OPT2024: 16th Annual Workshop on Optimization for Machine Learning

BlockLLM: Memory-Efficient Adaptation of LLMs by Selecting and Optimizing the Right Coordinate Blocks

Amrutha Varshini Ramesh AVRAMESH@CS.UBC.CA
University of British Columbia

Vignesh Ganapathiraman VIGNESH.GANAPATHIRAMAN@YAHOOINC.COM
Yahoo! Research

Issam H. Laradji ISSAM@SERVICENOW.ORG
ServiceNow Research

Mark Schmidt SCHMIDTM@CS.UBC.CA
University of British Columbia, Canada CIFAR AI Chair (Amii)

Abstract

Training large language models (LLMs) for pretraining or adapting to new tasks and domains has become increasingly critical as their applications expand. However, as model and data sizes grow, the training process presents significant memory challenges, often requiring a prohibitive amount of GPU memory that may not be readily available. Existing methods such as low-rank adaptation (LoRA) add trainable low-rank matrix factorizations, altering the training dynamics and limiting the model's parameter search to a low-rank subspace. GaLore, a more recent method, employs Gradient Low-Rank Projection to reduce the memory footprint, in the full parameter training setting. However GaLore can only be applied to a subset of the LLM layers that satisfy the "reversibility" property, thus limiting their applicability. In response to these challenges, we introduce BlockLLM, an approach inspired by block coordinate descent. Our method carefully selects and updates a very small subset of the trainable parameters without altering any part of its architecture and training procedure. BlockLLM achieves state-of-the-art performance in both finetuning and pretraining tasks, while reducing the memory footprint of the underlying optimization process. Our experiments demonstrate that fine-tuning with less than 5% of the parameters, BlockLLM achieves state-of-the-art perplexity scores on the GLUE benchmarks. On a Llama model pretrained on the C4 dataset, BlockLLM is able to train with significantly less memory than the state-of-the-art, while still maintaining competitive performance.

1. Introduction

Pretraining and finetuning LLMs are resource-intensive processes, requiring substantial memory and computational power. For example, a $7B$ parameter Llama model requires approximately 14GB of memory for storing the weights, with an additional 14GB for gradients during backpropagation [27]. When using the Adam optimizer [9], which maintains first and second moment estimates, the memory requirement doubles, totaling around 56GB of VRAM. As models grow larger, this memory burden will continue to escalate, restricting large-scale LLM training to only researchers and organizations with the most advanced GPUs such as A100s or higher. This is a significant barrier to entry for researchers and practitioners who do not have access to such high-end hardware.

Existing strategies for memory-efficient training. To address these challenges, multiple strategies are being explored to reduce the number of parameters, gradients, and the corresponding optimizer state size. One popular strategy is *pruning*, where a large set of parameters or even entire

layers are removed from the model architecture [12, 22, 25]. However, pruning approaches often require extensive retraining to recover lost accuracy [2]. Furthermore, identifying which parameters are crucial before training is challenging [14, 20]. This challenge complicates implementation and can lead to generalization issues, particularly on diverse or unseen data [12].

PEFT (*Parameter-Efficient-Fine-Tuning*) methods [6, 7, 10] achieve memory efficiency by introducing low-rank matrices to the transformer architecture, significantly reducing the number of trainable parameters needed during fine-tuning. Although integrating these low-rank matrices alleviates the extensive retraining demanded by pruning techniques, they can alter the training dynamics and potentially lead to quality issues during the merging phase [5]. The low-rank assumption may also constrain the model's expressiveness, limiting its ability to fully capture complex patterns in the data. Furthermore, the additional parameters introduced by PEFT methods can increase the model's parameter size, countering efforts to reduce overall model size.

A recent work, *GaLore* [27], which focuses on full parameter training, achieves memory efficiency by performing low-rank factorization of the gradients in specific layers. However, GaLore does not achieve high memory efficiency across all model types, as its gradient factorization method can only be applied to layers that satisfy the reversibility property. This limitation restricts its applicability and efficiency in models where not all layers exhibit this property.

Techniques such as gradient and activation checkpointing [1], quantization [4], and parameter offloading [19] are also commonly used to achieve memory savings. However, these methods often come with its own trade-offs. For instance, checkpointing [1] reduces memory usage but requires recomputation, quantization lowers precision and can affect accuracy [4], and parameter offloading increases data transfer latency [19]. While these methods have their own limitations, many of these techniques are complementary to the approach presented in this work and provide additional opportunities for memory reduction when used in combination.

Block coordinate descent (BCD). BCD is popular algorithm in the large-scale optimization literature. At any training iteration t, only a subset or block of parameters b_t is updated, rather than the entire parameter set W. The block b_t can be chosen in several ways. In this work, we use greedy BCD, where the block b_t is selected greedily based on a function of the gradient. This selection strategy has been theoretically demonstrated to yield superior performance compared to random block selection [8, 15, 16, 18]. Building on these results, our approach leverages greedy block selection to enhance parameter and memory efficiency in LLM training. Additionally, the convergence of BCD has been theoretically proven on various problem architectures in the optimization literature [16, 18, 26], further inspiring us to adapt it to large-scale LLM training.

While training with BCD, b_t does not stay fixed across iterations. As a result, BCD does not constrain model performance, which is often the case with low-rank approximation methods. Moreover, BCD doesn't alter the model architecture in any way and preserves its structure throughout the training process. By preserving the model's architecture and reducing the active parameter set per iteration, BCD aligns naturally with reduced parameter training regimes, forming the cornerstone of our proposed BlockLLM framework.

Our Contributions. We propose a novel parameter and memory-efficient algorithm, **BlockLLM** where we dynamically select and train a block of parameters. BlockLLM primarily achieves memory savings by storing optimizer states in the VRAM only for these selected parameters. We demonstrate the superior performance of BlockLLM on finetuning and pretraining tasks. We also conduct thorough ablations studies on several aspects of BlockLLM.

2. BlockLLM methodology

In this section we introduce BlockLLM, a parameter and memory efficient training method designed to reduce the number of trainable parameters in large language models (LLMs) without compromising training performance. Akin to other parameter-efficient fine-tuning (PEFT) methods such as LoRA [6] and ReLoRA [10], BlockLLM updates only k parameters at any iteration t. Here $k \ll z$ where z is the total number of parameters. However, the main difference is that BlockLLM optimizes parameter selection by focusing on the most impactful parameters at different stages of the training process. The overall algorithm of BlockLLM is provided in Algorithm 1 in the Appendix.

Parameter Selection Criteria. In the context of LLMs, at iteration t the update to the parameter is the processed gradient $\tilde{G}_t$, calculated using an optimizer such as Adam [9]. Specifically, for any layer l of the model, the update is given by $\tilde{G}_t^l = M_t^l / \sqrt{V_t^l + \epsilon}$, where all the operations are applied element-wise. Here, $M_t^l = \beta_1 M_{t-1}^l + (1-\beta_1) G_t^l$ and $V_t^l = \beta_2 V_{t-1}^l + (1-\beta_2) {G_t^l}^2$. Here, β_1 and β_2 are hyperparameters of the optimizer. $G_t^l \in \mathbb{R}^{p \times q}$ is the gradient at layer l. M_t and V_t denote the bias-corrected first moment estimate and bias-corrected second moment estimate respectively.

BlockLLM achieves memory savings by storing these optimizer states M_t and V_t only for the currently selected layers, rather than for all parameters in the model. When the set of selected layers changes, the optimizer is reinitialized with these new layers. This means it no longer maintains the optimizer states for the previously selected layers, similar to methods such as ReLoRA [10]. As an alternative, we also tried to offload M_t and V_t of the selected parameters to CPU and re-load them as needed. But that did not improve the model performance. Thus we decided to adopt the former strategy to avoid the offloading operation in the interest of faster training.

Now, the parameter selection in BlockLLM relies on $\|\tilde{G}_l\|$ for each layer $l \in L$, which requires computing $\tilde{G}_l$ for all layers. This operation consumes significant memory required to store all the gradients. To reduce this, we sample a small number of p additional layers per iteration besides the current block. $|\tilde{G}_l|/f_l$ is computed for these p layers and their norms stored in a dictionary H, which is then used for efficient parameter selection.

During the backward pass, BlockLLM selects the layers with large values in H and updates only those layers. These selected layers are denoted as the set S. Note that by selecting full layers, we may not achieve the desired sparsity level s. Therefore, for each selected layer we construct a binary mask to retain only the top k parameters by gradient magnitude:

$$\text{mask}[i,j] = \begin{cases} 1 & \text{if } |\tilde{G}_t^l[i,j]| \geq \tau \\ 0 & \text{otherwise,} \end{cases}$$

where τ is an estimated threshold, computed by looking at the gradient values of each layer. Specifically, τ is obtained computing the $(1-\zeta)^{th}$ percentile in $\tilde{G}_t^l$. The value of ζ is defined as $\zeta = (\Sigma_p - n_s)/n_s$ (refer to Algorithm 2 for definitions of Σ_p and n_s). Then, in every iteration t, the selected parameters in layers $l \in S$ are updated using the computed masks. The update rule is given by $W_{t+1}^l = W_t^l - \eta \left(\text{mask} \odot \tilde{G}_t^l\right)$, where η is the learning rate. An illustration of the proposed parameter selection procedure is given in Figure 1.

However, there is one caveat with this approach. In the initial training iterations, the gradient estimates are known to be noisy. Additionally, in cases such as pretraining and finetuning with significant domain shifts, there is often very little useful inductive bias. Therefore, using gradients to select important parameters may prove to be detrimental to our cause in the initial few iterations.

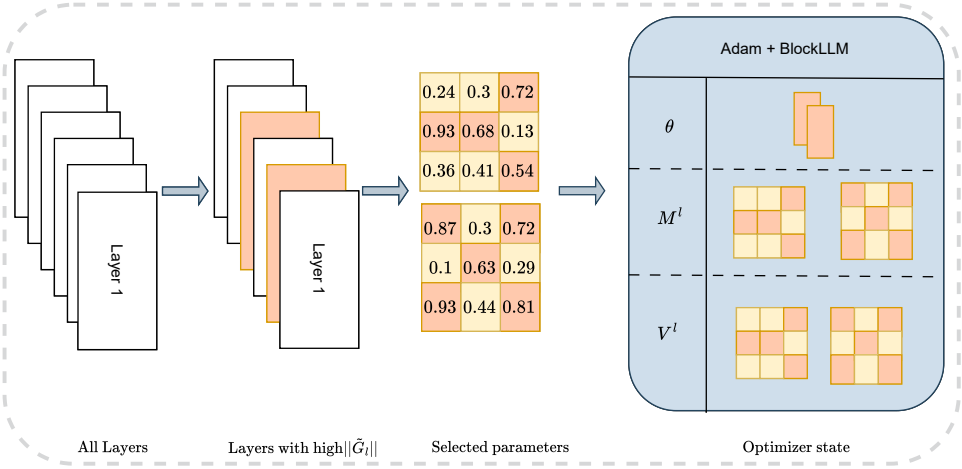

Figure 1: Given a large language model consisting of many layers, BlockLLM first finds the layers with the largest gradient $||\tilde{G}_l||$ (highlighted in orange) and selects a subset of the parameters. During optimization, only the selected layers will be updated (θ) and the Adam optimizer will only keep track of the bias corrected moments (M_l) and (V_l) for those selected parameters (shown in orange).

To address this challenge, our experiments incorporate *layer visit frequency* f into the selection criteria. Specifically, for any layer $l \in L = \{1, 2, \ldots, n\}$, then f_l represents the sum normalized number of times the layer has been selected. That is,

$$S_t^l = \begin{cases} 1 & \text{if layer } l \text{ is selected at time } t \\ 0 & \text{otherwise.} \end{cases}$$

The *layer visit frequency* f_l for layer l after T time steps is given by $f_l = \frac{1}{T}\sum_{t=1}^{T} S_t^l$. Consequently, the layer selection criterion is modified to $|\tilde{G}_l|/f_l$. This modification favors layers with high gradient norms while also giving priority to layers that have been selected less frequently in previous iterations. Our experiments show that this refined selection criterion enhances performance.

Parameter Selection Frequency. The natural next question is how many iterations to perform the update (1) with the same set S. BlockLLM addresses this by using the loss ϕ as a critical signal for determining when to change the parameter selection. Specifically, BlockLLM introduces a hyperparameter, patience m. At any iteration t, if ϕ_t equals to or exceeds the moving average of losses over the last m iterations, the set S is revised. The detailed parameter selection frequency algorithm is provided in Algorithm 2 and the ablation for the hyperparameter m is provided in the section E.4 in the Appendix.

3. Experiments

We evaluated BlockLLM on both finetuning tasks and pretraining tasks.[1] The finetuning experiments were conducted in Tesla V100 gpu (32 GB) and the pretraining tasks were conducted on NVIDIA A40 (48 GB) and A100 GPUs (80 GB) (one GPU per experiment).

1. All our experiments were based on the code released by [27]. We thank the authors for releasing their code and clearly documenting their experiment setup in the paper.

3.1. Finetuning on GLUE

We benchmarked the performance of BlockLLM against GaLore [27] and full finetuning (FFT) using the pretrained RoBERTa model [11] on GLUE tasks [24]. For BlockLLM, we conducted hyperparameter tuning for the learning rate, and our experimental setup is detailed in E.2. For GaLore, we used the learning rate specified in their original work [27]. In our experiments, we used $s = 0.95$. We evaluated both performance and memory consumption during the training process for all methods. The results, presented in Tables 1 and 2, indicate that BlockLLM outperforms the other models in all tasks while achieving approximately a 13.5% reduction in memory usage on average.

	MRPC	COLA	STS-B	RTE	SST2	MNLI	QNLI	QQP
Block-LLM	**3.97**	**2.8**	**3.48**	**9.1**	**3.6**	**13.7**	**12.8**	8.36
GaLore (rank=8)	4.52	4.2	4.8	9.7	3.87	15.1	14.8	8.43
GaLore (rank=4)	4.52	4.2	4.8	9.7	3.86	15.2	14.8	**8.03**
FFT	4.24	3.67	4.4	10.28	3.82	15.53	15.02	9.22

Table 1: VRAM Memory Comparison Across Different Tasks (measured in GB). VRAM memory usage was monitored as described in Section E.1.

	MRPC	COLA	STS-B	RTE	SST2	MNLI	QNLI	QQP
Block-LLM	91.8	**63.8**	90.02	80.14	**94.95**	**87.75**	92.95	91.36
GaLore (rank=8)	89.96	62.5	**91.1**	79.78	94.38	87.17	**92.97**	91.11
GaLore (rank=4)	91.7	61.67	91.09	79.78	94.04	87	92.65	91.06
FFT	**92.36**	62.84	91.1	**80.5**	94.57	87.18	92.33	**92.28**

Table 2: Score Comparison Across Different GLUE Tasks

3.2. Pre-training on Llama model

We also compared BlockLLM with GaLore [27] in pretraining LLaMA-based large language models [23] on the C4 (Colossal Clean Crawled Corpus) dataset [17]. Our experiment setup is similar to GaLore [27], following the setup from ReLoRA[10]. We ran BlockLLM with the experimental setup described in E.3 and computed the perplexity scores from final evaluation loss and maximum memory usage in GB(Gigabytes). The perplexity and memory are shown in Figure 4 and Table 3.

	60M		130M		350M	
	Perplexity	Memory	Perplexity	Memory	Perplexity	Memory
BlockLLM	**34.31**	**28.27**	**25.36**	**40.68**	19.02	**42.6**
GaLore	34.88	32.26	**25.36**	46.69	**18.95**	49.06

Table 3: Comparison of Accuracy and VRAM Memory (GB) of LLama Models with GaLore. Note that for GaLore, we used AdamW as the underlying optimizer.

We note here that BlockLLM takes a little bit longer to converge for some experiments ($130m$ and $350m$ models) compared to full parameter learning methods such as GaLore in the pretraining experiment. We suspect this is due to the fact that we are dealing with noisy gradients in the earlier iterations of training. However, BlockLLM converges to the state-of-the-art perplexity score in a few more iterations compared to GaLore. This is issue is not present in the finetuning experiments.

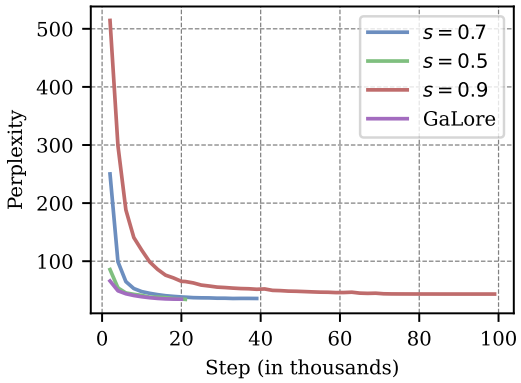

Figure 2: Perplexity of Models

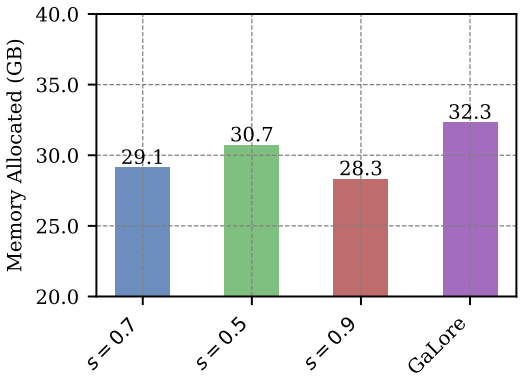

Figure 3: Memory usage

Figure 4: Comparison of perplexity (left) and memory usage (right) of Llama 60M. Here s denotes the specified model sparsity. As it can be seen, BlockLLM performs competitively with GaLore, but at a much lower memory footprint.

Effect of sparsity s. Here, we compare BlockLLM with sparsity values $s = \{0.5, 0.7, 0.9\}$ against full parameter training $(s = 0)$ and GaLore [27]. The results are in Figure 4. We observe that with $s = 0.5$, BlockLLM consumes about 1.5 GB less memory than Galore and higher sparsity values further reduce memory usage though this comes with the trade-off of requiring more training iterations for similar performance. This means that in a memory-constrained training setup, BlockLLM can be used with higher sparsity without compromising model performance. The trade-off is longer training, but this flexibility can be a game-changer for optimizing resource usage. In order to achieve speeds comparable or better than the baselines, we found that a sparsity value of 0.75 or higher was better for finetuning and sparsity of $0.5 - 0.7$ was optimal for pre-training experiments.

4. Conclusion & Future work

In this paper we introduced BlockLLM, a novel method for efficiently training large language models. By dynamically estimating and updating the importance of parameters during training, BlockLLM effectively achieves state-of-the-art performance while significantly reducing the memory footprint. Our method achieves the highest validation accuracy on GLUE finetuning tasks, sometimes even surpassing full finetuning. One key aspect of BlockLLM is that it does not presuppose the importance of layers but continuously evaluates and updates parameter importance throughout training. This adaptive approach allows for more flexible and efficient optimization compared to methods that assume certain parameters are critical from the outset. Additionally, BlockLLM preserves the original architecture without altering the model structure or restricting the parameter search space, making it suitable for various LLMs and tasks.

5. Acknowledgments

This research was partially supported by the Canada CIFAR AI Chair Program and the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants RGPIN-2022-03669.

References

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[2] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[3] Manas Gupta, Efe Camci, Vishandi Rudy Keneta, Abhishek Vaidyanathan, Ritwik Kanodia, Chuan-Sheng Foo, Wu Min, and Lin Jie. Is complexity required for neural network pruning? a case study on global magnitude pruning. *ArXiv*, abs/2209.14624, 2022. URL https://api.semanticscholar.org/CorpusID:252595918.

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

[5] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*, 2021.

[6] 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.

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

[8] Sai Praneeth Karimireddy, Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. Efficient greedy coordinate descent for composite problems. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2887–2896. PMLR, 2019.

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

[10] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

[11] 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.

[12] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.

[13] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

[14] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[15] Julie Nutini, Mark Schmidt, Issam Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning*, pages 1632–1641. PMLR, 2015.

[16] Julie Nutini, Issam Laradji, and Mark Schmidt. Let's make block coordinate descent converge faster: faster greedy rules, message-passing, active-set complexity, and superlinear convergence. *Journal of Machine Learning Research*, 23(131):1–74, 2022.

[17] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[18] Amrutha Varshini Ramesh, Aaron Mishkin, Mark Schmidt, Yihan Zhou, Jonathan Wilder Lavington, and Jennifer She. Analyzing and improving greedy 2-coordinate updates for equality-constrained optimization via steepest descent in the 1-norm. *arXiv preprint arXiv:2307.01169*, 2023.

[19] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[20] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. On the effect of dropping layers of pre-trained transformer models. *Computer Speech & Language*, 77:101429, 2023.

[21] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[22] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

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

[24] 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.

[25] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732*, 2019.

[26] Jinshan Zeng, Tim Tsz-Kit Lau, Shaobo Lin, and Yuan Yao. Global convergence of block coordinate descent in deep learning. In *International conference on machine learning*, pages 7313–7323. PMLR, 2019.

[27] 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.

APPENDIX

Appendix A. Analysis of parameter importance

The criteria for selecting the "right" subset of parameters during the training is not immediately clear. In the section A, we look at magnitude pruning as a tool to identify parameter importance, in the finetuning setting. Magnitude pruning is a widely recognized technique for reducing the parameter count in neural network models [3]. In this analysis, we use parameter magnitude as a measure of parameter importance and study the impact of training on the selected parameters. First, we trained DistilBERT [21] on the IMDb dataset [13] for sequence classification, achieving an accuracy of 92.02%. We then conducted inference on the GLUE-CoLA dataset [24] without fine-tuning, resulting in a significant drop in accuracy to 47.74%. This drop in performance, possibly due to domain shift, encouraged us to use this setup for our analysis.

Next, we performed magnitude pruning on the IMDb pre-trained model at various sparsity levels and fine-tuned these pruned models on the GLUE-CoLA dataset. Let s denote the sparsity level, W^t represent the model parameters at iteration t, and n be the total number of parameters. For each parameter w_i where $i = 1, \ldots, n$, we compute $|w_i|$. During training, we update only $S = \text{Top}_k|W^0|$, where $k = n \times (1 - s)$. The results of these experiments, detailing the relationship between sparsity and accuracy, are summarized in Table 4.

A.1. Sparsity Accuracy Trade-off

We performed magnitude pruning on the IMDb pre-trained model [13] weights at various sparsity levels and fine-tuned these pruned models on the GLUE-CoLA dataset [24]. The results of these experiments, detailing the relationship between sparsity and accuracy, are summarized in Table 4.

Sparsity	Accuracy
0.0	79.57
0.5	78.52
0.6	74.2
0.7	67.68
0.8	69.12
0.9	69.12

Table 4: Performance of pruned models at various sparsity levels on GLUE-CoLA dataset.

The accuracy generally declines with increasing sparsity. At 0.5 sparsity, the performance remains relatively high at 78.52%, close to the non-pruned model's 79.57%. However, accuracy drops more significantly to 67.68% at 0.7 sparsity. Interestingly, at sparsity levels of 0.8 and 0.9, accuracy stabilizes around 69.12%.

Interesting results emerged from this analysis: at 0.5 sparsity, the model retains a high accuracy of 78.52%, suggesting that up to 50% of parameters can be pruned with minimal performance loss. However, accuracy drops significantly at 0.7 sparsity to 67.68%. *This suggests that there is some*

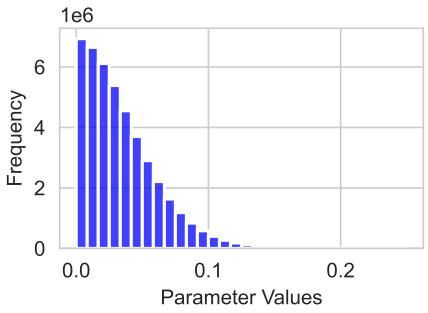

Figure 5: Histogram of the changed parameters

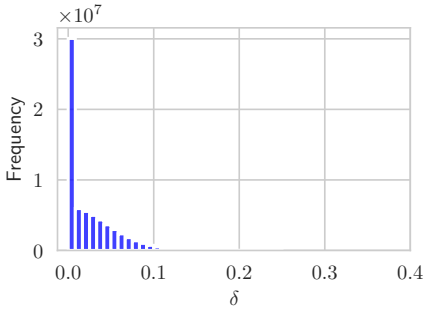

Figure 6: Histogram of δ

Figure 7: Analysis of weight magnitudes of DistilBERT model pretrained on IMDb dataset and finetuned on CoLA dataset.

inductive bias that the model can leverage when finetuning on a dataset with significant domain shift, under a reduced parameter setting. However, estimating the sparsity s apriori is hard, and its not clear what factors influence this.

Analysis of Weight Magnitudes. To further understand the effects of this parameter selection strategy, we analyzed $|W|$, before and after training to understand which weights are updated more frequently and their impact on model performance. Specifically, we compared the initial weights $|W^0|$ and final weights $|W^t|$ of the model after fine-tuning on the CoLA dataset [24]. Our findings are presented in Figure 7. The histogram on the left of Figure 7 shows $|w_i^t|$ for all i where $\delta_i > \eta$. Here, $\delta_i = |w_i^0 - w_i^t|$ and η is the threshold. The histogram on the right of Figure 7 plots the frequency of δ in the updates, revealing that a large percentage of the updates were minor.

Several additional insights and questions emerge from these observations. The histogram of δ indicates that most weight changes are minor, suggesting that significant updates are concentrated in a small portion of the weights. These experiments raise some pertinent questions:

1. Is it reasonable to judge the impact of a parameter during training based on its initial weight magnitude?
2. How should the appropriate sparsity level s be determined before commencing training?
3. Although we observed that only a few parameters undergo significant changes during training, which specific parameters are updated during various phases of the training process?

A.2. Analysis of Reduced Parameter Training

To address the above questions, we further investigated the pruning process. First, we updated the chosen parameter set S every m iterations, based on the weight magnitudes $|W^t|$, at current iteration t. This means that after every m, the parameter seletion criteria is revisited to obtain a new set of parameters to update. The objective is to understand if S changes significantly over time and if adaptively selecting S enhances the training performance. In this framework, we continue

to update only the top $k = n \times (1 - s)$ parameters in each iteration. However, the percentage of unique parameters q, updated throughout the entire training process can be greater than $(1 - s)$ of $\%$ parameters updated. (i.e $q \geq 1 - s$). Analysing q can reveal how much parameters are truly impactful for training, thereby addressing our second question.

We conducted this experiment using the same setup: fine-tuning the DistilBERT model on the GLUE-CoLA dataset after pre-training on the IMDb dataset. The results of this experiment are summarized in the Table 5. Additionally, we performed similar experiments on other GLUE [24] datasets, with the results presented in Section B.2

$1 - s$	q	m	**Accuracy**	**Matthews Correlation**
0.1	0.58	1000	82.55	0.5882
0.02	0.36	1000	82.45	0.5711
0.02	0.14	4000	82.45	0.5794
0.02	0.10	6000	82.07	0.5679

Table 5: Impact of Update Frequency and Sparsity on CoLA Dataset

Our observations indicate that s decreases, meaning more parameters are updated per iteration, the number of unique parameters q also increases, which results in better model performance. *The increasing number of unique updates indicates that different parameters may become important at different stages of training, necessitating a more dynamic approach to parameter updates.*

We also evaluated the model with varying update frequencies m. Naturally, as m increases, the model has fewer opportunities to update a larger number of parameters, leading to a decrease in q. Interestingly, there is an optimal point up to which performance either improves or remains stable, beyond which it begins to decline. This suggests that updating too many parameters or too few parameters can be detrimental to the training process. *These findings highlight the need for adaptive methods to determine the choice of parameters and the update frequency.*

The findings from the above analysis suggest that a parameter-efficient training method that updates a small set of parameters each iteration is feasible. However, some key aspects require clarification:

- **Parameter Selection Criteria**: Our analysis in Figure 7 indicates that the model frequently updates parameters with lower weight magnitudes. This contradicts the very premise of magnitude pruning. Therefore its not clear if adopting magnitude as a parameter importance criteria will help, in general. In our work, we bank on evidence from prior work on greedy parameter selection strategy [16, 18] and use gradient as the parameter importance criteria.
- **Parameter Selection Frequency**: As discussed earlier, determining when to change the set of parameters to update for a given training phase is crucial. In BlockLLM, we developed a strategy that uses the current loss objective value as a signal to decide if the parameter selection needs to be updated.

Appendix B. More analysis

B.1. Analysis of weight magnitudes

In this experiment, we pretrain DistilBERT on the IMDB dataset [13] and then finetune it on GLUE-CoLA [24] with sparsity $s = 0.7$. We then plot the histogram of the weight magnitudes W^t where $\delta = |w_i^0 - w_i^t| > \eta$, with η as the threshold. We set $\eta = 0.001$ in this case.

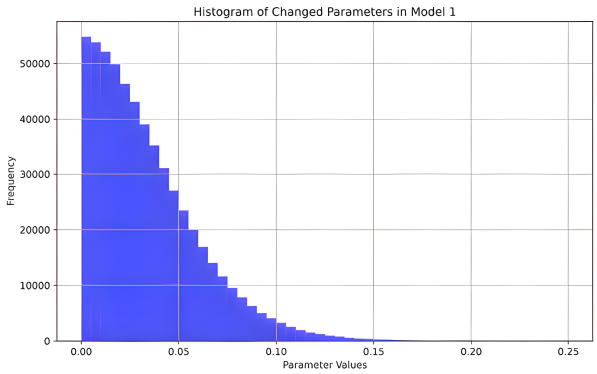

Figure 8: Histogram of changed parameters for $s = 0.7$

B.2. Analysis of reduced parameter training

Here, we fine-tuned the GLUE datasets [24] on the DistilBERT model [21] pretrained on the IMDb dataset [13]. We varied the sparsity s and update frequency m while monitoring the number of unique parameters updated q. We ran the GLUE-SST2 experiments for 8400 iterations and GLUE-STSB for 20000 iterations. Additionally, we tracked the VRAM usage for the GLUE-SST2 dataset to compare it with the memory consumption of full-parameter fine-tuning, which is 7.9 GB. This comparison aims to determine if reduced parameter training effectively decreases memory usage.

$1-s$	q	m	Spearman Correlation
0.01	0.05	5000	88.82
0.01	0.037	10000	88.77

Table 6: Impact of Update Frequency and Sparsity on STSB Dataset

Table 6 shows the impact of update frequency m and sparsity s on the STSB dataset, where the correlation remains stable despite changes in update frequency. As m increased too high, the performance declined.

$1-s$	q	m	Accuracy	VRAM
0.008	0.04	2400	91.97	4.4
0.01	0.11	3000	90.94	5.5
0.02	0.13	1000	90.59	5.5
0.02	0.16	2000	92.2	5.5

Table 7: VRAM Usage with Different Update Frequencies on SST2 Dataset

Appendix C. Algorithm

Algorithm 1 BlockLLM Training Algorithm

Input: Data X, initial model parameters W_0, sparsity s, set of layers L, learning rate η, patience parameter m, β_1, β_2, ϵ.
Initialize: $M_0 = 0$, $V_0 = 0$, $H = []$
while *each iteration* t **do**
  `// Forward and backward pass`
  Loss ϕ_t, G_t = COMPUTE GRADIENT(W_t)
  if *length(H)* $\geq m$ ***and***
  $\phi_t \geq \frac{1}{m}\sum_{i=t-m+1}^{t} H[i]$ ***or*** *(*$t = 0$*)* **then**
    mask, S = SELECTPARAM(G_t, s, L)
    $H = []$; `// Reset loss history`
  end
  `// Update selected parameters`
  foreach $l \in S$ **do**
    G_t^l = COMPUTE GRADIENT(W_t^l)
    $M_t^l, V_t^l, \tilde{G}_t^l$ = ADAM($M_{t-1}^l, V_{t-1}^l, G_t^l$)
    $\tilde{G}_t^l = \text{mask} \odot \tilde{G}_t^l$
    $W_{t+1}^l = W_t^l - \eta \tilde{G}_t^l$
  end
end

Algorithm 2 Select Parameters Function

Function `SelectParam(`G_t, s, L`)`**:**
  Compute $\|\tilde{G}_t^l\|$ for each layer $l \in L$
  $n = \sum_{l \in L} \text{count}(l)$, $n_s = (1-s) \times n$
  $D \leftarrow \text{sort}(L, \text{desc}, \|\tilde{G}_t^l\|/f_l)$
  Initialize $S = []$, $\Sigma_p = 0$
  foreach $l \in D$ **do**
    Σ_p += count(l), $S \leftarrow S \cup \{l\}$
    if $\Sigma_p \geq n_s$ **then**
      break
    end
  end
  foreach $l \in S$ **do**
    Let $\text{mask}_l = \mathbf{0}_{|G_t^l|}$
    foreach i, j *in mask* **do**
      if $\|\tilde{G}_t^l[i,j]\| \geq \tau$ **then**
        $\text{mask}_l[i,j] = 1$
      end
    end
  end
  return *mask,* S

Figure 9: (Left) The main BlockLLM algorithm, (right) SELECTPARAM function that performs parameter selection at iteration t.

Appendix D. Approximate BlockLLM

In the current implementation of BlockLLM, masks are binary matrices matching the dimensions of the gradients in their respective layers. Consequently, the memory footprint scales with the number of selected layers, as each layer requires a new mask. Therefore, for our experiments, we consider an approximation where we skip the mask computation step and update all parameters in S. The

update step is as follows:

$$W_{t+1} = \begin{cases} W_t^l - \eta \tilde{G}_t^l & \text{if } l \in S, \\ \text{Freeze } W^l & \text{otherwise.} \end{cases} \tag{1}$$

Our experimental results demonstrate that this approximate block selection strategy is effective in practice. Nonetheless, we believe that a more refined implementation, capable of achieving a more granular parameter selection, is feasible. We reserve this as a potential area for future exploration.

Appendix E. Experiment details

E.1. VRAM memory

All memory values presented in our tables represent actual observed memory usage in gigabytes (GB) rather than estimates. Memory consumption was monitored using the "nvidia-smi" command, and the maximum memory usage recorded during the training process was noted.

E.2. Finetuning on GLUE

The General Language Understanding Evaluation (GLUE) benchmark [24] is widely used to evaluate the performance of NLP models across a range of tasks, including sentiment analysis, question answering, and textual entailment. We fine-tuned the pre-trained RoBERTa model [11] , available in the HuggingFace library, on the GLUE benchmark tasks [24]. The batch size was set to 32 for the CoLA dataset, and 16 for all other datasets. For all tasks, we used $s = 0.95$ and $m = \frac{1}{4}\times$ total number of iterations. VRAM memory usage was monitored and recorded as described in Section E.1. The learning rates for the different tasks are as follows.

Table 8: Hyperparameter details for the GLUE experiments

	MRPC	COLA	STS-B	RTE	SST2	MNLI	QNLI	QQP
Learning rate	3E-05	5E-05	3E-05	3E-05	3E-05	3E-05	1E-05	3E-05

E.3. Pre-training on Llama

We present the hyperparameters utilized for training the LLama models with sizes 60M, 130M and 350M in 9. We fine-tuned the learning rate for BlockLLM while keeping all other hyperparameters fixed across experiments. We ran GaLore for 10% of total iterations to observe memory consumption, and used the results from [27] for comparison. For 60M and 130M experiments, the maximum sequence length was set to 256 with a gradient accumulation of 2,and for 350M with a batch size of 128 with a gradient accumulation of 4. A cosine annealing schedule was employed for learning rate adjustment, decaying to 10% of the initial learning rate. For BlockLLM, no learning rate warm-up was applied. However, for GaLore, the learning rate was warmed up for the first 10% of training, following the approach outlined in [27]. The parameter m was set to 50 for all the experiments.

E.4. Ablation on the hyperparameter m

We investigated the sensitivity of the model to the patience parameter m in both fine-tuning and pre-training setups. These experiments were conducted using the GLUE benchmark and the LLaMA

	60M	130M	350M
Learning rate	1E-03	1E-03	1E-03
Total training steps	10K	20K	60K
s	0.5	0.5	0.5

Table 9: Hyperparameter Details for Pre-training LLaMA Models with BlockLLM on the C4 Dataset

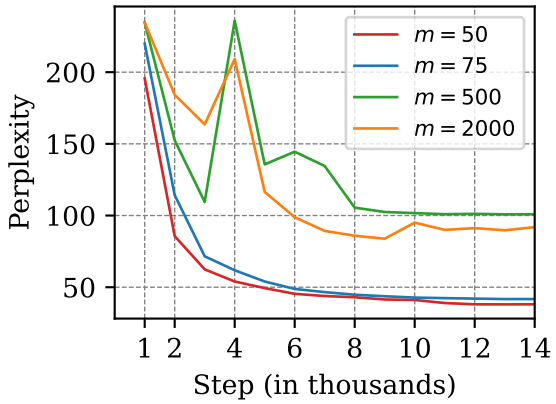

Figure 10: Ablation on the patience parameter m. We see that m affects the algorithm and model performance significantly in our pre-training experiments.

2 model on the C4 dataset. Throughout the experiments, we fixed all parameters of the Adam optimizer and maintained a sparsity level of $s = 0.5$ while varying m. The results are presented in Figure 10. Our observations indicate that in the fine-tuning setting, the model is relatively insensitive to variations in m. Specifically, setting $m = 50$ or $m = 1000$ did not result in significant performance differences. This finding aligns with the observations reported in [27], which suggest that gradients change more slowly. The gradual nature of gradient changes implies a correspondingly gradual variation in the optimal parameter set, thereby reducing sensitivity to changes in m. In contrast, in the pretraining setting, smaller values of m lead to faster convergence. This behavior can be attributed to the presence of noisy gradients in the earlier iterations of pretraining. Consequently, a smaller m helps maintain impactful parameter selection particularly in the initial phase of training, thereby facilitating faster convergence.

E.5. Ablation on Parameter Selection Strategy

We investigate whether selecting parameters with higher gradient norms, which are less frequently chosen, benefits training. We observed the training performance of the LLaMA 60M model on the C4 dataset while employing the strategy of randomly selecting block parameters for updates. Specifically, let $L = [l_1, l_2, \ldots, l_n]$ represent the list of all layers. At iteration t, we randomly select k layers from L with equal probability until the required number of parameters is met according to

the sparsity s. Let $D \subseteq L_{\text{shuffled}}$ be the subset of selected layers, defined as

$$D = \{l_i \in L_{\text{shuffled}} : \sum_{l_i \in D} \text{param\_count}(l_i) \leq s \times \sum_{l_i \in L} \text{param\_count}(l_i)\},$$

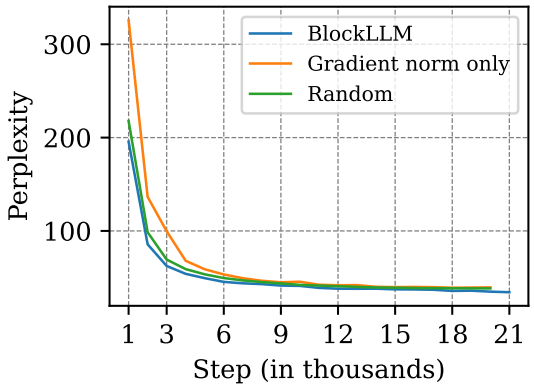

Figure 11: Comparison of random vs greedy BlockLLM on Llama 60m model with 50% parameters chosen.

where param_count(l_i) denotes the number of parameters in layer l_i. In each iteration t, we employ the Adam optimizer to update only the parameters in the layers belonging to the set D. We conducted hyper parameter tuning for the patience parameter m, which controls the frequency of updates. For a sparsity level of $s = 0.5$, we determined the optimal value of $m = 75$. To evaluate the effectiveness of the random parameter selection method, we compared evaluation loss with that of BlockLLM. Given that both methods involve updating a similar number of parameters, we expect their memory usage to be comparable. The comparative results are presented in Figure 10.

Our observations indicate that BlockLLM converges faster than the random parameter selection method. Intuitively, this suggests that parameters with higher gradient norms may facilitate more rapid learning. One possible explanation is that parameters with higher gradient norms contribute to larger updates, thereby playing a more significant role in the training process during each iteration. Consequently, these parameters might accelerate the model's convergence by making more substantial contributions to the optimization trajectory.

Effect of Layer Visit Frequency f. To assess this, we conducted an experiment where parameters were selected based solely on their gradient norms. Specifically, for each layer $l \in L$, we computed the processed gradients $\tilde{G}_l$. We then sorted the layers in descending order based on the their processed gradient norms, $||\tilde{G}_l||$. From this ordered list, we selected the top k layers that satisfied the sparsity requirement s. The results of this experiment are presented in Figure 10. We observed that when parameters were selected solely based on $||\tilde{G}||$, without considering the parameter visit frequency f_l for any layer $l \in L$, the loss converged more slowly and to a higher value compared to BlockLLM.

Broader Impacts Our work aims to reduce the memory and computational requirements of training LLMs. First, our technology democratizes access to LLM training, making it more feasible for

student researchers and institutions with limited computational resources to participate in cutting-edge AI research. Furthermore, the low-memory requirements of our method means that one can train with larger batch sizes and achieve faster convergence. This has a direct effect on the environment.

Future works. Future work on BlockLLM could explore several promising avenues. Currently, our research has focused on parameter selection based on gradient norms, but BlockLLM can be seen as a framework for parameter-efficient training rather than a single algorithm. This opens the door to investigating alternative criteria for parameter selection, potentially tailored to specific problems or tasks. Moreover, while our ablation studies on BlockLLM's hyperparameters have provided insights into their impact on training, further research is needed to understand how different layers might be affected by greedy parameter selection strategies. BlockLLM also complements existing memory-optimized training techniques, including those discussed in this paper. Exploring the integration of BlockLLM with methods like quantization or GaLore could further reduce memory consumption.