LoRD: Low-Rank Decomposition of Monolingual Code LLMs for One-Shot Compression

Ayush Kaushal [*1] **Tejas Vaidhya** [*1 2] **Irina Rish** [1 2]

Abstract

We propose using low-rank matrix decomposition (LoRD), which splits a large matrix into a product of two smaller matrices, to compress neural network models and thereby enhance inference speed. Unlike quantization, LoRD maintains fully differentiable, trainable parameters and leverages efficient floating-point operations. We investigate its advantages for compressing Large Language Models (LLMs) for monolingual code generation, demonstrating that linear layer ranks can be reduced by up to 39.58% with less than a 1% increase in perplexity. Specifically, we use LoRD to compress the StarCoder 16B model to 13.2B parameters with no performance drop and to 12.3B parameters with minimal performance drop in the HumanEval Pass@1 score, all within 10 minutes on a single A100 GPU. The compressed models achieve up to a 22.35% inference speedup with just a single line of code change in Hugging Face's implementation with Pytorch backend.

1. Introduction

Code LLMs are becoming essential tools for enhancing developers' productivity in LLM-assisted products like Copilots(Peng et al., 2023) and in LLM-based agents(Wang et al., 2023). However, their large size (e.g., 34B parameter publicly available models (Rozière et al., 2023) and 175B+ proprietary ones (Chen et al., 2021a)) leads to high operational costs and slow inference, particularly for Copilot applications.

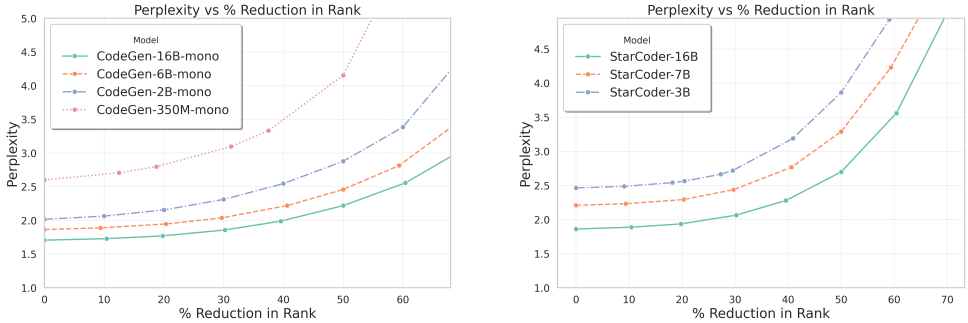
The need for smaller and faster LLM models motivated a variety of recent methods for model compression and inference speed-up. Quantization (Frantar et al., 2023; Dettmers et al., 2023b) reduces the number of bits per weight parameter by lowering precision; it can significantly compress LLMs and speed up inference in low-batch decoding phases (Kim et al., 2023). Pruning compresses models by removing connections from neural networks, thereby sparsifying the weight matrices. Distillation trains a smaller model using a larger teacher model for supervision but requires significant computational resources for retraining. In this work, we propose using eigenvector-based compression, referred to as LOw Rank Decomposition (LoRD) for Code LLMs. This method does not require expensive retraining and addresses several limitations of quantization and pruning. Low-Rank Decomposition factorizes a dense matrix of a neural network as a product of two smaller dense matrices. The LoRD model can leverage the highly optimized floating-point dense matrix multiplication kernels (NVIDIA, 2007; Blackford et al., 2002) that have been written for modern hardware. In contrast, quantized models often require specialized hardware-dependent implementation in order to enable fast inference. Moreover, the neural network remaining fully-differentiable and all the parameters remaining trainable even after compression, unlike quantization.

Several previous works have attempted to apply matrix decomposition methods like SVD, Tucker or Kronecker decomposition for compression (Ben Noach and Goldberg, 2020; Tahaei et al., 2022; Edalati et al., 2022). However, these have been limited to small language models like Bert (Devlin et al., 2019) and GPT2 (Radford et al., 2019), and have shown success only on narrow task-specific use cases or after retraining of neural network, often only with teacher-guided distillation supervision. These works have observed that weight matrices are not low rank and adapt methods like Singular Value Decomposition for data-aware decomposition of weights (Chen et al., 2021b; Hsu et al., 2022; Yu and Wu, 2023). We adapt these approaches for Large Language Models over Python code, and show that these models can be low-rank decomposed to compress and speed up inference without the need for retraining entire neural network.

To summarize, the contributions of this work are as follows: This work studies low-rank decomposition across two families of code LLMs - StarCoder and CodeGen (section 2.2) for varying parameter sizes, establishing the potential for reducing the rank of models through decomposition. It ex-

[*]Equal contribution [1]University of montreal [2]Mila. Correspondence to: Tejas Vaidhya <iamtejasvaidhya@gmail.com>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.



(a) Perplexity vs % Rank Reduction for CodeGen Models.

(b) Perplexity vs % Rank Reduction for StarCoder Models.

Figure 1. Perplexity vs % Reduction in Rank for Different Models.

amines these trends across different kinds of linear layers in a transformer block, observing the potential for up to a 39.58% rank reduction with less than a 1% change in perplexity. Next, we proposes various considerations regarding layer selection, grouping, and optimal rank for compressing the models to achieve optimal compression and inference speedup (section A.3). Notably, the StarCoder 16B model, scoring 31.67 in HumanEval Pass@1 (Chen et al., 2021a), is compressed to 13.2B parameters with a similar HumanEval score of 31.57, and further down to 12.3B parameters with a score of 29.22 (section 3.2). LoRD models offer an inference speedup of as high as 22.35% with just one line of change in huggingface's transformer library (section 3.3). It demonstrates that these LoRD models can be further compressed via the near-lossless quantization method of SpQR (Dettmers et al., 2023b) to reduce their precision to 8 and 4 bits without any further reduction in HumanEval performance (section A.6.1). Finally, these decomposed models also reduce the memory requirements of adapter finetuning by 21.2% over QLoRA (section A.5).

2. Code LLMs are Low Rank Decomposable

2.1. Experimental Settings

We limit ourselves to evaluating the Python code-writing capabilities of monolingual code LLMs. We take Python calibration dataset from *the stack* (Kocetkov et al., 2022) and consider the corresponding subset of the stack smol (Bigcode, 2022) as validation data. We filter out those sequences which are less than 1024 tokens or 10240 characters in length. In our case study, we consider the **CodeGen mono** and **StarCoder** model families. The CodeGen mono models are available in various sizes with 350M, 2B, 6B, and 16B parameters. On the other hand, StarCoder models include a specialized version of StarCoderBase 16B, which is a 16B model further trained on Python code sourced from the stack dataset's training split. Additionally, within the StarCoder family, we include StarCoderBase models at 3B and 7B parameter scales, due to the lack of their monolingual counterparts. All our experiments were performed on a single A100 GPU.

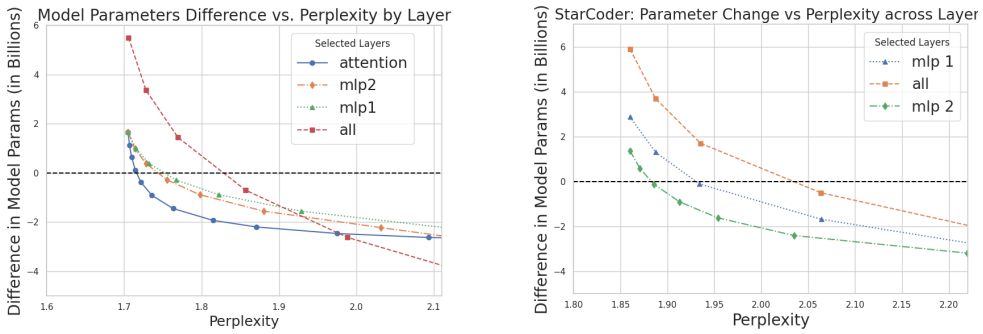
2.2. Change in Perplexity across Reduction in Rank

For studying the trends of increase in perplexity with a reduction in rank across different model sizes focusing on CodeGen and StarCoder models as case studies. We set a fixed low-rank r for all the layers. Later, we will discuss how to achieve optimal compression and inference speedup via low-rank decomposition in section §3.

Results: Analysis of the perplexity versus rank reduction trends in both CodeGen and StarCoder as depicted in Figures 1(a) and 1(b) show the trends of increase in perplexity across reduction in rank of the weight matrix. For the largest models in both families, we observe only about a 1% increase in perplexity for 10% reduction in rank, and upto 35% reduction in rank for less than 10% increase in perplexity. However, the smallest model i.e. CodeGen Mono 350M can only be decomposed to 35% rank reduction for a similar drop in perplexity. We observe that the perplexity changes much slower for larger models as the % rank reduces, and hence can be compressed more. This is similar to observations in quantization and pruning (Li et al., 2020). It should be noted that for most models, more than 50% leads to significant output quality degradation. The detailed table is provided in appendix A.5

3. Consideration for Optimal Compression and Speedup through Decomposition

In this section, We discuss how to adapt the low rank decomposition optimally for model size reduction and achieving inference speedup without significant reduction in models' output quality. Following (Kim et al., 2023), memory bandwidth is assumed to be the bottleneck for inference, making speedups for decoding directly proportional to the size of the transformer model.



(a) CodeGen 16B. (b) StarCoder 16B.

Figure 2. Parameter Reduction vs perplexity for decomposition across various layers.

3.1. Low-Rank Decomposition in Transformer: To which weight matrices do we apply LoRD?

LoRD can be applied to any subset of weight matrices in a neural network. Transformer models have varying aspect ratios across their linear layers: $\alpha = 1.00$ for output projections after attention, $\alpha = 0.96$ for multi-query attention projections, $\alpha = 0.25$ for typical MLP projections with an expansion factor of 4, and as low as $\alpha = 0.12$ for the embedding and language model head projection in CodeGen 16B with a vocab size of 51,200. Figure 4 (in appendix) plots the % change in model sizes across various percentage reductions in rank for matrices with different aspect ratios.

For square and near-square matrices, a small rank reduction doubles the size of the linear layer after decomposition. Only after a 50% reduction does the size after decomposition match the original matrix. A significant degradation in performance is observed at this level of rank decomposition, as discussed in §2.2. All the previous works on smaller models, address this by retraining the model (Yu and Wu, 2023; Chen et al., 2021b; Hsu et al., 2022; Ben Noach and Goldberg, 2020), often via knowledge distillation supervision (Hinton et al., 2015; Sanh et al., 2019) on specific narrow tasks. However, retraining is infeasible for larger models, prompting us to skip decomposing matrices with very high aspect ratios, such as output projection or multi-query attention. In contrast, the weights in MLP achieve parity at only 20% rank reduction. While embedding and LM head layers can be compressed through decomposition, as has been done for smaller transformer models (Baevski and Auli, 2019; Lan et al., 2020), they contribute only a very small portion of the model's weight. Therefore, we do not consider decomposing these matrices. Below, we will describe the set of considerations and analyses for helping with optimal compression via LoRD.

Weight Sharing and Aspect Ratio Reduction through Layer Grouping: To reduce the aspect ratio of matrices, we group layers with the same input vector to share the same bottleneck matrix after decomposition. This enables re-use of computation and sharing of weights, thereby reducing the aspect ratio and achieving compression with lower rank reduction. Candidate layers for grouping include the query, key and value projection matrices in multi-headed attention with aspect ratio reduced to $\alpha = 0.33$ and the gating layer in SwiGLU (Shazeer, 2020) with the first linear layer of MLP in models like LLaMa (Touvron et al., 2023) with $\alpha = 0.1875$.

Layer Sensitivity for Optimal Low-Rank Decomposition: We conduct empirical analyses to study the sensitivity of different layers to low-rank decomposition across the largest model among the two model families. Figure 2 illustrates the increase in perplexity versus reduction in model parameters. Decomposing all linear layers achieves the parity point much later than any single linear layer with a low aspect ratio. For CodeGen, the attention weight matrix (query, key, and value projections) offers the least increase in perplexity for the highest drop in parameter count. This makes these these layers most suitable for decomposition, showing less than 1% increase in perplexity even after 39.58% rank reduction. We also observe the mlp2 (downscaling mlp) suited better for decomposition over mlp1 (upscaling mlp), making mlp2 a promising candidate for low-rank decomposition in the StarCoder model.

Preferred ranks for efficient computation: On modern hardware accelerators like GPU and their corresponding software stacks, matrix multiplication kernels are faster. Given their dimensions are divisible by a high factor of 2.

3.2. Performance on LoRD models

We perform low-rank decomposition on the largest models of the StarCoder and CodeGen families (16B), varying the ranks for both. The detailed framework for building compressed LoRD models are provided in Appendix A.4. We consider decomposing layers that offer the most parameter reduction (§A.3) with the least increase in perplexity - MLP 2 for StarCoder and attention for CodeGen. We report the Pass@1 and Pass@10 scores over the Human Eval dataset

Table 1. Human Eval Score of LoRD across StarCoder and CodeGen.

StarCoder 16B				CodeGen 16B Mono			
Model Type	**Rank**	**Pass @ 1**	**Pass @ 10**	**Model Type**	**Rank**	**Pass @ 1**	**Pass @ 10**
Base Model	6144	31.67	48.28	Base Model	6144	29.02	46.34
LoRDCoder 14.9B	4480	33.18	48.41	LoRDCoder 15.9B	4480	29.08	46.95
LoRDCoder 14.5B	4096	31.69	45.12	LoRDCoder 15.6B	4096	28.90	46.24
LoRDCoder 13.8B	3584	30.90	47.56	LoRDCoder 15.1B	3584	28.54	45.73
LoRDCoder 13.2B	3072	31.57	45.36	LoRDCoder 14.7B	3072	27.99	43.29
LoRDCoder 12.6B	2560	29.84	42.31	LoRDCoder 14.3B	2560	27.32	45.12
LoRDCoder 12.3B	2304	29.22	40.12	LoRDCoder 14.1B	2304	27.07	41.46

(Chen et al., 2021a) using the code-eval GitHub repo (Bacaj, 2023) in Table 1.

Results: We observe that StarCoder models can be low-rank decomposed to 13.2B parameters (50% rank reduction) with no drop in Pass@1 performance. CodeGen models also exhibit a consistent trend in Human Evaluation performance when assessed in terms of rank reduction. However, when comparing reductions in parameter count, CodeGen models experience a significant decline in HumanEval scores due to the higher aspect ratio of the decomposed matrix. It noteworthy that certain compressed models exhibit a slight improvement in Pass@1 compared to the base model. This trend of slight enhancement resulting from compression is also observed across various metrics and benchmarks in other compression efforts (Frantar and Alistarh, 2023; Cerebras, 2022).

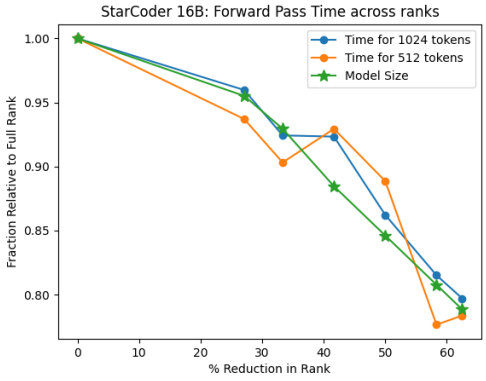


Figure 3. Time and Model size of StarCoder 16B across ranks.

3.3. Speedup from LoRD

Next, we examine the inference speedup of the models over the standard cuBLAS floating point kernels. We use the standard Huggingface implementation (Wolf et al., 2020) of Starcoder with pytorch backend (Paszke et al., 2019) utilizing standard cuBLAS kernels on A100 GPUs. LoRD decomposed models were implemented by modifying just one line of code to replace an MLP with an extra linear layer [1] for inference. We benchmark over 1024 tokens and 512 tokens sequence, averaged across 10 runs with warm up of 3 runs. The relative time taken and model size were plotted against the reduction in rank, as shown in Figure 3.

Results: Inference speedups as high as 22.35% are observed for decomposed models. Generally, the lines in the graph exhibit a downward slope, indicating that a reduction in rank beyond 25% typically results in decreased inference time and model size. However, it's important to note that the underlying hardware, as well as associated software kernels, significantly influence the speedup gains. We notice huge gains, when the rank is rounded off to a multiple of higher power of 2 (like 4096 and 2560 at 33% and 58% rank reduction), despite slight reduction in model size. In contrast, for certain ranks with multiples of lower powers of 2 (like 3584 and 2304 at 41% and 62% rank reduction) are slower than those at slightly higher ranks. It is worth noting that the impact of hardware inefficient matrix shape is less significant for longer tokens sequence of 1024. This is a consequence of $O(n^2)$ attention overhead starting to become more significant, especially in the absence of SoTA attention implementation techniques (Rabe and Staats, 2021; Dao et al., 2022; Dao, 2023).

[1] `nn.Linear(in, out) -> nn.Sequential(nn.Linear(in, rank), nn.Linear(rank, out))`

4. Conclusion

We investigated the compression of monolingual code generation model family using one-shot compression paradigm: low-rank decomposition. Analyzing the change in perplexity with the change in rank across model families such as StarCoder and CodeGen, as well as their individual layers. We observed that the rank of these models can be reduced by up to 39.58% with less than a 1% change in perplexity. Subsequently, we proposed considerations for one-shot compression of these models using LoRD, achievable under 10 minutes on Nvidia A100 GPUs. We compressed StarCoder 16B to 13.2B parameters with no drop in HumanEval pass@1 and only a minor drop in HumanEval pass@1 to 12.3B parameters, achieving speedups of up to 22.35%. Furthermore, the LoRD models are compatible with near-lossless quantization techniques such as SpQR, offering additional gains from quantization-based compression in addition to those from decomposition.

5. Acknowledgements

We acknowledge the support from the Mozilla Responsible AI Grant, the Canada CIFAR AI Chair Program and the Canada Excellence Research Chairs Program. This research was enabled by the computational resources provided by the Summit supercomputer, awarded through the Frontier DD allocation and INCITE 2023 program for the project "Scalable Foundation Models for Transferable Generalist AI" and SummitPlus allocation in 2024. These resources were supplied by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, with support from the Office of Science of the U.S. Department of Energy. We extend special thanks to Jens Glaser for his assistance with the Summit and Frontier supercomputers. We would also like to thank Ishita Hirmath for helping with proofreading.

References

A. Bacaj. code-eval. https://github.com/abacaj/code-eval, July 2023.

A. Baevski and M. Auli. Adaptive input representations for neural language modeling. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ByxZX20qFQ.

M. Ben Noach and Y. Goldberg. Compressing pre-trained language models by matrix decomposition. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 884–889, Suzhou, China, Dec. 2020. Association for Computational Linguistics. URL https://aclanthology.org/2020.aacl-main.88.

P. Bigcode. The stack smol, 2022. URL https://huggingface.co/datasets/bigcode/the-stack-smol.

L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

T. Cerebras. Creating sparse gpt-3 models with iterative pruning, 11 2022. URL https://www.cerebras.net/blog/creating-sparse-gpt-3-models-with-iterative-pruning.

S. Chaudhary. Code instructions dataset. https://huggingface.co/datasets/sahil2801/code_instructions_120k, Jun 2023.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021a.

P. Chen, H.-F. Yu, I. Dhillon, and C.-J. Hsieh. Drone: Data-aware low-rank compression for large nlp models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 29321–29334. Curran Associates, Inc., 2021b. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/f56de5ef149cf0aedcc8f4797031e229-Paper.pdf.

T. Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Re. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx.

T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023a.

T. Dettmers, R. Svirschevski, V. Egiazarian, D. Kuznedelev, E. Frantar, S. Ashkboos, A. Borzunov, T. Hoefler, and D. Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression, 2023b.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

A. Edalati, M. Tahaei, A. Rashid, V. Nia, J. Clark, and M. Rezagholizadeh. Kronecker decomposition for GPT compression. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*

(Volume 2: Short Papers), pages 219–226, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-short.24. URL https://aclanthology.org/2022.acl-short.24.

R. Feng, K. Zheng, Y. Huang, D. Zhao, M. Jordan, and Z.-J. Zha. Rank diminishing in deep neural networks. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=tIqzLFf3kk.

E. Frantar and D. Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015.

Y.-C. Hsu, T. Hua, S. Chang, Q. Lou, Y. Shen, and H. Jin. Language model compression with weighted low-rank factorization. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=uPv9Y3gmAI5.

E. J. Hu, yelong shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer. Squeezellm: Dense-and-sparse quantization, 2023.

D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries. The stack: 3 tb of permissively licensed source code, 2022.

Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, and J. E. Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

C. NVIDIA. Compute unified device architecture (cuda). Website, 2007. URL https://developer.nvidia.com/cuda-toolkit. Accessed: 2023-09-17.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, chapter ., page . Curran Associates Inc., Red Hook, NY, USA, 2019.

G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, A. Cappelli, H. Alobeidli, B. Pannier, E. Almazrouei, and J. Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data, and web data only, 2023.

S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023.

M. N. Rabe and C. Staats. Self-attention does not need $o(n^2)$ memory, 2021.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code, 2023.

V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019.

N. Shazeer. Glu variants improve transformer, 2020.

B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, Y. Guo, and Q. Wang. Pangu-coder2: Boosting large language models for code with ranking feedback, 2023.

M. Sun, Z. Liu, A. Bair, and J. Z. Kolter. A simple and effective pruning approach for large language models, 2023.

M. Tahaei, E. Charlaix, V. Nia, A. Ghodsi, and M. Rezagholizadeh. KroneckerBERT: Significant compression of pre-trained language models through kronecker decomposition and knowledge distillation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2116–2127, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.154. URL https://aclanthology.org/2022.naacl-main.154.

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023.

L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J.-R. Wen. A survey on large language model based autonomous agents, 2023.

T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6.

H. Yu and J. Wu. Compressing transformers: Features are low-rank, but weights are not! *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(9): 11007–11015, Jun. 2023. doi: 10.1609/aaai.v37i9.26304. URL https://ojs.aaai.org/index.php/AAAI/article/view/26304.

A. Appendix

A.1. Background on Low-Rank Decomposition

Let L denote the linear layer of an LLM model, associated with the weights $W \in \mathbb{R}^{d_1 \times d_2}$ and bias $b \in \mathbb{R}^{d_1 \times 1}$, and let $\mathrm{d}_{\min} = \min(d_1, d_2)$ and $\mathrm{d}_{\max} = \max(d_1, d_2)$. A low-rank decomposition (a.k.a. low-rank factorization) of L yields a new layer $\tilde{L}$, with two weight matrices $A \in \mathbb{R}^{r \times d_2}$ and $B \in \mathbb{R}^{d_1 \times r}$, and a bias $\tilde{b} \in \mathbb{R}^{d_1 \times 1}$, where $r << \mathrm{d}_{\min}$, such that for a batch of n input vectors, $X \in \mathbb{R}^{d_2 \times n}$, the batch of output vectors $Y \in \mathbb{R}^{d_1 \times n}$ is given by

$$Y = \tilde{L}(X) = BAX + \tilde{b} \approx L(X) = WX + b. \quad (1)$$

The Singular Value Decomposition (SVD) offers the best r-rank approximation of matrix $W \in \mathbb{R}^{d_1 \times d_2}$. First W can be decomposed as $W = USV^T$, where $U \in \mathbb{R}^{d_1 \times d_1}$ and $V \in \mathbb{R}^{d_2 \times d_2}$ are orthogonal matrices and $S \in \mathbb{R}^{d_1 \times d_2}$ is a diagonal matrix of singular values with entries in decreasing order. Selecting the top k singular values and associated vectors gives a decomposition of W into a product of two low-rank matrices $W \approx BA$ as follows

$$W = \underbrace{(U_{:,:r} S_{:r,:r})}_{B \in \mathbb{R}^{d_1 \times r}} \underbrace{(V_{:r,:})}_{A \in \mathbb{R}^{r \times d_2}}, \quad (2)$$

where $_{:a,:b}$ denotes a slice operation over a matrix that gives its first a rows and b columns.

Eigendecomposition is another decomposition method. We can represent the eigendecomposition of a square matrix $\hat{W} \in \mathbb{R}^{d_1 \times d_1}$ as $\hat{W} = Q\Lambda Q^T$, if $\hat{W}$ is positive semi-definite. Here $Q \in \mathbb{R}^{d_1 \times d_1}$ is an orthogonal matrix whose columns are the eigenvectors of W, and $\Lambda \in \mathbb{R}^{d_1 \times d_1}$ is a diagonal matrix whose entries are the eigenvalues of W sorted in decreasing order. Similar to SVD, we can decompose $\hat{W}$ as a product of two low ranked matrices $\hat{W} \approx BA$ by retaining only the largest r eigenvalues (and corresponding eigenvectors) as follows:

$$\hat{W} = \underbrace{(Q_{:,:r} \Lambda_{:r,:r})}_{B \in \mathbb{R}^{d_1 \times r}} \underbrace{(Q^T_{:r,:})}_{A \in \mathbb{R}^{r \times d_1}} \quad (3)$$

Since Q is orthonormal and the eigenvalues Λ are sorted in descending order, $Q_{:,:r} Q^T_{:,:r} \approx \mathbf{I}$ where $\mathbf{I}$ is identity matrix of dimension d_1.

SVD gives the optimal low-rank decomposition of matrix in terms of Frobenius norm; however it does not take the data distribution into account. Approaches like weighted SVD (Hsu et al., 2022) and SVD over both weight and data (Chen et al., 2021b) have been proposed but are prohibitively expensive to scale to larger models for their requirement of backpropagation over calibration dataset. SVD over very large weight matrices is also very computationally expensive. So, we instead leverage the observation that activations in transformers are low-ranked (Feng et al., 2022) and adapt the more heuristically driven approach of Atomic Feature Mimicking (AFM) (Yu and Wu, 2023) that creates low rank matrices conditioned on a small amount of calibration data. Specifically, consider the eigen-decomposition of covariance over Y as

$$\mathbb{E}[yy^T] - \mathbb{E}[y]\mathbb{E}[y]^T = \hat{Q}\hat{\Lambda}\hat{Q}^T \quad (4)$$

Here $\hat{Q}$ is a matrix of its eigenvectors, hence $\hat{Q}_{:,:r}\hat{Q}^T_{:,:r} \approx \mathbf{I}$. Using this, we can write the output vector Y as $Y \approx \hat{Q}_{:,:r}\hat{Q}^T_{:,:r}Y$. By writing Y in terms of W, X and b from Equation 1, we have:

$$Y \approx \hat{Q}_{:,:r}\hat{Q}^T_{:,:r}WX + \hat{Q}_{:,:r}\hat{Q}^T_{:,:r}b \quad (5)$$

Comparing to Equation 1, this gives us $B = \hat{Q}_{:,:r} \in \mathbb{R}^{d_1 \times r}$, $A = \hat{Q}^T_{:,:r} W \in \mathbb{R}^{r \times d_2}$ and $\tilde{b} = \hat{Q}_{:,:r} \hat{Q}^T_{:,:r} b \approx b$. This approach is also straightforward to adapt for LLMs like LLaMa (Touvron et al., 2023), Falcon (Penedo et al., 2023), CodeLLaMa (Rozière et al., 2023) which do not have a bias term in the linear layer by setting $\tilde{b}$ to zero vector.

A.2. Dataset and Model

Our study focuses on evaluating Python, one of the most popular programming languages. This choice is driven by the popularity and availability of datasets, open monomodels, and more comprehensive evaluation suites compared to other languages.

A.2.1. Dataset

The Stack (Kocetkov et al., 2022) is a large dataset of permissively licensed source code in 30 programming languages created by the BigCode Project for training code-generating AI systems.

Stack Smol(Bigcode, 2022): A small subset (0.1%) of the dataset from the Stack comprises 10,000 random samples from each programming language in the original dataset.

In this paper, we utilize a calibration dataset derived from Python subset of the Stack. For validation purposes, we rely on the Python subset of Stack Smol. We also filter out sequences that are less than 1024 or 10240 characters in length.

A.2.2. Models

In this paper, we consider the following Code models for experimentation

CodeGen {350M, 2B, 6B, 16B}-Mono (Nijkamp et al., 2023) is family of an open-source autoregressive language model for program synthesis trained sequentially on code dataset. We specifically use Codegen Mono family of models, which are initialized with CodeGen-Multi and further pre-trained on a Python programming language dataset. In our study, we use four different sizes of the CodeGen Mono family: 350M, 2B, 6B, and 16B.

StarCoderbase is a family of open-source models trained on over 80 programming languages from The Stack dataset. These models employ Multi Query Attention and were trained using the Fill-in-the-Middle objective, with a context window of 8,192 tokens. In this study, we use and refer the StarCoderBase 3B and 7B models as StarCoder 3B and 7B, respectively due to the unavailability of their Python fine-tuned counterpart. The StarCoder 16B model is a version of the StarCoderBase 16B model that has been further trained on a Python dataset. It should be noted that this model hasn't been further trained on Python datasets. However, our results for this model are consistent with the results of models that have been further fine-tuned on Python Datasets.

A.3. Key concepts for compression through Decomposition

Threshold for Effective size reduction: Consider a weight matrix $W \in \mathbb{R}^{d_1 \times d_2}$ of a transformer layer with low rank decomposed $A \in \mathbb{R}^{r \times d_2}$ and $B \in \mathbb{R}^{d_1 \times r}$. The number of parameters before and after decomposition respectively are $d_1 d_2$ and $r(d_1 + d_2)$. Therefore, if $r > \frac{d_1 d_2}{(d_1 + d_2)}$, (i.e a decomposition with small rank reduction), then the size of the model after decomposition can even be higher than the original models.

Impact of Matrix Aspect Ratio on Compression: Let the ratio of the smaller dimension to the larger dimension of the matrix (i.e. the aspect ratio) be $\alpha = \frac{d_{min}}{d_{max}}$. For square matrix, $\alpha = 1$ and for tall or fat matrices $\alpha << 1$.

The percentage change in parameters from decomposition, in terms of percent change in rank $\%\Delta r = 100 * \frac{d_{min} - r}{d_{min}}\%$ and aspect ratio can be expressed as:

$$100 * \frac{r(d_{max} + d_{min}) - d_{max} d_{min}}{d_{max} d_{min}} = 100\alpha - (1 + \alpha)\%\Delta r \tag{6}$$

It should be noted that change in parameters from decomposition can either be positive (the number of parameters increased after decomposition), or negative (the number of parameters decreased after decomposition). In order to achieve model compression and consequently inference speedups, one would want a very high negative percentage change in parameters.

Parity Point for Compression: Using Eq. 6, one can observe that little reduction in rank may lead to an increase in model parameters instead of decreasing. For instance, square matrices ($\alpha = 1$) will have 100% increase (i.e doubling in size), then $\%\Delta r \to 0_+$ and only after the rank is reduced by more than 50%, will the **Parity Point** of the rank reduction be reached, that offers same or lesser number of a parameter in the decomposed layer as the original matrix. This parity point for tall or fat matrices ($\alpha \to 0_+$), can be achieved with a very small percent reduction in rank and can start giving a reduction in model size. For compression to be achieved, we would want to reduce the rank by an amount to cross this parity point threshold. However, reducing the rank by a lot can degrade performance significantly. So we must take the aspect ratio into account, in order to achieve compression without much reduction in rank.

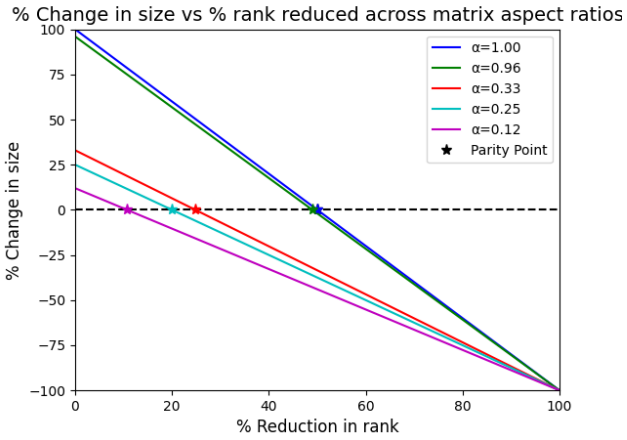


Figure 4. Parity Point across various aspect ratios (α) of the different linear layers in transformers.

A.4. Compressed LoRD Models

In this section, we provide a framework for creating optimally compressed LoRD models that are used to create a family of LoRD models for both StarCoder 16B and CodeGen16B-Mono.

Algorithm 1 LoRD model framework

Input: Large language model M, target rank reduction percentage r, proxy dataset D
Output: Compressed language model M'
Perform Layer Grouping to reduce aspect ratio.
Calculate the aspect ratios α of all layers
Identify critical layer(s) L in M (using layer sensitivity test and α)
for each layer l in L **do**
  Compute rank k of weight matrix W_l
  Set target rank $k' = (1 - r) \times k$
  Invoke Low Rank Decomposition Step on layer l with M, D, and k'
  Update M with compressed layer weights and biases
end for
Evaluate M' performance (e.g., Pass@k, perplexity)
Optionally, adjust r and repeat steps 2-6 for desired compression-performance trade-off
Return M'

A.5. Parameter Efficient tuning of LoRD models

We test the potential for using LoRD to further reduce the memory usage over existing parameter-efficient techniques. We consider the code instruction dataset (Chaudhary, 2023) and filter those examples that pertains to python programming language. We use QLoRA (Dettmers et al., 2023a), which is an even more memory efficient version of LoRA (Hu et al., 2022) (shown in figure 5)storing the weights in quantized format, for fine-tuning for 1 epoch. We compare results from fine-tuning two of the decomposed models LoRDCoder 13.2B and LoRDCoder 12.3B model to the StarCoder model. We observe a HumanEval pass@1 of 37.80 and 37.62 across LoRDCoder 13.2B and LoRDCoder 12.3B fine-tuning, competitive to the performance of 37.74 offered by StarCoder model.

Algorithm 2 Low Rank Decomposition Step (Adopted Atomic Feature Mimicking for LLMs)

Input: Original model M with weight W in the l-th layer, calibration dataset D, pre-set rank k.
Output: Two decomposed layer with weights B and A .
for each sample x in D **do**
  Forward pass $M(x)$ to get the output feature y in the l-th layer and update $E[yy^T]$ and $E[y]$.
end for
Calculate the eigenvectors $\hat{Q}$ based on Eq. 4.
Extract the first k columns of $\hat{Q}$ into $\hat{Q}_k$, and obtain $B = \hat{Q}_k$, and $A = \hat{Q}_k^T W$.
Return B, A.

A.6. Combining LoRD with Quantization and Pruning

A.6.1. Quantization

While LoRD enables compression at similar precision levels, we study whether the decomposed models can be further compressed through quantization. Table 2 shows the HumanEval pass@1 results for the different LoRDCoder (from StarCoder 16B) across 8 and 4 bit quantization levels, using near-lossless quantization technique of SpQR (Dettmers et al., 2023b).

We observe that the LoRD models can be combined with quantization for further compression, showing no performance drop for 8-bit and very little performance drop on 4-bit quantization for majority. Slight increase in HumanEval after quantization is also observed, similar to Pangu-Coder2 (Shen et al., 2023).

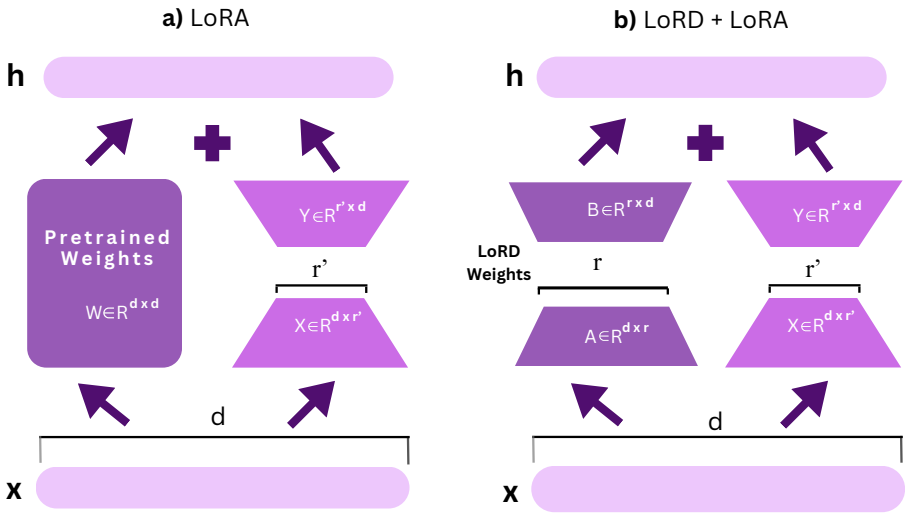


Figure 5. Comparing LoRA vs LoRD + LoRA.

Table 2. Human Eval score of quantized LoRDCoder models

Model	Pass@1 (FP16)	Pass@1 (8-bit)	Pass@1 (4-bit)
LoRDCoder 14.9B	33.18	33.17	32.01
LoRDCoder 14.5B	31.69	31.58	32.74
LoRDCoder 13.8B	30.90	31.10	30.73
LoRDCoder 13.2B	31.57	31.52	32.01
LoRDCoder 12.6B	29.84	29.87	30.22
LoRDCoder 12.3B	29.22	29.14	29.45

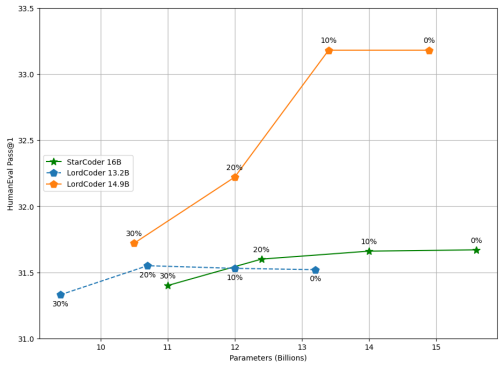


Figure 6. Model params vs human eval pass@1 across different sparsity.

A.6.2. Pruning

To explore the possibility of further compressing the LoRD models, we investigated their potential for additional sparsification. This is performed using state-of-the-art (SoTA) LLM pruning methods as described by Sun et al. (2023). The experimental results (shown in figure 6), particularly on the HumanEval benchmark, suggest that pruned LoRD models (LoRDCoder) maintain competitive performance compared to traditionally pruned models (e.g., StarCoder). This indicates that LoRD is not only effective as a standalone compression strategy but also as a foundational technique that can be combined with pruning for greater efficiency and retention of performance in the future.

A.7. Comparison of Storage Efficiency of pruned and LoRD models.

In this section, we delve into a comparative analysis between pruning and LoRD (Low-Rank Decomposition) Models. Traditional pruning techniques, which create sparse matrices by setting certain weights to zero, only show benefits in model size and inference speed at high sparsity levels. In contrast, the LoRD (Low-Rank Decomposition) compression technique offers a promising alternative by maintaining dense matrices that are inherently more compatible with the matrix multiplication operations on GPUs. This section compares the efficacy of LoRD with traditional unstructured pruning methods in terms of parameter reduction, and model size efficiency.

Pruned models produce sparse matrix weights in the neural network. Matrix multiplication over sparse matrices is much slower than the resulting dense matrices in LoRD on most GPUs. Dense matrices, in addition avoid representation format overhead that sparse matrices incur from parameter reduction [2] and often requires specialized kernels for reduc-

[2]This overhead in sparse matrix occurs from having to store indices/bitmasks to indicate which values are present and not. This can be very significant at low levels of sparsity. PyTorch's sparse formats (CSR, CSC, COO) all store indices at int64 format, and for moderate levels of sparsity (<50%), the sparse matrix takes up more space than a dense matrix with zero-ed out values.

ing this overhead (Dettmers et al., 2023b). Dense matrix multiplication is also easier to implement than sparse matrix multiplication, especially over quantized models.

The LoRD compression approach not only reduces the parameter count but also significantly decreases the model size when compared to pruned models using various sparse formats. We quantified this by considering a 4-bit quantized model size across similar parameter counts and evaluated the storage requirements for LoRD compressed models versus traditionally pruned models in three sparse formats: PyTorch's CSR, SpQR's custom kernels, and a hypothetical bitmask representation.

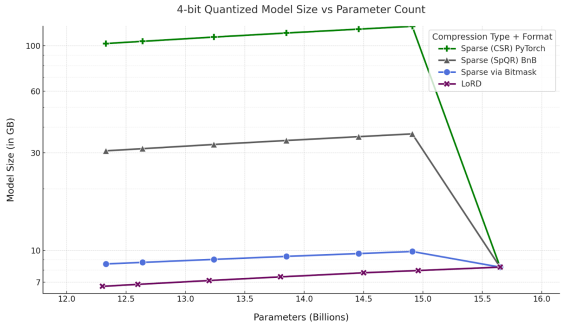


Figure 7. 4-bit Quantized Model Size vs Parameter Count

- **PyTorch CSR Format**: The CSR format, while being the most efficient sparse format in PyTorch, stores indices of nonzero elements at 64 bits. For a 4-bit quantized model, this accumulates to more than 68 bits per parameter due to the inability of PyTorch to support CSR matrices at lower than 16-bit precision. Comparatively, LoRD compressed models can save up to 96 GB at the same parameter count, highlighting a significant efficiency in storage.

- **SpQR Sparse Kernels**: These kernels, designed for floating-point matrices, store indices at 16 bits. Assuming these could support 4-bit quantization, the storage per nonzero element would exceed 20 bits. The LoRD models demonstrated a potential saving of up to 24.6 GB compared to SpQR's format, even before its integration into mainstream libraries like bitsandbytes.

- **Bitmask Sparse Format**: Although using a bitmask for nonzero elements leads to a low 5 bits per parameter for a 4-bit quantized model, this method suffers from practical deployment challenges such as the lack of supportive low-level software and difficulties in parallelization. LoRD models outperform this method by at least 28% in storage efficiency at equivalent parameter counts.

A.8. Limitations of LoRD

In our study, several limitations of the LoRD (Low-Rank Decomposition) method for neural network compression have been identified. First, we observed a notable performance degradation at high levels of rank reduction. Moreover, the effectiveness of LoRD is constrained by the aspect ratios of weight matrices. Our findings indicate that compressing square or near-square matrices is inefficient, and in certain cases, small rank reductions can increase model size due to the inherent aspect ratio of these matrices. Additionally, the application of LoRD across different transformer architectures requires sensitivity analysis of individual layers to determine optimal sections for decomposition. This variability limits the method's generalizability across diverse transformer architectures, necessitating tailored sensitivity analyses for each model type.

A.9. Perplexity and Reduction in Ranks

We present the changes in perplexity, parameter count, and percentile ranking resulting from rank reduction. Table 3 details this for the StarCoder model, while Table 4 presents it for the CodeGen models.

Table 3. Perplexity and Parameter Count Variations in StarCoder Models Across Rank Reduction

Name	Rank	Percent Rank	Layers	perplexity	Params
StarCoder-16B	1856	69.79	all	5.002	6.76B
StarCoder-16B	2432	60.42	all	3.56	8.75B
StarCoder-16B	3072	50.00	all	2.698	10.96B
StarCoder-16B	3712	39.58	all	2.278	13.17B
StarCoder-16B	4288	30.21	all	2.064	15.15B
StarCoder-16B	4288	30.21	all	2.064	15.15B
StarCoder-16B	4928	19.79	all	1.935	17.36B
StarCoder-16B	5504	10.42	all	1.888	19.35B
StarCoder-16B	6144	0	all	1.86	21.56B
StarCoder-7B	1664	59.38	all	4.228	4.26B
StarCoder-7B	2048	50.00	all	3.289	5.19B
StarCoder-7B	2432	40.63	all	2.766	6.12B
StarCoder-7B	2880	29.69	all	2.437	7.20B
StarCoder-7B	3264	20.31	all	2.293	8.13B
StarCoder-7B	3712	9.38	all	2.231	9.22B
StarCoder-7B	4096	0	all	2.208	10.15B
StarCoder-3B	1152	59.09	all	4.932	1.81B
StarCoder-3B	1408	50.00	all	3.862	2.17B
StarCoder-3B	1664	40.91	all	3.189	2.54B
StarCoder-3B	1984	29.55	all	2.718	2.99B
StarCoder-3B	2048	27.27	all	2.663	3.09B
StarCoder-3B	2240	20.45	all	2.563	3.36B
StarCoder-3B	2304	18.18	all	2.541	3.45B
StarCoder-3B	2560	9.09	all	2.486	3.82B
StarCoder-3B	2816	0	all	2.463	4.19B

Table 4. Perplexity and Parameter Count Variations in CodeGen Models Across Rank Reduction

Name	Rank	Percent Rank	Layers	Perplexity	Params
CodeGen-16B-mono	1856	69.79	all	3.037	6.93B
CodeGen-16B-mono	2432	60.42	all	2.556	8.76B
CodeGen-16B-mono	3072	50.00	all	2.218	10.90B
CodeGen-16B-mono	3712	39.58	all	1.989	13.04B
CodeGen-16B-mono	4288	30.21	all	1.857	14.96B
CodeGen-16B-mono	4928	19.79	all	1.770	17.10B
CodeGen-16B-mono	5504	10.42	all	1.728	19.03B
CodeGen-16B-mono	6144	0	all	1.706	21.17B
CodeGen-6B-mono	1280	68.75	all	3.425	2.77B
CodeGen-6B-mono	1664	59.38	all	2.814	4.02B
CodeGen-6B-mono	2048	50.00	all	2.458	4.85B
CodeGen-6B-mono	2432	40.63	all	2.218	5.68B
CodeGen-6B-mono	2880	29.69	all	2.037	6.65B
CodeGen-6B-mono	3264	20.31	all	1.946	7.48B
CodeGen-6B-mono	3712	9.38	all	1.888	8.45B
CodeGen-6B-mono	4096	0	all	1.863	9.28B
CodeGen-2B-mono	768	70	all	4.443	2.78B
CodeGen-2B-mono	1024	60	all	3.384	1.61B
CodeGen-2B-mono	1280	50	all	2.877	1.94B
CodeGen-2B-mono	1536	40	all	2.546	2.28B
CodeGen-2B-mono	1792	30	all	2.311	2.61B
CodeGen-2B-mono	2048	20	all	2.155	2.95B
CodeGen-2B-mono	2304	10	all	2.063	3.28B
CodeGen-2B-mono	2560	0	all	2.015	3.62B
CodeGen-350M-mono	384	62.5	all	6.332	230.99M
CodeGen-350M-mono	512	50	all	4.153	272.95M
CodeGen-350M-mono	640	37.5	all	3.333	314.90M
CodeGen-350M-mono	704	31.25	all	3.095	335.88M
CodeGen-350M-mono	832	18.75	all	2.797	377.83M
CodeGen-350M-mono	896	12.5	all	2.706	398.81M
CodeGen-350M-mono	1024	0	all	2.600	440.76M