/4}\cdot\frac{1}{L^{3/4}d\sqrt{\alpha(1+\alpha-\alpha^2/2)}},$$

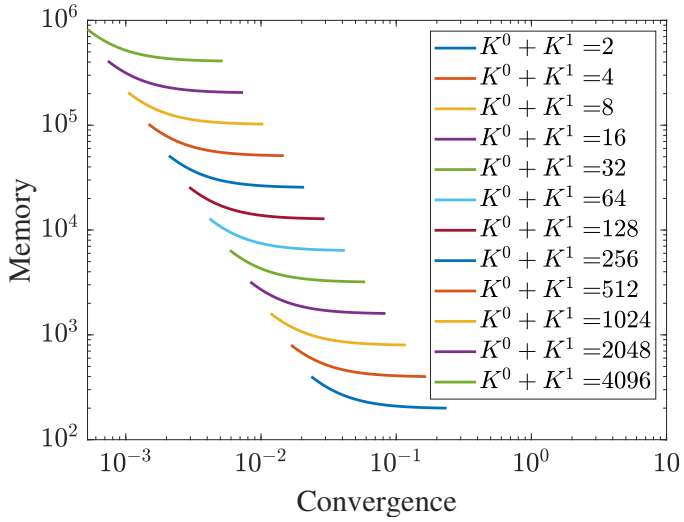


Figure 8: An illustration for the memory-convergence trade-off of Addax. The x-axis represents the expected gradient size after a fixed number of iterations.

Algorithm 1 converges with rate

$$\mathbb{E}[\|\nabla\mathcal{L}(\boldsymbol{\theta}_t)\|^2] \leq 5\sqrt{2L} \cdot \frac{\sqrt{\frac{(1-\alpha)^2}{K^1} + \frac{\alpha^2 d}{K^0}}}{2-\alpha} \cdot \sigma\sqrt{\frac{\mathcal{L}(\boldsymbol{\theta}_0) - \mathcal{L}_\star}{T}}$$
$$= \mathcal{O}\left(\frac{1}{\sqrt{T}} \cdot \sqrt{\frac{(1-\alpha)^2}{K^1} + \frac{\alpha^2 d}{K^0}}\right)$$

Remark 1. [Trade-off between convergence and memory] We obtain a trade-off between the convergence speed and the memory cost of Algorithm 1 as follows: For a model with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, the memory cost for estimating a zeroth-order gradient on one sample with SPSA is $\approx d$, and for estimating a first-order gradient on one sample is $\approx 2d$. Then we have the following trade-off shown in Figure 8