Kronecker Products and GPT2

Ayoub @ Uni Passau

June 8, 2024

Abstract

We introduce Krony-PT, a compression technique of GPT2 [1] based on Kronecker Products. We specifically target the MLP layers of each transformer layer, and systematically compress the feed forward layer matrices to various degrees. We use optimal Van Loan decompositions to initialize the new matrices, and also introduce a new pruning-based initialization trick. Our method compresses the original GPT2 124M parameters to various smaller models, 67M being the smallest, and 95M being the largest compressed model. Our 81M model variant outperforms distilgpt2 on next-token prediction on all standard language modeling datasets, and shows competitive scores or performs on par with other Kronecker Products based compressed models of GPT2 that are significantly higher in size.

Contents

1 Introduction

With (/ Given) their rapid and unprecedented development, Large Language Models (LLMs) are revolutionizing many industries and changing the way we perceive and interact with machines. Their strong ability to understand, generate, and perhaps even "generalize" in numerous domains including automated text generation, language translation, and notably coding, has led many to believe that scaling up models is all you need to achieve Artificial General Intelligence (AGI) (i.e., The Scaling Hypothesis [2]). Despite their advanced features, the core of LLMs still fundamentally relies on the initial Transformer architecture [3]. And specifically the encoder only variants that was first introduced with GPT2 [1].

Although GPT-4 [4] has fairly led the race in the past year, a lot of the big actors of the industry seem to have recently caught up (e.g., Anthropic [5], Meta [6], [7] , and Google [8]), as demonstrated by the independent and open LMSys Leaderboard [1]. Subsequently, many believe that the two main ingredients are **compute** and **data**. And while this seems affordable for big corporations, it poses significant challenges for individuals and smaller organizations to reproduce these SATA models. Hence, advancing work on the compression and factorization methods of LLMs is essential to make these powerful tools more accessible and sustainable for broader use, and a necessity for deployment on edge-devices.

Our contributions could be summarized as follows:

- We use weight tying, i.e., the embedding and softmax matrices are identical. This makes our model, the only "effective" 81M model compared to (X,Y). Which significantly reduces the model size by 38M compared to other methods (see appendix for details regarding parameter count).
- We have a systematic compression scheme, i.e., compress all layers the same way. We don't see any reasons to why we would only compress odd layers (besides to "force" the number of parameters to be under certain threshold).
- We try different compression schemes (67M, 81M, and 95M)
  - We propose a new simple initialization for the 95M model.
  - We use multiple factors for the VL based units.
  - We introduce scalers to each factor, which (apparently) helps with generalizing (why? the performance on OWT seems to be ok, but the ppl is better on other benchmarks, e.g., wiki and lamabada)
- We don't use distillation (empirical evidence shows no benefits compared to only Vanilla supervised learning).

[1]https://chat.lmsys.org/?leaderboard

2 Related work

Compressing Large Language Models is currently a major focus in research. In this context, Matrix Factorization provide the perfect framework for compression. There are three major research directions in the compression of LLMs, **Distillation**, **Quantization**, and finally **Factorization**, which is the focus of our work. Generally speaking, Matrix Factorization techniques aim to combine (e.g., a product) lower rank matrices to reconstruct an original (typically bigger) matrix. A lot of work has been done on the compression of LLMs using factorization methods, including Kronecker Products. Most notably for encoder based based, and rarely applied to decoder based methods ([9], [10], [11]).

Motivated by how dense the attention layers are [], we opted not to compress the QKV matrices.

3 Methodology

3.1 Kronecker Decomposition

In this section, we study different Kronecker decomposition setups, and the percentage of compression it would lead to. So far we only decompose the weights `c_fc.weight` and `c_proj.weight` (each has $2.3M$ in the original GPT2-small architecture.). Each transformer layer (there are 12 in total) has two of these weights. They count to $56.6M$ in total (45% of GPT2 $124M$). Hence any significant reduction to these matrices would lead to a remarkable compression ratio of the whole model. We choose not to compress the other weights, namely, attention weights and embedding matrix for various reasons that we will expose later on.

3.2 The 95M model

The most basic strategy is to divide one of the dimensions of each W by 2, this would lead to a 95M model. The parameter $p_1 = \texttt{c\_fc.weight}$ (resp. $p_2 = \texttt{c\_proj.weight}$) has a shape of $(3072, 768)$ (resp. $(768, 3072)$). We first try the following decomposition: $p_1 = \underbrace{W_{11}}_{(3072,384)} \otimes \underbrace{W_{12}}_{(1,2)}$ and $p_2 = \underbrace{W_{21}}_{(384,3072)} \otimes \underbrace{W_{22}}_{(2,1)}$

This decomposition would lead to to reduction of $28M$ (50%). The new network would have approx $95M$. Our goal is to eventually reach the $82M$ mark, similar to DistilGPT2, and other Factorized models (inserts other refs here).

3.3 Different decomposition schemes:

It is reasonable to aim for a decomposition that guarantees the maximum rank we can get. Since the Rank of Kronecker products is multiplicative, i.e.,

$\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$ [2], we can easily compute the rank of each possible decomposition. In our case, we have $W \in R^{(m,n)}$ where $m = 3072$ and $n = 768$. Hence, for each layer of GPT2, we aim to find the "optimal" $A \in R^{(m_1,n_1)}$, and $B \in R^{(m_2,n_2)}$, i.e.,:

$$W \approx A \otimes B, \qquad m = m_1 m_2, n = n_1 n_2$$

.

W.l.o.g, for each decomposition (A, B) the maximum rank we can reach is $\min(m_1, n_1) \cdot \min(m_2, n_2)$. And each of the reduced decompositions would have exactly $m_1 n_1 + m_2 n_2$ parameters. Hence, theoritically, the maximum rank we can get is 768 of a $(3072, 768)$ matrix. The following table summarizes some possible combinations, alongside the reduction it would lead to per layer, and the total number of parameters in GPT2, for only those decompostions of maximal attainable rank. We are particularly interested in 3 class of models, the 67M, the 81M and the 96M. (Need to add this) Furthermore, we add multiple factors to the models labeled with **MF** (second table / a few decompositions are missing, check this out. e.g., 3072, 384).

[2]Link to proof: https://math.stackexchange.com/questions/707091/elementary-proof-for-textrank-lefta-otimes-b-right-textranka-cdot

Name	Dimension	params	Model size
67M	(64, 32)	3200	67,893,504
	(64, 48)	3840	67,908,864
	(96, 32)	3840	67,908,864
	(64, 64)	4672	67,928,832
	(128, 32)	4672	67,928,832
	(96, 48)	5120	67,939,584
	(96, 64)	6528	67,973,376
	(128, 48)	6528	67,973,376
	(128, 64)	8480	68,020,224
	(96, 96)	9472	68,044,032
	(192, 48)	9472	68,044,032
	(128, 96)	12480	68,116,224
	(192, 64)	12480	68,116,224
	(128, 128)	16528	68,213,376
MF1	(256, 64)	16528	68,213,376
	...	...	...
MF2	(1024, 256)	262153	74,108,376
	(768, 384)	294920	74,894,784
	(1024, 384)	393222	77,254,032
81M	(768, 768)	589828	81,972,576
	(1536, 384)	589828	81,972,576
	(1024, 768)	786435	86,691,144
96M	(1536, 768)	1179650	96,128,304
GPT2	(3072, 768)	2359297	124,439,832

Table 1: Different compression schemes

Name	Dimension	params	1 factor	2 factors	3 factors
MF1	(256, 64)	16528	68,2M	68,6	69M
MF2	(1024, 256)	262153	74M	80M	86M

Table 2: Adding multiple Kronecker Factors

3.4 Initialization

Since we inherit a GPT2 checkpoint that was trained for multiple epochs on the Open Web Text (OWT) (cite here), we want to initiliaze our weights in a way that leverages the old pre-training as much as possible. This is of course obvious for the parameters that are common between **GPT2** and **KronyPT** (i.e., we match). But more tricky for the weights that are decomposed into Kronecker Factors. In our work, we try two different approaches, Van Loan decomposition (cite here), and a Pruning based method exclusively for the 95M model.

		Datasets		
# Params	Model	wikitext-103	wikitext-2	Lambada
95M	**Krony-PT1**	41.80	35.50	61.34
95M	**Krony-PT1**	41.81	36.02	59.95

Table 3: Comparison of different models on various datasets.

3.5 Van Loan Decomposition

The Van Loan decomposition is a mathematical technique used to represent a matrix as a sum of Kronecker products, which is particularly useful in numerical linear algebra for simplifying various matrix computations. This method is named after Charles F. Van Loan, who developed this approach.

3.6 Overview

The Van Loan decomposition provides a framework to express a matrix Z in terms of Kronecker products of smaller matrices. This is beneficial in applications such as signal processing, systems theory, and solutions to tensor equations where the representation of a matrix in a reduced form can significantly simplify computations and analyses.

3.7 Decomposition Process

The basic principle of the Van Loan decomposition is to express a matrix Z as a sum of Kronecker products of smaller matrices. The general form of the decomposition is given by:

$$Z = \sum_{i=1}^{k} A_i \otimes B_i, \tag{1}$$

where A_i and B_i are matrices of appropriate dimensions, and $\otimes$ denotes the Kronecker product. The challenge lies in determining the matrices A_i and B_i such that the sum of their Kronecker products faithfully reconstructs the matrix Z.

3.8 Kronecker Product

The Kronecker product, central to this decomposition, is defined for two matrices A of size $m \times n$ and B of size $p \times q$ as:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}, \tag{2}$$

which results in a block matrix of size $(mp) \times (nq)$.

3.8.1 Van Loan Method

The Van Loan decomposition is a method based on the SVD that can be used to approximate a given matrix as a sum of k Kronecker products, optimizing the Frobenius norm of the approximation. Originated from Van Loan and Pitsianis's 1993 work on approximations with Kronecker products [12]. The algorithm described below decomposes a given matrix A, with dimensions $(..., m{\cdot}m_2, n{\cdot}n_2)$, into a sum of k Kronecker products, i.e., returns $\{U_i\}$ and $\{V_i\}$ such that $A \approx \sum_{i=1}^{k} U_i \otimes V_i$.

Algorithm 1 Kronecker Product Decomposition

```
1: **Input:** Matrix $A$, integers $m, m_2, n, n_2, k$
2: **Output:** Matrices $\{\mathrm{U}_i\}$ and $\{V_i\}$
3: Rearrange $A$ to form a new matrix for decomposition.
4: Perform low-rank SVD to approximate $A$ by $USV^T$, where $S$ contains the top $k$ singular values.
5: Reshape and scale $U$ and $V$ matrices to match the desired Kronecker product forms.
6: Return $U$ and $V$ matrices scaled by the square root of the singular values.
```

The returned matrices, suitable for reconstructing A with minimized Frobenius error. The matrices U and V are of shape $(..., k, m, n)$ for U and $(..., k, m_2, n_2)$ for V. A simple Python script to generate the $\mathrm{U}_s and V_s is as follows:$

```
def kronecker_decompose(A, m: int, n: int, *, k: int = 1, niter: int = 10):

    m2, n2 = A.shape[-2] // m, A.shape[-1] // n

    A = rearrange(A, "... (m m2) (n n2) -> ... (m n) (m2 n2)", m=m, m2=m2, n=n, n2=n2)

    u, s, v = torch.svd_lowrank(A, q=k, niter=niter)

    u = rearrange(u, "... (m n) k -> ... k m n", m=m, n=n, k=k)
    v = rearrange(v, "... (m2 n2) k -> ... k m2 n2", m2=m2, n2=n2, k=k)

    scale = s[..., None, None].sqrt()
    return u * scale, v * scale
```

3.8.2 Pruning based Initialization

We propose a new initialization strategy by inducing sparsity in the first factor of the Kronecker Product, and prune it by half. This is equivalent to picking the second factor as $\begin{bmatrix}1\\0\end{bmatrix}$. Now, we illustrate how this procedure works with a random matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} \end{bmatrix} \xrightarrow[\text{pruning}]{} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & 0 & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \\ 0 & 0 & 0 & 0 & 0 \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

4 Experiements

4.1 Training setup

For pre-training, we follow general industry standards, namely of Chinchilla [13]. We noticed more stable loss curves with higher **batch sizes**, typically 128 batch size (attained with a gradient accumulation of 4, and each forward pass seeing a batch of 32), and constant **learning rate** of 6×10^{-5} (better than a cosine scheduler with a warm-up). After pre-training on approximately one epoch, we pick the best checkpoint and train further on $\approx$ 500k tokens per step (as suggested in the literature) which amounts in our set up (sequence length of 1024) to a batch size of 512. All experiments have been conducted using a single A100 80 GB, for one epoch of on Open Web Text [14], an open source reproduction of OpenAI's WebText dataset.

After training for approximately one epoch, we usually "fine-tune" aggressively with very large batch size (approx 500k tokens per iteration) for approx. 10% of data. We had best success with this strategy.

4.2 Results

We refer to our models with the nomenclature **Krony-PT-XM{-Y}**, with X representing the number of the parameters, and optionaly Y referring to separate checkpoints of the same model class. We namely focus on 2 class models, the **Krony-PT-81M** and **Krony-PT-95M**. We evaluate our models next-token prediction capabilities on 3 datasets, wikitext103, wikitext2 [15] and Lambada [16].

4.3 The 81M class:

Krony-PT-81M is our model class of 81M parameters, and the suffixes 1350 and 3950 refer to different checkpoints. Both models clearly outperform distilGPT2 ([17]) on the 3 datasets, with a noticeable better perplexity on Lambada [16].

		Datasets		
# Params	Model	wikitext-103	wikitext-2	Lambada
124M	GPT2	29.16	24.67	45.28
82M	DistilGPT2	44.53	36.48	76.00
81M	**KronyPT-81M-1350**	**41.98**	**34.99**	-
81M	**KronyPT-81M-3950**	-	-	**64.92**

Table 4: Perplexity results of Krony-PT and DistilGPT

		Datasets		
# Params	Model	wikitext-103	wikitext-2	Lambada
81M	**KronyPT-81M-1350**	41.98	34.99	-
81M	**KronyPT-81M-3950**	-	-	64.92
119M	TQCompressedGPT2	40.28	32.25	64.72
119M	KEPT-2	40.97	32.81	67.62

Table 5: Perplexity of Krony-PT against other Kronecker based models.

4.4 The 95M class:

4.5 Scale test

How far can we push down the size?

Convergence starts to emerge starting from 80M models. 80M with single factors is higher than 80M with small dims and high factors.

4.6 Multiple Kronecker factors, with scalers.

In this section, we study the effect of adding multiple Kronecker Factors, and additionally adding scalers. In this setting, we compare 3 models:

Setting number 1 $\rightarrow$ 96M class (i.e., the biggest model class with one factor)

Setting number 2 $\rightarrow W = \sum_{n=1}^{r} A_i \otimes B_i$

Setting number 3 $\rightarrow W = \sum_{n=1}^{r} s_i(A_i \otimes B_i)$

To keep things at a comparable level, we train this class of 92.2M parameters, attained with 4 factors of size $(256, 1024)$.

Note: Adding scalers barely adds any complexity to the parameter count, for the setting number 3, we add exactly 96 parameters, depicted as follows: $96 = (\underbrace{4}_{\text{number of factors}} \times \underbrace{2}_{\text{MLP matrices per layer}}) \times \underbrace{12}_{\text{number of layers}}$. Moreover, it doesn't add any additional latency to inference compared to only using multiple factors (without scalers), as we can multiply the scaler s_i with the first Kronecker factor A_i first and then compute the Kronecker product with B_i.

Conclusion: Adding multiple factors do not help much, but adding scalers can during training improve model performance.

		Datasets		
# Params	Model	wikitext-103	wikitext-2	Lambada
96M	**KronyPT 96M Parameters**	41.80	35.50	61.34
92.2M	**4 factors without scalers**	-	-	
92.2M	**4 factors with scalers**	**38.15**	**33.08**	**61.04**

Table 6: Effect of adding scalers to multiple Kroneckers

4.7 Embeddings freezing:

In this setting we test the impact of freezing on the model's performance. I should probb test on the 82M models. Code should be done today. We also noticed that the scalers in the case of 4 factors, – no matter what the initialization is –, always do converge to around 4. Hence, we tried to to freeze them as well at 4. And see how the learning would react.

4.8 Aggressive learning rate.

As opposed to Chinchilla recommendations, we noticed that having an aggressive learning rate helps convergence. (to be confirmed)

4.9 General trends:

Things to add here:

- Models do not get better at Wiki, but do improve on Lambada.
- Distillation is not helpful
- Does pruning help? For training and for inference?
- does re-plugging the matrix work?

5 Discussion

We clarify in this section some design choices of the other papers:

5.1 Parameter count:

The other papers ([9], [11]), count the number of parameters differently than how our method and DistilGPT2 do, which makes the comparison not tit for tat. The difference is that they do not include the output matrix in the parameter count, which is not of a negligible size (approximately 40M), especially when initialized using pre-trained GPT2 weights. Quoting from KnGPT2 [9]: "Note that number of parameters of the models are reported excluding the output embedding layer in language modelling which is not compressed, is equal to row

Parameters". TQCompressor [11] follow their lead as well, which was clarified on a GitHub issue [3]. This difference in counting, makes their 81M models, a 120M in reality. The reason why we (and GPT2 paper) don't count the output matrix is because it is identical to the embedding matrix, this known as weight tying or weight sharing. Which does not seem the case with the other papers (I have contacted the other paper for clarifications through github).

5.2 Only 3% of data

Both papers claim that they only used 3% of the training data to train their models. We argue that in this set up, limiting your training data to only a fraction does not imply a better compression/factorization method, for (the simple and) following reasons:

1. They inherit almost half the weights from the old GPT2 checkpoint that was trained for numerous epochs, and has seen the whole data.
2. They use Van Loan (VL) method to initialize the Kronecker products, hence, even, when they don't match the original checkpoint, VL is not a random initialization, some knowledge is definitely passed through the SVD.
3. The use the same output matrix as GPT2, without weight trying (one can be fairly sure that this matrix "distills" all knowledge that was learned).

Hence, we can't quantify exactly "how much knowledge" has been passed through the already trained parameters. A fair comparison would be to initialize **all the parameters** of the new compressed model with random values, and not rely on any of the other pre-trained ones. Which is clearly not the case here.

[3]Link to GitHub issue: https://github.com/terra-quantum-public/TQCompressedGPT2/issues/1

References

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[2] G. Branwen, "The scaling hypothesis," 2021.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[4] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[5] Anthropic, "The claude 3 model family: Opus, sonnet, haiku."

[6] Meta, "Llama 3."

[7] AI@Meta, "Llama 3 model card," 2024.

[8] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J.-b. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser, *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024.

[9] 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*, pp. 2116–2127, 2022.

[10] A. Edalati, M. Tahaei, A. Rashid, V. P. Nia, J. J. Clark, and M. Rezagholizadeh, "Kronecker decomposition for gpt compression," *arXiv preprint arXiv:2110.08152*, 2021.

[11] V. Abronin, A. Naumov, D. Mazur, D. Bystrov, K. Tsarova, A. Melnikov, I. Oseledets, S. Dolgov, R. Brasher, and M. Perelshtein, "Tqcompressor: improving tensor decomposition methods in neural networks via permutations," *arXiv preprint arXiv:2401.16367*, 2024.

[12] C. F. Van Loan and N. Pitsianis, *Approximation with Kronecker products.* Springer, 1993.

[13] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models. arxiv," *arXiv preprint arXiv:2203.15556*, 2022.

[14] A. Gokaslan and V. Cohen, “Openwebtext corpus.” `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

[15] S. Merity, C. Xiong, J. Bradbury, and R. Socher, “Pointer sentinel mixture models,” *arXiv preprint arXiv:1609.07843*, 2016.

[16] D. Paperno, G. Kruszewski, A. Lazaridou, Q. N. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández, “The lambada dataset: Word prediction requiring a broad discourse context,” *arXiv preprint arXiv:1606.06031*, 2016.

[17] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” in *NeurIPS* $EMC^2 Workshop$, 2019.