

Imperial College London
Department of Electrical and Electronic Engineering

Generative Language Model Compression with Algebraic Approaches

Mingxue Xu

Submitted in part fulfilment of the requirements
for the degree of Doctor of Philosophy
April 10, 2026



Statement of Originality

I hereby declare that this thesis and the work reported herein was composed by and originated entirely from me. Information derived from the published and unpublished work of others has been acknowledged in the text and references.

Mingxue Xu
April 10, 2026

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

Generative language models have empowered daily and professional AI applications, yet their internal mechanisms and deployment on affordable hardware remain underexplored. This thesis addresses generative language model compression, which serves dual purposes: understanding the inner workings of language models and enabling lightweight deployment.

The work targets three gaps: ① current methods rely on empirical heuristics rather than principled theories, lacking clear connections between compression parameters and fundamental properties such as capacity and expressivity; ② compression pipelines are bespoke scripts tightly coupled to specific cases, hindering reproducibility and systematic comparison; ③ many techniques assume abundant retraining resources, poorly matching training-free adaptation on lower-end devices.

To address ①, the thesis develops a geometric and algebraic framework that treats embeddings, attention, and feed-forward layers as compositions of linear operators on structured subspaces. This perspective reveals where low-rank structure is compatible with model semantics, unifying matrix and tensor factorisations under one taxonomy grounded in direct products, tensor products, and contractions. To address ②, the thesis introduces an object-oriented compression toolbox that operationalises these ideas through reusable abstractions, separating model management, algorithms, evaluation, and profiling behind stable interfaces for systematic comparison across open-source models. To address ③, a runtime study of tensor-train decomposition for embedding layers, which consume nearly half the parameters in sub-billion parameter models, on Raspberry Pi 5 shows that compression preserves or slightly improves accuracy while halving energy usage with negligible latency overhead.

Together, this thesis bridges theory, software engineering, and hardware-aware evaluation, illuminating language models through geometric subspaces and linear operators for controllable compression with predictable trade-offs.

Keywords: Generative Language Model, Model Compression, Tensor Decomposition, Low-rank Factorisation, Edge Computing, Geometric Transformation

Contents

1	Introduction	13
1.1	Background and Motivation	13
1.2	Model Compression - From Systematic Benefits to Intrinsic Understandings of Deep Networks	14
1.3	Current Research Gaps in Model Compression	17
1.4	Algebra for Model Compression	19
1.5	Contributions for this Thesis	20
1.6	Thesis Structure	22
2	Literature Review	23
2.1	Common Language Tasks and Their Evaluations	23
2.2	Generative Language Model Compression Based on Matrices	26
2.2.1	Low-Rank Factorisation	26
2.2.2	Structured and Semi-Structured Pruning	28
2.3	Generative Language Model Compression Without Structured Matrices Concerns . .	29
2.3.1	Unstructured Pruning	29
2.3.2	Quantisation	30
2.4	Interpretability of Neural Representations	31
2.4.1	Probing for Linguistic Knowledge	31
2.4.2	The Geometry of Representation Spaces	31
2.4.3	Spectral Analysis of Model Operators	33
2.4.4	Mechanistic Interpretability	33
2.5	Chapter Summary	36
3	Interpretation and Analysis of Generative Language Model Mechanism	39
3.1	Analysis of Model Weights and Activations	39
3.1.1	Parameter Proportion for Different Layer Types	40
3.1.2	Singular Value Analysis of Weight Matrices	42
3.1.3	Activation Through Transformer Block(s)	43
3.2	Local Geometric Understanding of Language Models	48
3.2.1	Framework Overview	48
3.2.2	Parameter Space in Model Compression	50

3.2.3	Generative Language Modelling with Parameter Vector	51
3.2.4	Formalizing Generative Language Model Compression	51
3.2.5	Weight Matrices/Tensors Composed by Subspaces	54
3.2.6	Common Algebraic Operations on Subspaces	57
3.2.7	Typical Matrix Factorisation Approaches	58
3.2.8	Tensors and Tensor Factorisation	61
3.2.9	Interpretation of Transformer Modules with Subspaces	65
3.2.10	Geometric View of the Recent Research on Low-Rank Compression	66
3.3	Global Information Transmission Through the Generative Language Model Computational Graph	70
3.3.1	Functionality Shifting Caused by Dimension Reduction	72
3.3.2	Fitting Error Summary	74
3.4	Chapter Summary	75
4	Implementation with Object-Oriented Programming	77
4.1	Relevant Open-Sourced Projects	77
4.1.1	Efficient Inference	78
4.1.2	Language Model Evaluation Harness	78
4.1.3	Representation Quality Metrics	81
4.2	Requirements and Design Principles	83
4.3	Factory Mode	84
4.4	Resource Management	89
4.5	Integration with Third-Party Projects	91
4.6	Case Study: A Full Compression-and-Evaluation Process for Qwen and MiniCPM Models	98
4.6.1	Actual Workflow	98
4.6.2	Quantitative Experimental Results and Discussion	100
4.7	Chapter Summary	103
5	Resource-Constrained Deployment and Algebraic LLMs Runtime Analysis	105
5.1	LLM-Profiler: a Fine-Grained LLM FLOPs Calculator	105
5.1.1	Core Arithmetic Operations in Model Forward Passes	105
5.1.2	Measurement for Llama3 Models	107
5.1.3	Comparative Analysis of llm-profiler and Other Profilers	107
5.2	Facing Resource-Constrained On-Device Compression	110
5.2.1	Adaptability for SLM Applications	113
5.2.2	Energy Efficiency for SLM Applications	114
5.3	Methodology	115
5.3.1	Individual Embedding Vector Compression	116
5.3.2	Language Model Inference Process with the Compressed Embeddings	116
5.3.3	Dense Layer Compression	118

5.4	Empirical Evaluation for Language Task Performance	118
5.4.1	Experimental Setup	118
5.4.2	Evaluation Metrics	119
5.4.3	Experimental Results	120
5.5	Low Energy Consumption	125
5.6	On-Device Latency	126
5.7	Chapter Summary	129
6	Conclusion	131
6.1	Contribution	131
6.2	Future Work	133

List of Tables

2.1	Summary of common language tasks and representative benchmarks discussed in this thesis. Each row shows one example from the dataset with its ground truth label. The test samples column reports the number of items in the evaluation split: tokens for language modelling, reviews for text classification, and question-answer (QA) pairs for all QA-style benchmarks. Where test set labels are not publicly available (BoolQ, HellaSwag, PIQA, WinoGrande), the validation split is used, following standard practice in the LM Evaluation Harness.	38
3.1	Transformer configuration of Llama models used for analysis, extracted from the Hugging Face model cards Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct and Meta-Llama-3.1-8B-Instruct. For these decoder-only transformers, each hidden layer contains one attention block and one feed-forward layer block.	43
3.2	Operations on subspaces (i.e. composition or decomposition) in the Euclidean space. Direct sum ($\oplus$) extends coordinates additively while tensor product ($\otimes$) extends representational capacity multiplicatively. In the following text, $\times$ is used between vector spaces (and the vectors in these vector spaces) to represent a composed vector space (a vector formed by the other two vectors), and $\otimes$ only refers to the Kronecker product. In Sec. 3.2.5, when using a vector to represent tensors, tensor product (and even tensor contraction) is prevalent with the direct product.	55
3.3	Interpretation of current literature under the proposed taxonomy. Since this section focuses on model compression, only the experimental model compression results claimed in the sections are extracted, with the compression ratio as defined in Def. 2. The additional information after the compression ratio, for example, “small”, “large”, etc. reflects the different versions of the base models. The definition of two optimisation goals, fidelity and accuracy are given in Sec. 3.2.4. “Not mentioned” in the last column means the focus of the investigated work is not model compression (e.g. acceleration or reduced computations), but intrinsically reduces the model size at the same time. All the experimental results are comparable to language task performance in the uncompressed models.	68
3.4	Fitting perplexities in Fig. 3.16a and accuracies in Fig. 3.16b.	75

4.1	Common building blocks of Llama3, Qwen3 and Gemma3 family models.	90
4.2	Managers and the resources they coordinate in toggle	91
4.3	Summary of the Comparison of Our Approach and SliceGPT	102
5.1	Differences between llm-profiler and other profilers.	108
5.2	Training-free is a crucial requirement in model compression.	112
5.3	Comparison with our approach and the relevant research.	113
5.4	Approximate energy consumption of different operations (1nJ=1000pJ). For servers, communication with the wired network (e.g. ethernet or optical fibre) is preferred; for edge devices, it is preferred to use wireless networks (e.g. Wi-Fi or cellular network).114	
5.5	Computation and memory complexity during compression and inference. The key tradeoff: storage complexity reduces from $\mathcal{O}(Vd)$ to $\mathcal{O}(VNlr^3)$ (where $V \gg l$ for vocabulary size vs. input length), while inference reconstruction incurs $\mathcal{O}(Nlr^2)$ overhead per token. This tradeoff favors scenarios where the memory and storage are the bottleneck.	116
5.6	Zero-shot reasoning performance for OPT-125M, OPT-350M and their compressed versions. The best result in each column is shown in bold , and the second best is <u>underlined</u>	123
5.7	Zero-shot reasoning performance for CerebrasGPT-111M, CerebrasGPT-256M, and CerebrasGPT-590M and their compressed versions. The best result in each column is shown in bold , and the second best is <u>underlined</u>	124
5.8	The latency (ms/token) of tensorisation & decomposition token embedding vectors and reconstruction on the high-end and lower-end devices. PPL_α means the compressed model with a negligible task performance drop, and the symbol $\varphi_{\max}$ represents the case with a maximum compression ratio. d_{emb} is the embedding dimension of the token embedding vector, and the tested models are GPT-2 and GPT-2-M. On the CPU level, for single token embedding vector decomposition and reconstruction, both server and edge devices have no significant computation overhead.	127
5.9	Parameters, number of floating-point operations (FLOPs) of the compressed and uncompressed sub-billion models, and latency on Raspberry Pi CPU. For FLOPs, the token number of the input texts is 100, while for latency on Raspberry Pi, the token number is 50.	128
5.10	Parameters, number of floating point operations (FLOPs) of the compressed and uncompressed sub-billion models, and latency on Raspberry Pi CPU. For FLOPs, the token number of the input texts is 100, while for latency on Raspberry Pi, the token number is 50.	129

List of Figures

3.1	The parameter ratio of Norms (including layer norms), feed-forward layers (FF), attention layers (Attn), and embedding layers (Emb), and the average zero-shot reasoning score on HellaSwag [1], ARC [2], BoolQ [3], and PIQA [4] of several open-source model series. In a model series, smaller models have a higher token embedding layer ratio and lower feed-forward layer ratio, while the attention layer ratio is maintained.	40
3.2	(a) Weight matrices in a transformer block. (b)–(d) Histogram of the largest 10% singular values of these weight matrices (i.e. $\mathbf{W}_*$ in (a)), averaged across transformer blocks for three Llama 3.x models. The y-axis shows frequency, and the number of transformer blocks for each model is summarised in Tab. 3.1.	42
3.3	Activations Flow in Llama 3.x Series Models	44
3.4	Token-level input and input-layernorm activations A_{in} and A_{norm} across Llama-3.x models. The t-SNE axes are scaled independently in each panel, so cross-model comparisons are based on patterns of colour mixing across transformer blocks rather than absolute spatial extent.	45
3.5	Query, key and value activations A_q , A_k and A_v for different Llama-3.x model sizes. Each panel has its own t-SNE axis ranges. For example, the apparently compact 8B A_k and A_v clouds partly reflect this rescaling, so low-rank implications are drawn from relative cluster overlap and anisotropy rather than absolute radius.	46
3.6	Attention output and post-attention norm activations A_{attn} and A_{post} in Llama-3.x models. t-SNE coordinates are re-scaled independently in each panel. The apparent collapse of 8B A_{attn} should therefore be interpreted as increased colour mixing across blocks, not as direct evidence of a smaller numerical variance.	46
3.7	Feed-forward layer gate and up-projection activations A_{gate} and A_{up} across Llama-3.x models. Because each panel uses independently scaled t-SNE axes, the dense 8B clouds reflect both increased sample density and axis rescaling. The proposed low-rank interpretation is therefore based on relative clustering structure rather than absolute size.	47

3.8	Down-projection activations A_{down} before residual addition in Llama-3.x models. Since t-SNE axes differ across panels, the visually tight 8B cloud is interpreted as indicating a more mixed manifold across blocks, rather than as a precise statement about reduced numerical variance.	47
3.9	Addressed issues, solutions and structure overview of this section.	49
3.10	Parametrization of generative neural networks in a vector space, and the process of model compression based on, taking a typical transformer [5] forwarding pass as case study.	52
3.11	Typical geometric transformations in transformer models.	56
3.12	Matrix (Cayley) multiplication and Kronecker product.	59
3.13	Geometric interpretation of typical tensor decomposition formats, taking a 3-order tensor as an example. Their vector forms are Eqs. (3.19) and (3.20). The golden cubes are the current constructed subspaces, the light green vectors are the subspace expansion represented by shared order (e.g. $\mathcal{A}^{(i)} \cap \mathbf{M}_i$ in Tucker format, $\mathcal{A}^{(i)} \cap \mathbf{G}_i$ and $\mathcal{A}^{(i)} \cap \mathcal{G}_i$ in Tensor-Train format), the dark green vectors are the current resulting vectors, inside the current constructed subspace. The red cube is the final resulted tensor, and pink planes represent resized dimension.	63
3.14	Activation flow (with annotated activations) in the LLM transformer.	70
3.15	The linear transformations before (solid arrows) and after (dashed arrows) pruning in the model. The original model maps the input embeddings E_{in} from the input space (illustrated with the shading grey ellipse) to the output embedding E_{out} in the output space, through a series of linear transformations (i.e. those defined by the weight matrices $\mathbf{W}_1, \dots, \mathbf{W}_N$). After pruning, $\mathbf{W}_i$ is converted to $\mathbf{W}'_i$ ($i \in \{1, 2, \dots, N\}$), and the original output E_{out} is shifted to E'_{out} . The final predictions (y and $\hat{y}$) are generated (normally non-linearly) from E_{out} and E'_{out} . Denote the prediction domain of the pruned model as $\hat{Y}$ and that of the unpruned model Y , the mapping $g : Y \rightarrow \hat{Y}$ shifts the model performance.	72
3.16	Fitted evaluation results. The fitting coefficients and errors are in Sec. 3.3.2.	74
4.1	Main pipelines (with involved functions), fundamental resources (models) and third-party projects in toggle . The functions with yellow background are mutually used functions in different pipelines. The dependencies with blue background are the foundation stones for the whole codebase.	83
4.2	Core factory wiring with plugin handoff in toggle	86
4.3	Workflow wired with Weight/Activation Analyser in toggle	92
4.4	Workflow wired with Language Model Evaluation Harness (lm_eval) in toggle	95

4.5	Model compression with the proposed approach on embedding layers (left side) and dense layers (right side, including feed forward layers and attention layers), for MiniCPM4-0.5B, Qwen2.5-0.5B, Qwen3-0.6B	101
5.1	FLOPs and other statistics of Llama-3.x-{1B,3B,8B,70B}-Instruct forward passes measured by <code>llm-profiler</code> . The input text length is 200, and the batch size is 16. .	107
5.2	Typical SLM layer composition and the SLM application requirement of adaptability.	110
5.3	An overview of empirical results of the proposed approach. Significant proportion of the sub-billion LLMs; in around half of the common sub-billion LLMs, their embedding layer parameters take one-third of the total model parameters. (a) Language task performance (perplexity, the lower the better) for DistilGPT2 when the compression ratio is within $3.0\times$. From the empirical evaluation on 1BW [6], when the tensor order increases, the language task performance tends to improve first, then decrease after order 3. (b) The compression trade-off of TensorSLM on different sizes models in GPT-2 and CerebrasGPT series models. This trade-off is roughly measured by the ratio between perplexity and compression ratio of embedding layers; the lower the ratio value, the better the compression. It was found that the larger the model size, the better the trade-off, where CerebrasGPT has a smoother trend compared with GPT-2. (c) Finetuning for MiniCPM4-0.5B performance recovery after embedding layers are compressed. With the finetuning sample number increases, the compressed model performance increases first, then declines.	115
5.4	Language task performance change after the compression. (a)-(d): The accuracy, precision, F1-score and recall of sentiment classification, with the increasing compression ratio of the CerebrasGPT models. The higher the values, the better the classification performance. (e)-(j): The language modelling performance and compression ratio of the GPT-2 series models, where N denotes the tensor order. The best compression cases (higher compression ratio with negligible drop in accuracy) occur when $N = 3$, implying the optimal feature representation of the token embedding vectors may be three-dimensional. (g): Different tensor decomposition approaches, where PPL means perplexity.	121
5.5	Inference energy costs of different models.	126

Loyal to the Truth.

Chapter 1

Introduction

1.1 Background and Motivation

Generative Language Models (GLMs), which are the superset of Large Language Models (LLMs), have empowered conversational assistants such as ChatGPT and Claude, supporting millions of users daily¹, providing natural language understanding and generation across domains from customer service to education. Code generation tools such as GitHub Copilot assist developers by completing code and implementing entire functions from natural language descriptions.² In scientific research, LLMs support literature review, hypothesis generation and experiments design. Medical assistants built on foundation models help clinicians interpret symptoms and recommend treatments [7]. Creative professionals use generative models for writing, design and content creation. These diverse applications share a common foundation: large scale transformer architectures trained on web scale text corpora.

This production success has been powered by remarkable advances in model architecture and training. Transformer-based generative models, introduced in 2017 [5], have scaled to hundreds or more billions of parameters for better problem-solving. Both proprietary systems (e.g. GPT-5.1 and Claude 4.5 Sonnet) and open source model families (e.g. LLaMA [8], Mistral [9] and Qwen [10]) offer increasingly sophisticated language capabilities, from zero-shot reasoning to multi-hop dialogue, long context understanding and mathematical problem-solving. Training on massive data on the Internet has enabled unseen model behaviours that were not possible before, e.g. in context learning, chain of thought reasoning and scaling laws [11].

¹As of December 2024, ChatGPT had approximately 300 million weekly active users, growing to 800 million by late 2025. See <https://www.businessofapps.com/data/chatgpt-statistics/> and <https://www.demandsage.com/chatgpt-statistics/>. Accessed 8 December 2025.

²Over 15 million developers were using GitHub Copilot by early 2025, contributing an average of 46 % of code written by users. See <https://www.wearetenet.com/blog/github-copilot-usage-data-statistics> and <https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/>. Accessed 8 December 2025.

However, the success of these systems has also confronted a tension between system capacity and system availability. The state of the art LLMs are typically deployed on high-end, specialised accelerators in data centres, where memory bandwidth, storage and power budgets are beyond the reach of most users. In contrast to such unreachable high-end systems, many emerging applications require running smaller generative models on laptops, mobile devices, or single-board computers (e.g. Raspberry Pi) where resources are typically constrained. In those cases, even models with fewer than a billion parameters can be prohibitively expensive to store, serve and update.

These resource-constrained deployment cases present several challenges that need to be addressed. First, memory footprint and storage limits on lower-end devices and embedded boards often permit only a few gigabytes of model capacity. Second, latency and throughput constraints demand fast inference on modest hardware, yet large dense matrices incur high per token delays. Third, energy consumption on battery powered devices directly impacts the user experience. Fourth, the parameter distribution and model behaviours (e.g. compressibility) of small models, which low-end devices can afford, are different from those of large models. For instance, in the sub-billion parameter models studied in this thesis, embedding layers alone can account for up to 48.08% of all parameters. On a device such as the Raspberry Pi 5, this single layer dominates storage, memory usage and energy consumption, posing a bottleneck for practical deployment.

These challenges motivate the central question addressed by model compression: how much the parameter count of language models can be reduced while their language capabilities are preserved? In the context of deep learning, compression serves two sides of both opportunities practical deployment and scientific research. On the theoretical side, compression serves as a lens for understanding generalisation, intrinsic dimensionality and the internal mechanisms of neural networks. The following section explores these dual imperatives in depth, examining why compression is both an engineering necessity and a theoretical tool for interpreting how transformers learn and represent knowledge. On the practical side, it enables deployment on lower-end devices, lowers energy consumption and the barriers to access to the most advanced language processing technologies.

To address these challenges, this thesis investigates two problems: ① how to depict an accurate picture of how language models transform information internally and compression mechanisms that respect those internal structures, and ② focusing on that small scale, resource-constrained cases, how to exploit the intrinsic low dimensional structure of language models so that sub-billion parameter models become practical, reliable programs on low-end hardware.

1.2 Model Compression - From Systematic Benefits to Intrinsic Understandings of Deep Networks

Model compression addresses both immediate deployment challenges and fundamental questions about how neural networks learn and generalise.

Practical imperatives. For generative language models, compression substantially impacts the usability of the relevant applications. Modern LLMs of billions of parameters impose severe constraints on deployment infrastructure, affecting energy consumption and inference latency. On the environmental side, training and serving large models consume massive amount of electricity, contributing to carbon emissions [12]. From a hardware perspective, serving a single multi-billion parameter model can require multiple high-end accelerators, which is only affordable for well funded organisations [13]. Inference latency compounds these challenges: autoregressive generation in large models incurs high per token delays, degrading user experience in interactive applications. By reducing parameter counts and computational graphs, compression methods enable faster iteration, lower costs and broader deployment, making state of the art language technologies accessible on resource constrained platforms.

Systematic benefits. Model compression normally aims to eliminate computational cost, deployment constraints and environmental impact. Modern transformer architectures achieve strong performance by stacking many attention and feed forward blocks, each with large projection matrices. The resulting parameter counts translate directly into memory footprints, floating points operations (FLOPs) and energy usage at inference time. For the workloads in data centre, these costs drive the need for more advanced accelerators and higher electricity budgets. For the workloads on lower-end devices, model compression most probably necessitates more in the following perspectives:

- *Memory and storage.* Consumer devices (e.g. single-board computers) often have only a few gigabytes of RAM and limited non volatile storage. A single uncompressed model can saturate these budgets, preventing multi model deployments and frequent updates.
- *Latency.* Inference on large dense matrices leads to high per token latency, especially when running on CPUs or modest GPUs. Interactive applications suffer when users must wait hundreds of milliseconds per token.
- *Energy consumption.* Sustained inference workloads on large models consume substantial energy, raising both environmental concerns and operating costs. For battery powered devices, energy consumption directly impacts user experience.
- *Privacy concerns.* Many use cases require local inference on private data, or operation in environments with limited connectivity. In those settings, sending private data to the other companies/institutes is unavailable or against the interests of the stakeholders.

Model compression mitigates these issues by reducing model size and computational complexity. Common approaches include quantisation, pruning and low-rank factorisation for weight matrices or tensors. These methods can offer large reductions in parameter count, FLOPs and memory bandwidth, sometimes with negligible loss in task performance.

Scientific foundations. From a scientific perspective, compression also poses fundamental questions about how LLMs represent and transform information. If a model can be compressed significantly without degrading performance, then many of its parameters are redundant in terms of its desired functions. Understanding where that redundancy comes from, and how it interacts with data and model architecture, is essential for designing compression schemes that are not only for model inference efficiency, but also for controllable and predictable model behaviours.

Recent theoretical work has established that model compression is not merely a practical expedient but also a path to study generalisation of machine learning models. A model that can be compressed heavily without losing performance exhibits a form of implicit regularisation: its learned function is simpler, in a quantifiable sense, than the capacity of its architecture would suggest [14]. This connection has been formalised in information theoretic bounds that relate compressibility to PAC Bayes generalisation error [15, 16]. More recently, analyses of neural network pruning, quantisation and low-rank factorisation have shown that the minimal description length of a network’s weights correlates with test performance across domains [17, 18]. In essence, compression serves as a proxy for the complexity of the learned representation, offering a measure of how much information the model genuinely requires to solve a task.

Information theoretic perspective. Beyond generalisation in machine learning theory, compression study reveals the information theoretic characteristic of learning itself. Neural networks trained on complex tasks often exhibit representations whose intrinsic dimensionality is far lower than the ambient parameter space [19, 20]. This phenomenon suggests that learning discovers low dimensional manifolds embedded in high dimensional weight spaces. Compression methods that follow this intuition, such as low rank factorisation or subspace quantisation, can preserve task performance while drastically reducing model size. The information bottleneck principle formalises this idea by framing representation learning as a trade-off between compressing inputs and retaining task relevant information. Studying compression therefore provides:

- *Quantitative information measures.* A quantitative measure of how much information is genuinely stored in each layer.
- *Redundancy localisation.* Insights into where redundancy arises, for example, whether in token embeddings, attention projections or feed forward blocks.
- *Principled budget allocation.* Principled criteria for allocating compression budgets across components based on their information content rather than heuristic rules.

Interpretability. Compression also serves as a pathway to interpretability. Understanding how models compress features, a phenomenon known as superposition, where the same set of neurons represent multiple concepts rather than a single individual concept, reveals the internal circuits and computations that implement high level behaviours [21]. Recent work has also shown that

transformers form emergent sub networks combining attention heads, MLP sub layers and residual connections to execute specific tasks. By analysing which components admit compression and which resist it, researchers can:

- *Reverse engineer computational graphs.* Identify critical pathways for reasoning by tracing information flow through compressed and uncompressed components.
- *Isolate functional circuits.* Isolate attention patterns and feature storing neurons responsible for particular linguistic or logical operations.
- *Decompose representations.* Decompose dense representations into interpretable sparse components via techniques such as sparse autoencoders.

This view transforms compression from an engineering optimisation into a scientific tool for understanding how language models function.

Scaling laws and efficient architectures. Compression study also informs scaling laws and the design of efficient deep neural network architectures. Empirical investigations of model scaling have established power law relationships between parameter count, training compute and downstream performance. However, these relationships do not account for the effective dimensionality of learned representations or the redundancy inherent in over parameterised networks. Compression provides a complementary perspective: by characterising the minimal representational capacity required for a given task, it reveals when additional parameters yield genuine expressivity gains versus mere numerical redundancy. This understanding enables:

- *Efficiency performance frontiers.* Derivation of efficiency performance frontiers that guide hardware and architecture co design.
- *Targeted scaling strategies.* Principled prediction of which layers or modules benefit most from scaling, focusing resources on components that expand effective capacity.
- *Compressibility aware training.* Development of training strategies that encourage compressible solutions without sacrificing capability.

In this view, compression is not a post-hoc intervention but an intrinsic property of well generalising models, one that should be considered from the outset of architecture and training decisions.

1.3 Current Research Gaps in Model Compression

Despite rapid progress, practical LLM compression still faces several obstacles. These obstacles motivate the research problems addressed in this thesis.

Theoretical gap. Most current compression methods are guided by empirical heuristics rather than a principled theory of representation. Matrix and tensor factorisation for generative language model compression is poorly defined and understood. In the current literature, the application side (language tasks and compression) and the methodology side (matrix and tensor factorisation) are separate, and the potential interaction between algebraic structure and deep learning is ignored. This separation manifests in several ways:

- *Disconnection from model properties.* Compression methods do not explicitly tie their approaches to fundamental network characteristics such as model capacity (how much information networks can memorise [22]) or model expressivity (what functions networks can represent [23, 24]).
- *Matrix versus tensor divide.* There is an inappropriate separation between matrix based and tensor based solutions, despite matrices being order-2 tensors mathematically. This results in difficulty comparing algorithms and identifying research gaps.
- *Inconsistent terminology in low-rank factorisation.* Matrix algebra terminology is not always consistent [25], and tensor terminology is hard to understand because it originates from diverse fields (psychometrics, physics, quantum chemistry), blocking broader adoption in the compression community.

As a result, it is rarely clear in advance which layers can be compressed, to what extent and with which impact on different language capabilities. Without an algebraic understanding of the subspaces and operators that constitute a transformer, compression ratios are typically tuned as hyperparameters rather than derived from properties of the model.

Engineering gap. From a software perspective, compression pipelines are frequently implemented as bespoke scripts attached to individual experiments. Code for loading models, applying a specific compressor and running evaluations is tightly coupled, making it difficult to reuse or extend. This fragmentation has several concerns:

- *Reproducibility.* Comparing different compression methods across models and datasets is labour intensive.
- *Integration.* Integrating new metrics or profiling tools requires rewriting considerable proportion of the existing codebase.
- *Runtime evaluation.* Comprehensive runtime evaluations (latency, FLOPs, memory usage) on specific runtime environment are often missing or not comparable.

Without reusable tooling that treats compressors, evaluators and profilers as first class components, it remains difficult to connect theoretical ideas to practical deployment decisions.

Deployment gap. Many compression algorithms rely on additional training, retraining or fine tuning. Examples include quantisation-aware training, knowledge distillation and iterative pruning with retraining. While these approaches are appealing in server settings, these methods poorly match the conditions of low-end device applications:

- *Data and compute requirements.* The low-end device applications only assume the accesses to the limited data of customised task, and sufficient compute for inference, and optimisation loops for additional training are often prohibited.
- *Weight overwriting.* Occasionally the original weights may be discarded or replaced completely to adjust the dynamic deployment environment.
- *Limited local resources.* In resource-constraint scenarios, models must be compressed or updated with limited local resources.

In contrast, training free compression methods operate directly on pretrained weights and activations. Such methods are especially attractive for on device applications, but existing training-free techniques often underperform their retraining-based counterparts, particularly on reasoning and understanding benchmarks.

Safety and reliability. Compression can also alter the functional behaviour of language models in subtle and unintended ways. Aggressive pruning or factorisation may lead to model collapse [26], unexpected backdoor behaviours [27], or shifts in calibration and robustness. Because most methods do not explicitly tie compression parameters to representation properties, these risks are difficult to quantify in advance. Setting ranks or sparsity levels becomes an exercise in trial and error rather than controlled design, leaving practitioners without clear guidance on how compression choices affect model safety.

Taken together, these gaps motivate a more principled view of compression, one that treats the internal algebraic structure of generative language models as the starting point rather than an afterthought. Such a view should support training-free methods, yield practical speedups on constrained hardware and come with software abstractions that enable systematic experimentation.

1.4 Algebra for Model Compression

“The program of a neural network is its weight matrix.” [28]

— Jürgen Schmidhuber

Transformer-based language models can be viewed as compositions of linear and nonlinear maps.

Linear operations, matrix multiplications in attention and feed forward blocks, along with embedding and output projections, account for the majority of the computational cost. Non-linearities and normalisation layers modulate these linear transformations but do not change their fundamental algebraic nature.

Recent work provides growing evidence that generative models operate on representations whose effective dimension is much lower than their ambient parameter or hidden state dimension. Low intrinsic dimensionality has been observed in the input token embedding space [29], intermediate hidden states [19] and final representation spaces [20]. These findings suggest that an explicitly geometric and algebraic perspective on the forward pass can illuminate where redundancy arises and how it can be removed.

This thesis adopts such an algebraic lens. It treats key components of the transformer, embedding layers, attention projections, feed forward blocks and residual pathways, as operators that map between subspaces of high dimensional vector spaces. Concepts such as direct products, tensor products and tensor contractions are used to unify different matrix and tensor factorisation methods under a single taxonomy. By grounding compression in this subspace-oriented view, it becomes possible to question about:

1. How compression budgets should be allocated across layers and activations? (discussed in [Sec. 3.1](#))
2. Which components have a low-rank or sparse structure that is compatible with model functions? (discussed in [Sec. 3.2](#))
3. How representation dimension, sparsity and information flow interact? (discussed in [Sec. 3.3](#))

Crucially, this algebraic perspective is not purely theoretical. It informs the design of concrete compression schemes, software abstractions, and runtime analysis developed in the following chapters. Specifically, it motivates the use of tensor-based decompositions for generative language model compression, and guides the interpretation of empirical relationships between sparsity, perplexity and accuracy.

1.5 Contributions for this Thesis

The overarching aim of this thesis is developing an algebraic foundation for controllable compression of generative language models, realising this foundation with a practical toolbox, and finally implementing in resource-constrained deployment scenarios.

More concretely, the thesis addresses two central research questions:

1. *Interpretation.* How can the internal computations of transformer based language models be

described in a human understandable algebraic language?

2. *Compression*. How to design and implement a training-free compression schemes that preserve language modelling and reasoning performance, particularly for sub-billion parameter models running on low-end devices?

Specifically, the thesis explores the three themes: ① understand generative language models, ② implement the compression algorithm, ③ deployment for real-world scenarios, especially for resource-constraint edge devices.

Understanding (Chapter 3). Develop a geometric and algebraic framework that treats transformers as compositions of linear operators on structured subspaces, and analyse how information flows through linear and nonlinear mappings. The main contributions include ① a subspace-based taxonomy unifying matrix and tensor factorisations, which is publicly accessible on arxiv [30], and ② log-linear laws that predict the compressed model performance without actually compressing them, which was presented in [31] at ICLR 2025 Workshop on Sparsity in LLMs.

Implementation (Chapter 4). Design and implement a transformer compression toolbox, which combines multiple algebraic techniques, including matrix-level SVD, tensor-level decompositions and typical structured pruning (i.e., layer removal), and supports a variety of models and language tasks. The main contribution is a factory-based object-oriented architecture with registry plugins. This enables reproducible compression pipelines validated on latest Qwen and MiniCPM model families, which serves as the experimental validation of a paper accepted by Machine Intelligence Research (public access will be available shortly).

Deployment (Chapter 5). With a focus on Tensor-Train Decompositions for embedding and dense layers, implement the whole compression pipeline and evaluate their runtime, energy and accuracy trade-offs on the real resources-constrained hardware. The main contributions include TensorSLM, a training-free tensor-based scheme achieving $0.5\times$ – $2.0\times$ embedding compression while maintaining or improving performance, and on-device validation showing theoretically 50% energy reduction on Raspberry Pi 5. This part was presented in [32] at ICML 2025 Workshop on Tiny Titans: The Next Wave of On-Device Learning for Foundational Models.

In summary, the aim of this thesis is not only to achieve good empirical compression ratios, but also to characterise the resulting functionality shifts. By relating compression parameters to representation dimension, the thesis seeks to make compression a more controllable design choice rather than an opaque hyperparameter search.

1.6 Thesis Structure

The remainder of this thesis is organised to reflect the aims outlined above. At a high level, the structure follows the three themes given in [Sec. 1.5](#).

[Chapter 2](#) surveys the relevant literature on language modelling, interpretability, and model compression. It introduces common language tasks and evaluation metrics, reviews work on the geometry and intrinsic dimensionality of neural representations, and catalogues existing approaches to low rank factorisation, pruning and quantisation. This chapter identifies the theoretical, practical, and engineering gaps that motivate the algebraic perspective adopted in this thesis.

[Chapter 3](#) develops a geometric and information theoretical framework for understanding transformer-based language models. It formalises how parameters and activations can be viewed as vectors in high-dimensional spaces, analyses local operators such as attention and multilayer perceptron blocks, and studies global information transmission through the computational graph. This framework reveals where low-rank structure is compatible with model semantics and provides a principled basis for compression.

[Chapter 4](#) presents the object-oriented implementation that operationalises the algebraic ideas. It first reviews open source inference engines and evaluation harnesses, then derives software requirements for reusable compression tooling. Building on these, it introduces a factory-based architecture that encapsulates model managers, compressors, metrics backends and plugins behind stable interfaces, and demonstrates the framework through case studies on modern small language models.

[Chapter 5](#) investigates deployment on resource-constrained devices and quantifies the runtime and energy consumption of the proposed compression schemes. This chapter also introduces a profiling toolkit for computing fine grained FLOPs counts, describes the training-free tensor compression method, TensorSLM, and evaluates compressed models on language modelling, classification and zero-shot reasoning benchmarks. Runtime measurements on Raspberry Pi and server-level hardware, with the estimated reduced energy consumption.

Finally, [Chapter 6](#) summarises the main theoretical and practical contributions of this thesis, discusses their implications for controllable compression of language models, and outlines the directions for future work. These include hybrid compression schemes combining tensor decompositions with pruning, and adaptive compression policies that respond to dynamic deployment constraints.

Chapter 2

Literature Review

This chapter reviews relevant work that motivates and contextualises the algebraic framework developed in this thesis. Standard language tasks and their detailed evaluation metrics are clarified in [Sec. 2.1](#), which sets the foundation for predicting model performance after compression without actually compressing the model later on in this thesis ([Sec. 3.3](#)). Then, the recent compression techniques are surveyed and discussed in [Secs. 2.2](#) and [2.3](#), with an emphasis on the methods that exploit low-dimensional structure in parameter and activation spaces. Finally, this chapter reviews research in interpretability that analyses neural representations through mathematical views in [Sec. 2.4](#).

Putting the above together, these three perspectives serve as the prelude for the unified algebraic framework presented in subsequent chapters.

2.1 Common Language Tasks and Their Evaluations

In the context of natural language processing, a **token** is a basic unit of text—such as a word, subword, or character—used as the model’s input. A **text (sequence)** is an ordered list of tokens that represents a sentence or passage for processing. After going through a look-up table (i.e. “tokenizer”), the input text sequence is then expressed by a numerical list, so-called token embeddings. Since transformer models lack inherent awareness of token order (they process the token embeddings in parallel), they rely on positional embeddings to provide information about each token’s location within the text sequence. Positional embeddings are added to token embeddings, for informative sequential patterns.

Language modelling and perplexity. Language Modelling is to predict the next token in a sequence. Perplexity directly measures how well the model’s probability distribution matches the true data distribution, while accuracy measures exact token prediction correctness. This is the primary training objective for most generative language models, e.g. the training process in

GPT series models [33]. A considerable number of language model downstream tasks use language modelling as their first prior training aim, with many other complex tasks (e.g. complex reasoning tasks) as benchmarks to improve the model capabilities. A formal definition of generative language modelling is as follows.

Denote the token sequence $(x_1, x_2, \dots, x_t)$ as a vector $\mathbf{x}$, the probability of token sequence $\mathbf{x}_i$ of length i is $y = p(\mathbf{x}) = \prod_{i=1}^t p(\mathbf{x}_i | \mathbf{x}_{i-1})$, where $\mathbf{x}_i = (x_1, \dots, x_i, 0, \dots, 0)$.

For a given neural network model f , denote by $\mathbb{X}$ the input space of f , which contains the token sequences. Denote by $\mathbb{Y}$ the output probability space of f and $\forall y \in \mathbb{Y}, y \in [0, 1]$. Given some loss function $\mathcal{L}$, the loss of a model prediction $\hat{\mathbf{y}} = f(\mathbf{x})$ and the ground truth label $\mathbf{y}$ is $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$. In this way, the learning objective is $\min \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$.

A lower perplexity score indicates that the language model is better at predicting the sample text, or in other words, better at language modelling. Formally, given a token sequence $S = (x_1, x_2, \dots, x_{|S|})$ and a language model that assigns conditional probabilities $p(x_i | x_1, \dots, x_{i-1})$, the perplexity is defined as

$$\text{PPL}(S) = \left(\prod_{i=1}^{|S|} p(x_i | x_1, x_2, \dots, x_{i-1}) \right)^{-1/|S|}. \quad (2.1)$$

This formulation is extended in the context of model compression in Secs. 3.3.1 and 5.4.2.

Language tasks that are evaluated by accuracy. In the context of language models, there are plenty of tasks using accuracy as their evaluation metric.

One of the most common tasks is **Text Classification**, such as sentiment analysis, spam detection, topic categorisation, and intent recognition, where models assign input text to predefined categories/classes. Accuracy measures the percentage of texts correctly classified into their true categories, and accuracy is the most straightforward and widely-used metric for these tasks.

Another common task evaluated by accuracy is **Multiple Choice Question Answering**. Standardised benchmarks include MMLU [34], HellaSwag [1], and ARC [2] where models select the correct answer from several options. Accuracy simply measures the proportion of questions answered correctly, providing a clear performance indicator that's easily comparable across different models and datasets.

While accuracy is intuitive, it can be misleading, especially for datasets with imbalanced classes. A more detailed evaluation is often performed using a confusion matrix, which breaks down predictions into True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN):

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \quad (2.2)$$

Eq. (2.2) applies directly to binary classification tasks such as sentiment classification benchmark IMDB [35] in Tab. 2.1. For the multiple-choice benchmarks in Tab. 2.1, accuracy is more generally

the proportion of questions answered correctly:

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^N \mathbf{1}[\hat{c}_i = c_i], \quad (2.3)$$

where N is the number of questions, c_i is the ground-truth answer for question i , and $\hat{c}_i$ is the model’s prediction. In the LM Evaluation Harness [36], $\hat{c}_n$ is determined by selecting the candidate completion with the highest (optionally byte-length normalised) log-probability under the language model. For a more nuanced assessment, metrics like Precision, Recall, and F1-score are used:

- *Precision* measures the accuracy of positive predictions: $\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$;
- *Recall* (or Sensitivity) measures the model’s ability to identify all positive instances, and its calculation is $\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$;
- *F1-score* is the harmonic mean of Precision and Recall, providing a single score that balances both: $\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$.

Metrics for text generation. For generative tasks like machine translation, summarisation, long-form question answering, and dialogue, evaluation is more complex, as there can be multiple correct outputs. Historically, automatic evaluation relied on lexical n -gram overlap metrics such as BLEU (Bilingual Evaluation Understudy) [37] and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [38], which compares the generated text to one or more human-written references by measuring overlap of n -grams (sequences of n tokens). Recent leaderboards and benchmarks, however, have shifted toward semantically-aware and preference-based metrics that align better with human judgments:

- *Lexical metrics*, like BLEU and ROUGE, remain widely reported for machine translation and summarisation, but they mostly capture superficial similarity and are insensitive to paraphrases, style, or discourse-level correctness.
- *Embedding-based neural metrics*, such as BERTScore [39], BLEURT [40], and COMET/COMETKiwi [41, 42] compare system outputs and references in a learned embedding space. These metrics are now standard in recent WMT machine translation leaderboards¹ because they achieve substantially higher correlation with human judgments than BLEU/ROUGE.
- *LLM-as-a-judge and preference-based metrics* are increasingly used for open-ended generation (instruction following, reasoning, dialogue). Benchmarks and frameworks such as the LM Evaluation Harness [36] and recent instruction-following evaluations [43] use strong reference models to score or compare system outputs on dimensions like helpfulness, factuality, and safety.

¹<https://www2.statmt.org/wmt25/translation-task.html>

Overall, state-of-the-art text generation evaluation for language models has shifted from purely lexical metrics to human preference-based metrics that better reflect semantic adequacy and human preferences, especially on more recent open-ended benchmarks.

2.2 Generative Language Model Compression Based on Matrices

Generative language model compression aims to reduce the computational and memory demands of language models while maintaining the performance on language tasks. As the models grow larger, deployment becomes challenging due to hardware limitations. Effective compression is crucial for democratising generative language model access, and enabling real-world applications where computational efficiency takes the first priority.

Though the current literature normally categorises generative language model compression as three mainstream approaches: quantisation, pruning, and low-rank factorisation, their detailed methodologies can be categorised into two types, depending on whether they work on matrices or not: ① working on matrices, including low-rank factorisation and structured/semi-structured pruning, and ② not working on matrices, including quantisation and unstructured pruning. This section introduces the methods in ①, and ② will be introduced in [Sec. 2.3](#).

2.2.1 Low-Rank Factorisation

Low-rank factorisation for language model compression, breaks data structures into smaller ones (matrices or tensors).

Basic principle. The core principle is that a large matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ can be approximated by the product of two or more smaller matrices, i.e., $\mathbf{W} \approx \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{m \times r}$, $\mathbf{B} \in \mathbb{R}^{r \times n}$, and the rank $r \ll \min(m, n)$. This decomposition reduces the parameter count from $m \times n$ to $r(m + n)$, leading to significant memory savings and potential inference speedups. In generative language models, this technique is primarily applied to the large weight matrices found in feed-forward networks, attention projections, and embedding layers.

Matrix and tensor formulations. The most direct method is matrix decomposition, where a single weight matrix is split into two lower-dimensional ones, often conceptually based on Singular Value Decomposition (SVD) to find the optimal low-rank approximation [47, 48]. A well-known application of this principle is parameter sharing in ALBERT [49], which factorises the large vocabulary embedding matrix. A second category involves more complex tensor decomposition methods, where a weight matrix is first reshaped (or “tensorised”) into a higher-order tensor and then decomposed using techniques like Tucker, Tensor-Train, or CP decomposition [50, 51]. These methods can

achieve higher compression ratios by exploiting multi-dimensional redundancy. Finally, some works employ **structured matrix factorisation**, such as using Monarch matrices, which decompose a dense matrix into a product of sparse, structured matrices [52].

Algebraic view of matrix-based methods. Viewed through the geometric taxonomy, the matrix- and tensor-based approaches can be compared along how they carve the parameter space into subspaces and manipulate these subspaces. Matrix-based methods such as GroupReduce, DRONE, ALBERT and related work [47, 53, 49, 48, 54] typically operate with a small number of subspaces (often two), effectively splitting one subspace into two lower-dimensional subspaces or shrinking existing subspaces. This corresponds to factorising a matrix into a product of smaller matrices while preserving the original layer interface. Methods based on Kronecker product [55, 56] further exploit separable structure by rewriting weight matrices as Kronecker products of smaller factor matrices, reducing parameter counts while still acting similar functions when representing embedding, attention, and feed-forward layers. Structured matrix constructions like Monarch [52] and the subspace-reallocation strategies in Wang et al. [57], Qiu et al. [58] can be interpreted as generating additional first-order subspaces and reallocating dimensionality across them, trading dense isotropic geometry for a more anisotropic but hardware-friendly (especially for GPUs) structure.

Algebraic view of tensor-based methods. Tensor-based approaches, by contrast, deliberately increase the number of subspaces and treat layers as high-order tensors. A typical strategy is to map modes to dimensions such as heads, channels, or sequence positions. Works like Tensorised Embedding Layers [50], tensorised transformers for language modelling [59], and subsequent BERT-oriented tensorisations [51, 60, 61] “unfold” conventional matrices into tensors with multiple modes and then perform Tucker, Tensor-Train, or block-wise decompositions. Geometrically, this increases the number of low-dimensional subspaces while reducing their complexity in each individual subspace, enabling very high compression ratios (e.g., up to $7.88\times$ on LSTMs and $2\times$ on transformers in Hrinchuk et al. [50]), but often at the cost of more complex optimisation landscapes. These tensor methods mainly target attention and feed-forward layers, and the embedding layers are partially or fully tensorised only when vocabulary sizes make the embedding layers the dominant parameter bottleneck.

Fidelity and accuracy as objective functions. Another important concern for model compression with low-rank factorisation is the balance between fidelity-driven and accuracy-driven objectives. Here, fidelity refers to how closely the compressed model’s weight matrices or intermediate representations reproduce those of the original, uncompressed model, typically measured by reconstruction error such as the Frobenius norm. Early matrix-based approaches, such as GroupReduce and Kronecker decompositions [47, 55], explicitly optimise reconstruction error (e.g. in Frobenius norm) and then fine-tune for task accuracy, resulting in checkmarks in both columns of the optimisation-goal axis. More recent works increasingly prioritise downstream accuracy and treat fidelity as a soft con-

straint, reflecting the empirical observation that aggressively compressed, but well fine-tuned models can match or even surpass language task baselines, with substantial parameter reduction. Tensor-based methods have the potential to push this trend further - they can avoid strict reconstruction objectives (i.e. fidelity) and instead treat tensor ranks and subspace counts as hyperparameters to be tuned against validation performance. Overall, the recent relevant research demonstrated a gradual shift from analytically motivated, fidelity-preserving matrix decompositions to more empirical, accuracy-first tensor and structured-matrix designs that prefer heavier geometric distortion in exchange for much higher compression ratios.

2.2.2 Structured and Semi-Structured Pruning

Pruning aims to reduce model complexity by identifying and directly removing redundant parameters. The theoretical basis for this lies in measuring parameter importance, a concept pioneered by [62] using second-order derivative information. For generative language models, these importance metrics have been adapted to account for the specific architectures of transformers. While simple magnitude-based pruning, which removes weights with small absolute values, is a common baseline [63], more advanced methods now incorporate gradient and activation data to better identify inconsequential parameters.

Structured pruning. Structured pruning removes entire architectural components, such as attention heads, layers, or neurons, which preserves the model’s regular structure for valid computation on standard hardware. This approach is justified by the functional redundancy observed in many parts of a transformer. The main challenge in structured pruning is selecting which structures to remove to maximise compression without significantly harming performance, and this is often formulated as a combinatorial optimisation problem.

There are several components in structured pruning that are removed or shrunk. *Attention head pruning* removes specific heads within the multi-head attention layers. Research by [64, 65] showed that many heads are redundant or specialised in ways that make them disposable without major performance loss, allowing for significant compression. *Layer-wise pruning* is a more aggressive technique that removes the entire transformer layers. Methods like LayerDrop [66] train models to be robust to layer removal, making it easier to identify and discard less important layers. To avoid the potential loss of representational capacity, layer-wise pruning is often combined with knowledge distillation [67, 68]. Finally, *width pruning* reduces the dimensionality of model components, such as the hidden size of feed-forward networks, by identifying and removing redundant neurons [69].

Semi-structured pruning. Semi-structured pruning offers a compromise between the fine-grained flexibility of unstructured pruning and the hardware efficiency of structured pruning. It imposes regular sparsity patterns that can be accelerated by specialised hardware (typically GPUs).

One popular technique is *N:M sparsity*, where N out of every M consecutive weights are kept. For example, a 2:4 sparsity pattern achieves 50% sparsity while remaining compatible with hardware accelerators like sparse tensor cores [70]. Another approach is *block-wise sparsity*, which prunes entire blocks of weights, offering a balance between hardware efficiency and flexibility. *Regularisation-based methods* induce semi-structured sparsity during training by using techniques like group Lasso regularisation [71, 72], which utilises sparsity in a structured way and avoids a separate pruning step.

Comparison between structured pruning and semi-structured pruning. Structured pruning offers superior hardware efficiency but can lead to drastic performance loss, which is caused by the superposition of the knowledge/reasoning capabilities existing in deep neural networks (will be further introduced in Sec. 2.4.4). Semi-structured pruning provides a more balanced trade-off, often preserving performance better while still allowing for considerable computational speedups. However, since semi-structured pruning must fit 2:4 or 4:8 patterns exactly, it is only suitable for standard NVIDIA hardware (or similar), useless elsewhere.

2.3 Generative Language Model Compression Without Structured Matrices Concerns

As categorised in Sec. 2.2, model compression without structured matrices concerns include unstructured pruning and quantisation.

2.3.1 Unstructured Pruning

Unstructured pruning removes individual weights from a network, creating sparse matrices that can be accelerated with specialised hardware. The most common approach is *magnitude-based pruning*, which removes weights with the smallest absolute values, a simple and effective heuristic for reducing redundancy [63]. An iterative process of pruning and fine-tuning (i.e. iterative magnitude pruning) allows for high levels of sparsity without significant performance loss. More advanced techniques use *gradient-based criteria*, which assess a weight’s importance by its impact on the loss function. These methods, originating from concepts like optimal brain damage [62], use second-order information to guide pruning and have been adapted for large-scale transformer architectures [73, 74].

A particularly influential idea in unstructured pruning is the *Lottery Ticket Hypothesis*, which posits that dense networks contain smaller, sparse subnetworks (“winning tickets”) that, when trained in isolation from the start, can match the performance of the full network [75]. This finding challenged the conventional need for overparameterization. While initial applications were computationally expensive, subsequent research has made finding these tickets more practical for generative language

models. Techniques like “early bird tickets” identify promising subnetworks early in training [76], and gradual magnitude pruning offers a more stable removal process [77]. Furthermore, research has shown that these “winning tickets” can be transferred across different tasks [75, 78], benefiting the practical deployment of pruned models.

Dilemma of unstructured pruning for transformers. While unstructured pruning is mathematically superior for preserving accuracy, it provides almost no real-world speedup on commonly used hardware like GPUs, because GPUs are designed for dense matrix multiplication.

2.3.2 Quantisation

Quantisation reduces the precision of model weights and activations from standard 32-bit floating-point (FP32) to lower bit-widths, such as 8-bit integers (INT8), 4-bit integers (INT4), or even binary precisions. This process achieves substantial memory savings and computational speedups, making generative language models more accessible for deployment on lower-end devices and reducing inference costs in production environments [63, 79].

Quantisation schemes. Neural network quantisation can be categorised into uniform and non-uniform schemes. Uniform quantisation is the most common approach, which maps continuous values to discrete levels with equal spacing using a scale factor and a zero-point [80]. This affine mapping is computationally efficient and is defined as $r = S(q - z)$, where r is the real value, q is its quantised integer representation, S is the scaling factor, and z is the zero-point. Non-uniform quantisation allows for variable spacing to better capture the distribution of weights and activations, but often requires more complex hardware support. Quantisation can be applied post-training, also known as Post-Training Quantisation (PTQ), which is fast but may degrade accuracy. Another quantisation during training, Quantisation-Aware Training (QAT), which typically yields better performance at the cost of more complex and resource-intensive training [81].

Quantisation in generative language models. Early work on generative language model quantisation focused on 8-bit precision as a practical compromise between model size and performance. A key breakthrough was the identification of emergent outlier features in large models, which are crucial for performance and must be handled carefully. [82] showed that by keeping a small fraction of these outlier weights in higher precision (mixed-precision quantisation), 8-bit models could achieve performance nearly identical to their FP16 counterparts.

The pursuit of more aggressive compression led to 4-bit quantisation methods. GPTQ [73] adapted principles from optimal brain damage [62] to develop a layer-wise PTQ method that achieves high accuracy at 4-bit precision. Building on this, QLoRA [83] combined 4-bit quantisation with Low-Rank Adaptation (LoRA) to enable the fine-tuning of very large models on consumer-grade hardware, a

significant step in democratising generative language model research. Other advanced techniques like ZeroQuant [84] and group-wise quantisation [85] have further refined low-bit quantisation by developing more sophisticated schemes to handle the unique weight distributions in transformer architectures.

Challenges in transformer quantisation. The attention mechanism and activations present unique challenges while applying quantisation. The dynamic range and distribution of activations require specialised quantisation schemes to avoid significant performance loss [86]. Similarly, the attention mechanism’s softmax operation and its sensitivity to small numerical changes necessitate careful quantisation strategies to preserve the model’s ability to capture long-range dependencies [87].

2.4 Interpretability of Neural Representations

The remarkable capabilities of generative language models are coupled with a significant lack of transparency, making the field of interpretability crucial for understanding, debugging, and trusting these powerful models. Research in this area is quite diverse, but this review focuses on works that analyse the internal structure and representations of models from a mathematical perspective, providing the context for the geometric framework developed in [Chapter 3](#).

2.4.1 Probing for Linguistic Knowledge

A foundational line of interpretability research involves “probing”. Probes are typically simple models (e.g., linear classifiers) trained to predict human-understandable properties (such as part-of-speech tags or syntactic dependencies) from a model’s internal activation vectors. The success of a probe is taken as evidence that the corresponding information is encoded in the model’s representations. While simple and straightforward, probing only provides a correlational view and does not fully explain the internal causal mechanisms of models that are actually used to probe this information.

2.4.2 The Geometry of Representation Spaces

This section introduces the complementary approaches that investigate the geometric properties of the high-dimensional spaces, where model parameters and activations reside. This perspective aligns closely with the framework of this thesis, treating neural networks as geometric objects whose behaviour can be understood through the structure of their parameter manifolds and activation landscapes. Rather than viewing networks through a set of individual neurons or parameters,

this geometric approach emphasises global structural properties, symmetries, and the relationships between different regions of representation space.

Intrinsic dimensionality. A key finding in this group of works is the concept of *intrinsic dimensionality* [88], which posits that even in models with billions of parameters, the core learning dynamics and task-relevant representations occupy a much lower-dimensional manifold embedded within the full parameter space. This phenomenon suggests that the apparent high dimensionality of neural networks is largely illusory for practical purposes: the effective degrees of freedom are far fewer than the parameter count would suggest. Li et al. [88] measured intrinsic dimensionality by training networks within random low-dimensional subspaces of the full parameter space, finding that many tasks could be solved with surprisingly only a few dimensions. This measurement technique reveals that the optimisation landscape is highly structured, with successful solutions concentrated near low-dimensional submanifolds rather than distributed uniformly throughout the high-dimensional parameter space.

The practical implications of low intrinsic dimensionality are profound. Aghajanyan et al. [89] demonstrated that fine-tuning can be achieved by optimising a very small number of parameters in a random subspace, often requiring fewer than 1% of the parameters to match full fine-tuning performance. Their approach projects the gradient updates onto a random low-dimensional subspace, constraining all parameter changes to lie within this subspace throughout training. The success of this method across diverse tasks and model architectures suggests that pretrained models have a surprisingly low intrinsic dimension, and that the knowledge encoded during pretraining provides such strong geometric constraints that adaptation to new tasks requires movement along only a small number of critical directions in parameter space. This finding connects directly to parameter-efficient fine-tuning methods and provides theoretical justification for low-rank adaptation techniques.

Loss landscape geometry. This geometric viewpoint extends to the analysis of the loss landscape, where properties like the flatness of minima are studied in relation to model generalisation [90, 91]. The loss landscape perspective treats training as navigation through a high-dimensional error surface, where local minima correspond to trained models and the geometry surrounding these minima influences generalisation performance. Choromanska et al. [90] analysed the loss landscape of multilayer networks with random weights, demonstrating that under certain assumptions, local minima are likely to have similar loss values to global minima, and that the landscape exhibits a particular structure where poor local minima become very rare as network size increases. Dinh et al. [91] investigated the relationship between the sharpness of minima and generalisation, finding that flatter minima often correspond to better generalisation, though the relationship is complicated by reparameterization invariances. This line of work establishes that the local curvature of the loss landscape around converged solutions carries information about the model’s robustness and generalisation properties, connecting optimisation geometry to downstream performance.

Geometric deep learning framework. The broader field of Geometric Deep Learning seeks to exploit symmetries and geometric priors in data and model architectures, providing a rich theoretical background for such analyses [92]. Geometric Deep Learning provides a unified framework for understanding how neural networks can respect and exploit the geometric structure of their input and representation spaces, encompassing graph neural networks, equivariant networks, and transformers as instances of a general principle: architectures should preserve relevant symmetries of the problem domain. This framework emphasises that the choice of geometric structure determines which functions can be efficiently represented, and that incorporating appropriate geometric inductive biases dramatically improves sample efficiency and generalisation. The geometric perspective reframes architectural design as a problem of selecting appropriate symmetry groups and geometric priors, connecting deep learning to differential geometry, group theory, and topology.

These works collectively support the idea that a geometric analysis, focusing on subspaces and manifolds, is a powerful tool for understanding neural network behaviour. The convergence of intrinsic dimensionality measurements, loss landscape analysis, and geometric deep learning theory establishes geometry as a central organising principle for interpretability research, motivating the algebraic framework developed in subsequent chapters.

2.4.3 Spectral Analysis of Model Operators

Another relevant research area, which directly informs the statistical analysis in [Chapter 3](#), is the spectral analysis of a model’s weight matrices. By treating these matrices as linear operators, their properties can be studied through their singular value or eigenvalue spectra. Such analysis has revealed the effective dimensionality [93] in the weight matrices of pretrained transformers, often exhibiting a power-law spectral distribution [94, Figure 7]. This inherent low-rank structure is a key reason why low-rank factorisation methods for compression are effective. Analysing how these spectral properties evolve during training and vary across different layers provides insight into how the model allocates its capacity and how information is transformed as it flows through the network.

2.4.4 Mechanistic Interpretability

Mechanistic interpretability aims to reverse-engineer neural networks into human-understandable functions (e.g. concepts or algorithms), or “circuits” (the subgraphs of the neural networks). This research line seeks to identify specific, causally-verified subnetworks of components (like neurons and attention heads) that implement distinct functions, such as induction or factual recall. For example, the mathematical framework for transformer circuits proposed by [95] provides a vocabulary for describing how attention heads and MLP layers compose to perform computations. While this approach offers deep, causal explanations for specific model behaviors, it often involves specific, manual analysis of individual components. The framework in this thesis, however, takes a more

generic algebraic approach, seeking to describe the general transformational properties of layers rather than identifying specific, human-labelled circuits.

The reverse engineering perspective on neural network interpretability establishes a compelling analogy between understanding neural networks and decompiling computer programs [96]. Just as reverse engineers study compiled binary code rather than exhaustively testing all possible inputs, mechanistic interpretability seeks to leverage network parameters as a finite description of behaviour. This paradigm shift treats the network itself as the primary object of study, rather than treating it as a black-box function defined solely by its input-output behaviour. The analogy extends deeply: network parameters correspond to binary machine instructions, the architecture functions as a virtual machine specification, and neuron activations represent the program’s variable states and memory. This perspective fundamentally reframes the interpretability challenge from an empirical question of input-output relationships to a structural question of algorithmic decomposition. The curse of dimensionality, which renders exhaustive testing of high-dimensional input spaces intractable, motivates this focus on parameters as the finite, complete specification of network behaviour.

Polysemanticity and superposition. A central challenge in this reverse engineering endeavor is the phenomenon of *polysemanticity*, where individual neurons respond to multiple unrelated features, analogous to a single memory address storing different semantic variables. This phenomenon severely complicates interpretation, as the natural units of the neural network architecture do not necessarily align with the semantic units by which we would understand the computation. Elhage et al. [97] demonstrated through toy models that polysemanticity arises from *superposition*, a phenomenon where neural networks represent more features than the dimensions they have by exploiting the sparsity of feature activations. In the case of superposition, the network represents n features in a space of dimension m ($m < n$) by allocating each feature a direction in activation space, relying on the fact that only a small subset of features are active for any given input. This representation strategy allows networks to pack more computational capacity into their fixed dimensionality, but creates interference between features sharing the same neurons. The work revealed that the neural basis is not necessarily the privileged basis for interpretation, fundamentally complicating the decomposition of network activations into independently understandable components. Understanding superposition requires analysing the geometry of how features are positioned in activation space, including their angles, magnitudes, and interference patterns.

Sparse autoencoders (SAE) for feature extraction. Recent work has made substantial progress in addressing polysemanticity through *sparse autoencoders* (SAEs), which perform dictionary learning to extract monosemantic features from polysemantic neuron activations. **Dictionary learning** [98] is an unsupervised machine learning technique that decomposes high-dimensional data (such as neural network activations) into a sparse linear combination of fundamental “feature” vectors, enabling the identification of interpretable patterns often hidden in superposition. The

core idea of SAE is to train an autoencoder with a heavily overcomplete hidden layer and a sparsity penalty that encourages only a small fraction of hidden units to activate for any input. This architectural design forces the autoencoder to learn a sparse dictionary of features that can linearly reconstruct the original neuron activations. Each learned feature in the overcomplete representation tends to correspond to a single, interpretable concept, effectively disentangling the superposed features that were represented in the original neuron space. [98] pioneered this approach by decomposing a 512-neuron transformer layer into over 4,000 interpretable features representing diverse concepts such as DNA sequences, legal language, and HTTP requests. Their work demonstrated that features emerge across multiple levels of abstraction, from simple pattern matching to complex semantic concepts, and that many features exhibit striking specificity to particular domains or contexts.

Building on this foundation, Templeton et al. [99] scaled the technique to Claude 3 Sonnet, a production-scale language model several orders of magnitude larger than previous work, extracting millions of highly abstract, multilingual, and multimodal features that generalise between concrete and abstract references. This scaling revealed qualitatively new phenomena - features that respond to both textual descriptions and visual depictions of the same concept, features that activate across multiple languages while maintaining semantic coherence, and features that generalise from literal to metaphorical usage. The research uncovered features representing abstract concepts such as “secrecy”, “power dynamics” and “scientific reasoning” demonstrating that interpretable features exist even at the highest levels of semantic abstraction. Critically, this work revealed features related to safety concerns including deception, sycophancy, bias, and dangerous content, demonstrating the practical utility of mechanistic interpretability for model alignment and safety. The ability to identify and potentially modify such safety-relevant features opens new avenues for controlling model behaviour at a mechanistic level.

Community infrastructure. In parallel, Google DeepMind released Gemma Scope [100], a comprehensive suite of over 400 sparse autoencoders trained on every layer and sublayer of Gemma 2 models, totaling more than 30 million learned features. This represents a massive community resource, providing sparse autoencoders at multiple sparsity levels and hidden layer widths for systematic investigation. Utilising the novel JumpReLU SAE architecture, which improves upon traditional ReLU-based SAEs through better handling of activation distributions, Gemma Scope provides the research community with unprecedented access to interpretable features for studying the internal mechanisms of language models. The systematic coverage across all layers enables researchers to trace how features evolve and compose as information flows through the network, facilitating both detailed case studies and large-scale statistical analyses of feature behaviour.

Circuit tracing. More recently, Ameisen et al. [101] introduced a circuit tracing approach that replaces MLP layers of the original models with Cross-Layer Transcoders (CLT) [102], which are sparse dictionary learning models that read activations from the residual stream at one layer and

reconstruct the outputs at multiple subsequent layers. This methodology simplifies the model’s circuit by identifying features that persist across layers, thereby resolving “cross-layer superposition” and enabling the tracing of direct linear effects throughout the model’s depth. Ameisen et al. [101] freezes the non-linearities (non-MLP layers, including attention layers) in transformers, and maps out the causal chain for a specific prompt with *attribution graphs*. Attribution graph is a visualisation tool where ① nodes represent active features (from the transcoder), token embeddings, reconstruction errors, and output logits, and ② edges represent linear effects between these nodes. The replacement model uses linear mappings for features, and the attribution (how much one node contributes to another, measured by “indirect influence” with adjacency matrices) is a well-defined linear measurement for the model’s inner workings.

Circuit tracing extends beyond isolated feature identification to map the causal connections between features across layers, constructing end-to-end computational pathways from input to output. By training transcoders that directly map between feature spaces of different layers, the method bypasses the neuron space entirely, working in the more interpretable feature basis throughout. This approach has uncovered evidence that models perform abstract reasoning in a language-agnostic feature space before mapping these internal computations to linguistic outputs, indicating a deeper separation between conceptual processing and linguistic realization than previously estimated by the community.

Pack it up. The advances introduced by this section collectively establish reverse engineering through feature extraction as a foundational methodology for understanding the computational structures underlying generative language model capabilities, providing crucial tools for both scientific understanding and practical safety applications. The progression from identifying polyseman- ticity, to developing sparse autoencoders for feature extraction, to scaling these methods to larger commercialized models, and finally to tracing circuits across layers demonstrates the maturation of mechanistic interpretability into a systematic science of neural network understanding.

2.5 Chapter Summary

This chapter surveys relevant works on natural language tasks, compression methods and inter- pretability. It connects standard evaluation metrics, practical large language model compression techniques and mathematical views of neural representations to the geometric algebraic framework developed in later chapters.

Language tasks and evaluation metrics. The opening section [Sec. 2.1](#) formalises generative language modelling via the probabilistic definition and the associated perplexity measure. It then reviews accuracy based evaluation for classification and multiple choice tasks, including precision,

recall and F1 scores derived from the confusion matrix. For text generation, this section introduces several types of metrics, including lexical metrics, embedding-based neural metrics, generative language model-as-a-judge and preference-based metrics. Together these metrics clarify which aspects of behaviour compression and deployment experiments are expected to preserve.

Compression methods for generative language models. Secs. 2.2 and 2.3 present a taxonomy of generative language model compression that emphasises structured matrix relevant and non-structured matrix relevant approaches. Low-rank factorisation methods are reviewed first as a structured matrix relevant approach, including matrix decompositions, Kronecker style factorisations and tensor decompositions applied to embeddings, attention and feed-forward layers. Structured and semi-structured pruning methods, which are categorised as a structured matrix relevant approach, are then contrasted with unstructured pruning, which is categorised as a non-structured matrix relevant approach, focusing on the trade-off between hardware efficiency and accuracy preservation. Finally, the quantisation (Sec. 2.3.2) covers low-bit width representations, mixed precision schemes and challenges specific to transformer activations and attention. This literature motivates the tensor-based compression approach developed in subsequent chapters.

Interpretability of neural representations. Sec. 2.4 reviews interpretability work that studies internal representations through mathematical tools. It first introduces probing-based methods, which correlate activation patterns with linguistic properties, and then turns to geometric analysis of activation spaces, including intrinsic dimensionality and loss landscape structure. Mechanistic interpretability and reverse engineering perspectives are discussed in terms of circuits, polysemanticity, superposition in activation space and attribution graphs. Recent circuit tracing uses cross-layer transcoders to build interpretable attribution graphs for MLP layers while keeping attention mechanisms fixed. Together these justify the geometric algebraic framework in Chapter 3, which treats transformers as structured operators on high-dimensional vector spaces.

These diverse lines in Secs. 2.4.1 to 2.4.3 provide different windows into the inner workings of generative language models. However, they are often pursued in isolation. This motivates the need for a unified framework, as proposed in the next chapter, that uses the common language of geometric algebra to connect the static properties of weights (operators) with the dynamic geometry of activations (representations), providing a more holistic and principled view of model interpretation.

Table 2.1: Summary of common language tasks and representative benchmarks discussed in this thesis. Each row shows one example from the dataset with its ground truth label. The test samples column reports the number of items in the evaluation split: tokens for language modelling, reviews for text classification, and question-answer (QA) pairs for all QA-style benchmarks. Where test set labels are not publicly available (BoolQ, HellaSwag, PIQA, WinoGrande), the validation split is used, following standard practice in the LM Evaluation Harness.

Type	Benchmark	Cognitive Demand	Example (Input → Ground Truth)	Metric	Test Samples
Language Modelling	WikiText-2 [44]	Language modelling	<i>“Robert <unk> is an English film, television and theatre actor. He had a guest-starring role on the television series The Bill in 2000.”</i> → each token predicted from left context	Perplexity (the lower the better)	245,569 tokens
Text Classification	IMDB [35]	Pattern matching	<i>“Brilliant over-acting by Lekmesiter... the survey of the movie is an unrelenting slog.”</i> → negative	Accuracy (the higher the better)	25,000 reviews
Yes/No Question Answering	BoolQ [3]	Inferential reasoning	<i>Passage: “Persian [...] is the official language of Iran...” Question: “Do they speak Persian in Iran?”</i> → Yes		3,270 QA pairs
Multiple Choice Question Answering	ARC [2]	Scientific reasoning	<i>“Which statement best explains why magnets usually stick to a refrigerator door?” (A) smooth (B) contains iron (C) good conductor (D) has wires</i> → (B)		Easy Set (ARC-e): 2,376 QA pairs; Challenge Set (ARC-c): 1,172 QA pairs.
	HellaSwag [1]	Commonsense reasoning	<i>“The man writes over the snow covering the window of a car... Then,” (A) adds wax (B) boards ski lift (C) puts on coat (D) continues removing snow</i> → (D)		10,042 QA pairs
	PIQA [4]	Physical reasoning	<i>“Goal: To remove a grape from its stem.” (A) Pull the grape off the stem (B) Pull the stem off the grape</i> → (A)		1,838 QA pairs
	WinoGrande [45]	Commonsense reasoning	<i>“The trophy doesn’t fit into the brown suitcase because the _ is too large.” (A) trophy (B) suitcase</i> → (A)		1,267 QA pairs
	MMLU-Pro [34]	Multi-task reasoning	<i>“What is the embryological origin of the hyoid bone?” (A) 1st arch (B) 1st & 2nd arch ...</i> → (B)		12,032 QA pairs
Text Generation	WMT [46]	Pattern matching (machine translation)	<i>Source: “Le chat est sur le tapis.”</i> → <i>Ref: “The cat is on the mat.”</i>	BLEU and BERTScore (both the higher the better)	varies

Chapter 3

Interpretation and Analysis of Generative Language Model Mechanism

This chapter establishes a principled framework for understanding the internal workings of transformer-based language models, moving beyond heuristic-based model compression approaches.

This chapter makes three contributions. ① [Sec. 3.1](#) analyses parameter proportions, parameter distributions and activation geometry, demonstrating that larger models exhibit homogeneous singular spectra and that key/value activations cluster compactly while query activations spread broadly. ② [Sec. 3.2](#) proposes a unified taxonomy using subspace as a pivot concept, offering geometric intuition through visualisations of tensor decompositions ([Fig. 3.13](#)). It shows that tensor product and tensor contraction are fundamentally the same operation, distinguished only by whether the operand subspaces share a common basis, i.e. [Eq. \(3.9\)](#). ③ [Sec. 3.3](#) derives log-linear laws ([Prop. 3.2](#) and [3.3](#)) relating sparsity to perplexity and accuracy, thereby turning compression into a predictable design choice.

3.1 Analysis of Model Weights and Activations

This section presents an empirical analysis of model weights and activations across several generative language model families. It examines parameter proportion by layer type, showing that feed-forward layers become more responsible for complex reasoning as model size increases. It then analyses singular value spectra of weight matrices, revealing that larger models exhibit homogeneous spectra amenable to uniform low-rank compression. Finally, it investigates activation geometry, finding that key/value activations cluster compactly while query activations spread broadly.

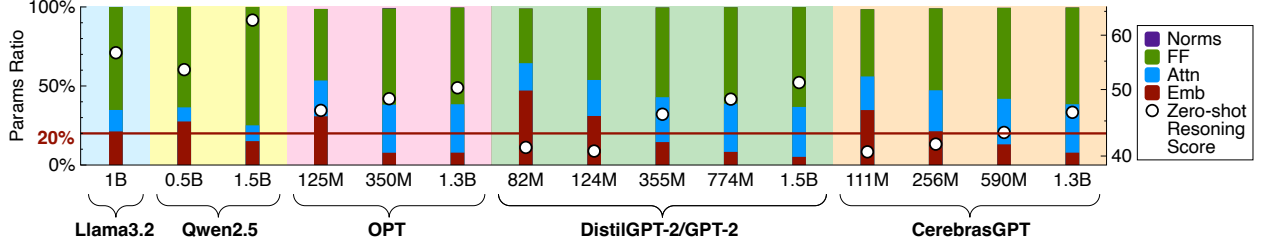


Figure 3.1: The parameter ratio of Norms (including layer norms), feed-forward layers (FF), attention layers (Attn), and embedding layers (Emb), and the average zero-shot reasoning score on HellaSwag [1], ARC [2], BoolQ [3], and PIQA [4] of several open-source model series. In a model series, smaller models have a higher token embedding layer ratio and lower feed-forward layer ratio, while the attention layer ratio is maintained.

3.1.1 Parameter Proportion for Different Layer Types

The transformer structure underpins the language task performance of generative language models. Fig. 3.1 plots, for each model within a series, the parameter ratio broken down by layer type (Norms, FF, Attn, Emb) alongside the average zero-shot reasoning score, including HellaSwag, ARC-Easy, ARC-Challenge, BoolQ and PIQA, as described in Tab. 2.1. This allows a direct comparison of architectural composition and task performance across model scales. The figure covers several popular open-sourced generative language model series, Llama3.2, Qwen2.5, OPT, DistilGPT-2/GPT-2, CerebrasGPT, with benchmark evaluations sourced from [103]. The scores shown are the highest average values among different model variants within the same pre-trained model series (e.g. models fine-tuned on different datasets), with the listed model serving as the pre-trained model.

From Fig. 3.1, it can be observed that for the same series models, with the number of parameters of the whole model increasing, the ratio of the attention layers roughly remains the same (though it decreases for DistilGPT-2/GPT-2), while that of feed-forward layers increases. The ratio of embedding layers decreases when the total parameter counts increase, though, during the ratio calculation, the embedding layer parameter counts actually increase. Regarding the benchmark scores, it can be observed that all five series show increasing scores as model size grows. The increasing scores within each model series may be attributed to two factors: ① the increasing total parameter counts; ② the dominant geometric transformations shift as the ratio of different layer types varies (the geometric transformation implemented by each layer type will be discussed in Sec. 3.2.9).

Recall that the attention layers map the input features into the value subspace $\mathbb{V}$, which is of a similar dimension as the query space $\mathbb{Q}$. However, the feed-forward layer in these generative language models typically maps the input features into a higher-dimensional latent space first, and then to a lower-dimensional subspace, whose dimension is similar to the outputs of attention layers.

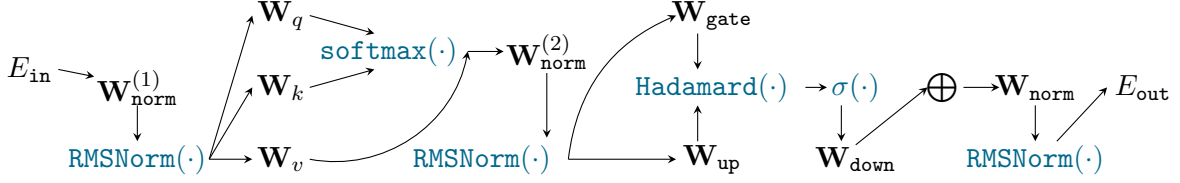
For smaller generative language models like GPT-2 , the model capability to memorise the distribution of data and more complex tasks (e.g. reasoning) does not diminish when the feed-forward layers parameter ratios decrease. From the discussion in [Sec. 3.2.5](#), it can be concluded that all kinds of matrix multiplication (the same function that the dense layers) can be seen as projections among different subspaces of the entire parameter space, thus the increasing ratio of the dense layers along with model size indicates the more and larger-scaled linear projections in GPT-2 series models. In practice, the model capability of “memorisation” (i.e. tasks relevant to memorising data like language modelling, summarisation and translation) is improved with the increasing model size [33]. This improvement can be explained as the more and larger-scaled projections help to reduce the approximation error between the predictions and actual data distribution. Such an explanation is consistent with [22, Result 5], which reports that for GPT-2 models, reducing or even eliminating all the feed-forward layers did not affect its model capacity for memorising data. However, for more complex tasks like reasoning, these larger-scale projections might introduce more noise, as the average zero-shot reasoning scores increase only modestly when the feed-forward layer ratios increase in [Fig. 3.1](#).

For generative language model series such as Llama3.2, Qwen2.5, OPT, and Cerebras-GPT, the model’s capability for more complex tasks (e.g. reasoning) improves notably when the feed-forward layer parameter ratio increases. Unlike the DistilGPT-2/GPT-2 series, the performance reports of tasks like language modelling of Llama3.2, Qwen2.5, OPT, and CerebrasGPT models could not be found, thus the memorisation capability of these models cannot be discussed. However, the model performance of more complex tasks can be explained with those shown in [Fig. 3.1](#). The increase in attention layers can be explained as the model can memorise finer-grained data distribution, and the rise of feed-forward layer ratio can be explained as the model’s ability to deal with higher-level, more abstract and implicit concepts and logic in the data, which might exist in the higher-dimension linear layers. Thus, for the benchmark scores, the model performance improves when the feed-forward layer ratio rises (sometimes also with increasing attention layers).

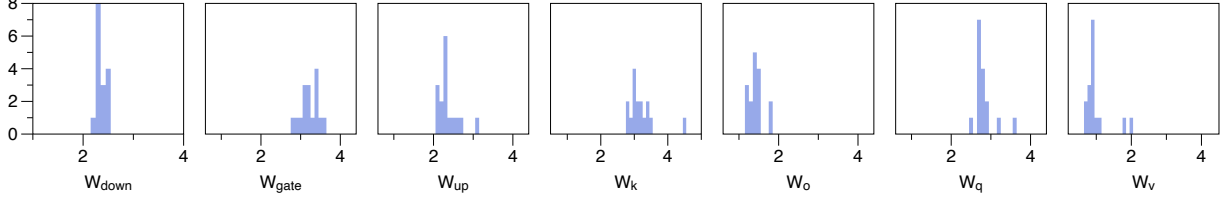
According to the above statistics and discussion, the following conclusion can be derived:

In larger generative language models, feed-forward layers are more responsible for complex reasoning capabilities.

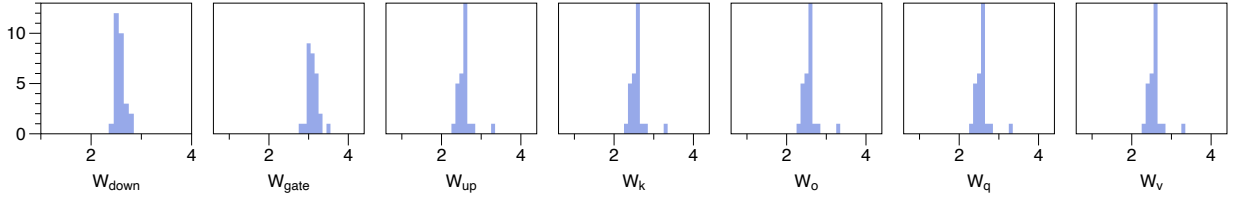
The role of the feed-forward layers in the generative language model will be further discussed in [Sec. 3.2.9](#).



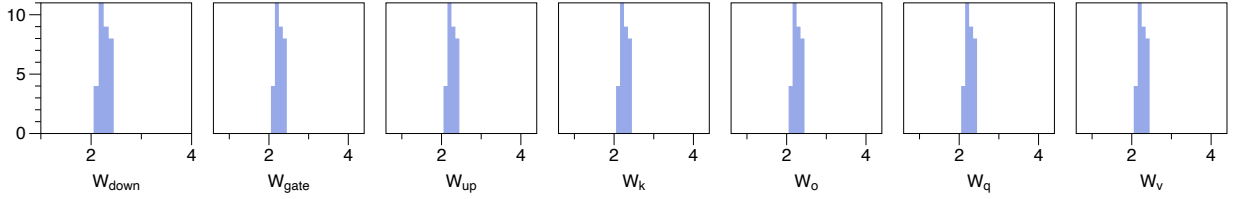
(a) Weight matrices in a single LLM transformer block, taking Llama 3.x series models as an example. A more detailed clarification is provided in [Sec. 3.3](#).



(b) Singular values of Llama-3.2-1B-Instruct weight matrices.



(c) Singular values of Llama-3.2-3B-Instruct weight matrices.



(d) Singular values of Llama-3.1-8B-Instruct weight matrices.

Figure 3.2: (a) Weight matrices in a transformer block. (b)–(d) Histogram of the largest 10% singular values of these weight matrices (i.e. $\mathbf{W}_*$ in (a)), averaged across transformer blocks for three Llama 3.x models. The y-axis shows frequency, and the number of transformer blocks for each model is summarised in [Tab. 3.1](#).

3.1.2 Singular Value Analysis of Weight Matrices

From [Fig. 3.2](#), it can be observed that the larger the models are, the more homogeneous the distribution of the singular values of the weight matrices are. This indicates more even orthogonal subspaces the weight matrices represent. This implies the larger models are more compressible, since the effective subspaces of the different weight matrices are more similar.

For Llama-3.2-1B-Instruct shown in [Fig. 3.2b](#), the mean of the largest 10% singular values varies noticeably across layer types. This suggests that, in the smallest model, modules retain distinct dominant directions and a relatively uneven spectral concentration, so effective low-rank approximations may need to be tailored per layer type (and possibly per block) rather than using a single global rank.

Table 3.1: Transformer configuration of Llama models used for analysis, extracted from the Hugging Face model cards Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct and Meta-Llama-3.1-8B-Instruct. For these decoder-only transformers, each hidden layer contains one attention block and one feed-forward layer block.

Model	# Transformer blocks	# Attention layers	# feed-forward layers	Hidden dimension
Llama-3.2-1B-Instruct	16	16	16	2048
Llama-3.2-3B-Instruct	28	28	28	3072
Meta-Llama-3.1-8B-Instruct	32	32	32	4096

For Llama-3.2-3B-Instruct shown in Fig. 3.2c, the top singular values become more tightly clustered within each layer family, and their means increase slightly and more uniformly across attention and feed-forward layer projections. As model size grows, different modules appear to calibrate toward a similar spectral scale and more closely aligned effective subspaces. Consequently, low-rank factorisation with a common rank across modules becomes more plausible, as the leading directions capture sufficient effectiveness (or, in other words, expressivity) for both attention and feed-forward layer weights.

For Llama-3.1-8B-Instruct shown in Fig. 3.2d, the means of the largest singular values are highly concentrated and almost indistinguishable across layer types, indicating a strong homogenization of spectral profiles throughout the network. This homogeneity implies that the dominant subspaces of different weight matrices overlap substantially, making the largest model particularly amenable to structured low-rank compression: a relatively small, shared rank can approximate many layers simultaneously without incurring large approximation errors.

Overall, these singular-value trends show a progression from heterogeneous, module-specific spectra in Llama-3.2-1B-Instruct to highly homogenised spectra in Llama-3.1-8B-Instruct. Smaller models, therefore, likely require layer- or block-dependent ranks or thresholds, whereas in Llama-3.2-3B-Instruct and especially Llama-3.1-8B-Instruct, a single shared rank (or a small set of ranks per layer family) can capture most of the spectral energy, enabling more uniform and structured weight-compression schemes.

In larger generative language models, the singular spectrum analysis of the weight matrices shows greater homogeneity.

This finding will be further verified by the experimental part of this thesis in Sec. 5.4.3, which indicates that “larger models are more compressible”.

3.1.3 Activation Through Transformer Block(s)

Fig. 3.3 describes how the activations flow within a single transformer block. Starting from the token-level embedding A_{in} , activations are first normalised to A_{norm} , which provides a shared, rescaled

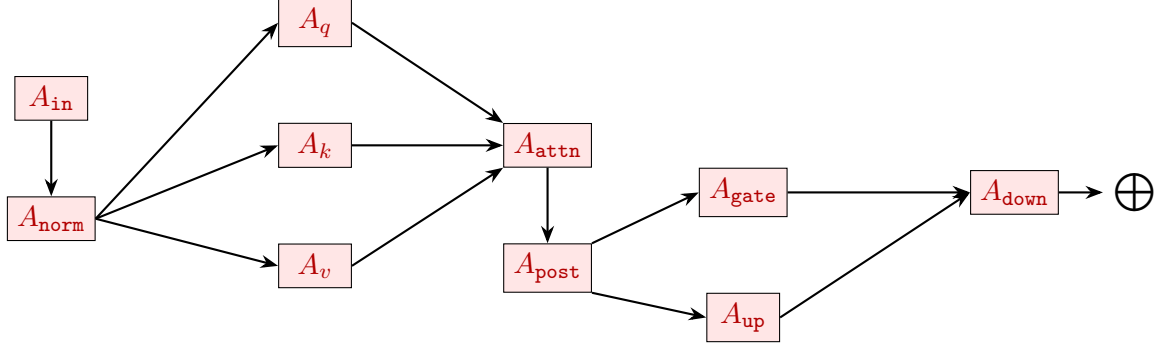


Figure 3.3: Activations Flow in Llama 3.x Series Models

representation space for subsequent projections. This normalised vector is then linearly projected into the query, key and value subspaces, producing A_q , A_k and A_v respectively. The attention mechanism combines these three streams into the aggregated attention output A_{attn} , which captures contextualised information from the entire sequence. After attention, a post-attention transformation and normalisation step produces A_{post} , aligning the attention result with the input of the feed-forward layer block. The feed-forward layer first expands A_{post} into a higher-dimensional gate pathway A_{gate} and an up-projection pathway A_{up} , allowing the model to separate feature selection from feature amplification. These two streams are recombined and projected back to the model dimension as A_{down} , which is then added via a residual connection to the attention-updated hidden state.

Figs. 3.4 to 3.8 show the t-SNE panels for different activations for Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct and Llama-3.1-8B-Instruct, which will be denoted as 1B, 3B, and 8B in the following text for simplicity, using the same input samples (16 samples from WikiText-2). Each t-SNE panel shows 3D embeddings of query-centric activations, where each point represents the activation vector of a single query. Different colours in t-SNE correspond to different transformer blocks, and the three columns compare the 1B, 3B and 8B models for the same activation type (e.g. A_{in} or A_{norm}). These plots reveal how the activations evolve with depth inside a given model, and how this evolution differs across model scales. Throughout this section, the following qualitative terminology is used for describing the t-SNE layouts:

- *Compact*: points of the same colour occupy a small, dense region, suggesting that a few shared basis directions should capture most variance under low-rank factorisation.
- *Separated*: colour clusters have little overlap and clear gaps between them, indicating that different blocks inhabit distinct subspaces and may require either higher rank or block-specific factors.
- *Dispersed*: points spread across a large volume with substantial cross-colour overlap, implying a higher intrinsic dimension where aggressive global rank reduction risks discarding important activation structure.

In Fig. 3.4, points (each corresponding to one query’s activation for the fixed input text) from

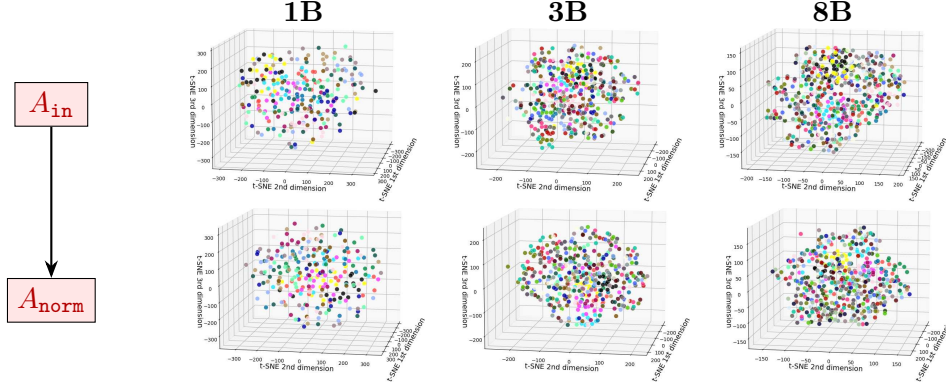


Figure 3.4: Token-level input and input-layernorm activations A_{in} and A_{norm} across Llama-3.x models. The t-SNE axes are scaled independently in each panel, so cross-model comparisons are based on patterns of colour mixing across transformer blocks rather than absolute spatial extent.

different transformer blocks largely occupy a single shared cloud for both A_{in} and A_{norm} across 1B, 3B and 8B models, with only mild tightening as model size and normalisation are applied. This heavy colour mixing suggests that early in the network, successive transformer blocks operate on a common representation manifold, mostly refining and re-scaling query activations rather than pushing them into clearly separated block-specific activation subspaces.

In Fig. 3.5, comparing columns (1B→3B→8B) for a fixed A_* suggests that larger models produce slightly tighter, more isotropic clouds, with the strongest contraction visible for A_k . This comparison is based on relative cluster shape rather than absolute axis ranges. Across rows for a fixed model size, A_q activations typically spread over a broader region than A_k and A_v , which concentrate into denser, more overlapping clusters. Together, these patterns suggest that scale is mainly used to organise key and value representations into a more regular geometry, while queries remain relatively dispersed so that different transformer blocks can interrogate the shared representation space along diverse directions. The activations of keys A_k and values A_v cluster compactly around a few dominant directions, so low-rank factorisation must preserve these axes to retain their essential structure. The activations of queries A_q , by contrast, spread broadly without concentrating along specific axes. Consequently, no particular directions are critical, and more aggressive rank reduction can be applied while still retaining sufficient directional variety for interrogating the representation space.

In Fig. 3.6, A_{attn} activations form well-separated block-wise clusters in 1B and 3B models but collapse into a single, highly mixed cloud in the 8B model, indicating that deeper attention stacks converge toward a shared representation manifold even after accounting for per-panel axis scaling. By contrast, A_{post} remains relatively spread evenly at 8B, suggesting that the post-attention layer norm reintroduces distributional diversity after attention has fused information. For low-rank factorisation, this pattern implies that aggressive compression is most appropriate on A_{attn} in larger models, where its effective dimension is already reduced, while A_{post} may require a higher retained rank to preserve its more heterogeneous structure.

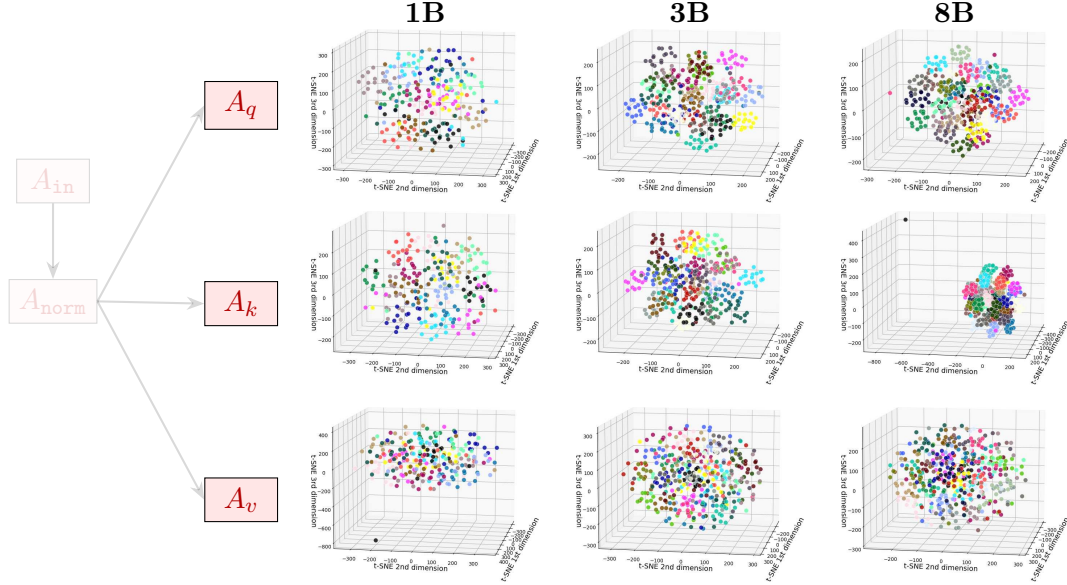


Figure 3.5: Query, key and value activations A_q , A_k and A_v for different Llama-3.x model sizes. Each panel has its own t-SNE axis ranges. For example, the apparently compact 8B A_k and A_v clouds partly reflect this rescaling, so low-rank implications are drawn from relative cluster overlap and anisotropy rather than absolute radius.

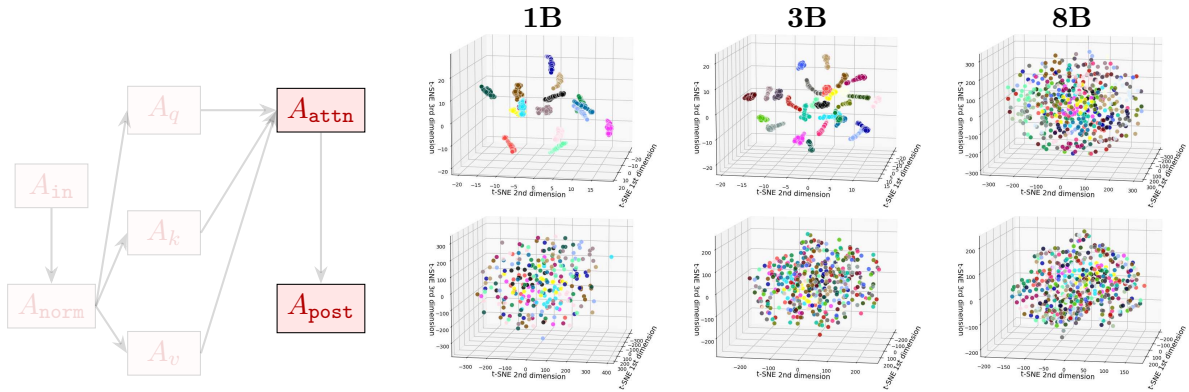


Figure 3.6: Attention output and post-attention norm activations A_{attn} and A_{post} in Llama-3.x models. t-SNE coordinates are re-scaled independently in each panel. The apparent collapse of 8B A_{attn} should therefore be interpreted as increased colour mixing across blocks, not as direct evidence of a smaller numerical variance.

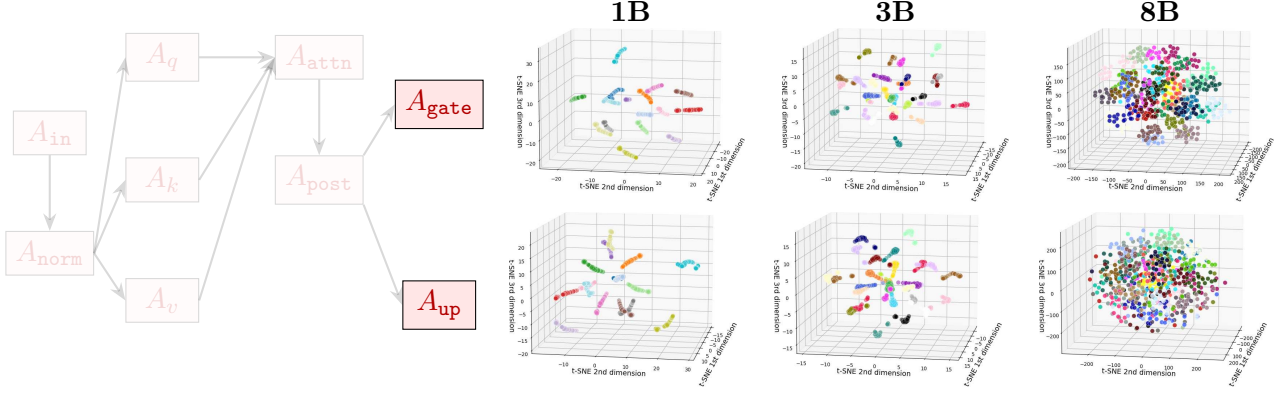


Figure 3.7: Feed-forward layer gate and up-projection activations A_{gate} and A_{up} across Llama-3.x models. Because each panel uses independently scaled t-SNE axes, the dense 8B clouds reflect both increased sample density and axis rescaling. The proposed low-rank interpretation is therefore based on relative clustering structure rather than absolute size.

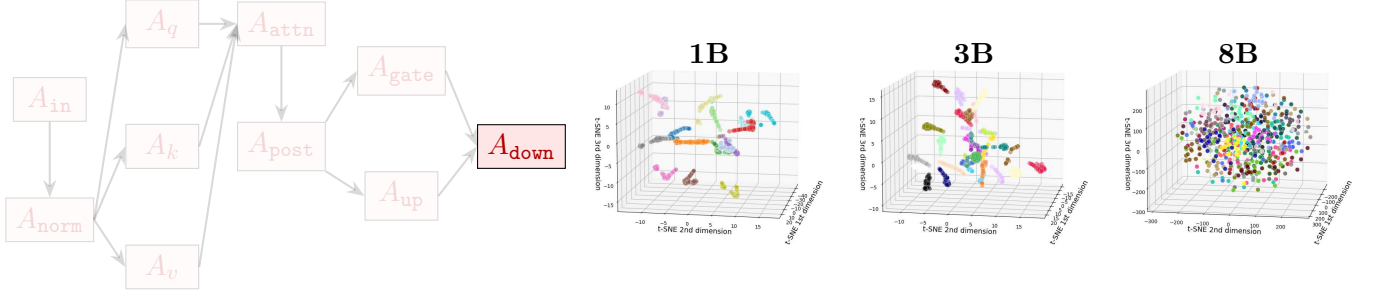


Figure 3.8: Down-projection activations A_{down} before residual addition in Llama-3.x models. Since t-SNE axes differ across panels, the visually tight 8B cloud is interpreted as indicating a more mixed manifold across blocks, rather than as a precise statement about reduced numerical variance.

In Fig. 3.7, both A_{gate} and A_{up} activations exhibit clear block-wise clusters for 1B and 3B models, which then merge into a densely packed, approximately spherical cloud in the 8B model. Within a fixed model, A_{gate} tends to separate transformer blocks slightly more than A_{up} , reflecting that gating emphasises which directions are amplified while the up-projection spreads information across a broader subspace. This suggests that in larger models, low-rank factorisation of the feed-forward layer activations can focus on a small number of dominant directions shared across blocks, particularly for A_{up} , while preserving a somewhat richer basis for A_{gate} to retain block-specific modulation.

In Fig. 3.8, the A_{down} activations for 1B and 3B models still form distinguishable block-wise strands, but these gradually coalesce into a single, highly entangled cloud in the 8B model, similar to the pattern observed for A_{attn} and A_{up} . This indicates that the final feed-forward layer output lives in a shared, low-dimensional manifold across transformer blocks, with scale primarily increasing the density of samples rather than the effective geometric complexity. Consequently, low-rank factorisation schemes that target A_{down} in larger models can reasonably employ more aggressive

rank reduction, provided they preserve a small set of global basis directions capturing this common manifold before the residual connection is applied.

3.2 Local Geometric Understanding of Language Models

This section proposes a unified taxonomy using subspace as a pivot concept to bridge matrix and tensor factorisations. It formalises parameter space, low-rank compression, fidelity and accuracy definitions, and interprets existing compression methods under this framework. A key finding is that tensor product and tensor contraction are fundamentally the same operation, distinguished only by whether the operand subspaces share a common basis (Eq. (3.9)).

This section reformulates the concepts and problems regarding deep neural network compression and natural language processing, in order to “compile” matrix and tensor factorisation under the same framework as generative language model compression – a framework built upon geometric algebra.

3.2.1 Framework Overview

Matrix and tensor factorisation for generative language model compression refers to approximating weight matrices with a set of smaller matrices or tensors (higher-dimensional matrix) to reduce the model size, thereby alleviating memory usage or computation overhead while maintaining language task performance. Since the emergence of successful solutions regarding neural network compression (see Fig. 3.9), such a factorisation approach has attracted massive interest. Compared with other model compression approaches like knowledge distillation and quantisation, matrix and tensor factorisations have unique advantages: ① they require less training effort than that based on knowledge distillation, ② they can be hardware-independent, where quantisation cannot.

However, factorisation for generative language model compression is poorly defined and understood. In the current literature, the application side (i.e., language tasks and compression) and the methodology side (matrix and tensor factorisation) are treated separately, and the potential interaction between algebraic structure and deep learning research is ignored. This includes model capacity, that is, how much information deep neural networks can memorise [22], and model expressivity, that is, what kind of functions deep neural networks can deal with [23, 24]. Furthermore, there is also an inappropriate separation between matrix-based solutions and tensor-based solutions (i.e. two separate reference lines in the central literature part of Fig. 3.9), though mathematically matrix algebra represents tensor algebra as order-2 tensors. This results in difficulties in comparing the compression algorithms and the empirical results of the two, and in identifying the current research gap. Both these two isolated issues discard the possibilities of matrix or tensor factorisation, promoting deep learning research with its algebraic properties. The third dilemma comes from

matrix and tensor research. The terminologies of matrix algebra are not always consistent (e.g. the definition of “direct sum” [25, Section 2.1]). The terminologies for tensors, on the other hand, are hard to understand since they come from various scientific fields (e.g. psychometrics, physics and quantum chemistry). These inconsistent and unfriendly terminologies block the model compression community from using these algebra structures to the full for further practical impacts.

To address the three above issues, this section proposes an elementary and unified terminology of matrix and tensor factorisation for generative language model compression. Specifically, this section utilises a concept from undergraduate-level linear algebra (also the core concept in geometric algebra), that of a *subspace*, to unify the concepts in tensor- and matrix-based approaches, as shown in Fig. 3.9. It is hoped that in this way these methodologies can become more approachable for people with broader science, technology, engineering, and mathematics backgrounds. The concepts in generative language model compression are also formalised in the context of “subspace”, e.g. parameter space, neural network compression and language modelling. The geometric interpretation of these reformulated concepts and common matrix/tensor compression methodologies (e.g. SVD, Kronecker product and tensor decomposition) is then provided, smoothly bridging compression methodologies and deep learning concepts, while aiding the understanding of tensor operations. Based on the proposed taxonomy, in Sec. 3.2.10, the recent literature is listed, their approaches and results are interpreted and compared, and the current research gaps are revealed together with suggesting further directions.

The structure of this section is shown in the middle panel of Fig. 3.9. The formalisation preliminaries and the reformulated concepts of generative language model compression are introduced in a geometric way. The interpretation of matrix and tensor factorisation methodologies is then provided geometrically. Subsequently, everything is put together, the existing relevant work is interpreted under the proposed terminology framework, compared, and the current research gap is pointed out.

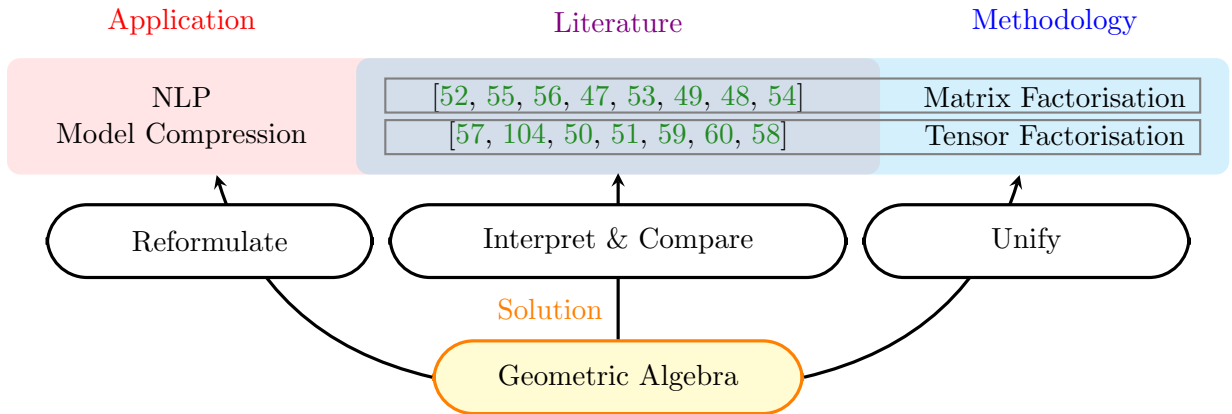


Figure 3.9: Addressed issues, solutions and structure overview of this section.

3.2.2 Parameter Space in Model Compression

A typical parametrization of neural networks [90, Section 3.3], which is followed in the proposed taxonomy, is as follows. Let $W = \{\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_l\}$ be the set of all weight matrices of a deep neural network model f , where l is the number of neural network layers. Denote by $\mathbb{R}$ the set of real numbers, and n the parameter count of the model f , yielding the upper bound of model parameter count $\text{sup}(f) = \sum_{i=1}^l |\mathbf{W}_i|$, $i = 1, 2, \dots, l$, where $|\mathbf{W}_i|$ stands for the number of elements of matrix $\mathbf{W}_i$ and $\text{sup}(f)$ is the upper bound parameter count of deep neural network model f .

As a step forward from this parametrization, the layers (i.e. layer number) and dimensionality (i.e. the concept “matrices”) of the set W , flatten the elements of all the weight matrices into a vector $\mathbf{v}$, with a comma-delimited form $\mathbf{v} = (v_1, v_2, \dots, v_n)$. It is also possible to represent $\mathbf{v}$ with $\text{sup}(f)$ -dimensional unit vectors $\mathbf{e}_1 = (1, 0, 0, \dots, 0)$, $\mathbf{e}_2 = (0, 1, 0, \dots, 0)$, $\dots$, $\mathbf{e}_n = (0, 0, 0, \dots, 1)$, like $\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \dots + v_n\mathbf{e}_n$. Here, $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$ is a basis set of the vector space which contains $\mathbf{v}$, while $\{v_1, v_2, \dots, v_n\}$ is the set of real value coefficients of vector $\mathbf{v}$.

Denote by V a set that contains all possible linear combinations of $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$ and $\{v_1, v_2, \dots, v_n\}$, and a vector space $\mathbb{V}$ which is an algebra structure that contains vector set V and scalar multiplication and vector addition on set V . For simplicity, $\mathbb{V} \subseteq \mathbb{R}^n$ denotes a vector space that contains all vectors in $\mathbb{R}^n$, where $\mathbb{R}$ is a set of real numbers. The following definition of the model parameter vector and parameter space is now given.

Definition 1 (Parameter Vector and Parameter Space). *Given a vector space $\mathbb{P} \subseteq \mathbb{R}^n$, a vector $\mathbf{v} \in \mathbb{P}$ is the parameter vector with a coefficient set $\{v_i | i = 1, \dots, n\}$. Then, $\mathbb{P}$ is called the parameter space, and $s(\mathbf{v}) = \sum_{i=1}^n (\text{sgn}(v_i))^2$ is the parameter count of $\mathbf{v}$, where sgn is the sign function.*

Note that $s(\mathbf{v})$ is the number of the non-zero elements in W , $s(\mathbf{v}) \leq \text{sup}(f)$. If $s(\mathbf{v})$ is significantly smaller than $\text{sup}(f)$, we say the weight matrices of model f are sparse. The simplest version of unstructured pruning, which has been introduced in Sec. 2.3.1, reduces the size of a neural network by removing less important parameters (i.e. zeros or nearly zero) or weights. This unstructured pruning can be viewed as removing subvectors with zero coefficients from $\mathbb{V}$. However, a systematic implementation of this, like only saving the indices of the less important parameters is outside the scope of this section.

Suppose there is another subset of V , denoted as $\tilde{V}$. The vector space built up on $\tilde{V}$ is called the subspace of $\mathbb{V}$, and denoted as $\tilde{\mathbb{V}}$. Together with Def. 1, this now gives the following formal definition of deep neural network compression.

Definition 2 (Low-rank Neural Network Compression). *Let $\tilde{\mathbb{P}}$ be a subspace of $\mathbb{P}$, and $\exists \tilde{\mathbf{v}} \in \tilde{\mathbb{P}}$, $s(\tilde{\mathbf{v}}) < s(\mathbf{v})$. The mapping $g : \mathbb{P} \rightarrow \tilde{\mathbb{P}}$ for extracting such a subspace $\tilde{\mathbb{P}}$ is called low-rank neural network compression, where $\text{CR}(\mathbf{v}, \tilde{\mathbf{v}}) = \frac{s(\mathbf{v}) - s(\tilde{\mathbf{v}})}{s(\tilde{\mathbf{v}})}$ is the compression ratio function, while a compression instance with a given threshold η , which satisfies $\text{CR}(\mathbf{v}, \tilde{\mathbf{v}}) > \eta$, is called η -compression.*

The formalisation in Def. 2 is rather general. Taking the notion of a *subspace* in a broad sense, the

compression output - smaller matrices/tensors can be represented with the same unit vectors as the original weight matrices, and the shrunk parameters can be seen as the corresponding coefficients of these changes to 0. For example, the most straightforward approach for [Def. 2](#) would be to break the original weight matrices into smaller matrices with SVD, Kronecker product [\[55, 56\]](#) or block-diagonal matrices approaches [\[52, 105, 106\]](#)

3.2.3 Generative Language Modelling with Parameter Vector

For a more general language task description rather than case-by-case, the focus is on the modelling process from [\[33\]](#), which is based on language modelling task and takes other language tasks as downstream tasks.

Definition 3 (Generative Language Modelling). *Given a sequence of tokens $(x_1, x_2, \dots, x_t)$ of length t , the goal of generative language modelling is to find a factorised joint probabilities*

$$p(x_t) = \prod_{i=1}^t p(x_i | x_1, \dots, x_{i-1}).$$

Denote the token sequence $(x_1, x_2, \dots, x_t)$ as a vector $\mathbf{x}$, the probability of token sequence $\mathbf{x}_i$ of length i is

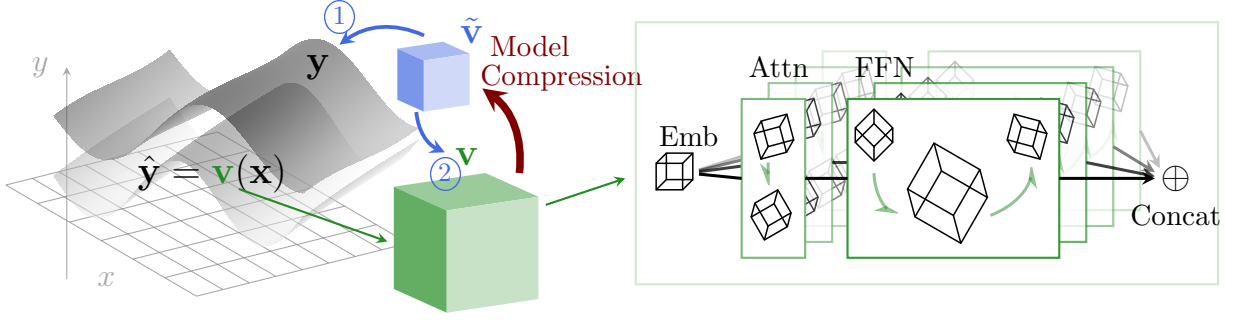
$$y = p(\mathbf{x}) = \prod_{i=1}^t p(\mathbf{x}_i | \mathbf{x}_{i-1}), \quad \mathbf{x}_i = (x_1, \dots, x_i, 0, \dots, 0). \quad (3.1)$$

For a given neural network model f and its parameter vector $\mathbf{v}$ defined in [Def. 1](#), denote by $\mathbb{X}$ the input space of f , which contains the token sequences. Denote by $\mathbb{Y}$ the output probability space of f and $\forall y \in \mathbb{Y}, y \in [0, 1]$. Given some loss function $\mathcal{L}$, we can obtain the loss of certain model prediction, $\hat{\mathbf{y}} = f(\mathbf{x})$, and the ground truth label, $\mathbf{y}$, as $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$. In this way, the learning objective for [Def. 3](#) is $\min \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$.

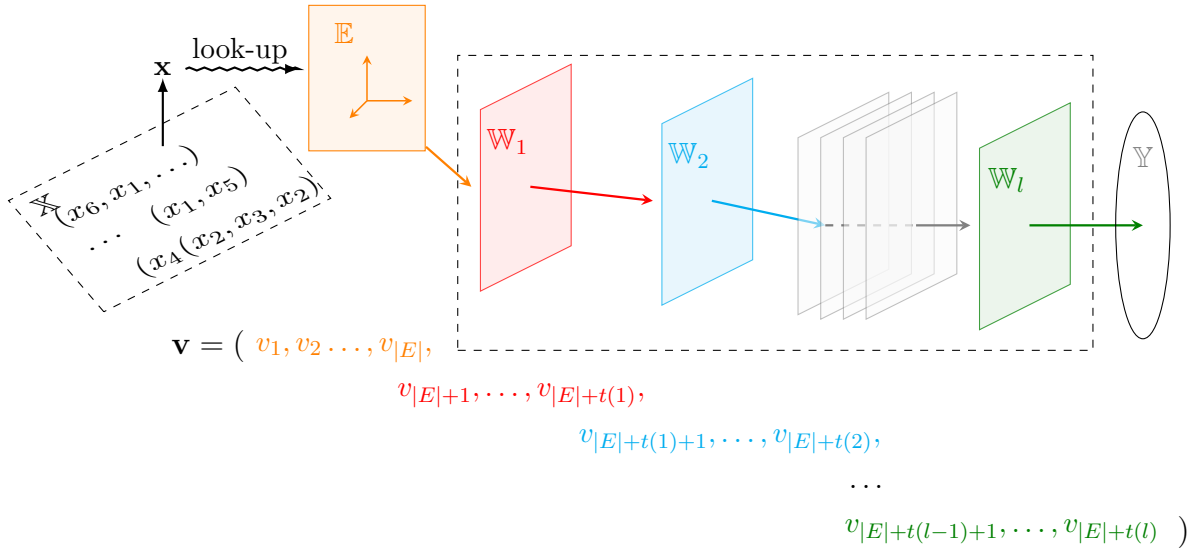
Informally, the prediction can also be represented as $\hat{\mathbf{y}} = \mathbf{v}(\mathbf{x})$, here $\mathbf{v}(\cdot)$ is a series of transformations defined by the vector $\mathbf{v}$. A vector space version of neural probabilistic language modelling [\[107\]](#) is shown in [Fig. 3.10b](#) and on the left side of [Fig. 3.10a](#).

3.2.4 Formalizing Generative Language Model Compression

Model compression is somewhat similar to data compression, that is, using fewer data/parameters to achieve similar performance or representation as the original ones. However, when the objective changes from data to a neural network model, the goals and evaluation metrics change. For data compression, the goal is to reduce the size of the original data, but to retain the necessary information that can be effectively restored to the original data. However, for model compression, except restoring to the original model weights, we care more about the task accuracy of the neural network



(a) Model compression (defined in Def. 6) in the context of the parameter vector, with a transformer structure as a case study. On the left side, the two optimisation goals for model compression are ① accuracy ($\hat{\mathbf{y}} \rightarrow \mathbf{y}$ or optimise $\text{ACC}(\mathbf{y}, \hat{\mathbf{y}})$ in Def. 4), and ② fidelity ($\mathbf{v} \rightarrow \tilde{\mathbf{v}}$ in Def. 5). The right side is the transformer structure discussed in Sec. 3.2.9.



(b) Generative language modelling with a layerwise parameter vector; $t(\cdot)$ is the index indicator function of $\mathbf{v}$, $t(i) = \sum_{k=1}^i |\mathbf{W}_k|$.

Figure 3.10: Parametrization of generative neural networks in a vector space, and the process of model compression based on, taking a typical transformer [5] forwarding pass as case study.

model that can be reserved. Thus, when talking about language model compression, the concerns for this section from both data compression and model performance perspectives are as follows:

1. **Accuracy:** To what extent, after adopting the smaller matrices/tensors into the neural network models, the model can have a similar performance (e.g. modelling the data distribution) to that of the original model;
2. **Fidelity:** The extent to which the smaller matrices/tensors can be recovered from the original ones.

This leads to the formal definition of these two model compression optimisation goals: fidelity and

accuracy.

Definition 4 (Accuracy). *Given a dataset $D = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^N$, a loss function $\mathcal{L}$ (typically returns non-negative values), and a model with parameter vector $\mathbf{v}$, the accuracy metric for low-rank language model compression is defined as the average loss over the dataset:*

$$ACC(D; \mathbf{v}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \quad \hat{\mathbf{y}}^{(i)} = \mathbf{v}(\mathbf{x}^{(i)}) \quad (3.2)$$

Definition 5 (Fidelity). *Let $g : \mathbb{P} \rightarrow \tilde{\mathbb{P}}$ be a compression mapping and $\tilde{g} : \tilde{\mathbb{P}} \rightarrow \mathbb{P}$ be a reconstruction mapping. Note that $\tilde{g}$ is generally an approximate reconstruction rather than an exact inverse of g (e.g., when g performs truncation such as in truncated SVD). The fidelity loss function $FDL(\cdot)$ is defined as the Mean Absolute Error (MAE) between the original parameter vector $\mathbf{v}$ and the reconstructed parameter vector $\hat{\mathbf{v}}$:*

$$FDL(\mathbf{v}, \hat{\mathbf{v}}) = \frac{1}{n} \sum_{i=1}^n |v_i - \hat{v}_i|, \quad \hat{\mathbf{v}} = \tilde{g}(g(\mathbf{v})). \quad (3.3)$$

When g is a truncated decomposition, $\tilde{g}$ reconstructs from the truncated representation, yielding non-zero fidelity loss in general.

By considering [Defs. 1 to 5](#) together, we obtain an integrated formalisation of generative language model compression as follows.

Definition 6 ($(\delta_1, \delta_2, \eta)$ - Language Model Compression). *Consider an input domain, $\mathbb{X}$, output domain, $\mathbb{Y}$, and a loss function, $\mathcal{L}$, for language modelling. For a dataset $D = \{(\mathbf{x}, \mathbf{y}) | \mathbf{x} \in \mathbb{X}, \mathbf{y} \in \mathbb{Y}\}$, if the following conditions are satisfied, such a compression scheme is termed as a $(\delta_1, \delta_2, \eta)$ - Generative Language Model Compression:*

1. $\forall (\mathbf{x}, \mathbf{y}) \in D, FDL(\mathbf{v}, \hat{\mathbf{v}}) < \delta_1 \wedge ACC(\mathbf{y}, \hat{\mathbf{y}}) < \delta_2$, where $\hat{\mathbf{y}} = \mathbf{v}(\mathbf{x})$;
2. for $g : \mathbf{v} \rightarrow \tilde{\mathbf{v}}$, compression ratio $CR(\mathbf{v}, \tilde{\mathbf{v}}) > \eta$.

From [Def. 6](#), if the focus is only on the accuracy of the compressed language model, $\delta_1 \rightarrow \infty$ (can be set as a large value in practice), the fidelity will be ignored during the optimisation process. Similarly, $\delta_2 \rightarrow \infty$ if only fidelity is required.

[Fig. 3.10a](#) provides an interpretation of [Def. 6](#), taking the language modelling task as an example. As addressed in [Def. 3](#), language modelling uses a model $\mathbf{v} \in \mathbb{P}$, $\mathbb{P} \subset \mathbb{R}^n$ to approximate the distribution of $\mathbf{y}$, that is $\mathbf{v}(\mathbf{x}) \rightarrow \mathbf{y}$. Model compression aims to find a vector $\tilde{\mathbf{v}}$, which lies in a subspace of $\mathbb{P}$, such that $s(\tilde{\mathbf{v}}) < s(\mathbf{v})$. There are two criteria for the process $\mathbf{v} \rightarrow \tilde{\mathbf{v}}$: ① whether $\tilde{\mathbf{v}}$ can well approximate $\mathbf{y}$, and ② whether $\tilde{\mathbf{v}}$ can be well reconstructed to $\mathbf{v}$. Criterion ② is the same criterion as for data compression, while criterion ① indicates that *language modelling is compression* [\[108\]](#).

3.2.5 Weight Matrices/Tensors Composed by Subspaces

Layer-wise parameterisation. In Defs. 1 and 2, all the parameters of a model were considered in one parameter vector for easing the problem definition. However, the current literature typically considers the factorisation of each layer, that is, factorise weights for each layer, as shown in Fig. 3.10b. Thus, for the following sections, model compression is discussed layer by layer.

For a neural network model $f = f_1 \circ f_2 \circ \dots \circ f_l$, where l is the maximum layer index of model f , and $\circ$ denotes function composition, by representing this form with parameter vectors in the parameter space $\mathbb{P}$, the following is obtained:

$$\mathbf{y} = f(\mathbf{v}, \mathbf{x}) = f_l(\mathbf{v}_l, \dots, f_3(\mathbf{v}_3, f_2(\mathbf{v}_2, f_1(\mathbf{v}_1, \mathbf{x}))) \dots), \quad (3.4)$$

$$\mathbf{v} = \sum_{i=1}^l \mathbf{v}_i, \quad \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_l \in \mathbb{P}. \quad (3.5)$$

In Eq. (3.5), the summation for $\mathbf{v}_i$ can also represent concatenation in neural networks, where outputs are appended by treating their subspaces as orthogonal and forming a larger composite space. If $\mathbf{v}_i$ is expressed in a comma-delimited form, we can define a vector index function $\text{idx}(\mathbf{v}, i) = \sum_{j=1}^i s(\mathbf{v}_j)$, which computes the cumulative parameter count up to and including layer i , along with its increment $\Delta \text{idx}(\mathbf{v}, i) = \text{idx}(\mathbf{v}, i) - \text{idx}(\mathbf{v}, i-1) = s(\mathbf{v}_i)$, representing the parameter count of layer i alone. Using these, we have

$$\mathbf{v}_i = (\underbrace{0, \dots, 0}_{\text{idx}(\mathbf{v}, i-1)}, \underbrace{v_{\text{idx}(\mathbf{v}, i-1)+1}, v_{\text{idx}(\mathbf{v}, i-1)+2}, \dots, v_{\text{idx}(\mathbf{v}, i)}}_{\Delta \text{idx}(\mathbf{v}, i)}, \underbrace{0, \dots, 0}_{\text{idx}(\mathbf{v}, l) - \text{idx}(\mathbf{v}, i)}). \quad (3.6)$$

Matrix and tensor composition. Given $\mathbf{v}_i$ as the parameter vector for the weight matrix of a concerned layer i , to form it into a matrix, a $\mathbb{P}^{(i)}$ subspace is denoted whose basis set is same as $s(\mathbf{v}_i)$, and two other subspaces $\mathbb{P}_1^{(i)}, \mathbb{P}_2^{(i)} \subseteq \mathbb{P}^{(i)}$, $\mathbb{P}_1^{(i)} \oplus \mathbb{P}_2^{(i)} = \mathbb{P}^{(i)}$, where $\oplus$ denotes the direct sum, which means the union of basis sets of two subspaces and the formation of a new vector space on this basis set union. Recall $\mathbf{v}_i$ is originally a weight matrix of layer i , $\mathbf{v}_i$ can then be considered as a vector in $\mathbb{P}$ which is composed by $\mathbb{P}_1^{(i)}$ and $\mathbb{P}_2^{(i)}$.

Then, the (j, k) th element of the matrix represented by $\mathbf{v}_i$ is $v_j v_k$, with two unit vectors $\mathbf{e}_j \in \mathbb{P}_1^{(i)}$ and $\mathbf{e}_k \in \mathbb{P}_2^{(i)}$. When considering $\mathbb{P}_1^{(i)} \subseteq \mathbb{R}^m$ and $\mathbb{P}_2^{(i)} \subseteq \mathbb{R}^n$, the vector space $\mathbb{P}^{(i)}$ composed by $\mathbb{P}_1^{(i)}$ and $\mathbb{P}_2^{(i)}$ should be contained in $\mathbb{R}^{m \times n}$. This composition is mathematically represented as $\mathbb{P}_1^{(i)} \otimes \mathbb{P}_2^{(i)} \subseteq \mathbb{R}^{m \times n}$, where $\otimes$ refers to direct product.

The above statement involves two distinct linear operations that must be carefully distinguished:

- **Direct sum ($\oplus$):** Given $\mathbb{P}_1 \subseteq \mathbb{R}^m$ and $\mathbb{P}_2 \subseteq \mathbb{R}^n$, the direct sum $\mathbb{P}_1 \oplus \mathbb{P}_2 \subseteq \mathbb{R}^{m+n}$ has dimension $\dim(\mathbb{P}_1 \oplus \mathbb{P}_2) = \dim(\mathbb{P}_1) + \dim(\mathbb{P}_2)$. This represents coordinate/dimension extension.

Table 3.2: Operations on subspaces (i.e. composition or decomposition) in the Euclidean space. Direct sum ($\oplus$) extends coordinates additively while tensor product ($\otimes$) extends representational capacity multiplicatively. In the following text, $\times$ is used between vector spaces (and the vectors in these vector spaces) to represent a composed vector space (a vector formed by the other two vectors), and $\otimes$ only refers to the Kronecker product. In [Sec. 3.2.5](#), when using a vector to represent tensors, tensor product (and even tensor contraction) is prevalent with the direct product.

Operation	Dimension change	Geometric meaning	Order change	Occurrence in this section
Direct Sum ($\oplus$)	$\dim(\mathbb{P}_1) + \dim(\mathbb{P}_2)$	Coordinate/dimension extension of vectors	0	Defs. 7 and 8 Fig. 3.11c and Eq. (3.6)
Tensor Product ($\otimes$) = Direct Product	$\dim(\mathbb{P}_1) \times \dim(\mathbb{P}_2)$	Space of bilinear maps, or outer products of vectors	+1	Defs. 7 and 8 Figs. 3.11a and 3.11b Eq. (3.11) and Thm. 2
Flatten	$\text{vec}(\mathbb{R}^{m \times n}) \rightarrow \mathbb{R}^{mn}$	Hypercube volume / calculus of dimensions	-1	Thms. 1 and 2 Eq. (3.11)

- **Tensor product ($\otimes$):** The tensor product $\mathbb{P}_1 \otimes \mathbb{P}_2 \subseteq \mathbb{R}^{m \times n}$ has dimension $\dim(\mathbb{P}_1 \otimes \mathbb{P}_2) = \dim(\mathbb{P}_1) \times \dim(\mathbb{P}_2)$. This represents the space of all bilinear maps, or equivalently, the outer products of vectors from each space.

As summarised in [Tab. 3.2](#), direct sum extends coordinates while tensor product extends the representational capacity (order) of the vectors. In this section, to avoid confusion, we use $\mathbb{P}_1 \times \mathbb{P}_2 \rightarrow \mathbb{P}$ to denote a general vector space composition process from $\mathbb{P}_1$ and $\mathbb{P}_2$ to $\mathbb{P}$, where the specific operation (direct sum or tensor product) is determined by context.

Definition 7 (Rank-1 Matrix Composition in Euclidean Space). *A rank-1 matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$ can be represented as the outer product of two vectors. Specifically, let $\mathbf{a} \in \mathbb{R}^m$ and $\mathbf{b} \in \mathbb{R}^n$. We embed these in a common vector space $\mathbb{P} = \mathbb{R}^{m+n}$ as $\tilde{\mathbf{a}} = (a_1, a_2, \dots, a_m, \overbrace{0, \dots, 0}^n)$ and $\tilde{\mathbf{b}} = (\overbrace{0, \dots, 0}^m, b_1, b_2, \dots, b_n)$. The (i, j) th element of $\mathbf{C}$ is $c_{i,j} = a_i b_j$, yielding the outer product representation:*

$$\mathbf{C} = \mathbf{a}\mathbf{b}^\top = \mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m b_1 & a_m b_2 & \dots & a_m b_n \end{bmatrix} \in \mathbb{R}^{m \times n} \quad (3.7)$$

Note that $\text{rank}(\mathbf{C}) = 1$ by construction. A general matrix of rank r can be expressed as a sum of r such rank-1 matrices, which connects directly to the SVD in [Thm. 1](#).

Two $(m + n)$ -dimensional vectors $\mathbf{a}$ and $\mathbf{b}$ are used to represent a matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$ in [Def. 7](#). An attractive property for this kind of representation is that $\mathbf{a}$ and $\mathbf{b}$ are in the same subspace, $\mathbb{R}^{m+n}$, which has fewer dimensions than $\mathbb{R}^{m \times n}$. In doing this, their interactions can be described in the whole parameter space $\mathbb{P}$ and common language model modules can be interpreted in [Sec. 3.2.9](#), which cannot be achieved when putting two vectors in disjoint subspaces $\mathbb{R}^m$ and $\mathbb{R}^n$.

The geometric intuition of matrix composition is visualised in [Fig. 3.11a](#). Suppose $\mathbb{P}$ has a basis

set $(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m, \mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n)$, so that $\mathbf{a} = \sum_{i=1}^m a_i \mathbf{v}_i$ and $\mathbf{b} = \sum_{i=1}^n b_i \mathbf{w}_i$. The subspace $\mathbb{P}_1$ is defined with the base $(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m)$, and subspace $\mathbb{P}_2$ is defined with the base $(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n)$, thus $\mathbf{a} \in \mathbb{P}_1$ and $\mathbf{b} \in \mathbb{P}_2$. Denote a linear transformation $\mathbf{C} : \mathbb{P}_1 \rightarrow \mathbb{P}_2$, such that $\mathbf{C}\mathbf{v}_i = \sum_{j=1}^n c_{ij} \mathbf{w}_j$. Here, matrix $\mathbf{C}$ acts like a dictionary mapping any vector from $\mathbb{P}_1$ to a certain vector in $\mathbb{P}_2$. During this linear transformation, vector $\mathbf{a}$ is the original operand, and vector $\mathbf{b}$ defines the actual transformation $\mathbf{C}$, or in other words, operation.

Remark 3.1. Matrix composition defined in Def. 7 can be seen as a process of transforming a vector $\mathbf{a} \in \mathbb{P}_1$ into another vector $\mathbf{c} \in \mathbb{P}_2$, where the transformation is defined by $\mathbf{b}$. Inversely, there exists a linear transformation $\mathbf{C}^\top : \mathbb{P}_2 \rightarrow \mathbb{P}_1$, with a mapping matrix as $\mathbf{C}^\top$.

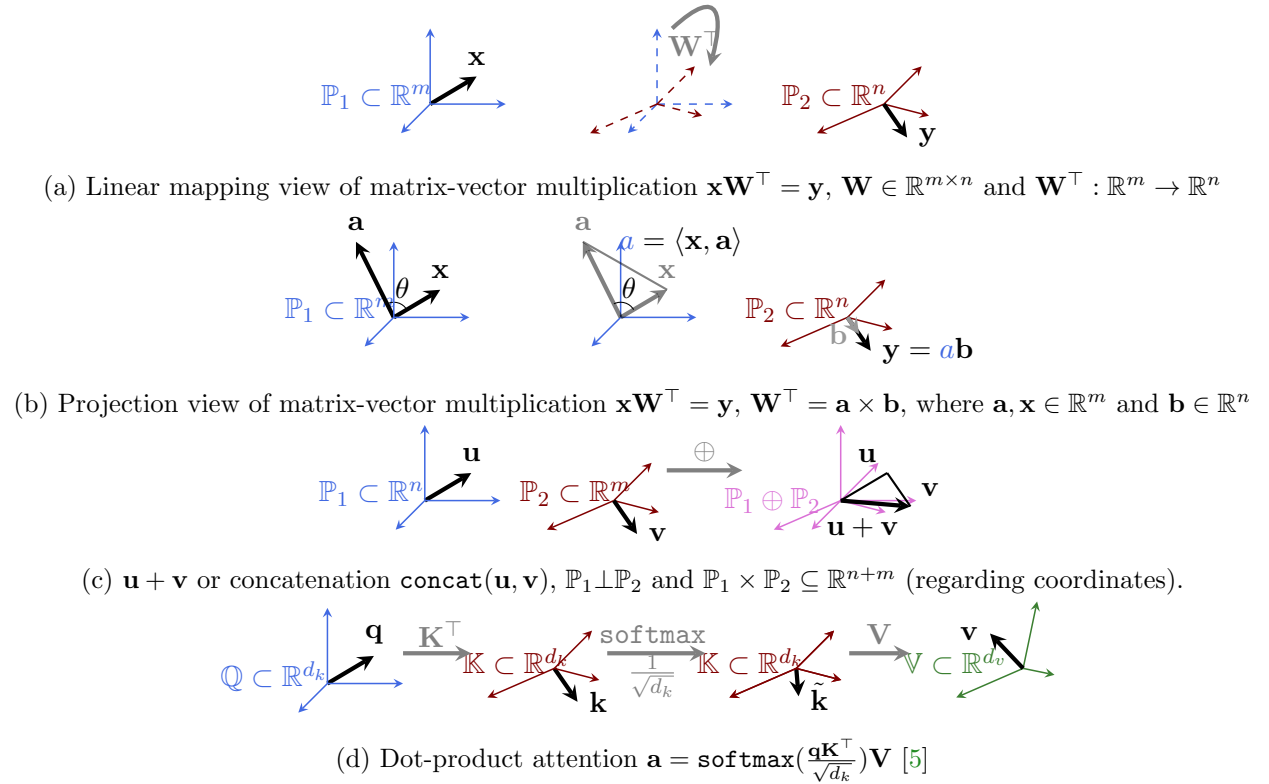


Figure 3.11: Typical geometric transformations in transformer models.

The procedure of defining the matrix with two subspaces in Def. 7 can be easily extended to tensors.

Definition 8 (Rank-1 Tensor Composition in Euclidean Space). A rank-1 n -order tensor $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$ can be represented as the outer product of n vectors. Let $\mathbf{v}^{(1)} \in \mathbb{R}^{d_1}$, $\mathbf{v}^{(2)} \in \mathbb{R}^{d_2}$, ..., $\mathbf{v}^{(n)} \in \mathbb{R}^{d_n}$. We embed these in a common vector space $\mathbb{P} = \mathbb{R}^{\sum_{i=1}^n d_i}$ with disjoint supports. The $(i_1, i_2, \dots, i_n)$ th entry of the tensor is:

$$\mathcal{A}[i_1, i_2, \dots, i_n] = v_{i_1}^{(1)} \cdot v_{i_2}^{(2)} \dots v_{i_n}^{(n)} = \prod_{k=1}^n v_{i_k}^{(k)}. \quad (3.8)$$

This is the n -way outer product $\mathcal{A} = \mathbf{v}^{(1)} \circ \mathbf{v}^{(2)} \circ \dots \circ \mathbf{v}^{(n)}$. Note that $\text{rank}(\mathcal{A}) = 1$ by construction.

A general tensor of rank r can be expressed as a sum of r such rank-1 tensors, which is the CP decomposition in [Sec. 3.2.8](#).

In [Def. 8](#), there are n subspaces, and these subspaces construct an n -order tensor. The parameter space $\mathbb{P} \subseteq \mathbb{R}^{d_1+d_2+\dots+d_n}$ can therefore represent tensors in $\mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$.

3.2.6 Common Algebraic Operations on Subspaces

The operations on subspaces in [Tab. 3.2](#) are revisited in this section, with additional clarification in the light of matrix factorisation ([Sec. 3.2.7](#)) and tensor factorisation ([Sec. 3.2.8](#)).¹

Cartesian product and tensor product on 1-order subspaces Taking a vector space $\mathbb{V} \subseteq \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$ as an example, it is an n -order vector space constructed by n 1-order subspaces $\mathbb{V}_1 \subseteq \mathbb{R}^{d_1}$, $\mathbb{V}_2 \subseteq \mathbb{R}^{d_2}$, $\dots$, $\mathbb{V}_n \subseteq \mathbb{R}^{d_n}$. The basis set of $\mathbb{V}$ is $\mathcal{V}$, $\mathcal{V} = \mathcal{V}_1 \times \mathcal{V}_2 \times \dots \times \mathcal{V}_n$, where $\mathcal{V}_i$ is the basis of $\mathbb{V}_i$, $|\mathcal{V}_i| = \dim(\mathbb{V}_i) = d_i$. Similar to what is discussed in [Sec. 3.2.5](#), this section uses the notation $\mathbb{V}_1 \times \mathbb{V}_2 \times \dots \times \mathbb{V}_n = \mathbb{V}$ to represent the process of constructing a new vector space $\mathbb{V}$ upon $\mathbb{V}_1, \mathbb{V}_2, \dots, \mathbb{V}_n$. This process is officially called *Cartesian product*, which means any vector $\mathbf{v} \in \mathbb{V}$ can be represented as a linear combination of the orthonormal vectors in $\mathcal{V}$. Specifically for [Eq. \(3.6\)](#), $\mathbf{v} = (v_1^{(1)}, v_2^{(1)}, \dots, v_{d_1}^{(1)}, v_1^{(2)}, v_2^{(2)}, \dots, v_{d_2}^{(2)}, \dots, v_1^{(n)}, v_2^{(n)}, \dots, v_{d_n}^{(n)})$, where the length of the vector representation is $|\mathcal{V}| = \dim(\mathbb{V}) = \sum_{i=1}^n d_i$, so that the representation range of the vector $\mathbf{v}$ grows to $\prod_{i=1}^n d_i$.

Tensor product is a widely used term for such dimension increase caused by “order concatenation”, whose operands are order-1 vectors, or in other words, vectors in order-1 subspaces. Obviously, direct product of subspaces in [Tab. 3.2](#) is a generalised version of the tensor product, and in the following texts refer to both cases as “product” with notation $\times$.

Product and contraction on n -order subspaces Similar to $\mathbb{V}$ in [Sec. 3.2.6](#), suppose there is another order- m vector space $\mathbb{W} \subseteq \mathbb{R}^{h_1 \times h_2 \times \dots \times h_m}$ consists of subspaces $\mathbb{W}_1 \subseteq \mathbb{R}^{h_1}$, $\mathbb{W}_2 \subseteq \mathbb{R}^{h_2}$, $\dots$, $\mathbb{W}_m \subseteq \mathbb{R}^{h_m}$, with a basis $\mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2 \times \dots \times \mathcal{W}_m$. $\mathcal{W} \cap \mathcal{V} = \emptyset$. Then, the tensor product of $\mathbb{V}$ and $\mathbb{W}$ is the same as order-1 tensor product in [Sec. 3.2.6](#), such that $\mathbb{V} \times \mathbb{W} = \mathbb{V}_1 \times \dots \times \mathbb{V}_n \times \mathbb{W}_1 \times \dots \times \mathbb{W}_m$, with a basis set $(\mathcal{V} \times \mathcal{W}) = \mathcal{V}_1 \times \dots \times \mathcal{V}_n \times \mathcal{W}_1 \times \dots \times \mathcal{W}_m$.

Contraction can be seen as a general version of tensor product. For the vector spaces $\mathbb{V}$ and $\mathbb{W}$, suppose $|\mathcal{V}_i| = |\mathcal{W}_j|$. Then, the mode- (i, j) contraction of $\mathbb{V}$ and $\mathbb{W}$ yields $\mathbb{V}_1 \times \dots \times \mathbb{V}_{i-1} \times \mathbb{V}_{i+1} \times \dots \times \mathbb{V}_n \times \mathbb{W}_1 \times \dots \times \mathbb{W}_{j-1} \times \mathbb{W}_{j+1} \times \dots \times \mathbb{W}_m$. In short, contraction “absorbs” the common subspaces shared by $\mathbb{V}$ and $\mathbb{W}$. Then the cardinality of the basis set after tensor product/contraction of $\mathbb{V}$ and $\mathbb{W}$ is

¹As noted by Gentle [25], terminology in matrix algebra varies across communities. In this section, we use “factorisation” and “decomposition” interchangeably, and distinguish direct sum ($\oplus$) from tensor product ($\otimes$) following Axler [109].

$$|\mathcal{V} \times \mathcal{W}| = |\mathcal{V} \cup \mathcal{W}| - |\mathcal{V} \cap \mathcal{W}| = \begin{cases} |\mathcal{V}| + |\mathcal{W}|, & \mathcal{V} \cap \mathcal{W} = \emptyset, \text{ tensor product,} \\ |\mathcal{V}| + |\mathcal{W}| - 2|\mathcal{V} \cap \mathcal{W}|, & \mathcal{V} \cap \mathcal{W} \neq \emptyset, \text{ tensor contraction.} \end{cases} \quad (3.9)$$

3.2.7 Typical Matrix Factorisation Approaches

Among different matrix factorisation algorithms in this section, the focus of this section is Singular Value Decomposition (SVD) and Kronecker Product.

Singular value decomposition (SVD) and matrix (Cayley) multiplication We refer to the following representation from [109, page 275, 277] for a clear interpretation of SVD via subspaces.

Theorem 1 (Singular Value Decomposition [109]). *Let $T : \mathbb{P}_1 \rightarrow \mathbb{P}_2$ be a linear map between finite-dimensional inner product spaces, with $\dim(\mathbb{P}_1) = m$ and $\dim(\mathbb{P}_2) = n$, where $m \leq n$. Then there exist:*

- an orthonormal basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\}$ of $\mathbb{P}_1$,
- an orthonormal set $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m\}$ in $\mathbb{P}_2$, and
- non-negative real numbers $s_1 \geq s_2 \geq \dots \geq s_m \geq 0$ (the singular values of T),

such that for every $\mathbf{v} \in \mathbb{P}_1$:

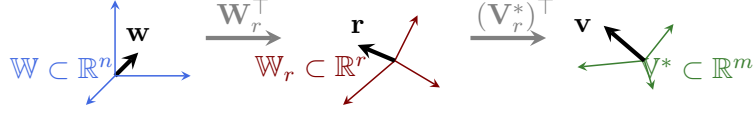
$$T(\mathbf{v}) = \sum_{i=1}^m s_i \langle \mathbf{v}, \mathbf{v}_i \rangle \mathbf{w}_i, \quad (3.10)$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product. In matrix form, if $\mathbf{v} = \sum_{i=1}^m v_i \mathbf{v}_i$, then $T(\mathbf{v}) = \sum_{i=1}^m s_i v_i \mathbf{w}_i$.

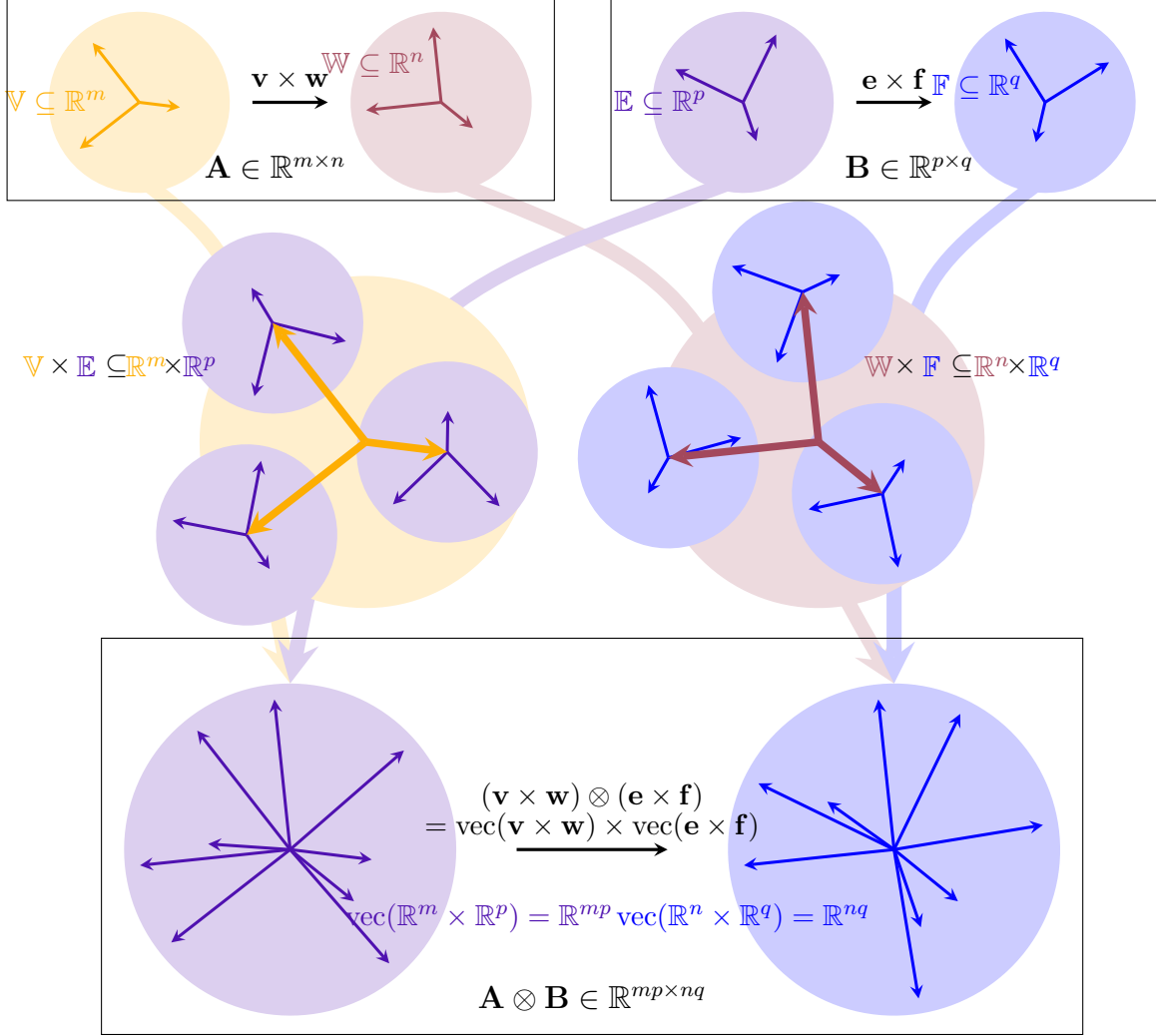
In **Thm. 1**, $\mathbf{v}_i$ and $\mathbf{w}_i$ are unit vectors in $\mathbb{P}_1$ and $\mathbb{P}_2$ respectively, and $s_1, s_2, \dots, s_m$ are the singular values of $\mathbf{T}$. In this way, the linear transformation of $\mathbf{v}$ from $\mathbb{P}_1$ to $\mathbb{P}_2$ can be represented with m orthogonal vectors in $\mathbb{P}_2$. The geometric illustration is exactly the same as that in **Fig. 3.11b**, where $\langle \mathbf{x}, \mathbf{a} \rangle \rightarrow a$, $\mathbf{x} \mathbf{W}^\top = a \mathbf{b}$ changes to $\langle \mathbf{v}, \mathbf{v}_i \rangle \rightarrow v_i$, $\mathbf{v}_i \mathbf{T}^\top = s_i \mathbf{w}_i \Rightarrow \mathbf{v} \mathbf{T}^\top = \sum_{i=1}^m v_i (s_i \mathbf{w}_i)$.

In **Eq. (3.10)**, for the i -axis in $\mathbb{P}_2$, singular values s_i are part of the coefficients of the unit vectors $\mathbf{w}_i$. If $\mathbf{w}_i$ is small enough, the orthogonal vector $s_i v_i \mathbf{w}_i$ can be ignored. If $(m - k)$ orthogonal vectors are removed in **Eq. (3.10)**, the representation of the result vector in $\mathbb{P}_2$ is reduced from m dimensions to k dimensions. This elimination process is also called *Truncated SVD*, as described in **Thm. 2**.

Theorem 2 (Truncated SVD and Matrix (Cayley) Multiplication [109]). *Given rank r , Truncated SVD for a linear transformation $\mathbf{T}$ is given by $\mathbf{T} \mathbf{v}_i = s_i v_i \mathbf{w}_i$, $i = 1, 2, \dots, r$. Suppose $\mathbf{v}_i \in \mathbb{R}^m$ and $\mathbf{w}_i \in \mathbb{R}^n$, and denote $\mathbf{V}_r = (\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_r) \in \mathbb{R}^{m \times r}$ and $\mathbf{W}_r = (s_1 v_1 \mathbf{w}_1, s_2 v_2 \mathbf{w}_2, \dots, s_r v_r \mathbf{w}_r) \in \mathbb{R}^{n \times r}$, so $\mathbf{T} \mathbf{V}_r = \mathbf{W}_r \Rightarrow \mathbf{T} = \mathbf{W}_r \mathbf{V}_r^*$ is a matrix multiplication form of $\mathbf{T}$, where $\mathbf{V}_r^*$ is the conjugate transpose of $\mathbf{V}_r$.*



(a) Matrix (Cayley) multiplication $\mathbf{W}_r \mathbf{V}_r^* : \mathbb{R}^n \rightarrow \mathbb{R}^r \rightarrow \mathbb{R}^m$ in [Thm. 2](#), where $\mathbf{W}_r \in \mathbb{R}^{n \times r}$ and $\mathbf{V}_r^* \in \mathbb{R}^{r \times m}$



(b) Kronecker product $\mathbf{A} \otimes \mathbf{B} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{mp \times nq}$. The matrix $\mathbf{A}$ and $\mathbf{B}$ are composed according to [Def. 7](#). That is, the first matrix, $\mathbf{A}$, is composed as $\mathbf{A} = \mathbf{v} \times \mathbf{w}$, and the second matrix, $\mathbf{B}$ is composed as $\mathbf{B} = \mathbf{e} \times \mathbf{f}$. The two vectors that compose matrix $\mathbf{A}$ serve as “parent nodes” and the two vectors that compose matrix $\mathbf{B}$ serve as “child nodes”. The final results of Kronecker products $\mathbf{A} \otimes \mathbf{B}$ is the composition of all the “child nodes”. Here, the four vectors $\mathbf{v} \in \mathbb{R}^m$, $\mathbf{w} \in \mathbb{R}^n$, $\mathbf{e} \in \mathbb{R}^p$ and $\mathbf{f} \in \mathbb{R}^q$.

Figure 3.12: Matrix (Cayley) multiplication and Kronecker product.

Kronecker product The discussion so far has assumed that all subspaces are equivalent in the composition of a matrix or tensor. In practice there are cases where some subspaces operate on top of others, forming hierarchical structures. The Kronecker product is such a case, where all the elements of the first operand are scalar multiplied by the second operand, as shown in [Fig. 3.12b](#).

For two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$, the Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ is

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \in \mathbb{R}^{mp \times nq}. \quad (3.11)$$

From the matrix composition perspective (Def. 7), suppose $\mathbf{A} = \mathbf{v} \times \mathbf{w}$, $\mathbf{B} = \mathbf{e} \times \mathbf{f}$, then (i, j) th element of $\mathbf{A} \otimes \mathbf{B}$ is given by

$$(\mathbf{A} \otimes \mathbf{B})_{i,j} = a_{\lceil i/p \rceil, \lceil j/q \rceil} b_{(i-1)\%p+1, (j-1)\%q+1} = v_{\lceil i/p \rceil} w_{\lceil j/q \rceil} (e_{(i-1)\%p+1} f_{(j-1)\%q+1}), \quad (3.12)$$

where $\%$ denotes the remainder when dividing. Moreover, its representation with four vectors $\mathbf{v} \in \mathbb{V}$, $\mathbf{w} \in \mathbb{W}$, $\mathbf{e} \in \mathbb{E}$ and $\mathbf{f} \in \mathbb{F}$ is

$$\mathbf{A} \otimes \mathbf{B} = \text{vec}(\mathbf{v} \times \mathbf{e}) \times \text{vec}(\mathbf{w} \times \mathbf{f}). \quad (3.13)$$

Fig. 3.12b illustrates the hierarchy of the Kronecker product, where $\mathbf{e}$ and $\mathbf{f}$ can be seen as the replicated child nodes of $\mathbf{v}$ and $\mathbf{w}$, respectively. The final result is composed of flattened “child nodes”, however, all “child nodes” are globally organised by “parent nodes” $\mathbf{v}$ and $\mathbf{w}$.

Block-diagonal matrices Block-diagonal matrices are blocks on the matrix diagonal while the rest elements are zeros. A remarkable example of such kind of matrices for model compression is butterfly matrix [52, 105, 106, 110], which can represent a matrix $\mathbf{C}$ with Eq. (3.7) as Eq. (3.14), where $\mathbf{C}_1 \in \mathbb{R}^{m \times m}$ and $\mathbf{C}_2 \in \mathbb{R}^{n \times n}$, $\mathbf{P}$ is a permutation matrix, and $\mathbf{D}$ is a butterfly factor [106] to yield

$$\mathbf{C}^\top = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix} \begin{bmatrix} \mathbf{C}_1 & 0 \\ 0 & \mathbf{C}_2 \end{bmatrix} \mathbf{P}. \quad (3.14)$$

According to Def. 7, $\begin{bmatrix} \mathbf{C}_1 & 0 \\ 0 & \mathbf{C}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{C}_1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{C}_2 \end{bmatrix}$ can be seen as the vector addition between two vectors $\mathbf{c}_1, \mathbf{c}_2 \in \mathbb{R}^{(m+n) \times (m+n)}$.

In Sec. 3.2.2, it is shown that $\mathbf{c}_1$ and $\mathbf{c}_2$ are composed by

$$\begin{aligned} \mathbf{a}_1 &= (a_1, a_2, \dots, a_m, \overbrace{0, \dots, 0}^{m+2n}), \\ \mathbf{b}_1 &= (\overbrace{0, \dots, 0}^m, b_1, b_2, \dots, b_m, \overbrace{0, \dots, 0}^{2n}), \\ \mathbf{a}_2 &= (\overbrace{0, \dots, 0}^{2m}, a_{m+1}, a_{m+2}, \dots, a_{m+n}, \overbrace{0, \dots, 0}^n), \\ \mathbf{b}_2 &= (\overbrace{0, \dots, 0}^{2m+n}, b_{m+1}, b_{m+2}, \dots, b_{m+n}), \end{aligned}$$

in a unified vector space $\mathbb{V} \subseteq \mathbb{R}^{2m+2n}$. This process can be further interpreted as the composition in two different subspaces $\mathbb{V}_1 \subseteq \mathbb{R}^{m \times m}$ and $\mathbb{V}_2 \subseteq \mathbb{R}^{n \times n}$, and $\mathbb{V} = \mathbb{V}_1 \times \mathbb{V}_2$ (regarding coordinates), with $\mathbf{a}_1, \mathbf{b}_1 \in \mathbb{V}_1$ and $\mathbf{a}_2, \mathbf{b}_2 \in \mathbb{V}_2$.

There may be transformations in $\mathbb{V}_1$ and $\mathbb{V}_2$ separately, which implies that in hardware the two processes can run in parallel. Then, the vectors $\mathbf{a}_1, \mathbf{b}_1$ and $\mathbf{a}_2, \mathbf{b}_2$ in two subspaces $\mathbb{V}_1$ and $\mathbb{V}_2$ are merged together and go through a transformation defined by $\mathbf{D}$.

Eq. (3.14) can be further expanded as

$$\mathbf{C}^\top = \left[\begin{array}{ccc|ccc} d_{1,1} & & & d_{2,1} & & \\ & \ddots & & & \ddots & \\ & & d_{1,m} & & & d_{2,n} \\ \hline d_{3,1} & & & d_{4,1} & & \\ & \ddots & & & \ddots & \\ & & d_{3,m} & & & d_{4,b} \end{array} \right] \left[\begin{array}{cccccc} a_1 b_1 & \dots & a_m b_1 & & & \\ \vdots & \ddots & \vdots & & 0 & \\ a_1 b_m & \dots & a_m b_m & & & \\ & & & a_{m+1} b_{m+1} & \dots & a_{m+n} b_{m+1} \\ & 0 & & \vdots & \ddots & \vdots \\ & & & a_{m+1} b_{m+n} & \dots & a_{m+n} b_{m+n} \end{array} \right], \quad (3.15)$$

where $\mathbf{C}^\top \in \mathbb{R}^{(n+m) \times (n+m)}$.

According to [106], the left matrix in Sec. 3.2.7 can be recursively factorised into smaller diagonal matrices, which finally results in a set of 2×2 diagonal matrices. These 2×2 diagonal matrices can be seen as 1-dimensional vectors in order-1 vector space, or equivalently, as scalars with index information in the parameter vector. The multiplication of such scalars and $\mathbf{C}_1$ and $\mathbf{C}_2$ can be seen as scaling in different subspaces. Herein, block-diagonal matrices can be geometrically interpreted as separate scalar scaling in smaller subspaces, and its following merge across these subspaces with concatenation.

3.2.8 Tensors and Tensor Factorisation

This section first gives the essential concepts about tensors and tensor operations. Then, it gives a composition view of tensor factorisation, based on the format built in Sec. 3.2.6.

Order- N tensor. An order- N real-valued tensor $\mathcal{A}$ is a high-dimensional matrix (or multi-way array), denoted by $\mathcal{A} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, where N is the order of the tensor (i.e., number of its modes), and I_k ($1 \leq k \leq N$) is the size (i.e., the dimension) of its k -th mode. In this sense, matrices (denoted as $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$) can be seen as order-2 tensors ($N = 2$), vectors (denoted as $\mathbf{a} \in \mathbb{R}^I$) can be seen as order-1 tensors ($N = 1$), and scalars (denoted as $a \in \mathbb{R}$) are order-0 tensors ($N = 0$).

Tensorisation. A vector $\mathbf{a} = (a_1, a_2, \dots, a_{I_1 I_2 \dots I_N}) \in \mathbb{R}^{I_1 I_2 \dots I_N}$, can be tensorised (or “folded”, “reshaped”) into an order- N tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, so that

$$\mathcal{A}[i_1, i_2, \dots, i_N] = a_{1+\sum_{k=1}^N (i_k-1) \prod_{p=1}^{k-1} I_p}, \quad 1 \leq i_k \leq I_k, \quad (3.16)$$

where $\mathcal{A}[i_1, i_2, \dots, i_N]$ denotes the $(i_1, i_2, \dots, i_N)$ -th entry of $\mathcal{A}$.

Vectorization. Given an order- N tensor, $\mathcal{A} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, its vectorization reshapes the high-dimensional matrix into a vector, $\text{vec}(\mathcal{A}) = \mathbf{a} \in \mathbb{R}^{I_1 \dots I_N}$.

Tensor contraction. The contraction of $\mathcal{A} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ and $\mathcal{B} \in \mathbb{R}^{J_1 \times \dots \times J_M}$, over the k th and p th modes respectively, where $I_k = J_p$ is denoted as $\mathcal{A} \times_k^p \mathcal{B}$ and results in a tensor

$$\mathcal{C} \in \mathbb{R}^{I_1 \times \dots \times I_{k-1} \times I_{k+1} \times \dots \times I_N \times J_1 \times \dots \times J_{p-1} \times J_{p+1} \times \dots \times J_M},$$

with entries

$$\begin{aligned} & \mathcal{C}[i_1, \dots, i_{k-1}, i_{k+1}, \dots, i_N, j_1, \dots, j_{p-1}, j_{p+1}, \dots, j_M] \\ &= \sum_{q=1}^{I_k} \mathcal{A}[i_1, \dots, i_{k-1}, q, i_{k+1}, \dots, i_N] \mathcal{B}[j_1, \dots, j_{p-1}, q, j_{p+1}, \dots, j_M] \end{aligned} \quad (3.17)$$

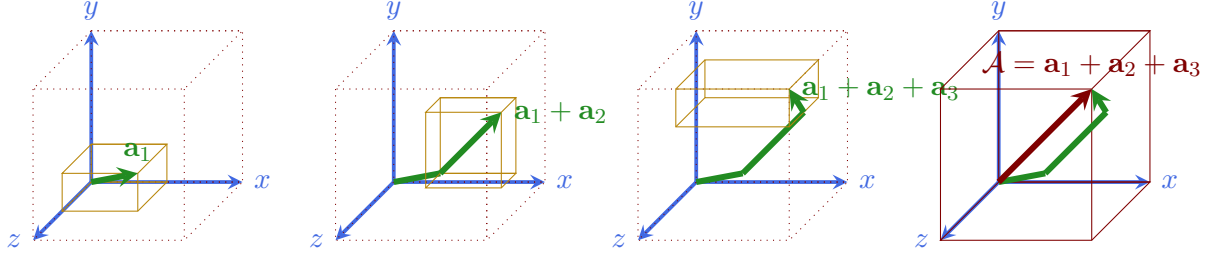
Tensor-train decomposition (TTD). The most common Tensor-Train Decomposition [111] is to format a tensor into a Matrix Product State (MPS or TT-MPS) form, which applies the Tensor-Train Singular Value Decomposition (TT-SVD) algorithm (see Sec. 5.3.1) to an order- N tensor, $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$. This results in N smaller 2-nd or 3-rd order tensors, $\mathcal{G}^{(k)} \in \mathbb{R}^{r_{k-1} \times I_k \times r_k}$ for $k = 1, \dots, N$, such that

$$\mathcal{X} \approx \mathcal{G}^{(1)} \times_2^1 \mathcal{G}^{(2)} \times_3^1 \mathcal{G}^{(3)} \times_3^1 \dots \times_3^1 \mathcal{G}^{(N)} \quad (3.18)$$

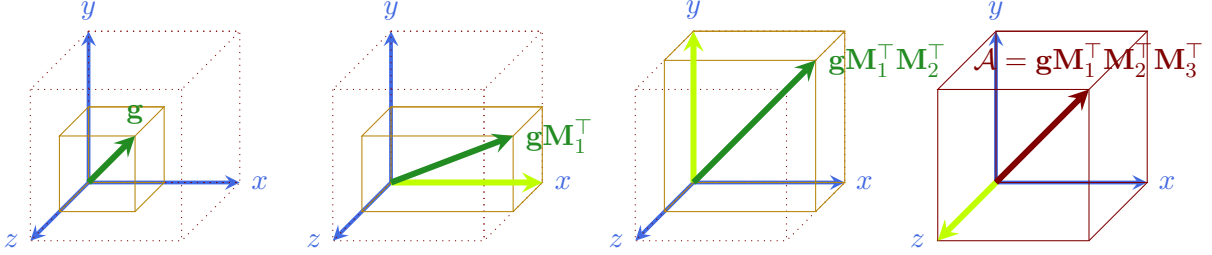
$\mathcal{G}^{(1)}, \dots, \mathcal{G}^{(N)}$ are referred to as the tensor cores, while the set $\{r_0, r_1, \dots, r_N\}$ represents the TT-rank of the TT decomposition ($r_0 = r_N = 1$).

Compared with matrix factorisations in Sec. 3.2.7, tensor factorisations are a bit more complex (but fortunately not too much), since there are more than two subspaces, and every two subspaces have interactions like in Thms. 1 and 2. All the interactions are not outside those in Tab. 3.2, which makes the various tensor factorisation orchestrated in the same style. The three tensor factorisation formats discussed in this section are the CP, Tucker and Tensor-Train. We later summarise that their forms represent roughly vector addition in the same subspace, dimension adjust intra subspace, and subspace order expansion.

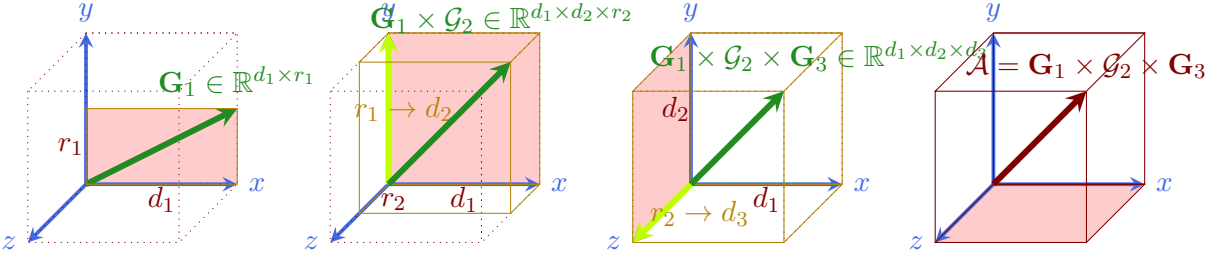
CANDE-COMP/PARAFAC (CP) format. The CP format represents a tensor with a sum of tensor product of vectors [112]. For a order- n tensor, $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$, there is $\mathcal{A} = \sum_{i=1}^r \mathbf{a}_i$, $\mathbf{a}_i =$



(a) CP format ($n = 3, r = 3$), the resulted tensor (red vector) represented as the addition of three 3-order tensors (green vectors).



(b) Tucker format ($n = 3$), matrix $\mathbf{M}_i \in \mathbb{R}^{h_i \times d_i}$ adjusts the tensor (green vector) on the current dimension (lime green axis).



(c) Tensor-Train format, $n = 3$, $\mathbf{G}_1$ and $\mathbf{G}_2$ are matrices and $\mathcal{G}_2$ is a 3-order tensor.

Figure 3.13: Geometric interpretation of typical tensor decomposition formats, taking a 3-order tensor as an example. Their vector forms are Eqs. (3.19) and (3.20). The golden cubes are the current constructed subspaces, the light green vectors are the subspace expansion represented by shared order (e.g. $\mathcal{A}^{(i)} \cap \mathbf{M}_i$ in Tucker format, $\mathcal{A}^{(i)} \cap \mathbf{G}_i$ and $\mathcal{A}^{(i)} \cap \mathcal{G}_i$ in Tensor-Train format), the dark green vectors are the current resulting vectors, inside the current constructed subspace. The red cube is the final resulted tensor, and pink planes represent resized dimension.

$\overbrace{\mathbf{v}_i \otimes \mathbf{u}_i \otimes \cdots \otimes \mathbf{w}_i}^{n \text{ vectors}}$, where r is the tensor rank and $\mathbf{v}_i \in \mathbb{R}^{d_1}$, $\mathbf{u}_i \in \mathbb{R}^{d_2}$, $\dots$, $\mathbf{w}_i \in \mathbb{R}^{d_n}$. We can easily convert this tensor format to a vector in $\mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$ as

$$\mathbf{a} = \sum_{i=1}^r \mathbf{a}^{(i)}, \quad \mathbf{a}^{(i)} = \mathbf{a}_1^{(i)} \times \mathbf{a}_2^{(i)} \times \cdots \times \mathbf{a}_n^{(i)}. \quad (3.19)$$

Such a format is an addition form of r n -order tensors. All these order- n tensors (represented as dark green individual vectors in Fig. 3.13a), are in the same subspace, as well as the resulting tensor $\mathcal{A}$ (represented as the final red vector in Fig. 3.13a).

Tucker format For a order- n tensor $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$, its Tucker format is $\mathcal{A} = \mathcal{G} \times_1 \mathbf{M}_1 \times_2 \dots \times_n \mathbf{M}_n$, where $\mathcal{G} \in \mathbb{R}^{l_1 \times l_2 \times \dots \times l_n}$ is a n -order tensor and matrix $\mathbf{M}_i \in \mathbb{R}^{h_i \times d_i}$ ($i = 1, 2, \dots, n$). Matrices $\mathbf{M}_i$ can be seen as a general linear transformation, $\mathbf{M}_i : \mathbb{R}^{h_i} \rightarrow \mathbb{R}^{d_i}$, which results in a transformation on the whole tensor $\mathcal{G} \times_1 \mathbf{M}_1 \times_2 \dots \times_i \mathbf{M}_i : \mathbb{R}^{d_1 \times \dots \times d_{i-1} \times \textcolor{blue}{h_i} \times h_{i+1} \times \dots \times h_n} \rightarrow \mathbb{R}^{d_1 \times \dots \times d_{i-1} \times \textcolor{blue}{d_i} \times h_{i+1} \times \dots \times h_n}$.

Suppose $(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n)$ is a basis of vector space $\mathbb{G} \in \mathbb{R}^{h_1 \times h_2 \times \dots \times h_n}$, the vector form of Tucker format is

$$\mathbf{a} = \mathbf{g} \times \mathbf{m}_1 \times \mathbf{m}_2 \times \dots \times \mathbf{m}_n, \quad \mathbf{m}_i \in \mathbb{M}_i \subseteq \mathbb{R}^{h_i \times d_i}, \quad \mathbb{G} \cap \mathbb{M}_i = \mathbf{v}_i \in \mathbb{R}^{\max(h_i, d_i)} \quad (3.20)$$

where $\mathbf{g} \in \mathbb{G}$. As shown in Fig. 3.13b, such a process represents changing the dimension range of each order of the subspace $\mathbb{G}$ via a tensor contraction mentioned in Eq. (3.9). Geometrically, the product between the current tensor $\mathbf{g} \times \mathbf{m}_1 \times \dots \times \mathbf{m}_{i-1}$ and $\mathbf{m}_i$ enlarges or shrinks the current dimension (highlighted on axis with lime green), reflecting the change of the current subspace (denoted as golden cubes in Fig. 3.13b).

Tensor-train format The Tensor-Train (TT) format [111] represents order- n tensor $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$ as $\mathcal{A} = \mathbf{G}_1 \times \mathcal{G}_2 \times_3^1 \dots \times_3^1 \mathcal{G}_{N-1} \times \mathbf{G}_n$, where $\mathcal{G}_i \in \mathbb{R}^{r_{i-1} \times d_i \times r_i}$, with the TT-rank set $\{r_1, \dots, r_{n-1}\}$, $\mathbf{G}_1 \in \mathbb{R}^{d_1 \times r_1}$ and $\mathbf{G}_n \in \mathbb{R}^{r_{n-1} \times d_n}$. The vector form of the Tensor-Train format is

$$\mathbf{a} = \mathbf{m}_0 \times \mathbf{m}_1 \times \dots \times \mathbf{m}_{n-1} \times \mathbf{m}_n, \quad \mathbf{m}_i \in \mathbb{M}_i, \quad (3.21)$$

where $\mathbb{M}_i \subset \{\mathbb{R}^{r_{i-1} \times d_i \times r_i} \mid i = 2, 3, \dots, n-1\} \cup \{\mathbb{R}^{d_i \times r_i} \mid i = 1\} \cup \{\mathbb{R}^{r_{i-1} \times d_i} \mid i = n\}$.

The vector form of Tensor-Train is quite simple, and the only point is that there is a dimension exceptions for $\mathbf{m}_0$ and $\mathbf{m}_n$ compared with other $\mathbf{m}_i$. Given that the tensor product and tensor contraction are considered identical in Eq. (3.9), where the contraction is a case where two subspaces share a dimension and “absorb” that dimension. In this sense, every product $\mathbf{m}_{i-1} \times \mathbf{m}_i$ means “enlarging or shrinking the shared dimension between $\mathbf{m}_{i-1}$ and $\mathbf{m}_i$, which is similar to the Tucker format, and then additionally “extend the new order introduced by $\mathbf{m}_i$ ” when $\mathbf{m}_i$ is three-dimensional.

Fig. 3.13c illustrates the above geometric transformation, maintaining the matrix and tensor notations to make it easier for the reader to transit from the classical tensor version to the proposed vector version. The first subspace is defined by $\mathbf{G}_1$, which is a matrix so that it settles down a 2-dimensional plane of size $d_1 \times r_1$. The product of $\mathbf{G}_1 \times \mathcal{G}_1$ is a subspace extension from 2-dimension to 3-dimension, along with the common dimension between $\mathbf{G}_1$ and $\mathcal{G}_1$ (r_1 , highlighted with lime green y -axis). Similarly, the next product along with the common dimension between $\mathbf{G}_1 \times \mathcal{G}_1$ and $\mathbf{G}_3$, but it does not add extra dimension since $\mathbf{G}_3$ is a matrix which only scales the existing dimension.

3.2.9 Interpretation of Transformer Modules with Subspaces

In this section, we give the geometric interpretation of common transformer modules, given in [Defs. 7](#) and [8](#), and graphically illustrated in [Fig. 3.10a](#). The transformer structure typically maps the outputs of embedding layers to a subspace of similar dimension, subsequently maps the output features from the attention layer to a higher-dimensional latent subspace (graphically represented by larger cubes), and finally to a lower-dimensional subspace (graphically represented by smaller cubes). For a multi-head transformer structure, this sequential mapping is executed multiple times in parallel, which is graphically represented as replicated groups of “Attn”(attention layer) and “FF”(feed-forward layer) modules.

Linear layer (the only parametric part of the feed-forward layer). A linear layer, also called a fully connected layer, dense layer, or affine layer, is the only component having weights in the feed-forward layer. Its output, $\mathbf{y} \in \mathbb{R}^n$, and input, $\mathbf{x} \in \mathbb{R}^m$, satisfy

$$\mathbf{y} = \mathbf{x}\mathbf{W}^\top + \mathbf{c}, \quad (3.22)$$

where $\mathbf{W} \in \mathbb{R}^{n \times m}$ is the weight matrix, and $\mathbf{c} \in \mathbb{R}^n$ is the bias. For simplicity, we only discuss the weight matrix rather than weight tensors here.

The forward pass of the linear layer represents basically matrix multiplication or vector-matrix multiplication; these two are treated as equivalent in this section. A common geometric interpretation of such a process is shown in [Fig. 3.11a](#), where the vector-matrix multiplication between vector $\mathbf{x}$ and matrix $\mathbf{W}^\top$ can be seen as the vector $\mathbf{x}$ being projected from the original subspace $\mathbb{P}_1$ to another subspace $\mathbb{P}_2$, where $\mathbb{P}_1, \mathbb{P}_2 \subset \mathbb{P}$.

However, based on the matrix composition defined in [Def. 7](#), such transformation can be interpreted as two operations shown in [Fig. 3.11b](#): ① $\mathbf{x}, \mathbf{a} \in \mathbb{P}_1, \langle \mathbf{x}, \mathbf{a} \rangle \rightarrow a \in \mathbb{R}$, ② $a\mathbf{b}, \mathbf{b} \in \mathbb{P}_2$, where $\langle \cdot, \cdot \rangle$ is inner product (projection product in sense of geometry), and $\mathbf{a}$ and $\mathbf{b}$ are the vectors composing weight matrix $\mathbf{W}^\top$.

Concatenation. Concatenation in the context of generative language models usually occurs when merging the outputs of the transformer heads (i.e. numeric vectors) by simply appending the outputs one-by-one. Since there are no arithmetic interactions among the elements in these vectors, this process can be seen as the merge of two orthogonal subspaces with two disjoint basis sets. This merging can be easily implemented via constructing a larger subspace $\mathbb{P}_3 = \mathbb{P}_1 \times \mathbb{P}_2 \in \mathbb{R}^{m+n}$, where $\mathbb{P}_1 \in \mathbb{R}^m$ and $\mathbb{P}_2 \in \mathbb{R}^n$, as shown in [Fig. 3.11c](#).

Attention layer. Attention layers are the core functional modules in the transformer-based language models. Take the original transformer architecture [\[5\]](#) as an example, the main mathematical

expressions of the attention and head (a module of the neural network) are as follows:

$$\text{attn}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \frac{\mathbf{q}\mathbf{K}^\top}{\sqrt{d_k}} \mathbf{V}, \quad (3.23)$$

where $\mathbf{q}$ is the query vector, $\mathbf{K}$ and $\mathbf{V}$ are respectively the key and value matrices, and d_k is the dimension of $\mathbf{q}$. This transformation process can be geometrically interpreted as in Fig. 3.11d. The query $\mathbf{q}$ is first projected into the subspace $\mathbb{K}$, which represents keys. Then, the resulting vector $\mathbf{k}$ is normalised via **softmax** and the coefficient $\frac{1}{\sqrt{d_k}}$ in the same subspace $\mathbb{K}$, to become another vector $\tilde{\mathbf{k}}$. Finally, the normalised vector, $\tilde{\mathbf{k}}$, is projected to the value subspace, $\mathbb{V}$.

Therefore, the attention layer is a series of linear transformations within three subspaces: query subspace $\mathbb{Q}$, key subspace $\mathbb{K}$ and value subspace $\mathbb{V}$, and its ultimate goal is to map a query vector to a value, and this value is named “attention”. Or in other words, the attention layer is a series of two projections, projecting a single query $\mathbf{q}$ to a value.

3.2.10 Geometric View of the Recent Research on Low-Rank Compression

The necessary issues regarding the terminologies of matrix and tensor factorisation for model compression have been addressed in Secs. 3.2.2 and 3.2.5. This subsection analyses relevant literature based on the proposed taxonomy. The core idea of the analysis is to **take every matrix or tensor as a geometric transformation**. Such analysis typically focuses on two questions:

1. How many and what subspaces are involved?
2. What geometric transformations (operations) occur among these subspaces?

Factual analysis of the comparison. In Tab. 3.3, only two investigated works compressed all core modules (i.e. embedding, attention and feed-forward layers) [55, 56]. This is because the forward passes of these three core modules are different, as discussed in Sec. 3.2.9 and in practical implementation, this difference may introduce new problems. Most of the investigated works did not consider fidelity as their compression goal, though fidelity is calculated without actual language tasks and thus easier to obtain than accuracy in practice. This phenomenon indicates that most works believe fidelity is not a shortcut to accuracy and should be considered differently. Note that there are some works that do not have an optimisation part in their methodologies at all [51, 59, 60, 61], mainly in the research work with tensor factorisation. This is probably because the fidelity metrics for tensors (e.g. Frobenius norm) are less indicative of model accuracy than those for matrices, especially when the compared methodologies have different tensor orders. A more valid way to obtain the model accuracy is just selecting some hyperparameters (e.g. tensor ranks), running the compressed models, and evaluating the compressed model predictions.

Implications for compression. The geometric analysis above suggests that compression strategies should differ based on model architecture:

- For GPT-2-style models where the MLP ratio is lower, attention layer compression (i.e., subspace dimension reduction in $\mathbb{Q}$, $\mathbb{K}$, $\mathbb{V}$) may be more impactful, as these models rely more heavily on attention mechanisms for their capabilities.
- For Llama-3.1/Gemma-2/Mistral-style models with higher MLP ratios, FFN layer compression (i.e., reducing latent space dimensions in the feed-forward layers) becomes critical, since the MLP layers constitute a larger proportion of total parameters.

These architectural differences motivate different geometric operations in our taxonomy (see [Tab. 3.3](#)): models with larger attention ratios benefit from “split one subspace into two lower-dimensional subspaces” operations applied to attention weights, while MLP-heavy models benefit from “shrink subspace dimensions” operations applied to feed-forward layers.

Table 3.3: Interpretation of current literature under the proposed taxonomy. Since this section focuses on model compression, only the experimental model compression results claimed in the sections are extracted, with the compression ratio as defined in Def. 2. The additional information after the compression ratio, for example, “small”, “large”, etc. reflects the different versions of the base models. The definition of two optimisation goals, fidelity and accuracy are given in Sec. 3.2.4. “Not mentioned” in the last column means the focus of the investigated work is not model compression (e.g. acceleration or reduced computations), but intrinsically reduces the model size at the same time. All the experimental results are comparable to language task performance in the uncompressed models.

Work	Base Model	Layer Types			#Subspaces	Geometric Operations	Optimisation Goal		Experimental Parameter Compression Ratio
		Emb	Attn	FF			Fidelity	Accuracy	
[52]	BERT		✓	✓	3,4,5	generate 1-order subspaces, projection	✓		$0.98\times$ (base)/ $1.33\times$ (large)
	GPT-2								$0.72\times$ (small)/ $1.15\times$ (medium)
[55]	GPT-2	✓	✓	✓	3	shrink subspace dimensions	✓	✓	$0.49\times$ (small)
[56]	BERT	✓	✓	✓	4				$6.6\times -18.3\times$ (base)
[47]	2-layer LSTM	✓			2	split one subspace into two lower-dimensional subspaces	✓	✓	$4\times -9\times$ (only Emb)
[53]	BERT		✓	✓					Not mentioned
[49]	BERT	✓							$0.24\times$ (base)
[48]	BERT		✓	✓					$0.69\times$ (base)
[54]	BERT, BART DeBERTaV3		✓	✓		split subspaces, use sparsity to reduce subspaces’ dimensions		✓	$1\times -19\times$
[57]	BERT, GPT-3	✓	✓	✓		reallocate subspace dimensions			$0.3\times -162.1\times$
[104]	GPT-2			✓	8	increase number of subspaces		✓	$0.67\times$ (small)
[50]	LSTM, Transformers	✓			6,8,12			✓	$7.88\times$ (LSTM)/ $2\times$ (Transformers)
[51]	BERT		✓	✓	2,3,5,7				$0.52\times -0.64\times$
[59]	Transformer		✓		3	change the projection sequence			$1\times -2\times$
[60]	Transformer		✓	✓	3				$2.6\times -4.5\times$
[61]	Transformer	✓			3	reorganize linear transformations			$0.24\times$
[58]	GPT-2		✓	✓	3-20	generate more 1-order subspaces	✓		Not mentioned

Geometric view of the comparison and current research gap. A straightforward idea of compressing neural networks is to directly break weight matrices into smaller matrices or tensors, which is equivalent to splitting high-dimensional subspaces into lower-dimensional subspaces. This idea and its variants can cover all aspects in Tab. 3.3, except for those changing the subspace projection sequences across layers [59, 60]. From the viewpoint of subspaces on Tab. 3.3, there are three obvious issues:

1. What are the criteria of constructing subspaces according to a given neural network;
2. How to identify the less important subspaces, merge such subspaces with others, or even discard them;
3. How to reallocate orders and dimensions in a collection of subspaces?

The three questions are rather abstract and should be implemented with specific factorisation methodologies like those in Sec. 3.2.6, as well as the actual forward pass of the considered neural networks. We now provide concrete examples connecting each issue to existing methods:

Issue 1: Criteria for constructing subspaces. Different methods implicitly adopt different criteria. For instance, SVD-based methods [49, 48] construct subspaces based on the spectral properties of weight matrices, where singular vectors define the basis. Kronecker product methods [55, 56] impose a hierarchical structure where subspaces are organised as “parent-child” relationships. Block-diagonal methods [52] partition the parameter space into independent subspaces that can be processed in parallel.

Issue 2: Identifying less important subspaces. The most common approach is singular value thresholding: in truncated SVD (Thm. 2), subspaces corresponding to small singular values $s_i \approx 0$ are discarded. For attention mechanisms, recent work on attention head pruning [48] identifies “less important” heads (subspaces in $\mathbb{Q}, \mathbb{K}, \mathbb{V}$) based on their contribution to downstream task performance. Magnitude-based pruning [54] identifies near-zero weight regions as candidates for removal.

Issue 3: Reallocating orders and dimensions. Adaptive rank selection strategies [57] dynamically adjust the dimensions of factorised matrices based on layer sensitivity analysis. The Tucker format (Eq. (3.20)) explicitly allows dimension adjustment via the factor matrices $\mathbf{M}_i$, where $h_i \neq d_i$ represents a dimension change. Cross-layer methods [59, 60] reallocate parameters by sharing subspaces across multiple layers, effectively changing the “order” of the global parameter tensor.

Furthermore, given that the target of model compression is to maintain comparable model performance with fewer parameters, it should be worthwhile to investigate the application requirements with the above three questions. For example, if the application requires the model to memorise more diverse data (i.e. more lower-dimensional subspaces in attention layers) or to have better reasoning

ability (i.e. a few higher-dimensional subspaces in feed-forward layers).

Pack it up. Secs. 3.2.1 to 3.2.7 propose a unified taxonomy related to the interpretation of the generative model compression, based on matrix and tensor factorisations. To bridge generative language model research and matrix/tensor factorisation, and also bridge the matrix and tensor research, this section has reduced the term subspace in geometric algebra as the pivot point to build these two connections. To ease the understanding of the relevant algebra structure, this section reduced and simplified the mathematical concepts as much as possible. For example, this section has induced that the direct product, tensor product, Cartesian product and tensor contraction are the different sides of the same process - constructing a new vector space from other subspaces to avoid confusion caused by too many terms. This section has finally provided the interpretation of the existing literature in the light of the proposed subspace-oriented taxonomy and pointed out a road when it comes to combining the required model ability (i.e. expressivity and capability) and the exact vector space implementations (i.e. certain types of matrix or tensor factorisation).

Since the focus has been on a unified taxonomy, this section has not covered as vast amounts of literature as possible, and the unique details of each investigated work have not been fully discussed either. Also, some subspace-related insights are not easy to align with specific matrix/tensor factorisation methods. Both are worthwhile for further investigation.

3.3 Global Information Transmission Through the Generative Language Model Computational Graph

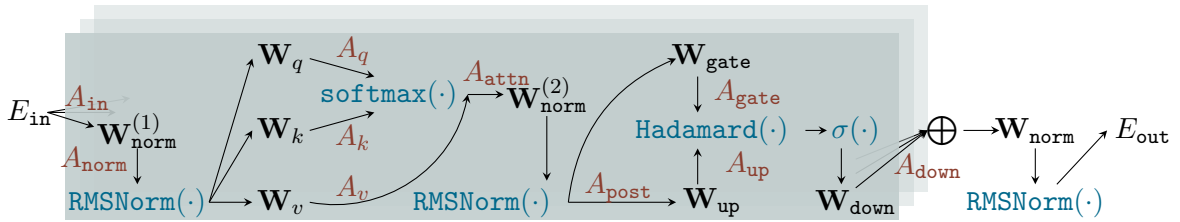


Figure 3.14: Activation flow (with annotated activations) in the LLM transformer.

This section clarifies how the linear representations are transmitted throughout the network, to analyse the impacts of dimension reduction on this transmission in Sec. 3.3.1.

A central concept in this analysis is the *representation dimension* (also referred to as the embedding dimension, model dimension, or the dimension of the residual stream in the relevant literature). This dimension, denoted d , determines the size of the vector space in which all token representations reside as they flow through the network. Structured pruning approaches such as SliceGPT [113] directly reduce the representation dimension to extract subnetworks of varying sizes. Understanding the

role of the representation dimension is essential because it dominates the linear transformations, model predictions, and, ultimately, model performance on downstream tasks such as ARC-Easy, ARC-Challenge [2], WinoGrande [45], and PIQA [4]. By mechanistically analysing the activation flow during model forward passes, it becomes clear that the representation dimension serves as the primary bottleneck governing how information is processed and transformed at each layer.

The transformer architecture structure in [5] is used, denoting the LLM transformer module as $\mathcal{M}$. $\mathcal{M}$ contains groups of linear and non-linear transformations in multi-head attention implementation, as shown in Fig. 3.14.

Proposition 3.1. *The mapping $M : E_{\text{in}} \rightarrow \bigoplus_{i=1}^h A_i$ defined by the transformer $\mathcal{M}$, consists of h groups of the following transformations:*

- **linear transformations** defined by $\{\mathbf{W}_{\text{norm}}, \mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v, \mathbf{W}_o, \mathbf{W}_{\text{gate}}, \mathbf{W}_{\text{up}}, \mathbf{W}_{\text{down}}\}$;
- **non-linear transformations** involved in $\{\text{RMSNorm}(\cdot), \text{softmax}(\cdot), \sigma(\cdot)\}$.

With the transformations in Propos. 3.1, the following analysis of the activation flow is provided.

Let d be the embedding dimension. Input a text of l tokens to the embedding layer first, we then have the encoded texts (embeddings) $E_{\text{in}} \in \mathbb{R}^{l \times d}$. E_{in} is then processed by a layer norm operation. Take the current widely-used RMSNorm [114] as a case study, the normalised embeddings after the layer norm operation is

$$\bar{E}_{\text{in}} = \mathbf{W}_{\text{norm}} E_{\text{in}} \cdot \text{RMSNorm}(E_{\text{in}}) \in \mathbb{R}^{l \times d}, \quad (3.24)$$

$$\text{RMSNorm}(E) = \sqrt{\frac{1}{d} \sum_{i=1}^d E^\top[:, i] E[:, i]} \in \mathbb{R}, \quad (3.25)$$

where $\mathbf{W}_{\text{norm}} \in \mathbb{R}^d$ is the weight of the RMSNorm layer, $E_{\text{in}}[:, i]$ is the i th column of E_{in} .

The attention layer typically consists of four linear layers with weight matrices $\mathbf{W}_q \in \mathbb{R}^{d \times h_{\text{attn}} h_{\text{dim}}}$, $\mathbf{W}_k \in \mathbb{R}^{d \times v h_{\text{dim}}}$, $\mathbf{W}_v \in \mathbb{R}^{d \times v h_{\text{dim}}}$ and $\mathbf{W}_o \in \mathbb{R}^{h_{\text{attn}} h_{\text{dim}} \times d}$. h_{attn} is the number of the attention heads, h_{dim} is the head dimension, and v is the number of key-value heads². In the default setting of Llama³, $h_{\text{attn}} h_{\text{dim}} = d$. The hidden states of query Q , key K and value V are then obtained as follows:

$$Q = \mathbf{W}_q \bar{E}_{\text{in}}, \quad K = \mathbf{W}_k \bar{E}_{\text{in}}, \quad V = \mathbf{W}_v \bar{E}_{\text{in}}, \quad (3.26)$$

and then the attention weights of parallel attention layers in $\mathcal{M}$, $\mathbf{W}_a$, and attention outputs A are obtained as

$$\mathbf{W}_a = \text{softmax}(QK^\top) \gamma \in \mathbb{R}^{h_{\text{attn}} \times h_{\text{attn}} \times l}, \quad A' = \mathbf{W}_o(\mathbf{W}_a V) \in \mathbb{R}^{l \times h_{\text{attn}} h_{\text{dim}}}, \quad (3.27)$$

where γ is the scaling factor. The attention outputs are then normalised again as $\bar{A}' = \text{RMSNorm}(A')$,

²The same notation names are used as listed in [Llama-3-8B Configuration](#).

³For simplicity, position encoding, dropout, attention mask, and MLP bias in LLMs are not considered.

then input to an MLP $\mathcal{M}_{\text{MLP}} = \{\mathbf{W}_{\text{gate}}, \mathbf{W}_{\text{up}}, \mathbf{W}_{\text{down}}\}$, such that we have the hidden state

$$A = \text{RMSNorm}(\mathbf{W}_{\text{down}}(\sigma((\mathbf{W}_{\text{gate}}\bar{A})(\mathbf{W}_{\text{up}}\bar{A}')))) \in \mathbb{R}^{l \times h_{\text{attn}} h_{\text{dim}}}, \quad (3.28)$$

where σ is the activation function. $\mathbb{R}^{h \times l \times d}$.

Suppose there are h parallel attention layers in $\mathcal{M}$, the attention outputs of the i th attention layer is A_i , so the final output of the transformer module $\mathcal{M}$ is

$$\bigoplus_{i=1}^h A_i = A_1 \oplus A_2 \oplus \dots \oplus A_h \in \mathbb{R}^{h \times l \times h_{\text{attn}} h_{\text{dim}}}. \quad (3.29)$$

Here, $\oplus$ denotes the direct sum, which is equivalent to concatenation in the actual implementation [30, 115]. Since in the default setting, $h_{\text{attn}} h_{\text{dim}} = d$, the final hidden states output $\bigoplus_{i=1}^h A_i \in \mathbb{R}^{h \times l \times d}$.

It can be observed that most of this transmission implemented by the transformer is composed of linear transformations, which are also the investigated objects in the scaling law [11]. **The representation dimension d dominates the linear transformation throughout the whole network.**

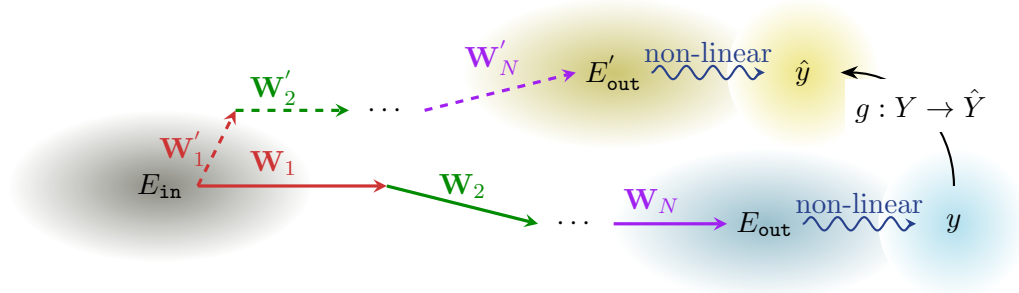


Figure 3.15: The linear transformations before (solid arrows) and after (dashed arrows) pruning in the model. The original model maps the input embeddings E_{in} from the input space (illustrated with the shading grey ellipse) to the output embedding E_{out} in the output space, through a series of linear transformations (i.e. those defined by the weight matrices $\mathbf{W}_1, \dots, \mathbf{W}_N$). After pruning, $\mathbf{W}_i$ is converted to $\mathbf{W}'_i$ ($i \in \{1, 2, \dots, N\}$), and the original output E_{out} is shifted to E'_{out} . The final predictions (y and $\hat{y}$) are generated (normally non-linearly) from E_{out} and E'_{out} . Denote the prediction domain of the pruned model as $\hat{Y}$ and that of the unpruned model Y , the mapping $g: Y \rightarrow \hat{Y}$ shifts the model performance.

3.3.1 Functionality Shifting Caused by Dimension Reduction

This section discusses the functionality shifting caused by the structured pruning, i.e., the mapping from the unpruned model prediction domain to that of the pruned model ($g: Y \rightarrow \hat{Y}$ in Fig. 3.15), and then the shift in the model performance. SliceGPT is adopted as the structure pruning approach, since it unifies the reduction of the representation dimension in the attention and feedforward layers.

Let s denote sparsity as s ($0 < s < 1$), $(1 - s)d \in \mathbb{Z}^+$ the representation dimension after pruning.

Perplexity. Recall that perplexity, as defined in Eq. (2.1), measures how well a language model predicts a token sequence. Since $\text{PPL}(S) = \exp(-\frac{1}{|S|} \sum_{i=1}^{|S|} \ln p(x_i | x_{<i}))$, taking the logarithm yields the average negative log-probability, which connects perplexity to the cross-entropy of the model’s predictive distribution. Consider a single token sequence $X = (x_1, x_2, \dots, x_l)$. Each token in X is assigned to an embedding vector $\mathbf{v}_i = (v_{i,1}, v_{i,2}, \dots, v_{i,d}) \in \mathbb{R}^{1 \times d}$, and obtaining the stacked embedding vectors (embedding matrix) $E = (\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_l) \in \mathbb{R}^{l \times d}$. The differential entropy of the stacked embedding vectors is $H(E_{\text{in}}) = H(E_{\text{out}}) = ld \cdot \kappa(t)$, where $\kappa(t) = -\int p(t) \log(p(t)) dt$. After pruning, the entropy of the output embeddings shifts from $ld \cdot \kappa(t)$ to $(1 - s)ld \cdot \kappa(t)$, such that $\frac{H(E'_{\text{out}})}{H(E_{\text{out}})} = 1 - s$. In this entropy-based view, perplexity satisfies $\text{PPL}(E) = 2^{H(E)}$. From this analysis, we derive the following proposition characterising the relationship between perplexity before and after pruning:

Proposition 3.2. *For structured pruning with representation dimension sparsity s , the perplexity of the pruned model, denoted as $\text{PPL}(D)$, and the perplexity of the original model, denoted as $\text{PPL}_0(D)$, satisfy*

$$\frac{\ln \text{PPL}_0(D)}{\ln \text{PPL}(D)} = 1 - s, \quad (3.30)$$

where D is the test dataset from the same distribution as the dataset used for pruning.

To verify Propos. 3.2, Llama-3-8B-Instruct [116] and Phi-3-mini-4k-Instruct [117] were pruned with different dimension sparsity with SliceGPT, and were then evaluated on WikiText2, as shown in Fig. 3.16a. It can be observed that the perplexities fit well with the linear function, which justifies the analytical expression Eq. (3.30).

Accuracy of generative language models is tied to the sequence probability, while perplexity is defined as the multiplicative inverse of the sequence probability. A well-generalised model would assign a high probability to the correct sequence (high accuracy but low perplexity). Therefore, intuitively, accuracy is inversely related to perplexity.

It is postulated that a logarithmic form like Eq. (3.30) might also exist in the correlation between accuracy and representation dimension sparsity. Note that here the accuracy of short, multiple-choice answers (e.g. “Yes”/“No”, or “A”/“B”/“C”) is considered, since this accuracy can be computed via exact match without ambiguity. Possible logarithmic expressions similar to $\frac{\ln \text{PPL}_0(D)}{\ln \text{PPL}(D)}$ were tested with the same LLMs used for perplexity evaluation. Perplexities were replaced with multiple-choice accuracies on ARC-Easy, ARC-Challenge, WinoGrande and PIQA (see Tab. 2.1 for the task clarification). Among these logarithmic expressions, the best-fitted form is $\ln \frac{\text{acc}}{\text{acc}_0}$, as shown in Fig. 3.16b, linking back to the weight- and activation-level trends discussed in Sec. 3.1.2. Based on these empirical observations, we propose the following proposition, which mirrors the analytical form of Propos. 3.2:

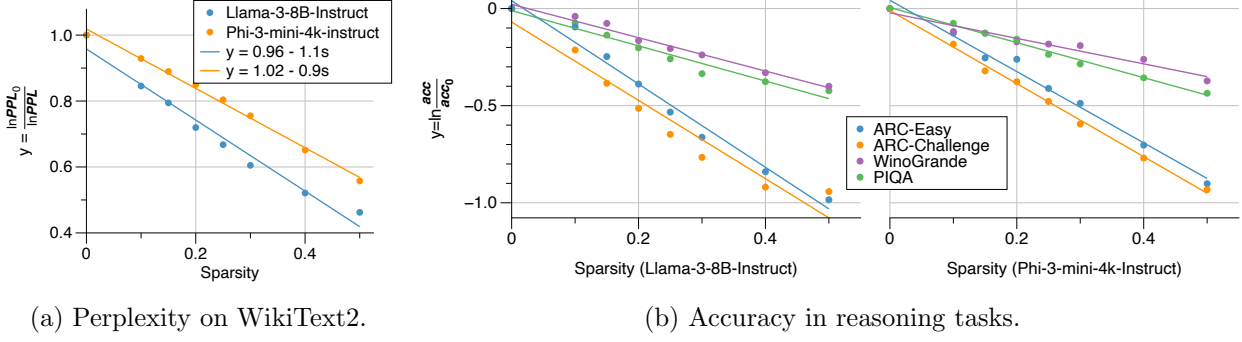


Figure 3.16: Fitted evaluation results. The fitting coefficients and errors are in [Sec. 3.3.2](#).

Proposition 3.3. *For structured pruning with representation dimension sparsity s , the multiple-choice accuracy of the pruned model, denoted as $acc(D)$, and that of the original model, denoted as $acc_0(D)$, satisfy*

$$\ln \frac{acc(D)}{acc_0(D)} \propto 1 - s, \quad (3.31)$$

where D is the test dataset for evaluating multiple-choice accuracy.

It can be observed that the left-hand side of [Eq. \(3.31\)](#) is almost the multiplicative inverse of the left-hand side of [Eq. \(3.30\)](#). This is consistent with the fact that $\frac{1}{PPL}$ and acc are positively correlated when the model generalises well, and they have (nearly) the same range ($\frac{1}{PPL} \in (0, 1]$ and $acc \in [0, 1]$). The right-hand sides of [Eqs. \(3.30\)](#) and [\(3.31\)](#) are exactly the same, indicating that the representation dimension dominates the model predictions. Though the multiple-choice accuracies fit [Eq. \(3.31\)](#) albeit less accurately than the perplexities, this formula consistency further empirically justifies what was claimed in [Sec. 3.3](#).

3.3.2 Fitting Error Summary

The fitting details in [Fig. 3.16](#) are clarified in [Tab. 3.4](#). For each sparsity configuration, the least-squares linear regression parameters are computed between representation dimension sparsity and perplexity/accuracy. The linear model $y = as + b$ is fitted, where s is the sparsity, $y = \frac{\ln PPL_0(D)}{\ln PPL(D)}$ for perplexity, and $y = \ln \frac{acc(D)}{acc_0(D)}$ for accuracy.

As shown in [Tab. 3.4](#), the slope a is consistently negative, confirming that increasing sparsity degrades model performance. The intercept b is close to the theoretical value (1 for perplexity ratio, 0 for log accuracy ratio), and the root-mean-squared errors (RMSE) remain small across all configurations. Specifically, perplexity fitting achieves RMSE values of 0.01–0.03, while accuracy fitting yields RMSE values below 0.09 across all benchmark datasets. These low errors confirm that the linear relations in [Eqs. \(3.30\)](#) and [\(3.31\)](#) adequately capture the empirical behaviour of structured pruning with representation dimension reduction.

Table 3.4: Fitting perplexities in Fig. 3.16a and accuracies in Fig. 3.16b.

Perplexity (Fig. 3.16a)	Dataset(D)	$y = as + b$ ($y = \frac{\ln \text{PPL}_0(D)}{\ln \text{PPL}(D)}$)		
		a	b	RMSE
Llama-3-8B-Instruct	WikiText2	-1.08	0.96	0.03
Phi-3-mini-4k-Instruct		-0.90	1.02	0.01

Accuracy (Fig. 3.16b)	Dataset(D)	$y = as + b$ ($y = \ln \frac{\text{acc}(D)}{\text{acc}_0(D)}$)		
		a	b	RMSE
Llama-3-8B-Instruct	ARC-Easy	-2.14	0.04	0.05
Phi-3-mini-4k-Instruct		-1.84	0.04	0.04
Llama-3-8B-Instruct	ARC-Challenge	-2.02	-0.07	0.09
Phi-3-mini-4k-Instruct		-1.88	-0.01	0.02
Llama-3-8B-Instruct	WinoGrande	-0.86	-0.02	0.02
Phi-3-mini-4k-Instruct		-0.66	-0.02	0.02
Llama-3-8B-Instruct	PIQA	-0.91	-0.01	0.03
Phi-3-mini-4k-Instruct		-0.90	0.01	0.01

3.4 Chapter Summary

This chapter develops a geometric and information-theoretic framework for understanding transformer based language models. By jointly analysing static weights and dynamic activations, it links layer-wise parameter structure, representation geometry and information flow to concrete compression opportunities, motivating later tensor-based algorithms and deployment-oriented design in Chapter 5.

Analysis of weights and activations. Sec. 3.1 examines parameter distributions and benchmark performance across several open source model families. It shows that smaller models allocate disproportionately many parameters to embeddings, while larger models invest more in feed-forward layers. The analysis connects these ratios to different capabilities, differentiating memorisation oriented behaviour from higher-level reasoning performance.

Local geometric understanding. Sec. 3.2 proposes a unified taxonomy for interpreting generative model compression, based on matrix and tensor factorisations. To bridge generative language model research and matrix/tensor factorisation, and also to connect matrix and tensor research more broadly, the term subspace in geometric algebra is used as a pivot for building these two connections. To ease the understanding of the relevant algebraic structure, the mathematical concepts are reduced and simplified as much as possible. For example, the direct product, tensor product, Cartesian product and tensor contraction are treated as different views of the same process: constructing a new vector space from other subspaces, thereby avoiding confusion caused by excessive terminology. The section finally provides an interpretation of the existing literature in the light of

this subspace-oriented taxonomy and points out a gap when combining the required model ability (i.e. expressivity and capability) with specific vector space implementations (i.e. particular matrix or tensor factorisation methods).

Since the focus has been on a unified taxonomy, the coverage of literature is not exhaustive, and the unique details of each investigated work are not fully discussed. Some subspace-related insights are also difficult to align with specific matrix or tensor factorisation methods. Both aspects are left for further investigation.

Global information transmission. The final section [Sec. 3.3](#) traces how activations propagate through the full computational graph under this geometric model. It studies representation dimension, norm growth and information flow across layers, proposes a quantitative way to estimate the compressed models’ performance without actually compressing them, and herein relates them to the scaling law.

Chapter 4

Implementation with Object-Oriented Programming

This chapter offers an object-oriented software design, aimed at eliminating repeated efforts when extending new models as much as possible.

Before discussing the implementation requirements and specific details, [Sec. 4.1](#) reviews how similar open-sourced projects are structured and summarise the current state of the third-party projects used in this thesis. Then a requirement list is discussed in [Sec. 4.2](#), providing a clearer blueprint for what the system should be. After that, a detailed solution is given in [Sec. 4.3](#), and finally, the system is tested with Qwen2/Qwen3 and MiniCPM4, with quantitative results reported in [Sec. 4.6](#).

4.1 Relevant Open-Sourced Projects

The implementation of algebraic compression involves some important third party projects as its bedrocks, such that the functionalities of this project can be up-to-date and meet the practical requirements, e.g. cover rich language task evaluation, and use more advanced weights/activation analysis approaches.

Before talking about these bedrocks third-party projects, this section starts with the open-sourced projects that implement the similar functionalities as the proposed project in this thesis ([Sec. 4.1.1](#)). Analysing what requirements they meet helps to identify what gaps this project should fill.

Then this section introduces the critical third-party projects, which are further extended how they are integrated in the proposed project in this chapter in [Sec. 4.5](#).

4.1.1 Efficient Inference

vLLM¹ and **DeepSpeed**² are current well acknowledged open-sourced projects focusing on efficient inference of deep learning models. vLLM focuses on production-grade LLM inference (with OpenAI-compatible server, SageMaker deployment), and is a modular, production-oriented build pipeline for serving LLMs at scale. DeepSpeed is more monolithic, inference-centric. It aims at HPC environments with OpenMPI, and only involves low-level networking (e.g. Mellanox, `nv_peer_mem`).

vLLM [118] has evolved into a production-grade, engine-centric inference stack whose core design revolves around KV-cache management and fine-grained request scheduling.

The engine coordinates a first-class scheduler and block manager over single- and multi-GPU executors, exposing a step-based API that upper-layer servers build on. The engine manages KV memory via a PagedAttention-style, block-based allocator (BlockSpaceManager and BlockTable) configured through CacheConfig, enabling prefix sharing, sliding-window or long-context behaviour, CPU/GPU offload, and explicit accounting of cache utilisation. The codebase reflects a coherent, cache- and scheduler-aware inference architecture that integrates memory layout, batching policy, compilation into a single, configurable engine.

DeepSpeed has evolved into a highly engineered transformer **inference runtime** that can convert an ordinary PyTorch model into a multi-GPU, **quantised** (or mixture-of-experts), kernel-optimised engine via a single high-level initialisation interface.

Its core is a hierarchy of strongly typed configuration objects that unify tensor and expert parallelism, optimiser-style memory partitioning, quantisation (including mixture-of-quantisation schemes), GPU-kernel choices, and checkpoint reshaping under one coherent abstraction. Rather than forcing users to rewrite their models, DeepSpeed relies on a generic “kernel injection” mechanism that inspects existing transformer layers and replaces them with specialised inference implementations, so model code and serving logic remain largely unchanged.

4.1.2 Language Model Evaluation Harness

Language Model Evaluation Harness [36] (denoted as `lm_eval` in the following) is an open-sourced framework maintained by EleutherAI (MIT-licenced, versioned as `lm_eval`) that provides a unified backend for evaluating generative language models across a large collection of standardized benchmarks. It is the engine behind Hugging Face’s Open LLM Leaderboard and has been adopted by both academic and industrial labs to compare models under a common, reproducible protocol.

The framework decouples task definitions, model backends, and metrics: tasks are described via

¹<https://github.com/vllm-project/vllm>

²<https://github.com/deepspeedai/DeepSpeed>

configuration and lightweight Python hooks, while models implement a small LM interface exposing `loglikelihood`, `loglikelihood_rolling`, `generate_until`, and `multiple_choice` requests.

This project supports models loaded through `transformers`, GPT-NeoX, Megatron-DeepSpeed, vLLM and a variety of commercial or local API endpoints, plus adapters such as LoRA. Evaluation uses public prompts and scripts, making results reproducible, and exposes a CLI (`lm_eval` / `lm-eval`) as well as a Python API, so that external training frameworks can invoke the same task suite and reuse cached model outputs.

Coverage of language tasks. Tasks in the harness are organised into families (`lm_eval/tasks`) that cover most of the commonly reported LLM capabilities:

1. *Knowledge and reading comprehension:* benchmarks such as SQuAD-style reading tasks, RACE, DROP, CoQA, QASPER, and open-domain QA datasets like NQ-Open and TriviaQA.
2. *Multi-choice knowledge and exam-style:* suites including ARC, MMLU and its variants, CEval, AGIEval, HeadQA, and medical and legal exams probing broad factual knowledge and discipline-specific understanding.
3. *Commonsense and reasoning:* tasks such as HellaSwag, PIQA, SIQA, WinoGrande, WSC, ANLI, BIG-Bench-Hard and inverse-scaling tasks that target everyday reasoning, distractor robustness, and counterintuitive phenomena.
4. *Mathematical and symbolic reasoning:* GSM8K, MGSM, Minerva-style math sets, Hendrycks-Math, HRM8K, and related quantitative benchmarks.
5. *Code and program synthesis:* HumanEval, MBPP and CodeXGLUE-style tasks focusing on functional correctness under unit tests.
6. *Summarisation, long-context and discourse:* SCROLLS datasets, LongBench, document summarisation and report-generation benchmarks.
7. *Multilingual and translation:* suites such as WMT, XNLI, XQuAD, XStoryCloze, Belebele, MLQA, and regional GLUE-style benchmarks.
8. *Safety, bias and toxicity:* RealToxicityPrompts, ToxiGen, WMDP, CrowS-Pairs, BBQ, plus domain-specific collections in medicine, law and finance.

Task groups and aliases allow users to select entire families (e.g. “leaderboard” or “japanese_leaderboard”) and to obtain aggregated scores over dozens of subtasks.

Metrics and integration with model outputs. Internally, each evaluation example is represented as an `Instance` (`lm_eval/api/instance.py`) holding the document, its `request_type` (one of the four output types), model arguments and metadata.

During evaluation, tasks build a list of instances, which are then grouped by request type in `evaluator.evaluate`; the selected LM backend is called on each group, and its raw outputs are stored in `Instance.resps`.

Filter ensembles defined in the task configuration (e.g. for taking the first generation, normalising text, or extracting choices) populate `Instance.filtered_resps` under named filter keys.

For each document, the task-specific `process_results` method receives the original doc together with the filtered responses and returns a dictionary mapping metric names to per-example scores (for example, 0/1 accuracy, log-probabilities, BLEU tuples, or structured values). These per-example values are accumulated in a `TaskOutput.sample_metrics` structure, keyed by metric and filter. Global metrics are then computed in `TaskOutput.calculate_aggregate_metric`, which looks up aggregation functions from the task’s `aggregation()` map and from the metric registry in `metrics.py`.

Most metric functions registered via `@register_metric` are simple pass-throughs that declare the *metric name*, its *output type compatibility*, an *aggregation function* (e.g. mean, perplexity, BLEU, F1), and whether higher values are better. The aggregator functions (`@register_aggregation`) compute dataset-level scores and, via `stderr_for_metric`, optional bootstrap-based standard errors, so that the final result table contains both point estimates and uncertainty for each metric, task and group.

simple_evaluate pipeline. The top-level helper `simple_evaluate` in `evaluator.py` wraps the full evaluation lifecycle from a user-facing interface and proceeds as follows:

1. Configure logging and random seeds (Python, NumPy, PyTorch, and a separate few-shot sampler seed), validate that either a global `limit` or an explicit index list `samples` is used, emit warnings when an instruction-tuned model is evaluated without a chat template, and delete any existing request cache if requested.
2. Instantiate the model: if the `model` argument is a string, look up the corresponding LM subclass in the registry and call its creation helpers with `model_args`, batch size and device; otherwise validate that a pre-initialised LM object was passed.
3. Optionally wrap the model in a `CachingLM` with per-rank SQLite databases to enable response caching.
4. Build a `TaskManager` and `task_dict`, adjust each task’s configuration (generation parameters, number of few-shot examples, predict-only mode), and optionally run task-specific integrity tests.

5. Log configuration for experiment tracking (model source, arguments, system prompt, chat template) to an `EvaluationTracker` if provided, delegate the core loop to `evaluate()`, and on rank 0 enrich the returned metrics with execution metadata (model name, arguments, device, batch sizes, seeds, cache path, bootstrap settings, git hash, environment and tokenizer information, timestamp) before returning a single nested result dictionary, while non-zero ranks return `None`.

4.1.3 Representation Quality Metrics

`information_flow` [94] couples a unified, matrix-entropy view of representations with broad layer-wise experiments. The authors show that a single Gram-matrix entropy lens bridges diversity, geometry, and invariance and that seven metrics derived from it correlate tightly with downstream accuracy. Across Pythia decoders, BERT encoders, Mamba state-space models, and finetuned Llama-3 variants, intermediate layers often outperform final layers, scale amplifies mid-layer compression, and finetuning reshapes entropy profiles. The open-source `information_flow` codebase operationalizes this framework, exposing a harness that sweeps layers, computes the metrics, and evaluates MTEB tasks under a unified configuration so empirical results directly test the theoretical claims.

Matrix-based entropy. Let token (or sample) embeddings form $\mathbf{Z} \in \mathbb{R}^{N \times D}$ with Gram matrix $\mathbf{K} = \phi(\mathbf{Z})\phi(\mathbf{Z})^\top$ using a positive definite kernel. The matrix Rényi entropy of order $\alpha > 0$ over eigenvalues $\lambda_i(\mathbf{K})$ is

$$S_\alpha(\mathbf{Z}) = \frac{1}{1 - \alpha} \log \left(\sum_{i=1}^r \left(\frac{\lambda_i(\mathbf{K})}{\text{tr}(\mathbf{K})} \right)^\alpha \right), \quad (4.1)$$

providing a shared foundation for all downstream metrics. The trace normalisation makes S_α invariant to rescaling, while the choice of α controls sensitivity to heavy tails versus uniform spectra, letting a single expression capture compression, clustering, and smoothness effects.

Prompt entropy. Applying (4.1) to the Gram matrix of token embeddings inside a single prompt, $\mathbf{Z}^{(\text{prompt})}$, yields $S_\alpha(\mathbf{Z}^{(\text{prompt})})$, which grows when tokens occupy diverse directions and shrinks when they compress to a few principal components. Higher prompt entropy signals rich, non-redundant token-level structure; low values indicate aggressive compression or over-regularisation.

Dataset entropy. Averaging token embeddings per prompt produces $\bar{\mathbf{Z}} \in \mathbb{R}^{N \times D}$. The dataset entropy $S_\alpha(\bar{\mathbf{Z}})$ measures how broadly prompts spread in representation space, revealing whether layers cluster inputs tightly or maintain global diversity across samples. In the chapter, entropy dips mark compression zones that often coincide with peaks in downstream accuracy.

Effective rank. Taking the Shannon limit $\alpha \rightarrow 1$ gives S_1 , and the effective rank follows as

$$\text{EffRank}(\mathbf{Z}) = \exp(S_1(\mathbf{Z})), \quad (4.2)$$

which lower-bounds the number of energetic dimensions in $\mathbf{K}$. A high effective rank implies near-uniform eigenvalues (distributed variance), while a low rank reveals heavy-tailed spectra where few directions dominate. The authors of `information_flow` point out that RankMe [119] and LiDAR [120] are effectively reparameterisations of this entropy-derived quantity.

Curvature. Viewing a prompt as a path in $\mathbb{R}^D$, set $\mathbf{v}_k = \mathbf{z}_{k+1} - \mathbf{z}_k$ for consecutive tokens. The average turning angle

$$\bar{C} = \frac{1}{L-2} \sum_{k=1}^{L-2} \arccos\left(\frac{\mathbf{v}_{k+1}^\top \mathbf{v}_k}{\|\mathbf{v}_{k+1}\| \|\mathbf{v}_k\|}\right) \quad (4.3)$$

is high when trajectories bend sharply (local features) and low when they move smoothly (global features). Empirically, curvature spikes in early layers for discrete cleanup and smooths in middle layers as models integrate longer-range semantics.

InfoNCE [121]. For paired augmentations $(\mathbf{z}_i, \mathbf{z}_i^+)$ and negatives $\mathbf{z}_j^-$, invariance is scored by

$$\mathcal{L}_{\text{InfoNCE}} = -\frac{1}{N} \sum_{i=1}^N \log \frac{\exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_i^+)/\tau)}{\sum_{j=1}^N \exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_j^-)/\tau)}, \quad (4.4)$$

so a lower loss denotes stronger augmentation invariance. The similarity is cosine in the codebase, and the temperature τ (typically 0.1–0.2) tunes how sharply positives must align for a layer to be considered invariant.

LiDAR. Treating each prompt as a class, LiDAR compares between-class scatter S_b to within-class scatter S_w :

$$\text{LiDAR} = \frac{\text{tr}(S_b)}{\text{tr}(S_w + \epsilon I)}, \quad (4.5)$$

where S_b uses class means and S_w uses deviations of augmentations around their class mean. Higher values mean augmentations cluster tightly and are well separated from other prompts. In practice, LiDAR rises in layers that also show high effective rank, indicating that spread across classes coexists with tight within-class cohesion.

DiME [122]. A matrix-entropy analogue of mutual information, DiME compares the joint Gram matrix of paired augmentations against their marginals:

$$\text{DiME}(\mathbf{Z}_1, \mathbf{Z}_2) = S_\alpha\left(\begin{bmatrix} \mathbf{Z}_1 \\ \mathbf{Z}_2 \end{bmatrix}\right) - \frac{1}{2}(S_\alpha(\mathbf{Z}_1) + S_\alpha(\mathbf{Z}_2)). \quad (4.6)$$

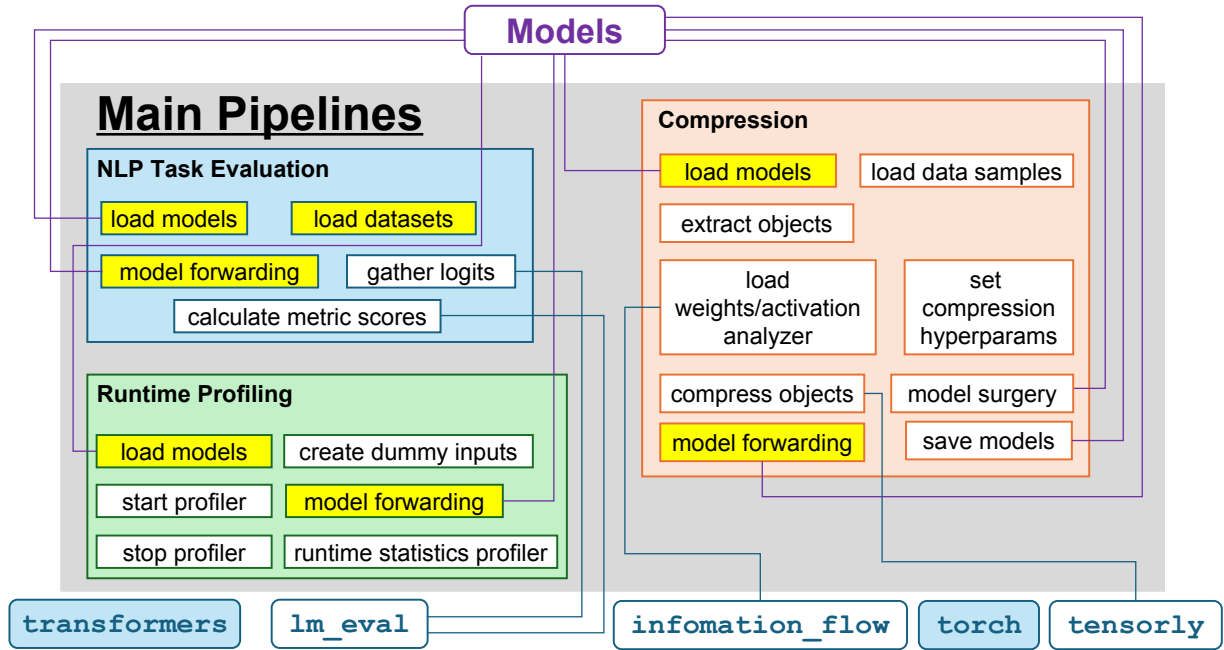


Figure 4.1: Main pipelines (with involved functions), fundamental resources (models) and third-party projects in `toggle`. The functions with yellow background are mutually used functions in different pipelines. The dependencies with blue background are the foundation stones for the whole codebase.

Large DiME indicates that true pairs share structure beyond what either set shows alone, signalling robust alignment under perturbations. The implementation additionally computes a shuffled baseline so the gap isolates true cross-augmentation signal rather than spurious batch correlations.

4.2 Requirements and Design Principles

`toggle` is designed as a unified tooling project to compress model weights and activations with low-rank factorisation (matrices and tensors) and pruning. It supports a large variety of language models and also compresses based on entropy or matrix analysis with actual datasets.

According to the relevant open-sourced projects discussed in [Sec. 4.1](#), and the required functionalities mentioned in [Chapter 3](#), `toggle` should contain pipelines of compression, NLP task evaluation, and runtime profiling, summarised in [Fig. 4.1](#).

In [Fig. 4.1](#), there is significant reuse of several functions (e.g. `load models`, `model forwarding`) and resources (i.e. models). Pipelines that involve these functions and resources should avoid redundancies as much as possible. Moreover, the involvement of third-party projects (`lm_eval` and `information_flow`) spans multiple functions. Interfaces between these two third-party projects and

`toggle` functions should also avoid redundancies.

In summary, the design principles for `toggle` are:

1. **Avoid two kinds of redundancies:**
 - (a) re-used functions across different pipelines;
 - (b) interfaces with third-party dependencies and functions.
2. **Frequently accessed resources (i.e. models) should be managed independently from any pipeline.**

For reducing redundancies, factory mode and plugins are the ideal practice in software engineering, which will be discussed in [Sec. 4.3](#). For model management, except for defining a specific class `ModelManager`, the internal components of different models should also be investigated for compact and easy-to-extend implementation, which will be discussed in [Sec. 4.4](#).

4.3 Factory Mode

`toggle` follows a set of registries (model loaders, compressors, evaluators, profilers) and factory functions that stitch them together according to the configuration. Each *pipeline* is assembled by the factory so that cross-cutting concerns—logging, reproducible random seeds, device/dtype policy, and checkpoint paths—are handled once and shared. *Plugins* supply new compression algorithms or adapters to third-party projects (`lm_eval`, `information_flow`) by implementing a thin interface; the factory then wires them into the existing pipeline without touching other components. This pattern eliminates copy-pasted initialisation code across pipelines and reduces the surface area that must change when adding a model family or swapping third-party dependencies.

Workflow. In the context of software engineering, a workflow denotes an ordered set of tasks with explicit dependencies, predicates, and failure semantics, capturing how discrete computational units advance state. In `toggle`, `Workflow` is the executable graph that organises plugin invocations into ordered `WorkflowStep` nodes (see [Fig. 4.2](#)) with explicit dependencies, readiness checks, and lifecycle states (pending, ready, running, completed, failed, skipped). Each step carries configuration, retry and timeout policy, optional parallelism, and emits execution metadata. The workflow object aggregates these steps, validates the dependency Directed Acyclic Graph (DAG), tracks completion and failure sets, and produces execution summaries.

Pipeline. Compared with workflow, a pipeline represents a broader, end-to-end data or model processing path, often spanning data acquisition, transformation, analysis, and reporting. A pipeline

provides the broader operational context in which workflows run. In `toggle`, it is represented by the `PipelineContext` and `PipelineOrchestrator`: shared configuration, workspace, and state persistence (`PipelineState` via `StateManager`).

A pipeline run proceeds in four phases. ① The controller builds `PipelineContext` (workspace, config, state) and loads a `Workflow` description. ② Available plugins are registered in the registry, then the `Workflow` is handed to the scheduler. ③ The scheduler normalises settings, binds status defaults, and primes the `EventBus` with `pipeline.started`. ④ Model/tokenizer handles are registered once through `ModelManager` so subsequent steps share the same resources.

Event. An event is a discrete, timestamped notification that something occurred, often broadcast to decouple producers from consumers via publish/subscribe. In `toggle`, events serve pipelines, flowing through `EventBus` as `EventMessage` objects carrying payloads and source metadata. Standard event names are enumerated in `PipelineEvent` (pipeline, workflow, model, compression, evaluation, system), and plugins use `PluginEventManager` to emit started/progress/completed/failed signals. This synchronous bus enables observability, coordination, and lightweight extensibility without hard-wired caller/callee dependencies.

Relationships between Workflow, Pipeline and Event. In this system, pipelines, workflows, and events form complementary layers of coordination.

A pipeline establishes the operational context: shared configuration, workspace, and state managed by `PipelineContext` and executed under the `PipelineOrchestrator`. Within that context, a `workflow` encodes the executable control flow as a dependency-resolved graph of `WorkflowStep` nodes, each binding a plugin, configuration, and execution policy (retry, timeout, parallelism). The `pipeline` provisions resources and lifecycle boundaries; the `workflow` specifies the sequence and concurrency of tasks that realise the pipeline's objective.

Events provide the communication fabric that links these layers without tight coupling. The `EventBus` transports `EventMessage` instances tagged by `event.type`, source, and timestamp. Core event types are enumerated in `PipelineEvent`, covering pipeline lifecycle (started, completed, failed), workflow progression (step started/completed/failed), and domain-specific signals (model, compression, evaluation). The orchestrator emits pipeline and workflow events to track progress and failure, while plugins use `PluginEventManager` to publish started/progress/completed/failed signals during step execution. Handlers can subscribe by event type (or wildcard), enabling logging, monitoring, or adaptive behaviour (e.g., conditional retries or cancellation) without modifying the emitting components.

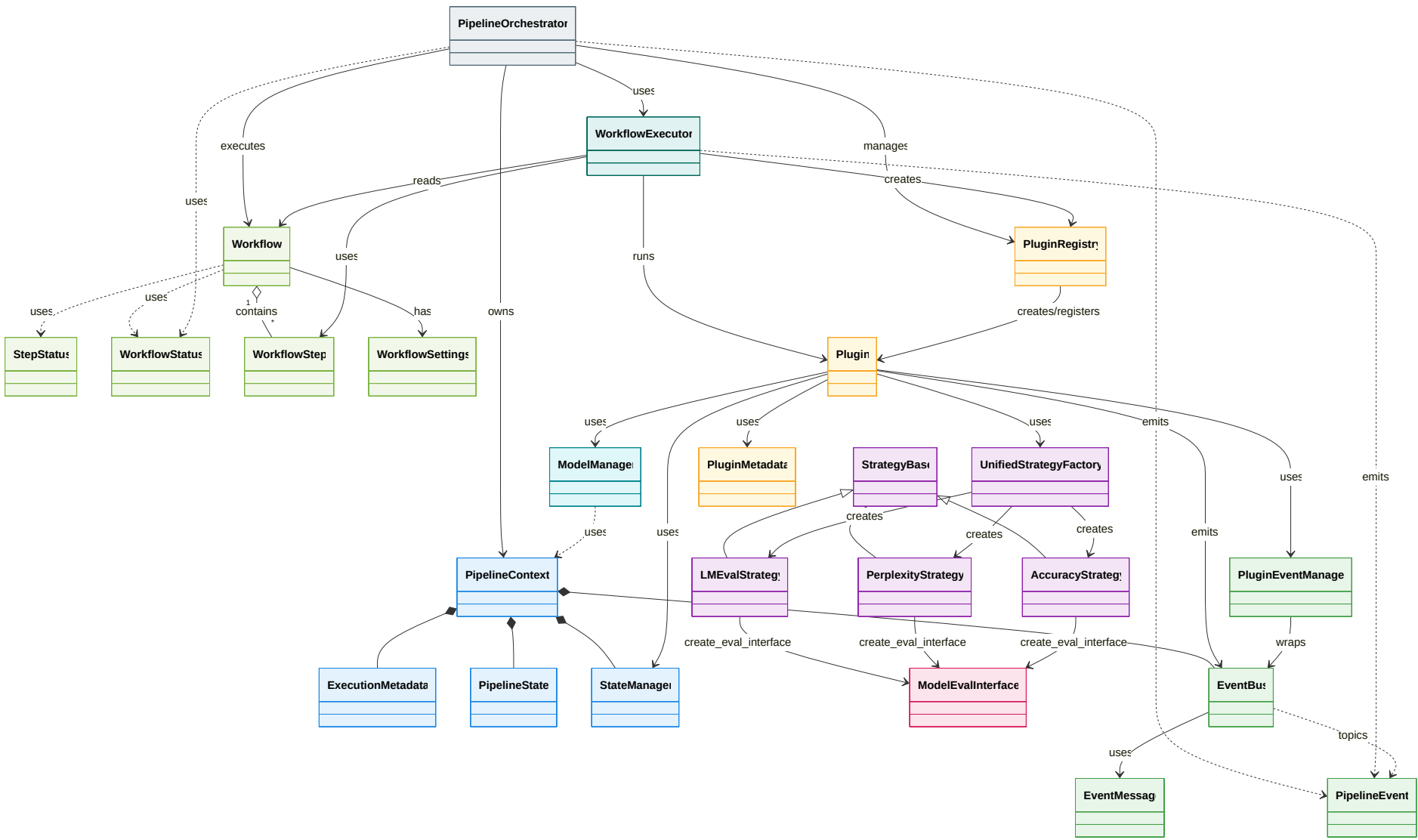


Figure 4.2: Core factory wiring with plugin handoff in toggle.

The functionalities and relationship between pipeline, workflow, and event can be summarised as

1. `pipeline` supplies scope and resources;
2. `workflow` provides structured execution;
3. `event` system mediates observation and coordination across both.

This separation of concerns yields **modularity** - steps can be added or reordered without altering observers, and **extensibility**, as new event consumers can introduce analytics, dashboards, or policy enforcement without invasive code changes. Together, they support reproducible, observable, and resilient end-to-end runs.

Plugins. A plugin is an extensible execution unit that encapsulates pipeline logic behind a uniform lifecycle. Specifically, ① The plugin registry/factory materialises extension objects by name, keeping discovery out of hot-path execution. ② Each plugin follows a uniform lifecycle (`initialize`, `execute` → `do_execute`, `cleanup`) and emits telemetry via standardized event helpers. ③ Algorithm choice is delegated to a strategy factory so compression/evaluation/analysis behaviours are injected, preserving polymorphic substitution. ④ Lightweight metadata attaches category/version labels to plugins for reflective inspection without coupling to execution logic.

In `toggle`, `PluginRegistry` supplies registration, discovery, and instantiation, while execution telemetry is mediated by the event system. The plugins span compression (low-rank/pruning), evaluation (LM harness adapters), analysis (weight/activation metrics), model loading, and reporting.

As shown in Fig. 4.2, `PluginRegistry` is the factory and registry that materializes Plugin subclasses by name, enforcing discoverability and decoupled creation. Each `Plugin` implements a uniform lifecycle (`initialize`, `execute` → `do_execute`, `cleanup`) and emits metrics and events. `PluginMetadata` carries classification data (category, version) to keep reflective operations out of hot-path execution.

Strategy. A strategy is a narrower, reusable computational policy - e.g., a compression algorithm, an evaluation routine, or an analysis method: subclassing `StrategyBase` and exposing an `execute` method. `UnifiedStrategyFactory` resolves strategy names to concrete implementations, validates parameters, and returns configured strategy instances across compression, evaluation, and analysis domains.

In `toggle`, strategies cover ① compression methods (`tensor_train`, `svd`, `tucker`, `cp`), ② evaluation routines (LM harness, perplexity, accuracy), and ③ analysis types (weight metrics, activation metrics), all resolved through the unified factory `UnifiedStrategyFactory`. Strategy selection is delegated to `UnifiedStrategyFactory` so that compression/evaluation/analysis algorithms are injected rather than hard-coded, allowing polymorphic substitution.

The relationship between plugins and strategies. Plugins and strategies support each other by decoupling orchestration concerns from algorithmic policy. Plugins orchestrate I/O, state updates, and event emission inside workflow steps, and delegate core computation to strategies obtained via the factory. This division allows plugins to remain thin coordinators, while strategies remain more focused, testable components. It also enables dynamic substitution (e.g., selecting `svd` vs. `tensor_train` without altering plugin code) and shared reuse across multiple plugins.

Separating plugins and strategies yields several benefits: ① single-responsibility alignment: plugins manage lifecycle, context, and observability; strategies only encapsulate algorithmic behaviour; ② reduced code redundancy: common orchestration boilerplate resides in the base plugin, while algorithm implementations are centralised in strategy classes; ③ configurability and extensibility: factories admit registration of new strategies without refactoring plugin call sites; ④ improved testability: strategies can be unit-tested independently of workflow execution, and plugins can be exercised with mock strategies; ⑤ clearer dependency management - dynamic imports in the factory avoid circular references and constrain coupling between orchestration and algorithm modules.

“Communication” in Pipelines. The shared context objects in pipelines split three duties. ① `PipelineContext` is the facade for configuration, workspace resolution, and the `EventBus`, holding `PipelineState` plus its `StateManager` for persistence. ② `EventBus` and `PluginEventManager` provide pub-sub and standardised payloads, enabling loose coupling among orchestrator, plugins, and observers. ③ `ModelManager` centralizes model/tokenizer acquisition and validation, preventing duplicated device/dtype policy in plugins. These context objects separate ownership (state/resources) from behaviour (plugins), while enforcing safe access to scarce resources. A more detailed clarification for this kind of “communication” will be introduced in [Sec. 4.4](#).

Scheduling and lifecycle during execution. The scheduler loop repeats three actions. ① Query `Workflow.get_ready_steps` to select dependency-satisfied steps. ② Drive each `WorkflowStep` through `PENDING` → `READY` → `RUNNING` → `COMPLETED`/`FAILED`/`SKIPPED` with retries and timeouts, dispatching in parallel when allowed. ③ Emit `workflow.step.*` events on each transition so observers (e.g., profilers) react without in-band coupling.

Plugin invocation and resource coordination Step execution couples four coordinated operations. ① Instantiate the named plugin via the registry/factory. ② Initialise with the shared context, pulling the appropriate model/tokenizer variant via `ModelManager`. ③ Load strategy implementations through the strategy factory and write artifacts into `PipelineState`. ④ Emit events via `PluginEventManager`; failures surface as `plugin.failed` without corrupting shared state because updates are scoped per step.

Feedback, persistence, and reporting. System feedback is concentrated in three channels. ① The scheduler aggregates per-step results into `Workflow.get_execution_summary`. ② `StateManager` persists `PipelineState` snapshots for reproducibility or later inspection, while `EventBus` retains a bounded history for timeline reconstruction. ③ Report builders (e.g., statistics managers) read state and emit JSON/CSV artifacts without knowing workflow internals, reinforcing separation of concerns between orchestration and analytics.

Summary. This architecture minimises redundancy by centralising shared resources (models, state, events) and delegating behaviour to interchangeable plugins and strategies, aligning with the requirements in [Sec. 4.2](#). Avoiding duplicated interfaces to third-party tooling and isolating frequently accessed resources satisfy the stated design principles, while the clear lifecycle boundaries improve extensibility and fault isolation needed for compression, evaluation, and profiling pipelines.

4.4 Resource Management

In object-oriented programming, a class named `*Manager` is usually a sign that the class controls, coordinates, or owns other objects or system resources. In `toggle`, such frequently used resources are

1. fundamental resources: language models;
2. orchestration resources (as introduced in [Sec. 4.3](#)): plugins and pipeline state;
3. analyser resources: weight and activation statistics used by `information_flow`.

ModelManager provides a central facade over shared model/tokenizer resources in the pipeline; it controls retrieval/setting for baseline vs compressed checkpoints, validates structure (including compressed-layer detection), computes comparisons, retrieval, validation, and lifecycle for plugins. It provides:

1. unified loading with consistent dtype/device policy and cache reuse across runs;
2. lightweight adapters that map model-specific attribute names to the common handles used by compression code (e.g. `model.layers`).

Unlike existing projects like SliceGPT [\[113\]](#) and ASVD [\[123\]](#), which write specific hooks for each model family, `toggle` uses the common components of the language models to avoid case-by-case hooks. In other words, `toggle` **implements compression algorithms in a model-agnostic manner**.

The reason we can do this is that the shared components across different model families take the dominant portions of model inference runtime. Many recent decoder-only models share the same core building blocks, the manager can present one consistent interface while still honouring

family-specific switches (e.g. sliding attention or logits softcapping). These shared components are summarised in [Tab. 4.1](#).

For instance, `toggle` normalises access to the main transformer blocks through a unified path, typically `model.layers`, regardless of the underlying model’s specific naming conventions (e.g., `model.transformer.h` or `model.model.layers`). This abstraction allows compression strategies like layer pruning and low-rank factorisation to be applied uniformly. As a result, extending `toggle` to support a new model family that adheres to this common architectural pattern becomes a matter of configuration rather than significant engineering effort, primarily involving mapping the model’s unique layer names to `toggle`’s standardised representation. A more detailed common blocks summary is in [Tab. 4.1](#).

System orchestration relevant Managers contain `StateManager` and `PluginEventManager`. ① `StateManager` owns persistence lifecycle for `PipelineState`, serializing/deserializing with locking and metadata to disk. It centrally controls how the state is saved/loaded rather than just exposing the raw state. ② `PluginEventManager` mediates event emission for a plugin, standardises event names/payloads and routes through the shared `EventBus`. Acts as a registrar/dispatcher layer, matching the registration/dispatch heuristic.

Statistics-relevant Managers serve with `information_flow` to assist hyperparameter setting for model compression. ① `StatisticsManager` coordinates sampling and aggregation runs and persists statistics artifacts for reuse instead of scattering ad-hoc files. ② `WeightsStatisticsManager` shapes raw weight-metric samples into structured reports and aggregated reductions, defining reporting policies across layers/module types/global summaries and coordinating multiple objects’ data into one artifact. ③ `ActivationsStatisticsManager` plays the same role for activation metrics, aggregating series, shape histograms, and reductions into reports, and centralising complex aggregation logic and output schema.

Table 4.1: Common building blocks of Llama3, Qwen3 and Gemma3 family models.

Feature	Llama	Qwen3	Gemma3
Decoder-only text LM	✓	✓	✓ (text backbone)
RMSNorm	✓	✓	✓
Gated MLP (gate/up/down)	✓	✓	✓
RoPE + <code>rope_scaling</code>	✓	✓	✓
GQA/MQA via <code>num_key_value_heads</code>	✓	✓	✓
Cache support (<code>Cache/DynamicCache</code>)	✓	✓	✓
Causal mask support	✓	✓	✓
Attention backend switch (<code>_attn_implementation</code>)	✓	✓	✓
Generation head (<code>ForCausalLM + GenerationMixin</code>)	✓	✓	✓
Logits slicing (<code>logits_to_keep</code>)	✓	✓	✓

Table 4.2: Managers and the resources they coordinate in `toggle`.

Managed Resources/Objects	Manager Responsibilities	Key Functions in <code>toggle</code>
Model	Centralizes model/tokenizer retrieval, validation, and lifecycle for plugins.	<code>get_model</code> , <code>set_model</code> , <code>get_tokenizer</code> , <code>set_tokenizer</code> , <code>validate_model</code> , <code>compare_models</code> , <code>get_model_info</code>
PluginEvent	Standardizes plugin event emission and dispatch via the shared EventBus.	<code>emit_started</code> , <code>emit_progress</code> , <code>emit_completed</code> , <code>emit_failed</code>
State	Persists and restores pipeline state snapshots to controlled storage.	<code>save_state</code> , <code>load_state</code>
Statistics	Provides workspace-aware report pathing and JSON/CSV persistence helpers.	<code>build_report_path</code> , <code>save_json</code> , <code>save_csv</code> , <code>load_json</code>
WeightsStatistics	Shapes weight-metric results into layered/global summaries with reductions.	<code>build_report</code> , <code>aggregate_by_module_type</code> , <code>aggregate_global</code> , <code>extract_csv_rows</code>
ActivationsStatistics	Aggregates activation metric series, shapes, and reductions across modules.	<code>build_report</code> , <code>aggregate_by_module_type</code> , <code>aggregate_global</code> , <code>extract_csv_rows</code>

4.5 Integration with Third-Party Projects

The integration with third parties mainly lies in the subclasses of `Plugin` (except for foundation dependencies like `torch` and `transformers`, which are at the same layer as `Plugin`).

Weights/activation analyser with `information_flow`. `ExternalMetricsBackend` functions as a reflection-driven integration layer that enables this codebase to incorporate third-party metrics providers without tight coupling. It extends the `MetricsBackend` abstraction, preserving a stable interface for metric discovery and invocation while permitting late binding to external code. The design treats external metrics as dependencies resolved at runtime via configuration or ambient environment, aligning with dependency injection principles and minimising build-time assumptions about available metric libraries. This approach is essential for both activation-level and weight-level analyses described in the implementation plans, where metric functions are treated as pluggable computation kernels rather than built-ins.

The backend’s resolution strategy follows a deterministic multi-stage pipeline. First, it attempts to honour explicit configuration by importing a module path or loading a file path supplied at construction. If neither is provided or the import fails, it evaluates dedicated environment variables that specify equivalent module or file targets. Should those sources be absent or invalid, it probes predefined default module paths associated with a canonical third-party metrics library. Only after

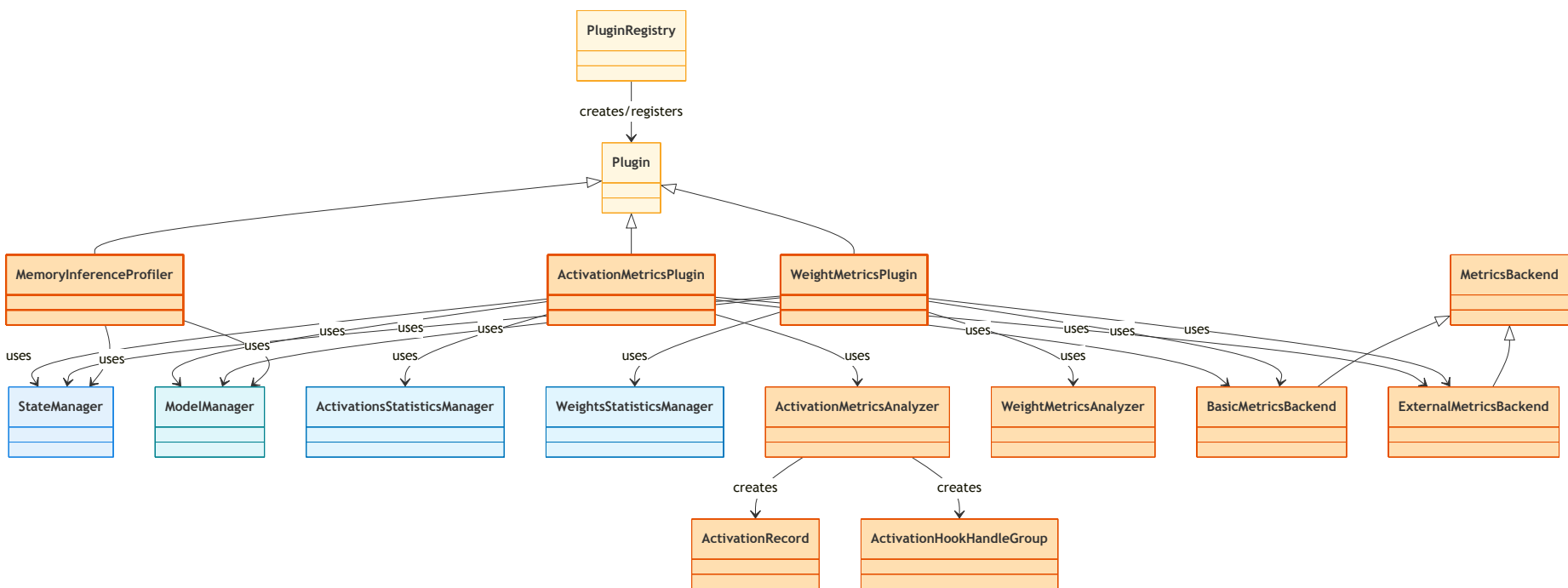


Figure 4.3: Workflow wired with Weight/Activation Analyser in toggle.

these attempts fail does it fall back to an empty registry while signalling failure via provenance metadata. This ordered resolution embodies a clear precedence model: user configuration supersedes environment hints, which in turn supersede curated defaults, ensuring predictable dependency binding while maintaining robustness in heterogeneous deployment environments.

Two complementary import mechanisms operationalise this resolution. For module-based integration, the backend delegates to the runtime import system, relying on the interpreter’s module search semantics to locate the external dependency. For file-based integration, it constructs a loader through the import machinery, synthesises a module specification from the concrete file location, materialises a module object, executes it in a controlled scope, and registers it in the process module cache. This method enables ad hoc integration of metrics shipped as standalone files outside the standard package boundaries while preserving module-level semantics such as global state and subsequent imports. Error handling is intentionally permissive: any exception during import is absorbed, and the backend proceeds to the next resolution stage, preventing a failed third-party binding from cascading into system-wide failures.

Once a module is successfully materialised, **ExternalMetricsBackend** performs reflective introspection to construct its registry. It enumerates public attributes, filters for callables with at least one positional parameter, and applies optional name normalisation by stripping a configured prefix. The resulting mapping is stored with lowercased keys to enable case-insensitive lookup, yet it retains original callable references to preserve third-party behaviour. This introspective curation decouples the system from specific naming schemes used by external libraries, facilitating interoperability even when third-party functions expose prefixes or stylistic conventions divergent from the host codebase. The registry thereby acts as an adapter that translates a foreign metric surface into a normalised internal contract, consistent with the **MetricsBackend** interface.

Provenance tracking accompanies registry construction to maintain observability over dependency resolution. The backend records whether an external source was found, which import path or file was used, and which resolution stage succeeded. This metadata is exposed through a property for downstream components to embed in analysis reports, enabling reproducibility and auditability of metric computations. In scenarios where no external dependency is available, the **BasicMetricsBackend** provides a deterministic fallback set of scalar computations over tensors, ensuring analytical routines remain functional albeit with a reduced metric vocabulary. The dual-backend pattern thus satisfies fault tolerance requirements while promoting optimal utilisation of richer third-party metrics when present.

Integration with the weight-only analysis workflow leverages this backend to supply metric functions to the analyser and subsequent reporting pipeline. The **WeightMetricsAnalyzer** receives callable metrics from **ExternalMetricsBackend** (or the fallback) and executes them over selected parameter tensors under no-gradient contexts, ensuring computational efficiency and immutability of model parameters. The resulting scalar outputs are aggregated by the statistics manager into structured per-layer, per-module-type, and global summaries, with provenance annotations that capture ex-

ternal metric resolution outcomes. This arrangement exemplifies inversion of control: the analyser is agnostic to the origin of the metric functions, depending solely on the abstract backend contract, while the plugin orchestrator configures the concrete backend according to user configuration, environment, or defaults.

In the activation metrics workflow, the same backend underpins runtime metric computations applied to intermediate activations captured via forward hooks. The `ActivationMetricsAnalyzer` invokes metrics provided by `ExternalMetricsBackend` on tensors emitted during model forward passes, enabling real-time statistical instrumentation without modifying model code. Reports generated from these metrics include provenance data describing whether third-party metric sources were engaged, thereby supporting experimental reproducibility and comparative evaluation across runs with differing metric providers. Because the activation pipeline uses standardised strategy and plugin abstractions similar to the weight analysis, the backend’s pluggable design ensures consistent metric semantics across both tensor categories while maintaining decoupling from any specific external library implementation.

Architecturally, the backend’s behaviour embodies reflective loading, runtime binding, and graceful degradation. By encapsulating import logic and registry normalisation, it isolates third-party integration concerns from the analytical algorithms themselves, reducing the surface area for compatibility issues when upgrading external libraries or altering deployment environments. The use of environment-variable fallbacks acknowledges operational contexts where configuration files cannot be modified, while the inclusion of default search paths targets common third-party packages to streamline onboarding. In combination, these mechanisms illustrate a design that balances extensibility, resilience, and operational pragmatism within a plugin-based analytics architecture.

This integration model also addresses software engineering concerns of maintainability and testability. The backend’s deterministic resolution order and explicit provenance facilitate unit testing of import scenarios and behaviour under missing dependencies, as outlined in the weight-only testing plan. Its separation from computation logic allows analytical components to be validated independently using the fallback metrics, while integration tests can assert correct binding to third-party metrics in controlled environments. Consequently, `ExternalMetricsBackend` serves as a cohesive adapter layer that harmonises third-party metric sources with the system’s analytical pipelines, ensuring that both weight-centric and activation-centric analyses can exploit external metric ecosystems without sacrificing reliability or clarity of execution.

Language task evaluation with `lm_eval`. The analysis subsystem integrates with the LM evaluation stack by orchestrating model forward execution through the `LMHarness` abstraction defined in the evaluation plugin layer. Within the activation analytics pathway, the `ActivationMetricsPlugin` treats the evaluation component as a driving mechanism that triggers forward passes while capture hooks are active. This design decouples metric computation from data generation: evaluation infrastructure supplies semantically meaningful inputs and batching policy, whereas the analyser remains

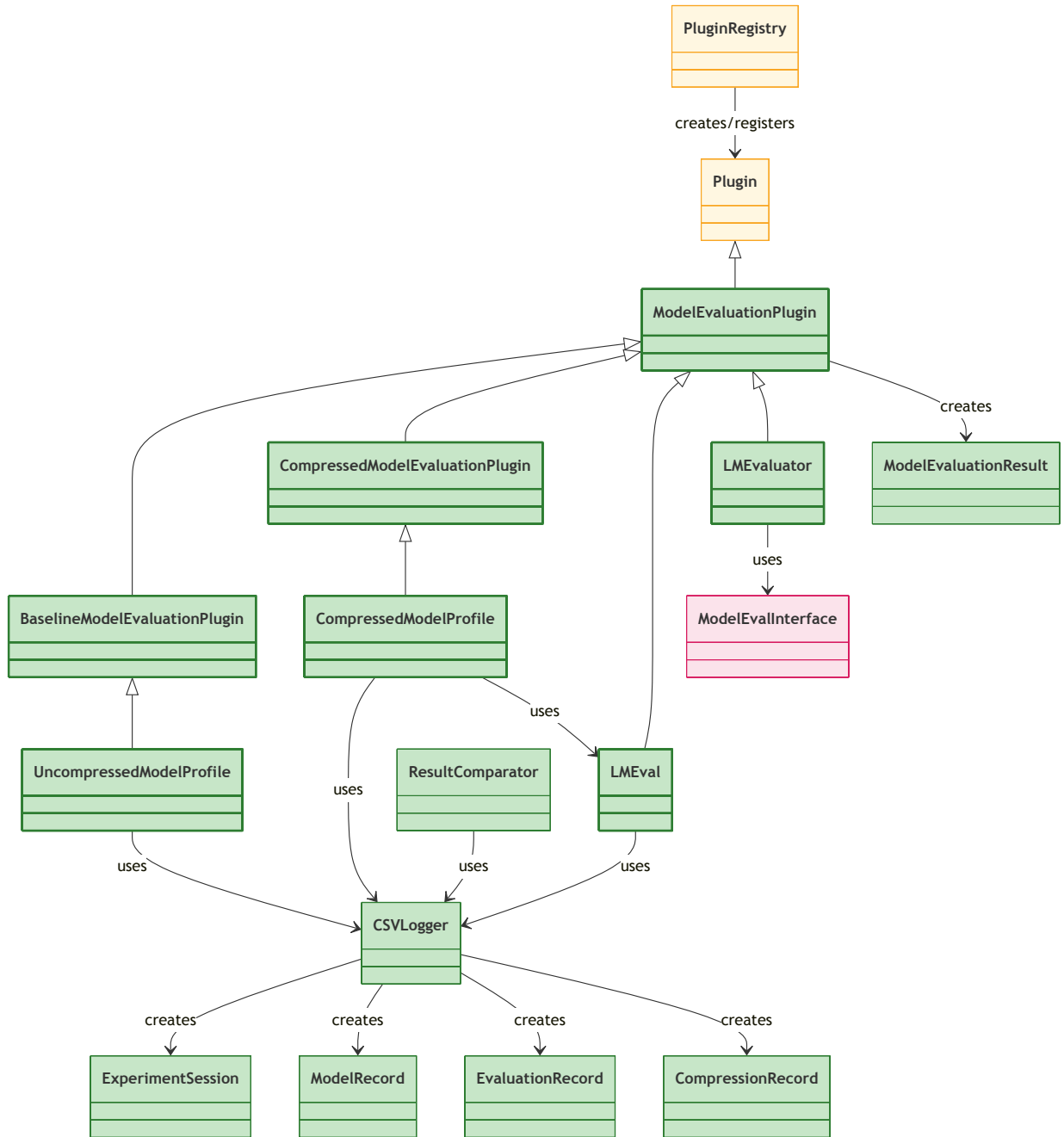


Figure 4.4: Workflow wired with Language Model Evaluation Harness (`lm_eval`) in `toggle`.

focused on tensor instrumentation. The arrangement leverages dependency injection; the plugin resolves evaluation parameters from runtime configuration, passes them to **LMHarness**, and relies on the harness to materialize the appropriate task workload.

The integration follows a guarded resolution strategy. The plugin attempts to import **LMHarness** and its availability flag from the evaluation module. If the optional dependency is present and the configuration provides a task and tokenizer, the plugin constructs a harness instance parameterised by task name, batch size, sample limit, and device auto-selection. The harness then executes the specified benchmark, invoking model forward passes in a manner consistent with the EleutherAI LM Evaluation Harness semantics. Hooks registered by the analyser intercept the model outputs during these passes, enabling activation-level metric computation without altering the evaluation workflow. This coupling is intentionally loose: the plugin does not assume a particular evaluation backend beyond the harness interface, thereby preserving architectural modularity and enabling substitution or extension.

In the absence of the external LM evaluation dependency, or when required inputs such as the tokenizer are missing, the plugin degrades gracefully to a minimal forward-driving routine. This fallback synthesizes a small random input based on introspected linear layer dimensions to trigger at least one forward path, ensuring that metric capture does not deadlock the pipeline. Although this path lacks semantic task alignment, it preserves system liveness and supports smoke-testing scenarios where full evaluation resources are unavailable. The fallback is encapsulated within the same plugin to retain a unified control surface for both comprehensive and minimal execution contexts.

The hook-based analyser, **ActivationMetricsAnalyzer**, operates independently of the evaluation source. Prior to hook registration, the plugin assembles a metric registry by merging contributions from **BasicMetricsBackend** and **ExternalMetricsBackend**, preferring externally supplied implementations when they overlap. The analyser receives this registry alongside compute and reduction directives, then registers forward hooks on selected modules according to declarative selection criteria. During evaluation, each hook normalises the module output, optionally migrates tensors to CPU to standardize metric execution, and dispatches the tensor to the metric functions. The analyser aggregates results as time series, captures shape statistics for non-scalar outputs, and produces raw layer-wise records for subsequent reporting. LM evaluation is thus one of several orchestrated triggers; the analyser itself remains agnostic to whether activations originate from benchmarked text or synthetic input.

The interaction with **LMHarness** preserves measurement provenance. The plugin records whether the external LM evaluation pathway was utilised, the task identifier, and runtime characteristics. This metadata is conveyed to the statistics manager, **ActivationsStatisticsManager**, which embeds it into the final report under analysis and provenance sections. Downstream utilities such as the report loader parse the LM evaluation task field to support summarisation and filtering, evidencing the system’s concern for traceability when external evaluation sources influence activation statistics.

From a control-flow perspective, the plugin adheres to the framework’s plugin lifecycle. It resolves models and tokenizers via the model manager, configures metric backends, initialises the analyser, registers hooks, delegates forward execution to `LMHarness` when available, and finally invokes the statistics manager to construct and persist reports. The hook removal is guaranteed via a finalization step, preventing resource leakage regardless of evaluation success. This lifecycle underscores separation of concerns: evaluation drives data, analysis computes metrics, and reporting persists structured artifacts.

The weight-centric analysis pathway (`WeightMetricsPlugin`) omits LM evaluation entirely, underscoring that the LM harness integration is purposefully scoped to activation capture where semantic inputs affect intermediate representations. Weight analysis operates purely on static parameters under inference-mode safeguards, sourcing metrics from the same backend abstraction but requiring no task-driven stimuli. This contrast highlights the architectural flexibility afforded by shared abstractions: a common metric backend and statistics manager can serve both dynamic and static measurement contexts, while evaluation coupling is engaged only when dynamic activations are of interest.

The evaluation module itself provides the `LMHarness` class as a façade over the EleutherAI harness, handling model/tokenizer resolution, backend selection (either HuggingFace-provided models or in-memory adapters), task validation, and metric extraction. It wraps the harness results into the framework’s standardized evaluation result object and optionally logs to CSV. The adapter class, `LMLMHarnessModelAdapter`, bridges the framework’s `ModelEvalInterface` to the harness expectations, implementing tokenization, detokenization, loglikelihood, rolling loglikelihood, and text generation functions. These capabilities allow activation analysis to benefit from a mature evaluation workload generator without reimplementing dataset handling or scoring logic. By depending on this adapter, the analysis plugin gains uniform access to task-conditioned forward passes across both baseline and compressed models.

Architecturally, the integration embodies inversion of control and optional dependency patterns. The analysis plugin does not embed task logic; instead, it delegates to `LMHarness`, which in turn may rely on the adapter when operating on in-memory models rather than externally hosted checkpoints. Conditional imports and feature flags shield the analysis stack from hard failures when the optional library is absent. Provenance emission and structured reporting ensure that any reliance on LM evaluation is explicit in downstream artifacts, supporting reproducibility and auditing.

Finally, the integration is designed with testability and maintainability in mind. Because the plugin can operate with a minimal forward path, unit and integration tests can exercise activation capture and metric aggregation without installing the LM evaluation dependency. When the dependency is present, end-to-end tests can validate that hooks fire during benchmark execution and that reports annotate the chosen task and evaluation parameters. This dual-mode capability balances robustness with feature richness, allowing deployments to exploit LM evaluation when available while retaining functional analysis in constrained environments.

4.6 Case Study: A Full Compression-and-Evaluation Process for Qwen and MiniCPM Models

This section gives a representative case study - a full compression-and-evaluation pipeline for a MiniCPM4-0.5B, Qwen2.5-0.5B and Qwen3-0.6B models, with the actual pipeline in [Sec. 4.6.1](#) and experimental results in [Sec. 4.6.2](#).

4.6.1 Actual Workflow

In this section, the involved classes in [Fig. 4.2](#) are highlighted in different colours.

It leverages the orchestration layer - [PipelineOrchestrator](#), [WorkflowExecutor](#), and [Workflow](#), and the plugin substrate defined by [Plugin](#), [PluginRegistry](#), together with event handling via [EventBus](#) and [PipelineEvent](#).

The following stepwise description mirrors a conceptual workflow composed of a list of [WorkflowStep](#) instances, each bound to a [Plugin](#) instance, while the [PipelineContext](#) (state, resources, workspace) anchors shared execution metadata.

Step 1: Configuration and Context Initialisation The script loads YAML configuration, extracting model identifiers, compression policy, and evaluation tasks. A [PipelineContext](#) is instantiated, provisioning ① a fresh [PipelineState](#) for cross-step artifacts, ② an [EventBus](#) to propagate lifecycle notifications, and ③ workspace bookkeeping. In a full orchestration, [PipelineOrchestrator.execute_workflow](#) would emit [PipelineEvent.WORKFLOW_STARTED](#), while [WorkflowExecutor](#) would later consume this context for step execution.

Step 2: Tokenizer/Model Loading Compatibility shims are applied (via `import _ALL_PARALLEL_STYLES`), followed by model and tokenizer construction. In a registered workflow, this would correspond to a [WorkflowStep](#) using a loader plugin (e.g., a class derived from [Plugin](#) that wraps [ModelLoader](#)) and emitting [PipelineEvent.MODEL_LOADING_STARTED/LOADED](#). The [PipelineState](#) can cache these objects for downstream steps.

Step 3: Baseline Profiling Random input IDs are generated (`random_ids_inputs`), and [MemoryProfiler](#) measures GPU/CPU memory during a forward pass. This phase could be framed as a “profiling” [WorkflowStep](#), with the plugin publishing `analysis.started` and `analysis.completed` events through [PluginEventManager](#) (constructed inside [Plugin.initialize](#)). Results would be persisted under `state.plugin_results.<name>` by the base [Plugin.execute](#).

Step 4: Evaluation (Original Model) An evaluation plugin `LMHarness` (inheriting the plugin contract) is configured with tasks such as `wikitext` and `arc_easy`, respecting batch size and device constraints. In a workflow, `WorkflowExecutor.execute_step` would instantiate `LMHarness` via `PluginRegistry.create_plugin`, honour step-specific timeouts, and emit `PipelineEvent`.`WORKFLOW_STEP_STARTED/COMPLETED`. The plugin emits `evaluation.started/completed` events, enabling auditability.

Step 5: Compression via ModelConsolidator The compression step uses `ModelConsolidator` (category: compression plugin helper) configured for SVD with a fixed rank/policy. On `initialize`, the consolidator receives `PipelineContext`, thereby gaining access to `UnifiedStrategyFactory` (for strategy resolution), `ModelManager`, and `StateManager`. During `compress_model_with_surgery`, it may emit `PipelineEvent`.`COMPRESSION_STARTED/COMPLETED` and plugin-scoped events `compression.progress`. Successful completion writes compression results into `PipelineState` (e.g., replaced layers, factorisation parameters), enabling subsequent steps to read from `state`.

Step 6: Activation Metrics `ActivationMetricsPlugin` is initialised with the same `PipelineContext`. It installs lightweight evaluation driving (`_drive_evaluation`) to collect activations. Its `execute()` method wraps `do_execute()`, emitting `analysis.started` and `analysis.completed` events, while storing metrics reports (shape histograms, per-layer statistics) into `PipelineState`. The plugin's lifecycle mirrors other plugins: `standardized.progress` (`.progress`), error propagation (`.failed`), and result persistence.

Step 7: (Optional) Serialization The script comments illustrate saving and reloading compressed artifacts via `save_compressed_to_safetensors` and `load_compressed_from_safetensors`. In a workflow, this would be a dedicated `WorkflowStep` whose plugin emits `PipelineEvent`.`COMPRESSION_COMPLETED` upon successful serialization and `PipelineEvent`.`MODEL_LOADED` when reconstructing the compressed model.

Step 8: Compressed Profiling and Evaluation A second profiling pass (`MemoryProfiler`) runs on the compressed model to compare memory footprints. Evaluation repeats using `LMHarness`, now on the compressed model, preserving identical task configuration. Each step continues to emit standardized events and record execution time (`WorkflowStep.execution_time`) via `WorkflowExecutor`.

Step 9: Metrics, Ratios, and Timing Report Final metrics include parameter counts (`params_count`), compression ratio, and JSON-formatted memory traces for baseline and compressed runs. The script aggregates timing measurements; in a fully declared workflow, these would correspond to per-step `WorkflowStep.execution_time` values exposed through `Workflow.get_execution_summary()` and surfaced by `PipelineOrchestrator.get_workflow_status()`.

Event-driven control flow Throughout, the `EventBus` transports notifications, enabling decoupled monitoring. The orchestrator subscribes to `PipelineEvent.WORKFLOW_STEP_COMPLETED/FAILED` for logging, while plugins publish `plugin.<name>.started/progress/completed` via `PluginEventManager`. This event spine ensures that retries, skips, or conditional branches (`WorkflowStep.condition`, evaluated by `WorkflowExecutor.evaluate_condition`) remain observable and auditable.

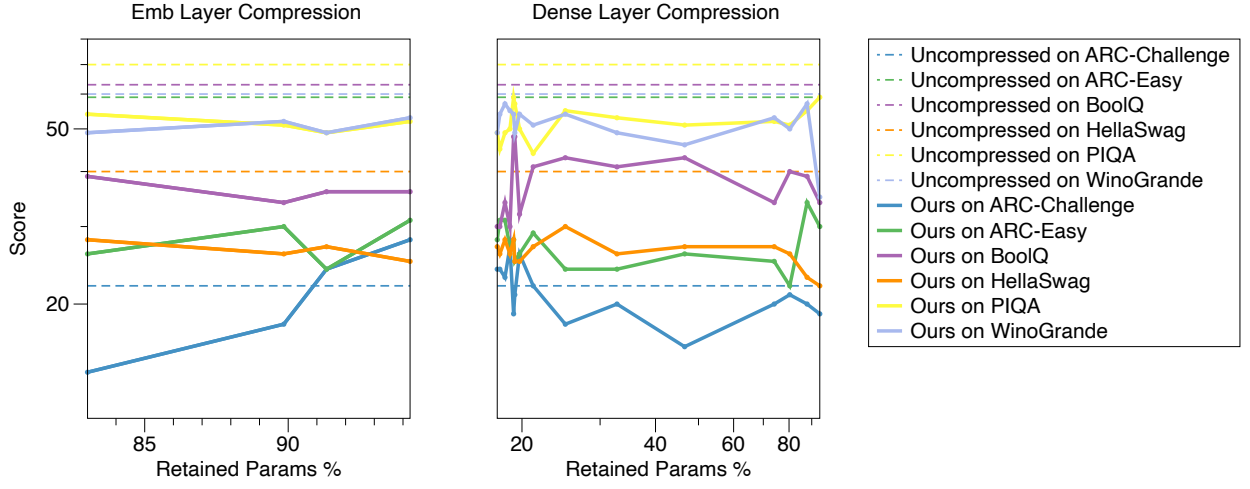
Above all, the given process in this section embodies a reproducible, stepwise pipeline: configuration ingestion, model/materialization, profiling, evaluation, compression, activation analysis, optional serialization, and post-compression evaluation. Each stage is encapsulated as a plugin instance executing within a workflow step, coordinated by `WorkflowExecutor` and overseen by `PipelineOrchestrator`. The `PipelineContext` centralizes state, resources, and events, while `Plugin` abstractions guarantee uniform lifecycle semantics. The result is a modular, event-driven architecture that couples compression research workflows with transparent observability and controlled concurrency.

4.6.2 Quantitative Experimental Results and Discussion

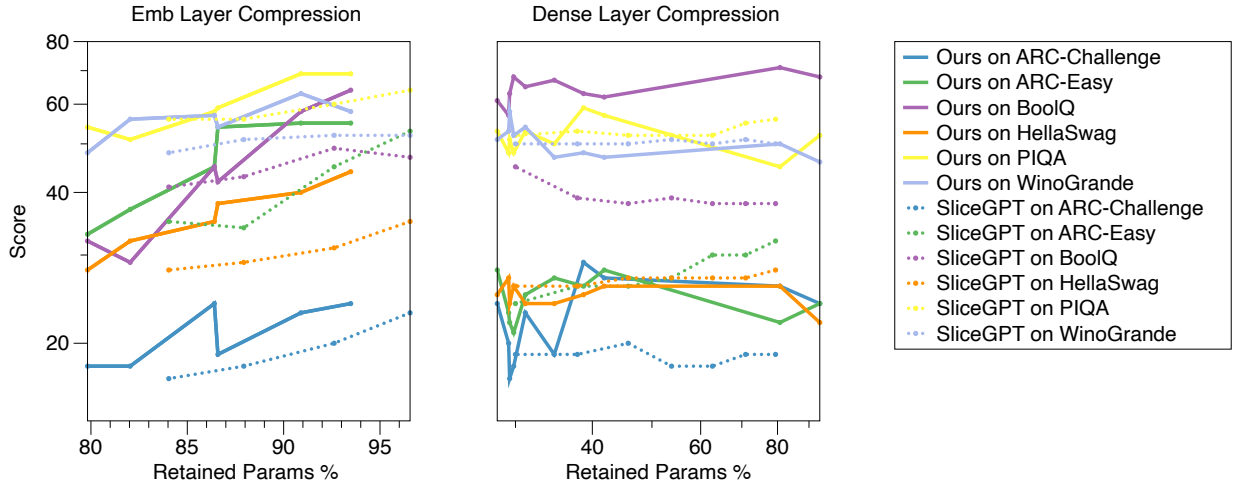
Zero-shot reasoning scores. To understand which linguistic capabilities different compression approaches preserve, the training-free tensor-based method is compared with SliceGPT [113], a representative training-free pruning approach. This comparative analysis reveals that different compression methods preserve complementary capabilities rather than uniformly degrading all performance. Fig. 4.5 reports accuracy on six zero-shot reasoning benchmarks: BoolQ [3], ARC-Easy and ARC-Challenge [2], HellaSwag [1], PIQA [4], and WinoGrande [45]. These benchmarks span inferential, scientific, commonsense, and physical reasoning, as detailed in Tab. 2.1. For each multiple-choice question, accuracy is computed as the proportion of questions answered correctly (Eq. (2.3)), where the model’s prediction is determined by selecting the candidate completion with the highest log-probability, following the LM Evaluation Harness [36] protocol described in Sec. 2.1.

The performance differences among the evaluated models sharpen this picture. MiniCPM4-0.5B’s dense layers, for instance, are **overparameterized for ARC-Challenge** — performance improves only when the model is shrunk to under 20% of its original size, a surprisingly aggressive level of compression. For Qwen2.5-0.5B and Qwen3-0.6B, the tensor-based method competes more closely with SliceGPT: it outperforms on BoolQ and ARC-Challenge, but trails on HellaSwag and WinoGrande, with Qwen3-0.6B peaking around 50% parameter retention. On ARC-Easy and PIQA, the tensor-based method excels at **mid-range compression levels (20–50%)** but falters outside that band. A closer look at task-relevant traits that contribute to these differences between the tensor-based method and SliceGPT leads to the summary of tasks, compression results and interpretation in Tab. 4.3.

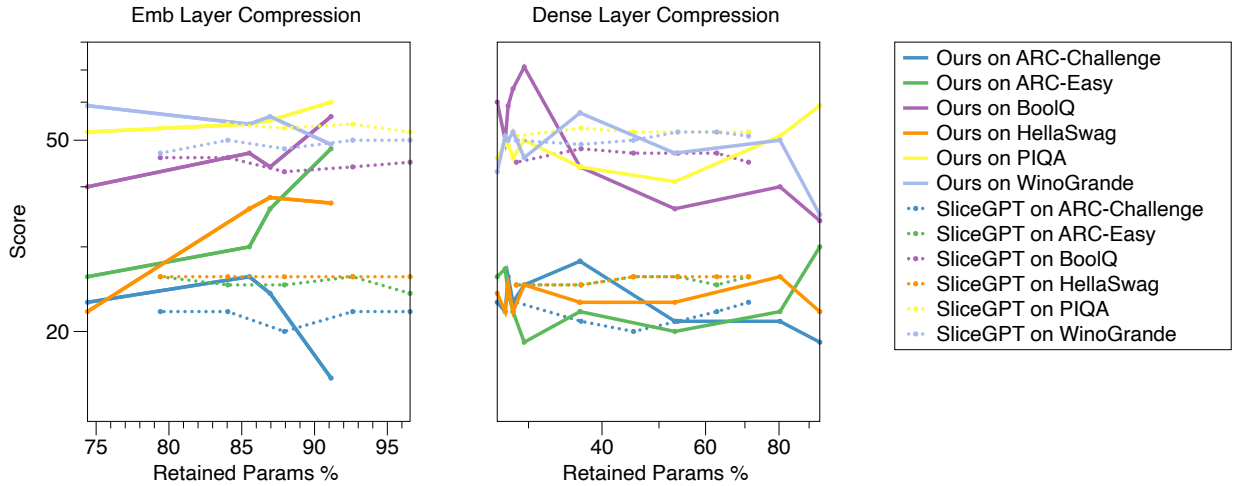
Tab. 4.3 implies that the tensor-based compression is better at maintaining higher-level logical



(a) MiniCPM4-0.5B compressed using the proposed approach on embedding and dense layers, with comparison to the performance of the original models (with dashed lines).



(b) Qwen2.5-0.5B compressed using the proposed approach on embedding and dense layers, with comparison to a pruning approach SliceGPT with different sparsity settings.



(c) Qwen3-0.6B compressed using the proposed approach on embedding and dense layers, with comparison to a pruning approach SliceGPT with different sparsity settings.

Figure 4.5: Model compression with the proposed approach on embedding layers (left side) and dense layers (right side, including feed forward layers and attention layers), for MiniCPM4-0.5B, Qwen2.5-0.5B, Qwen3-0.6B

Table 4.3: Summary of the Comparison of Our Approach and SliceGPT

Task Traits	Tasks	Compared with SliceGPT	Interpretation
Logical Reasoning	BoolQ, ARC-Challenge	Outperforms SliceGPT	Our approach filters out lower-level redundant dense capacity while keeping higher-level reasoning capabilities.
Fact Retrieval & Lexical Co-occurrence	ARC-Easy, PIQA	Better than SliceGPT when 20–50% params are retained, worse than SliceGPT for the rest.	Moderate compression removes redundancy but preserves distributed commonsense priors; too much harms them.
Adversarial Commonsense / Broad Correlation	HellaSwag, WinoGrande	Underperforms SliceGPT	These tasks rely on wide, weak lower-level feature coverage.

reasoning capabilities, but weaker than SliceGPT in retrieval and commonsense tasks, which has a theoretical guarantee in lower-level language modelling tasks [124].

Findings. In the zero-shot reasoning evaluation in Fig. 4.5, **compressing the embedding layers is consistently effective—and sometimes even improves model performance.** At similar parameter ratios, the tensor-based method outperforms the SliceGPT baseline for both Qwen2.5-0.5B and Qwen3-0.6B. MiniCPM4-0.5B even surpasses the original model on ARC-Challenge when more than 90% of parameters are retained. This suggests that embedding layers contain modest redundancy, and that a structured, task-aware compression method can enhance generalisation rather than just reduce size. In contrast, **dense-layer compression behaves much less predictably.** Performance does not simply drop as models are compressed — instead, each task has a “sweet spot,” where moderate compression maintains or boosts accuracy, while going too far in either direction hurts. This highlights that not all layers, or all tasks, carry the same amount of redundancy.

The performance differences among the evaluated models sharpen this picture. MiniCPM4-0.5B’s dense layers, for instance, are **overparameterized for ARC-Challenge** — performance improves only when the model is shrunk to under 20% of its original size, a surprisingly aggressive level of compression. For Qwen2.5-0.5B and Qwen3-0.6B, the tensor-based method competes more closely with SliceGPT: it outperforms on BoolQ and ARC-Challenge, but trails on HellaSwag and WinoGrande, with Qwen3-0.6B peaking around 50% parameter retention. On ARC-Easy and PIQA, the tensor-based method excels at **mid-range compression levels (20–50%)** but falters outside that band. Again, the task-specific traits that drive these differences between the tensor-based method and SliceGPT are summarised in Tab. 4.3.

Why tensor decomposition preserves different capabilities? Tensor decomposition enforces a low-rank factorisation structure that compresses by finding regularities across multiple modes simultaneously. This structured compression tends to preserve:

1. Abstract reasoning patterns that generalise across contexts (e.g. BoolQ, ARC-Challenge);
2. Higher-order feature interactions captured by multi-mode tensors.

In contrast, pruning methods like SliceGPT remove individual connections, which tends to preserve:

1. Distributed lower-level features stored redundantly (e.g. HellaSwag);
2. Statistical co-occurrence patterns robust to sparse removal.

4.7 Chapter Summary

This chapter presents the object oriented implementation that operationalises the proposed algebraic compression approach for practical language model deployment. It first reviews relevant open sourced infrastructures, then establishes concrete requirements for a reusable system, before detailing the factory style software design and validating it through case studies on modern small language models.

Relevant open sourced projects. [Sec. 4.1](#) surveys existing efficient inference stacks and evaluation frameworks that motivate the design of the implementation. Projects such as vLLM and DeepSpeed illustrate production grade engines for scalable inference, while the Language Model Evaluation Harness (`lm_eval`) provides a unified, benchmark rich evaluation backend. The analysis clarifies which capabilities can be reused and which integration gaps the implementation must address.

Requirements. The requirements section [Sec. 4.2](#) translates the high-level goals into concrete software constraints. It specifies support for multiple model families and tokenizers, training-free compression workflows, pluggable metrics backends, and explicit separation between core algebraic logic and deployment-specific plugins. These requirements guide the subsequent architectural decisions and ensure the system remains extendable as new models and evaluators emerge.

Solution design. The solution sections detail the object-oriented architecture centred on factory and registry abstractions ([Sec. 4.3](#)), resource management ([Sec. 4.4](#)) and third-party integration ([Sec. 4.5](#)). Core interfaces describe models, compression pipelines, metrics backends and plugins, while concrete implementations bind to third party libraries such as `transformers`, `vllm` and external metrics packages. The design emphasises late binding, dependency injection and inversion of control, so that new model families or analysis tools can be added with minimal code changes.

Case study. The final section [Sec. 4.6](#) demonstrates the implementation on representative sub-billion models, including MiniCPM4-0.5B, Qwen2.5-0.5B and Qwen3-0.6B. It instantiates the fac-

tory based workflows, runs end to end compression and evaluation pipelines, and reports quantitative results on language modelling and reasoning benchmarks. These case studies confirm that the implementation faithfully realises the theoretical design while remaining practical for real-world deployments.

Chapter 5

Resource-Constrained Deployment and Algebraic LLMs Runtime Analysis

Based on the theoretical discussion about the algebraic nature of transformer-based language models (Chapter 3) and the implementation with software engineering concerns (Chapter 4), this chapter will focus on actual deployment issues and quantitative experimental results. This section first suggests a fine-grained FLOPs profiler LLM-Profiler, then proposes an algorithm TensorSLM.

5.1 LLM-Profiler: a Fine-Grained LLM FLOPs Calculator

FLOPs (floating point operations), which often dominate inference latency, are one of the most important optimisation objectives in model compression. Though on the hardware level the exact FLOPs of matrix multiplication can be very different (e.g., matrix multiplication strategies may shift during the ARM architectural evolution [125]), higher-level FLOPs estimation (i.e., estimation based on Python implementation) is still a valuable guidance for the model compression.

5.1.1 Core Arithmetic Operations in Model Forward Passes

This section introduces the actual calculation of several commonly used PyTorch operations, including ① operator `*`, ② operator `@`, and `torch.matmul()`, in the forward pass LLMs, based on the open-sourced Python and C code of PyTorch¹ and the official Python documentation². For simplicity, the cases like Python broadcasting, which handle dimension mismatches of operands, are

¹<https://github.com/pytorch/pytorch>

²<https://docs.python.org/3/library/operator.html>

not considered in `llm-profiler`.

Python operator `*`. There are typically two cases for the Python operator `*`, which are categorised according to the dimension of the operands.

1. If the two operands are 1-D arrays (vectors), the operation is the dot product, also known as the scalar product. The equivalent PyTorch function is `torch.dot(·)`. Given two vectors $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{a} * \mathbf{b}$ involves $\text{ADD} = n - 1$ and $\text{MULT} = n$.
2. If the two operands are 2-D arrays (matrices), the operation is the Hadamard product, also known as the element-wise product. Given two matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{n \times m}$, $A * B$ involves $\text{ADD} = 0$ and $\text{MULT} = nm$.

PyTorch matrix multiplication operator `@`. Matrix multiplication is also known as Cayley matrix multiplication, and has been comprehensively discussed in [Sec. 3.2.7](#). Given two matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, $A @ B$ (or `torch.mm(A, B)`) involves $\text{ADD} = np(m - 1)$ and $\text{MULT} = nmp$.

`torch.matmul(·)`. Used as matrix product of two tensors³, `torch.matmul(·)` is more complex than the other two arithmetic operations in this section. Given two operands $\mathcal{A}$ and $\mathcal{B}$, there are five different cases as follows.

1. $\mathcal{A} \in \mathbb{R}^d$ and $\mathcal{B} \in \mathbb{R}^d$, the operation is equivalent to `torch.dot(·)`, used for dot product. Thus $\text{ADD} = d - 1$, and $\text{MULT} = d$.
2. $\mathcal{A} \in \mathbb{R}^{n \times m}$ and $\mathcal{B} \in \mathbb{R}^{m \times p}$, the operation is equivalent to `torch.mm(·)`, used for Cayley matrix multiplication, thus $\text{ADD} = np(m - 1)$ and $\text{MULT} = nmp$.
3. $\mathcal{A} \in \mathbb{R}^{n \times m}$ and $\mathcal{B} \in \mathbb{R}^m$, the operation is equivalent to `torch.mv(·)`, used for matrix-vector product, thus $\text{ADD} = n(m - 1)$ and $\text{MULT} = nm$.
4. $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_N}$, $\mathcal{B} \in \mathbb{R}^{d_N}$, there are three steps for such operation:

- (a) reshape $\mathcal{A}$ into $\mathcal{A}' \in \mathbb{R}^{(\prod_{k=1}^{N-1} d_k) \times d_N}$;
- (b) get matrix-vector product result $\mathcal{C}' = \text{torch.mv}(\mathcal{A}', \mathcal{B})$;
- (c) get the final result via reshaping $\mathcal{C}'$ into $\mathcal{C} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_{N-1}}$;

where the `torch.mv(·)` decides $\text{ADD} = (\prod_{k=1}^{N-1} d_k)(d_N - 1)$, and $\text{MULT} = \prod_{k=1}^N d_k$.

5. $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_N}$, $\mathcal{B} \in \mathbb{R}^{d_N \times d_{N+1}}$, there are three steps for such an operation:

- (a) reshape $\mathcal{A}$ into $\mathcal{A}' \in \mathbb{R}^{(\prod_{k=1}^{N-1} d_k) \times d_N}$;

³<https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/LinearAlgebra.cpp>

- (b) get matrix multiplication result $\mathcal{C}' = \text{torch.mm}(\mathcal{A}', \mathcal{B})$;
- (c) get the final result via reshaping $\mathcal{C}'$ into $\mathcal{C} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_{N-1} \times d_{N+1}}$;

where the $\text{torch.mm}(\mathcal{A}', \mathcal{B})$ decides $\text{ADD} = (\prod_{k=1}^{N-1} d_k)(d_N - 1)d_{N+1}$, and $\text{MULT} = \prod_{k=1}^{N+1} d_k$.

These three arithmetic operations are the core of calculating FLOPs. The source code of the model’s forward passes is analysed based on the appearances of these three operations, and ADD and MULT are accumulated to obtain the final FLOPs.

5.1.2 Measurement for Llama3 Models

FLOPs and other statistics of Llama-3.x-{1B,3B,8B,70B}-Instruct⁴ measured by `llm-profiler` are shown in Fig. 5.1. The profiling results reveal several scaling patterns across the Llama3 model family. For the 1B model, addition and multiplication operations total approximately 117.5 billion FLOPs, with 1.24 billion parameters requiring 59.1 million cache entries and 327.2 million activation values. Scaling to 3B parameters increases computational load to 315.6 billion FLOPs, with proportional growth in cache to 193.5 million and activations to 848 million. The 8B model demonstrates near linear scaling with 567 billion FLOPs, while the 70B variant exhibits substantial computational demands of 3.91 trillion FLOPs, 69.5 billion parameters, 551 million cache entries, and 6.43 billion activations. These measurements provide quantitative baselines for compression algorithm and deployment cost estimation across different hardware platforms.

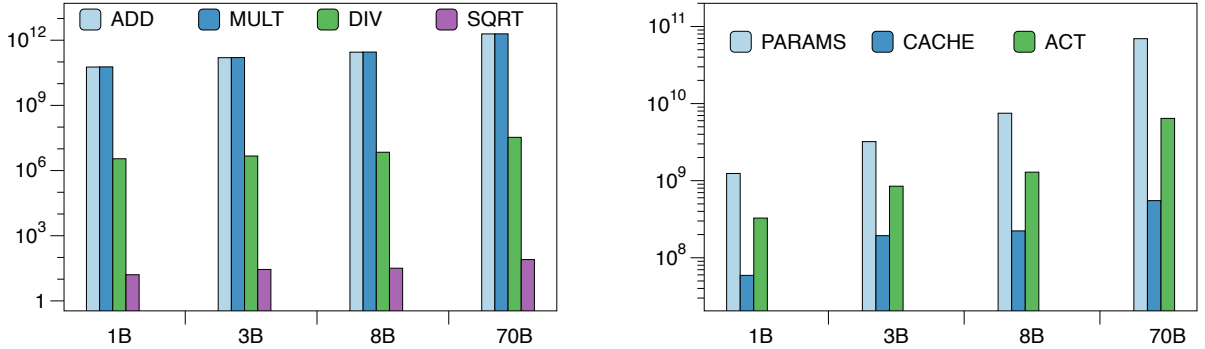


Figure 5.1: FLOPs and other statistics of Llama-3.x-{1B,3B,8B,70B}-Instruct forward passes measured by `llm-profiler`. The input text length is 200, and the batch size is 16.

5.1.3 Comparative Analysis of `llm-profiler` and Other Profilers

The rapid advancement of Large Language Models (LLMs) necessitates precise tools to analyse their computational behaviours and resource requirements. As these models become integral to diverse

⁴More precisely, these models are meta-llama/Llama-3.2-1B-Instruct, meta-llama/Llama-3.2-3B-Instruct, meta-llama/Llama-3.1-8B-Instruct and meta-llama/Llama-3.3-70B-Instruct

applications, distinguishing between the capabilities of various analysis instruments is crucial for effective optimisation. The profiling of LLMs is a multifaceted task, wherein various tools offer insights into distinct aspects of model performance and resource consumption, as illustrated in Tab. 5.1. `llm-profiler` is distinguished by its provision of a fine-grained, model-centric analysis, contrasting with other more general-purpose or hardware-centric profilers. This section clarifies these distinctions, categorising them by the level of abstraction, granularity, and the primary focus of the profiling tool.

Table 5.1: Differences between `llm-profiler` and other profilers.

Aspect	<code>llm-profiler</code>	DeepSpeed FLOPs Profiler ⁵	PyTorch Profiler ⁶ or <code>torch.autograd.profiler</code> ⁷	Holistic Trace Analysis ⁸
Abstraction	Algorithmic, model level	Model implementation level	System and framework level	End to end system level
Granularity	Layer and operation type	Per layer FLOPs, aggregated ops	Operator and kernel timelines, model level	Timeline and module level, no math counts
Main metrics	ADD, MULT, DIV, SQRT, PARAMS, ACT, CACHE	Total FLOPs and MACs, no separation of ADD, and MULT, no cache tracking	Wall time, CUDA time, memory, coarse FLOPs on specific hardware	Latency breakdown, overlap, throughput and utilisation for LLM pipelines
Typical use	Compare architectures and compression, hardware agnostic analysis	Estimate training and inference costs for parallel LLM workloads	Debug runtime bottlenecks and optimise kernels on concrete systems	Visualize and tune system-wide performance for LLM deployments

Level of abstraction (algorithmic vs. system-level profiling). A primary distinction of `llm-profiler` lies in its focus on the *algorithmic* level of the model, rather than the system or hardware level.

- **`llm-profiler`:** Operating at a high level of abstraction, this tool analyses the theoretical computational cost of the LLM architecture. It computes the number of algorithmic operations (e.g., additions, multiplications), parameter counts, and memory requirements for cache and activations. This hardware-agnostic approach provides a pure measure of the model’s intrinsic complexity.
- ***System-Level Profilers*:** These general-purpose profilers monitor program execution on specific hardware configurations. They trace function calls, measure CPU cycles, and identify code bottlenecks. Although effective for optimising model implementation, their output depends heavily on the underlying hardware and software stack. The insights provided pertain to the *execution* of the model rather than its inherent computational structure.
- ***Hardware-Specific Profilers*:** Offering the lowest level of abstraction, these tools provide deep insights into the utilisation of specific hardware components, such as GPU kernels, memory band-

⁵https://github.com/deepspeedai/DeepSpeed/tree/master/deepspeed/profiling/FLOPs_profiler

⁶<https://github.com/pytorch/kineto>

⁷<https://github.com/pytorch/pytorch/blob/v2.6.0/torch/autograd/profiler.py>

⁸<https://github.com/facebookresearch/HolisticTraceAnalysis>

width, and tensor core usage. While indispensable for performance engineering on a particular hardware target, the findings lack portability and do not directly translate to the algorithmic complexity of the model itself.

Granularity of analysis (fine-grained vs. coarse-grained). The granularity of analysis constitutes another critical point of comparison.

- **llm-profiler:** Facilitates fine-grained analysis by decomposing computational costs into individual layers or specific operations within the model. This granularity enables a detailed comprehension of which LLM components (e.g., attention mechanism, feed-forward network) are the most computationally intensive.
- *Coarse-Grained Profilers:* These tools yield a high-level overview of resource consumption. For instance, the `time` command measures the total execution time of a script, while a basic memory profiler might report peak memory usage. Although useful for rapid assessment, they lack the detail required to pinpoint specific areas for optimisation within the model architecture.

Primary focus (model-centric vs. performance-centric). The primary objective of the profiling tool also shapes its functionality.

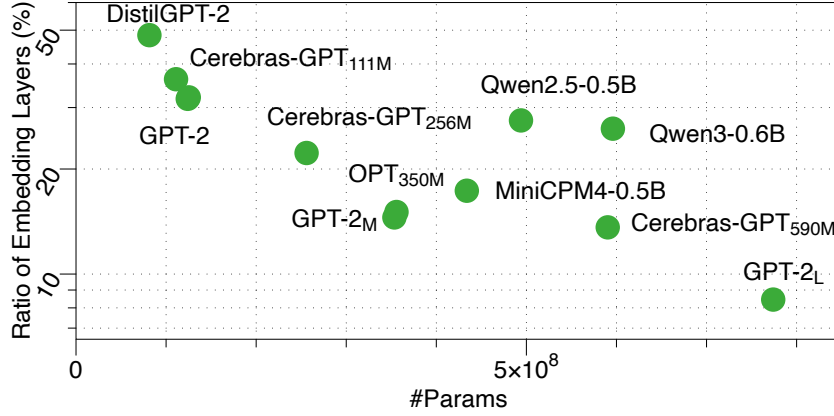
- **llm-profiler:** Fundamentally *model-centric*, its primary purpose is to analyse and comprehend the LLM architecture. This characteristic renders it an ideal instrument for researchers and model designers seeking to compare the theoretical efficiency of different model designs or to predict model performance on novel hardware platforms.
- *Performance-Centric Profilers:* Designed to optimise model implementation, these tools provide detailed timelines of operations, identify GPU idle time, and suggest improvements for data loading and kernel execution. Emphasis is placed on the *runtime performance* of the model rather than its theoretical complexity.

Scope (LLM-specific vs. general-purpose). Finally, the intended scope constitutes a major differentiator.

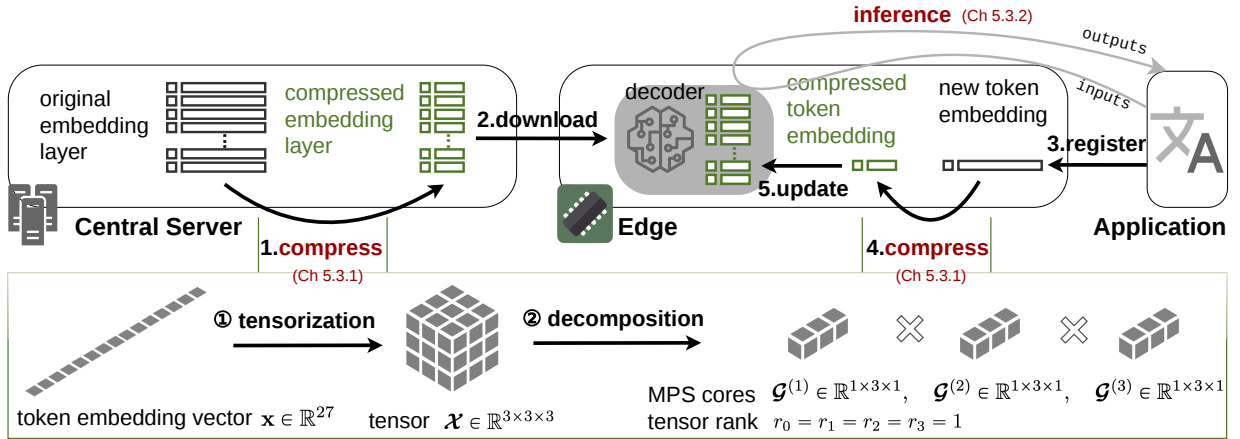
- **llm-profiler:** As a specialised tool designed for LLMs, it incorporates knowledge of the unique architectural components of transformers (e.g., self-attention, layer normalisation) and provides analysis tailored to these models.
- *General-Purpose Profilers:* While applicable to diverse program types, general-purpose profilers lack the domain-specific knowledge required to provide LLM-centric insights. For instance, distinguishing between the computational cost of the attention mechanism and the feed-forward network is often infeasible without significant manual interpretation of the results.

By focusing on the intrinsic, hardware-agnostic properties of LLMs, **llm-profiler** occupies a unique and critical niche in the landscape of profiling tools, serving as a bridge between theoretical model design and practical performance engineering.

5.2 Facing Resource-Constrained On-Device Compression



(a) Embedding layer parameter count ratio for the sub-billion language models involved in this chapter, which typically take one-fourth to one-third of the total model parameters.



(b) The workflow of SLM compression in an edge computing scenario with the proposed approach. Details about this pipeline are clarified in [Sec. 5.2.1](#).

Figure 5.2: Typical SLM layer composition and the SLM application requirement of adaptability.

Memory and storage efficiency currently prohibit unlocking the full potential of lightweight applications of generative language models for low-end devices. A straightforward approach to solving memory and storage problems in a unified manner is deep learning model compression, which reduces the parameters of a deep learning model.

Typical language models for low-end devices, which are called Small Language Models (SLMs),

can have less than one billion parameters, yet they are still a burden for low-end devices. In the composition of the language model parameters, the embedding layers account for a significant portion, especially for sub-billion language models. For the current typical sub-billion language models, the smaller the language models, the higher the embedding layer portion, which can be up to 48.08%, as shown in Fig. 5.2a.

Low-rank factorisation, which means breaking matrices or higher-dimensional tensors into smaller units, has recently been recognised as a promising solution for language model efficiency, since its successful application in parameter-efficient fine-tuning. There are already efforts aiming to utilise such a technique in language model compression; some are specialised for embedding layers while others are not. However, most of these require an extra training process (as summarised in Tab. 5.3), such as fine-tuning, meta-learning and training from scratch. In the real practice, training-free is a crucial requirement in model compression, with the reasons summarised in the following Tab. 5.2. In short, the reasons for training-free include ① extra training involves additional computation and training data, which may be unavailable for low-end devices; ② training the language model from scratch discards the valuable knowledge stored in the weights of the original models. The actual instances can be multiple models deployments (e.g. 5 models instead of 3 on 8GB devices), bandwidth-constrained distribution (e.g. saving 20 more minutes per 150MB reduction over 1 Mbps networks), and privacy-critical deployments (e.g., local data contains sensitive personal information).

Given these deployment contexts and benefits of training-free, a critical question arises:

Without extra training, how can the compressed models retain their language task performance?

To answer this question, this section proposes an approach that induces a high-dimensional tensor structure to capture the information stored in the embedding parameters, named as TensorSLM. The *Tensor-Train Decomposition (TTD)* [111], which excels in factorising high-order tensors, is used to represent embeddings in a lower-rank Matrix Product State (MPS) format. Due to the look-up table nature of the embedding layer, each token embedding is treated individually (i.e. each row of the token embedding matrix) rather than taking the token embedding matrix as a whole. This design prevents damaging the individual information of each token, and even applies to use cases with ever-changing vocabulary. A comprehensive discussion of the proposed approach is provided, including general perspectives such as compression ratio and language task performance, as well as technical perspectives such as the impact of different hyperparameter choices in TensorSLM and the effects of the original model size.

Taking GPT family, OPT models and MiniCPM4-0.5B as case studies, the contributions of this work are as follows:

1. This work comprehensively investigates which linguistic capabilities different compression meth-

Table 5.2: Training-free is a crucial requirement in model compression.

Why “training-free”?	Evidences	Deployment Scenarios
Massive computation costs for resource-constraints scenarios.	Training memory costs are typically $2\times$ to $4\times$ those of inference costs [126, Figure 1], which is usually unaffordable for lower-end devices like Raspberry Pi.	Emergency deployment: rapidly compress and deploy a newly released domain-specific language model to edge devices in the field within hours, with no access to training infrastructure.
Benefits for industrial developments.	In production environments, training-free methods: (1) eliminate the need for expertise in training dynamics, thus being more accessible to practitioners; and (2) integrate better into existing machine learning deployment pipelines [127].	Dynamic model switching: swap between multiple task-specific models on the same hardware, compressing each on-device as needed.
Data privacy and availability issues	Well-prepared training data are not always available, as low-cost data labelling can be challenging [128]. Additionally, collecting training data can result in privacy violations [129].	Privacy-critical on-premise: compress proprietary or sensitive models on-premises without sending data to cloud training services.

ods preserve. Tensor-Train Decomposition excels at preserving higher-level reasoning (BoolQ, ARC-Challenge), while pruning methods maintain lower-level lexical features (HellaSwag, WinoGrande), revealing complementary capability preservation.

2. As far as we know, TensorSLM is the first to provide training-free low-rank compression for **low-end device**. Tensor-Train Decomposition is adapted for token-by-token operations where block-wise approaches [52, 130] are incompetent, trading latency (no more than 0.5 ms/token overhead on Raspberry Pi) for storage and memory reduction.
3. Both simple language tasks (language modelling and sentiment classification) and complex language tasks (zero-shot reasoning) are evaluated, with comparison to matrix-based SVD and a representative pruning approach SliceGPT [113]. Embedding layers compress consistently ($0.5\times$ - $2.0\times$), while dense layers exhibit task-specific “sweet spots” (20-50% retention optimal for fact retrieval).
4. A set of ablation studies is provided on on-board inference latency, compression on different layer types (embedding and dense layers), and the extra finetuning efforts required for recovery of language task performance. Latency of various compressed cases is measured on the low-end Raspberry Pi 5⁹, and a detailed systematic analysis is given for running TensorSLM on low-end

⁹<https://www.raspberrypi.com/products/raspberry-pi-5/>

edge devices like Raspberry Pi and higher-end servers.

5.2.1 Adaptability for SLM Applications

Unlike the current LLM applications, which are mostly running on high-end GPU servers (e.g. in the data centres with numerous NVIDIA A100), SLMs are mainly for edge (or mobile) applications that require adapting to the environment with limited resources on lower-end devices. A common approach to adapting to the dynamic environment is updating the vocabulary according to the changes in input text distribution [47]. The reasons for this distribution change vary from case to case. For example, new user registration, or the frequently used tokens update with the users' changing daily lives. There are some existing works summarised in Tab. 5.3. It can be observed that none of them addresses the application scenario as just discussed.

Table 5.3: Comparison with our approach and the relevant research.

Works	Device		Training?	Structure		Layer		Focused Size	
	high-end	low-end		matrix	tensor	Emb	Linear	large	small
[47]	✓		✓	✓		✓			✓
[131]	✓				✓	✓			✓
[132]	✓		✓	✓		✓			✓
[133]		✓	✓	✓		✓			✓
[134]		-	✓	✓				-	
[135]	✓		✓	✓		✓			✓
[123]	✓			✓			✓	✓	
[136]	✓		✓	✓			✓		✓
[104]	✓		✓		✓		✓	✓	
[137]	✓			✓			✓	✓	
[52]	✓		✓	✓			✓		
[130]	✓		✓	✓	✓		✓		
[138]	✓		✓		-				✓
Ours		✓			✓	✓		✓	

To cope with the ever-changing input tokens and vocabulary, a straightforward strategy is to build a cloud-edge system, as shown in Fig. 5.2b, which is similar to the workflows in the field of edge computing, e.g. [139, Fig.1]. There are two kinds of devices in this workflow: 1) the central server, which is possibly a server in public or private cloud services, or a higher-end personal computer, and 2) the low-end edge device. The focus here is on a typical edge device – Raspberry Pi. Over a fairly long period (e.g. months or years), the central server only communicates with the edge device once to provide a brand-new pre-trained language model. Afterwards, the edge device should update the vocabulary on board according to the changes in the environment.

A detailed explanation of Fig. 5.2b is as follows:

Step 1. *The central server compresses the whole token embedding matrices on the token embedding level, according to Sec. 5.3.1.*

Step 2. The compressed vocabulary and other parts of the language model (e.g. the decoder) are downloaded and then deployed on a low-end device.

Step 3. During the application runs, the vocabulary updates for two cases:

1. a new token is required according to the actual application requirements, it will be registered by the service on the edge device. Jump to **Step 4**.
2. an old token is required to be removed (e.g. it has not been used for a long time), the edge device simply deletes the corresponding token embedding vector. Meanwhile, the application deregisters this token.

Step 4. The low-end device compresses the added token embedding vector as described in [Sec. 5.3.1](#).

Step 5. The current vocabulary of the language model. The compression process of a single token embedding follows a pipeline of ① tensorisation and ② decomposition.

5.2.2 Energy Efficiency for SLM Applications

From the workload of the high-end GPU servers (e.g. those equipped with NVIDIA A100) and low-end edge devices (e.g. Raspberry Pi 5) described in [Sec. 5.2.1](#), it is clear that the edge device only takes charge of light-weight essential tasks, since it has strict limitations in computation, memory and communication. Furthermore, since battery life directly impacts the user experience, energy consumption is also a significant concern.

Table 5.4: Approximate energy consumption of different operations (1nJ=1000pJ). For servers, communication with the wired network (e.g. ethernet or optical fibre) is preferred; for edge devices, it is preferred to use wireless networks (e.g. Wi-Fi or cellular network).

Energy Consumption		Raspberry Pi 5 (Cortex-A76 CPU)	GPU server (A100 GPU)
Computation (pJ/float32)	Add	1.0-2.5	5-12
	Mult	1.2-3	6-15
Memory (pJ/float32)		70-260	100-450
Communication (nJ/float32)	Wired	50-350	
	Wireless	400-6000	

The actual energy consumption of a device depends on various factors, like the semiconductor temperature, system workload, operating environment, etc. Thus, it is hard to precisely calculate the exact energy consumption of an algorithm on a certain hardware. However, it is still possible to estimate the range of energy consumption in the system as [Tab. 5.4](#), which leads to the following remarks:

Remark 5.1. Memory operations are more “expensive” than computation in terms of energy.

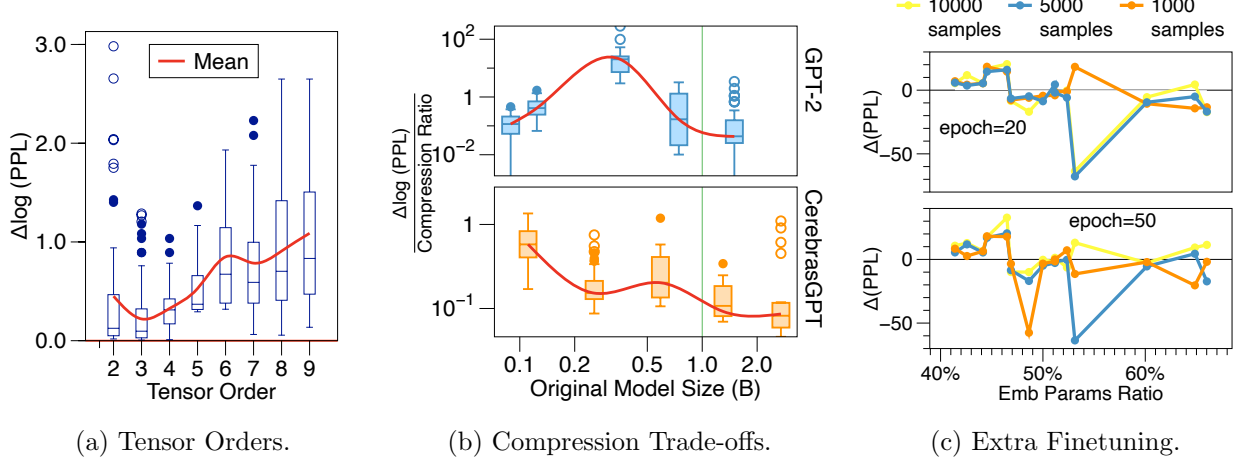


Figure 5.3: An overview of empirical results of the proposed approach. Significant proportion of the sub-billion LLMs; in around half of the common sub-billion LLMs, their embedding layer parameters take one-third of the total model parameters. (a) Language task performance (perplexity, the lower the better) for DistilGPT2 when the compression ratio is within $3.0\times$. From the empirical evaluation on 1BW [6], when the tensor order increases, the language task performance tends to improve first, then decrease after order 3. (b) The compression trade-off of TensorSLM on different sizes models in GPT-2 and CerebrasGPT series models. This trade-off is roughly measured by the ratio between perplexity and compression ratio of embedding layers; the lower the ratio value, the better the compression. It was found that the larger the model size, the better the trade-off, where CerebrasGPT has a smoother trend compared with GPT-2. (c) Finetuning for MiniCPM4-0.5B performance recovery after embedding layers are compressed. With the finetuning sample number increases, the compressed model performance increases first, then declines.

Remark 5.2. *Non-essential communication should be avoided for energy concerns.*

The workflow in Fig. 5.2b already satisfies Remark 5.2. For Remark 5.1, if real-time is *not* the most important concern in the edge application, memory can be “exchanged” for computation to obtain longer battery life. Further discussion and evaluation around these aspects are in Sec. 5.3.1 and Sec. 5.4.

5.3 Methodology

This section clarifies the technical cornerstones of the proposed approach. A practical pipeline of this approach is depicted in Fig. 5.2b. The whole vocabulary is processed on higher-end servers, while inference and vocabulary updates happen on lower-end edge devices.

5.3.1 Individual Embedding Vector Compression

For the compression of the embedding matrix, rather than decomposing the whole embedding weight matrix, each embedding vector is decomposed separately. The lower half of Fig. 5.2b is a simplified illustration of such a process, with a detailed description in Sec. 5.3.1.

Tensorisation. Each token embedding $\mathbf{x} \in \mathbb{R}^d$ is reshaped (or folded, tensorised) into an order- N tensor. Denote $\text{reshape}(\cdot)$ as the reshape function, $\mathcal{X} = \text{reshape}(\mathbf{x}, \{I_1, I_2, \dots, I_N\})$ and $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ such that $d = \prod_{k=1}^N I_k$. As an example, a token embedding vector $\mathbf{x}$ is a 27-dimensional vector, $d = 27$. In this way, vector $\mathbf{x}$ is reshaped into an order-3 ($N = 3$) tensor $\mathcal{X}$, with tensor size for each mode $I_1 = I_2 = I_3 = 3$.

Tensor decomposition. Tensor $\mathcal{X}$ is then decomposed and stored in a Matrix Product State (MPS) form as $\mathcal{X} \approx \mathcal{G}^{(1)} \times_3^1 \dots \times_3^1 \mathcal{G}^{(N)}$, with hyperparameters as TT ranks $r_0, r_1, \dots, r_N$. Instead of storing the entire token embedding vector $\mathbf{x} \in \mathbb{R}^d$, the corresponding MPS cores, $\mathcal{G}^{(k)} \in \mathbb{R}^{r_{k-1} \times I_k \times r_k}$, for $k = 1, \dots, N$, are stored. parameter count of the MPS cores $\{\mathcal{G}^{(k)}\}$ is $\sum_{k=1}^N |\mathcal{G}^{(k)}| = \sum_{k=1}^N r_{k-1} I_k r_k$, where $|\cdot|$ represents the parameter count.

Given the vocabulary size V , the original parameters of the embedding layers are compressed from Vd to $V \sum_{k=1}^N r_{k-1} I_k r_k$, and the compression ratio can be obtained via $\eta_{\text{TD}} = \frac{d}{\sum_{k=1}^N r_{k-1} I_k r_k} - 1$. The computation and memory complexities for all the above processes are summarised in Tab. 5.5.

Table 5.5: Computation and memory complexity during compression and inference. The key tradeoff: storage complexity reduces from $\mathcal{O}(Vd)$ to $\mathcal{O}(VNlr^3)$ (where $V \gg l$ for vocabulary size vs. input length), while inference reconstruction incurs $\mathcal{O}(Nlr^2)$ overhead per token. This tradeoff favors scenarios where the memory and storage are the bottleneck.

Memory	Original	Compressed	Encoded Texts	Input to $\mathcal{M}_{\text{trans}}$
	$\mathcal{O}(Vd)$	$\mathcal{O}(VNlr^3)$	$\mathcal{O}(lNlr^2)$	$\mathcal{O}(ld)$
Computation	TT-SVD		Reconstruction	
	$\mathcal{O}(Nlr^3)$		$\mathcal{O}(Nlr^2)$	

5.3.2 Language Model Inference Process with the Compressed Embeddings

The original inference process with embedding vectors is as follows: when the encoded texts (separated as tokens) are forwarded to the embedding layer, the embedding layer outputs the embedding vectors according to the input tokens; the embedding layer here acts like a look-up table. The embedding vectors are then forwarded to the hidden layers of the transformer, whose size is the same as the dimension of the embedding vectors. Thus, if there is no internal change in the hidden layers, the dimension of the embedding vectors should compile with the dimension of the hidden

layers. So, the compressed embeddings should be reconstructed to the original dimension to enable the forwarding process. This inference happens at the application phase shown in the upper right part of Fig. 5.2b.

Thus just before forwarding embedding vectors to the hidden layers, the memory usage increases from $l \sum_{k=1}^N r_{k-1} I_k r_k$ to ld . However, given that the vocabulary size V is normally much larger than the input token number l , that means $V \gg l$. Thus the proposed approach can still significantly reduce the memory usage if the embedding layer takes a significant part of the whole model parameters. The reconstruction process follows the tensor contraction in Eq. (3.17), turning the TT cores $\{\mathcal{G}^{(k)}\}$ into a N -order tensor $\mathcal{X}$ according to Eq. (3.18), and then vectorizing $\mathcal{X}$ into a full-size embedding vector according to Sec. 3.2.8.

Proposition 5.1. *For an order- N tensor whose dimension for each order is I , its TT-format yields the highest compression ratio when $I = 2$ and TT rank $r = 1$.*

Proof. Assume the tensor size $[I_1, \dots, I_N]$ for the tensor $\mathcal{X}$ to achieve the highest compression ratio, next the proof of this hyperparameter selection is given.

The compression ratio in Sec. 5.3.1 can be represented as

$$\eta = \frac{V \times d}{\sum_{j=1}^V \sum_{n=1}^N (r_{n-1} \times I_n \times r_n)_j} \quad (5.1)$$

$$= \frac{V \times d}{I_1 r_1 + r_1 I_2 r_2 + \dots + r_{N-2} I_{N-1} r_{N-1} + r_{N-1} I_N} \quad (5.2)$$

$$= \frac{V \times d}{\sum_{k=1}^{\lfloor \frac{N+1}{2} \rfloor} r_{2k-1} (r_{2k-2} I_{2k-1} + I_{2k} r_{2k+1})} \quad (5.3)$$

For the simplest case, assume $I_1 = \dots = I_N = I$ and $r_1 = \dots = r_N = r$. Given $d = \prod_{n=1}^N I_n = I^N$, we have $N = \log_I d$, and

$$\eta = \frac{V \times d}{rI [2 + (N-2)r]} = \frac{V \times d}{rI [2 + (\log_I d - 2)]}. \quad (5.4)$$

In Equation Eq. (5.4), the numerator is a constant, and in the denominator, R is a hyperparameter for the Tensor-Train Decomposition. Thus the objective function for the highest compression ratio η is

$$\min_{I, N} rI [2 + (N-2)] \quad \text{s.t.} \quad N = \log_I d \quad (5.5)$$

$$I, N, r \in \mathbb{Z}^+ \quad (5.6)$$

$$2 \leq I \leq N \leq \lfloor \log_2 d \rfloor \quad (5.7)$$

Regarding Eq. (5.5), if we eliminate N then we have a function $h = rI [2 + (\log_I d - 2)]$. Regarding

d in Eq. (5.7), the largest token embedding size of recent GPT-3 [140] is 12,888. Thus, for the GPT series models no later than GPT-3, Eq. (5.5) should be $2 \leq I \leq N \leq 13$. In this range, h is a monotonically increasing function, where the minimum h occurs at $I = 2$.

Therefore, for the simplest case, we have the best hyperparameter selection of $I_1 = I_2 = \dots = I_N = 2$, and $N = \lfloor \log_2 d \rfloor$.

□

5.3.3 Dense Layer Compression

For the dense layers (feed-forward layers and attention layers), this thesis adopts the view in [130], taking weight matrices as two-order tensors, and directly using Tensor-Train Decomposition to compress them.

5.4 Empirical Evaluation for Language Task Performance

This section presents empirical results on compression, language task performance, and latency across representative sub-billion models on both high-end servers and low-end edge devices.

5.4.1 Experimental Setup

Models, tasks and dataset. The sub-billion models considered are DistilGPT2 [68], GPT2, GPT2-M/L [33], CerebrasGPT-111M/256M/590M [141], OPT-125M/350M, Qwen2.5-0.5B [10], Qwen3-0.6B [142] and MiniCPM4-0.5B [143] .

Regarding the language tasks, two different levels of language tasks are considered:

- **Simple Tasks:** language modelling and sentiment classification. For language modelling, the considered datasets are WikiText2, WikiText103 [44] and 1BW [6]. For sentiment classification, the considered dataset is IMDB [35].
- **Complex Tasks:** zero-shot common sense reasoning tasks. The common sense reasoning datasets include ARC-Easy and ARC-Challenge [2], BoolQ [3], HellaSwag [144], PIQA [4] and Wino-Grande [45].

Hardware. Our main experiments were completed on a GPU workstation with an RTX A6000 48GB GPU and AMD Ryzen Threadripper PRO 5955WX CPU. The GPU resource was mainly used to fine-tune language modelling models for sequence classification, which is the requirement of

the sentiment classification task. The inference latency of the low-end devices was measured on a Raspberry Pi 5, with a 64-bit Arm Cortex-A76 CPU and 8GB SDRAM.

5.4.2 Evaluation Metrics

Compression ratio. Denote $\mathcal{M}$ as a model block set containing a list of model modules like embedding layers and attention layers. With $\mathcal{M}_0$ as the original model block set, $\mathcal{M}_{\text{cmpr}}$ as the compressed version of $\mathcal{M}_0$, and $|\mathcal{M}|$ as the parameter count of $\mathcal{M}$. The compression ratio η is defined as

$$\eta = \frac{|\mathcal{M}_0| - |\mathcal{M}_{\text{cmpr}}|}{|\mathcal{M}_{\text{cmpr}}|}. \quad (5.8)$$

Specifically, the embedding compression rate is $\eta_{\text{emb}} = \frac{|\mathcal{T}_0| - |\mathcal{T}_{\text{cmpr}}|}{|\mathcal{T}_{\text{cmpr}}|}$, where $\mathcal{T}$ only contains the token embedding layer and the position embedding layer. In addition, when comparing the compression of different layer types, a unified proportion is used of the compressed model parameter counts in the original model parameter counts $\frac{|\mathcal{M}_{\text{cmpr}}|}{|\mathcal{M}_0|}$.

Perplexity and logarithmic perplexity for language modelling. Perplexity, as defined in Eq. (2.1), is used as a performance evaluation metric of the language modelling task. When explicitly associating perplexity with a specific model $\mathcal{M}$, the notation becomes

$$\text{PPL}(S, \mathcal{M}) = \left(\prod_{i=1}^{|S|} p_{\mathcal{M}}(x_i \mid x_1, x_2, \dots, x_{i-1}) \right)^{-1/|S|}, \quad (5.9)$$

where S is an ordered set (token sequence), consisting of a set of tokens $\{x_t\}$, $t = 1, 2, \dots, |S|$, and $\mathcal{M}$ is the model block that contains all the modules of the language model under evaluation.

Notice that the compression ratio (5.8) has a linear form, while perplexity (5.9) has an exponential form, so it is hard to combine them as a description of a model compression result, since when compression ratio η linearly increases, the perplexity PPL explodes exponentially.

To this end, the following logarithmic form is used to describe the language modelling performance

$$\ln \text{PPL}(S, \mathcal{M}) = -\frac{1}{|S|} \sum_{i=1}^{|S|} \ln p_{\mathcal{M}}(x_i \mid x_1, x_2, \dots, x_{i-1}) \quad (5.10)$$

Now, the language modelling performance change before and after compression is given by

$$\Delta \ln \text{PPL}(S, \mathcal{M}) = \ln \text{PPL}(S, \mathcal{M}_{\text{cmpr}}) - \ln \text{PPL}(S, \mathcal{M}_0) \quad (5.11)$$

which can be expressed as

$$\Delta \ln \text{PPL}(S, \mathcal{M}) = \frac{1}{|S|} \sum_{i=1}^{|S|} \ln \frac{p_{\mathcal{M}_0}(x_i \mid x_1, x_2, \dots, x_{i-1})}{p_{\mathcal{M}_{\text{cpr}}}(x_i \mid x_1, x_2, \dots, x_{i-1})} \quad (5.12)$$

Observe that (5.11) exhibits linearity, like (5.8).

Metrics for sentiment classification. Four common evaluation metrics for classification are used – accuracy, precision, recall and F1-Score for sentiment classification – to comprehensively analyse the classification performance of the compressed model. between the metric values after and before the compression.

Zero-shot reasoning scores. The scores obtained by Language Model Evaluation Harness¹⁰ are directly used as zero-shot reasoning evaluation metrics. The evaluated benchmarks are BoolQ [3], ARC-Easy and ARC-Challenge [2], HellaSwag [1], PIQA [4], and WinoGrande [45], as introduced in Tab. 2.1.

5.4.3 Experimental Results

Compression ratio and language task performance The language modelling performance and the compression ratio of different tensor orders and models are shown in Fig. 5.4. There is no general conclusion as to whether higher-order tensors are better than lower-order tensors, but roughly speaking, compression ratio and language modelling performance in Fig. 5.4e-Fig. 5.4h are better than those in Fig. 5.4i-Fig. 5.4j. This implies that there may exist (most probably) three-dimensional representation in the weight of embedding layers that can best generalise the investigated language tasks. On the other hand, due to the combination of tensor size and TT ranks exponentially exploding, not all possible combinations can be tested. However, the results still show that independent of the tensor orders and the models used for the compression, significant language modelling performance loss tends to appear when the compression ratio exceeds $2.0\times$. The proposed method is further compared with the Tucker decomposition in Fig. 5.4k with the same tensorisation strategy in Sec. 5.3.1, and the adopted Tensor-Train Decomposition outperforms the Tucker Decomposition in perplexity.

The results of the sentiment classification task also show that the robustness of larger-scale models (Cerebras-590M and Cerebras-1.3B) is better than that of the smaller models (Cerebras-111M and Cerebras-256M), similar to the trend in language modelling tasks mentioned above. The compressed larger-scale models tend to outperform the original model in precision and F1-score, indicating that the proposed compression improves the ability of the larger models to recognise the positive texts. In

¹⁰<https://github.com/EleutherAI/lm-evaluation-harness>

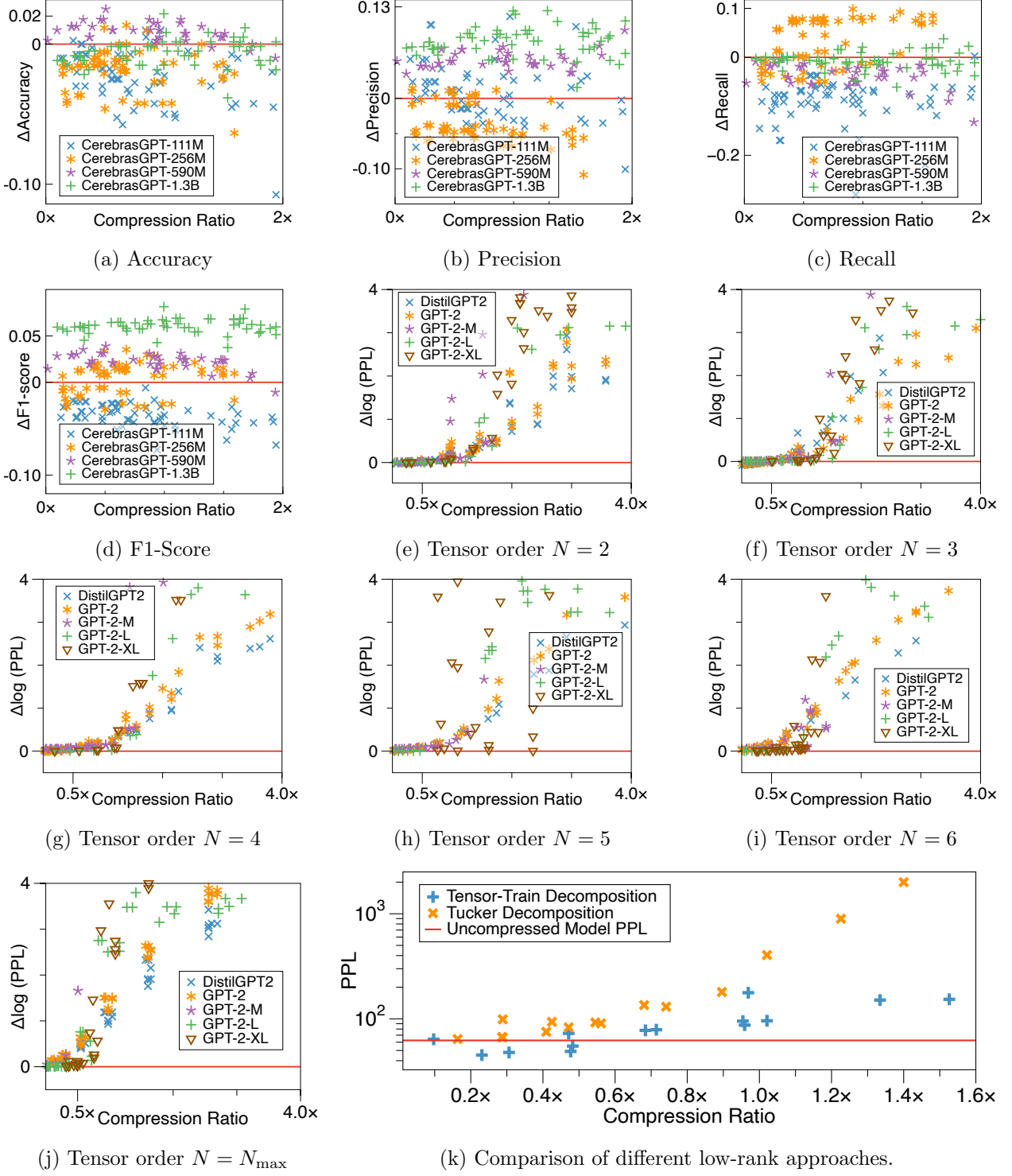


Figure 5.4: Language task performance change after the compression. (a)-(d): The accuracy, precision, F1-score and recall of sentiment classification, with the increasing compression ratio of the CerebrasGPT models. The higher the values, the better the classification performance. (e)-(j): The language modelling performance and compression ratio of the GPT-2 series models, where N denotes the tensor order. The best compression cases (higher compression ratio with negligible drop in accuracy) occur when $N = 3$, implying the optimal feature representation of the token embedding vectors may be three-dimensional. (g): Different tensor decomposition approaches, where PPL means perplexity.

contrast, the smaller models tends to have worse performance when the compression ratio increases.

A notable observation is that in both language modelling (Figs. 5.4e to 5.4j) and sentiment classification (Figs. 5.4a to 5.4d), the larger models (GPT-2-M, GPT-2-L, GPT-2-XL, CerebrasGPT-590M and CerebrasGPT-1.3B) are more robust to the compression ratio increase, compared with smaller models (DistilGPT2, GPT-2, CerebrasGPT-111M and CerebrasGPT-256M), especially when the compression ratio is less than $1.0\times$. This is probably because the embedding layers take a smaller proportion of the entire model, and also sheds light that TensorSLM might be used to improve the language task performance for large-scale models.

Sentiment classification. For sentiment classification, the common evaluation metrics for classification are adopted to comprehensively analyse the classification performance of the compressed model. To investigate the performance change before and after compression, this section uses the difference between the metric values after and before the compression.

Zero-shot reasoning. The comparison between TensorSLM with matrix-level SVD for embedding layer compression. The results and the discussion are in Tabs. 5.6 and 5.7. TensorSLM is compared with matrix-level SVD for embedding layer compression. The results are in Tabs. 5.6 and 5.7. The best result in each column (among compressed models) is shown in **bold**, and the second best is underlined.

Our approach has a higher chance of achieving better average reasoning task scores than the SVD-based approach, which implies that our tensors are better at extracting implicit representations in small-size models than matrices. Moreover, in our evaluation, our approach generally has higher scores than the SVD-based approach in ARC-challenge and BoolQ. Both of these datasets are more unprompted and unconstrained compared to the other evaluated datasets. This fact implies that our approach may be better at these difficult, unconstrained reasoning tasks.

Table 5.6: Zero-shot reasoning performance for OPT-125M, OPT-350M and their compressed versions. The best result in each column is shown in **bold**, and the second best is underlined.

	Param (%)	ARC-c	ARC-e	BoolQ	HellaS.	PIQA	WinoG.	Avg.
OPT-125M								
Original	100.00	23.38	57.11	57.74	41.53	71.71	59.35	49.33
SVD (matrices)	0.13	21.5	25.84	37.83	25.81	52.5	49.01	35.06
	26.57	20.05	31.14	37.83	26.59	56.31	50.59	36.65
	53.01	18.17	34.26	37.83	26.96	57.56	52.88	37.35
	79.45	18.77	<u>39.9</u>	45.63	27.38	59.9	50.83	39.54
	92.67	18.77	43.14	<u>47.37</u>	28.51	63	<u>51.3</u>	<u>40.89</u>
Ours (vectors)	2.47	<u>21.33</u>	26.39	37.83	25.63	52.94	50.59	35.53
	29.17	20.22	28.66	39.14	26.17	53.54	49.88	35.92
	50.78	21.25	29.55	40.15	26.19	54.9	50.12	36.50
	71.88	19.37	35.31	47.09	27.78	59.58	50.59	39.01
	87.11	19.03	39.6	59.51	<u>28.41</u>	<u>61.15</u>	33.78	41.80
OPT-350M								
Original	100.00	20.82	44.19	57.68	32.03	64.64	52.09	43.49
SVD (matrices)	0.20	21.5	25.25	37.83	25.67	51.36	49.17	34.87
	19.93	20.82	25.93	38.53	25.94	53.92	51.07	35.62
	39.66	20.22	25.8	38.62	26.22	53.26	50.59	35.41
	59.39	19.2	25.55	38.5	26.54	53.97	51.46	35.61
	79.12	19.2	27.53	37.83	27.16	55.93	49.33	35.80
	98.85	20.73	41.37	37.89	30.47	62.95	49.25	<u>39.48</u>
Ours (vectors)	3.52	21.08	24.92	45.47	25.66	53.1	51.14	36.58
	18.75	20.39	26.01	62.17	25.9	53.16	48.7	38.50
	28.13	20.05	24.87	62.17	26.08	53.54	49.33	38.60
	42.19	20.05	25.25	<u>48.44</u>	26.18	53.7	49.09	36.58
	70.31	20.48	25.42	62.17	26	53.16	<u>51.62</u>	38.87
	94.53	<u>21.42</u>	<u>36.15</u>	45.9	<u>29.59</u>	<u>61.92</u>	52.33	40.21

Table 5.7: Zero-shot reasoning performance for CerebrasGPT-111M, CerebrasGPT-256M, and CerebrasGPT-590M and their compressed versions. The best result in each column is shown in **bold**, and the second best is underlined.

	Param (%)	ARC-c	ARC-e	BoolQ	HellaS.	PIQA	WinoG.	Avg.
CerebrasGPT-111M								
Original	100.00	16.64	37.88	62.14	26.76	59.41	49.01	40.82
SVD (matrices)	0.13	<u>20.9</u>	26.52	37.86	25.46	52.45	48.93	35.10
	26.57	17.49	31.44	37.77	26.53	56.2	50.99	36.28
	53.01	17.06	35.27	44.16	26.57	56.75	50.28	37.74
	79.45	16.55	<u>37.08</u>	59.88	26.76	<u>58.81</u>	49.72	<u>40.38</u>
	92.67	15.44	37.92	<u>61.77</u>	26.84	59.19	49.17	40.56
Ours (vectors)	2.47	19.97	28.24	61.93	26.09	54.13	<u>50.36</u>	39.32
	29.17	20.48	29.84	59.85	26.26	55.66	50.04	39.55
	50.78	19.8	31.48	49.11	<u>26.78</u>	57.51	49.09	38.17
	71.88	17.92	34.51	58.32	26.74	58.05	50.28	40.05
	87.11	20.99	24.07	61.04	25.66	52.67	49.49	38.27
CerebrasGPT-256M								
Original	100.00	16.89	40.95	61.5	27.44	61.37	51.3	40.82
SVD (matrices)	0.09	21.16	26.73	37.83	25.74	52.29	51.22	35.40
	28.26	17.75	33	38.2	26.48	58.05	50.43	36.84
	47.05	17.15	35.61	39.97	26.83	59.03	<u>51.78</u>	37.79
	75.22	18.09	<u>39.44</u>	<u>61.01</u>	27.4	60.83	51.62	<u>41.77</u>
	94.00	18.17	40.74	59.94	27.4	<u>61.04</u>	50.91	41.70
Ours (vectors)	2.67	20.22	24.62	37.74	25.5	54.57	50.75	35.37
	38.60	<u>20.9</u>	27.57	37.83	25.8	53.48	49.96	35.57
	50.37	18.94	31.52	52.35	26.76	56.86	50.67	38.72
	61.40	20.05	35.02	59.91	27.3	57.94	50.59	40.67
	98.90	19.28	40.74	61.5	<u>27.39</u>	61.43	52.01	42.33
CerebrasGPT-590M								
Original	100.00	19.03	46.42	59.17	29.12	62.73	49.8	43.08
SVD (matrices)	0.07	21.33	27.1	37.92	25.79	52.45	47.99	35.23
	26.91	18.09	35.98	37.83	26.82	58	50.59	37.35
	47.03	17.24	39.31	37.83	27.56	59.74	<u>50.83</u>	38.18
	73.87	18.86	44.11	49.45	28.42	<u>61.64</u>	50.75	41.15
	94.00	19.8	46.17	52.72	<u>28.97</u>	62.19	49.88	<u>42.19</u>
Ours (vectors)	1.37	23.38	24.87	<u>56.61</u>	25.59	52.67	52.17	38.43
	19.21	19.97	26.81	49.82	25.66	52.72	49.17	36.91
	46.88	19.54	35.9	40.89	27.04	57.83	49.72	37.92
	66.41	20.31	38.09	58.87	28.21	60.28	49.88	41.42
	94.34	<u>22.1</u>	<u>44.7</u>	56.42	29.02	<u>61.64</u>	49.49	42.70

5.5 Low Energy Consumption

From the workload of the high-end GPU servers (e.g. those equipped with NVIDIA A100) and low-end edge devices (e.g. Raspberry Pi 5) described in [Sec. 5.2.1](#), it is clear that the edge device only takes charge of light-weight essential tasks, since it has strict limitations in computation, memory and communication. Furthermore, since battery life directly impacts the user experience, energy consumption is also a significant concern.

Energy Consumption Analysis. Following the methodology of [\[145\]](#), this section estimates the energy consumed when an SLM processes an input token through the embedding layer, prior to the decoder stage. Assuming in the same operating environment and other conditions (e.g. temperature), the memory energy cost of each `float32` is ν , and the computation energy cost of each `float32` is τ , all the model weights are represented in `float32`.

Cost comparison heuristic A heuristic comparing memory and compute costs guides design choices at runtime.

When inputting a text of length l , denote original model energy cost regarding memory as $\mathcal{E}_\nu$, model energy cost regarding computation is $\mathcal{E}_\tau$,

$$\mathcal{E}_\nu = \nu(dV + ld), \quad \mathcal{E}_\tau = 0, \quad (5.13)$$

and after compressing with TensorSLM, the energy costs are

$$\mathcal{E}'_\nu = \nu(VNIr^2 + lNIr^2 + ld), \quad \mathcal{E}'_\tau = \tau NIr^2. \quad (5.14)$$

Denote the SVD rank k , the energy cost after compressing with matrix-based SVD is

$$\mathcal{E}''_\nu = \nu[k(V + 2d + l + 1) + ld], \quad \mathcal{E}''_\tau = \tau(2ldk - ld + kd). \quad (5.15)$$

Therefore, the ratio of inference energy ω between the compressed language models and the uncompressed models. Denote $\omega_{\text{TT}} = \frac{\mathcal{E}'_\nu + \mathcal{E}'_\tau}{\mathcal{E}_\nu + \mathcal{E}_\tau}$ as the ratio with TensorSLM, and $\omega_{\text{SVD}} = \frac{\mathcal{E}''_\nu + \mathcal{E}''_\tau}{\mathcal{E}_\nu + \mathcal{E}_\tau}$ as the ratio with SVD. This thesis will give the estimated values of ω_{TT} and ω_{SVD} according to the hyperparameters of the investigated open-source SLMs.

The estimated inference energy costs are shown in [Fig. 5.5](#). The Y-axis indicates the ratio between the inference energy costs of the compressed model and that of the uncompressed model; the lower, the better energy saving. For each language model, the compression case that has a similar language task performance.

Note on evaluation protocol. Here, the other configurations (evaluated datasets) are aligned across models to ensure comparable language-task performance for reporting. The results show that

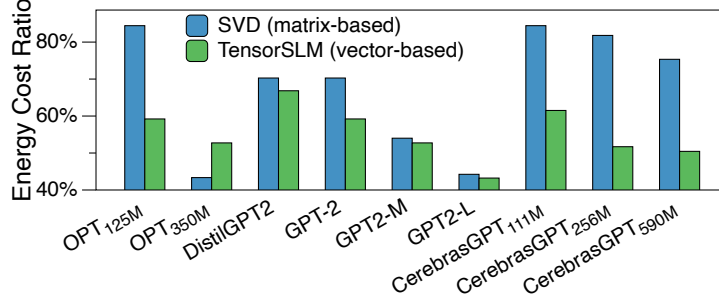


Figure 5.5: Inference energy costs of different models.

the tensor-based approach mostly outperforms the SVD-based approach. Furthermore, TensorSLM supports adaptivity in edge applications, while the SVD-based approach does not.

5.6 On-Device Latency

While TensorSLM significantly reduces the model parameters and even improves the language tasks performance, in practice it also introduced latencies - compression latency and inference latency.

Compression Latency

For the compression latency, the compression latency on the token level was investigated, as shown in Tab. 5.8. Here, “original” means the uncompressed model, while PPL_α means the compressed model with a negligible task performance drop. In this case, “negligible task performance drop” means in the language modelling task, the perplexity is no more than 100.0. The notation $\varphi_{\max}$ refers to the compressed model with maximum compression ratio. It was observed that for individual token embeddings, there was no significant latency difference between high-end servers and Raspberry Pi, typically no more than 2 milliseconds for each token. Thus, it is acceptable for the Raspberry Pi to compress the individual token embeddings.

Table 5.8: The latency (ms/token) of tensorisation & decomposition token embedding vectors and reconstruction on the high-end and lower-end devices. PPL_α means the compressed model with a negligible task performance drop, and the symbol $\varphi_{\max}$ represents the case with a maximum compression ratio. d_{emb} is the embedding dimension of the token embedding vector, and the tested models are GPT-2 and GPT-2-M. On the CPU level, for single token embedding vector decomposition and reconstruction, both server and edge devices have no significant computation overhead.

Device (CPU) (ms/token)	d_{emb}	tensorisation & decomposition		reconstruction	
		PPL_α	$\varphi_{\max}$	PPL_α	$\varphi_{\max}$
Server	768	0.627	1.429	0.117	0.238
	1024	0.452	1.512	0.114	0.261
Raspberry Pi 5	768	0.760	1.948	0.330	0.468
	1024	0.612	2.148	0.364	0.614

Inference Latency

For the inference latency of a single text with a typical length of 50 tokens (as shown in Tab. 5.10), the configurations “original”, PPL_α , and $\varphi_{\max}$ were used, same as those in Tab. 5.8, to represent the uncompressed model, the compressed model with a negligible task performance drop, and the model with a maximum compression ratio. A typical induced latency for an input text was no more than 0.3 seconds, which is acceptable for edge applications.

It should be noted that the embedding reconstruction latency depends on both tensor shapes and FLOPs, and the on-device memory management varies when models of different sizes are loaded. Consequently, in Tab. 5.10, FLOPs alone does not provide a complete predictor of on-device inference latency.

The cases of $\varphi_{\max}$ are typically slower than the cases of PPL_α . It has been demonstrated in Tab. 5.9 that the $\varphi_{\max}$ (the maximum compression ratio) corresponds to cases where embedding vectors are compressed into TT-formatted tensors of the highest orders. During the forward passes, the TT-format of these tensors is decompressed order by order. For example, for an N -order TT-formatted tensor, the decompression process involves $(N - 1)$ *serial* matrix multiplications.

This implies that the higher the tensor order, the more matrix multiplication rounds are executed, potentially resulting in slower decompression. A non-representative exception is the compression for CerebrasGPT-256M, which has the 5-order tensor shape $4 \times 2 \times 17 \times 4 \times 2$ for PPL_α , and 7-order tensor shape $2 \times 2 \times 2 \times 2 \times 17 \times 2 \times 2$ for $\varphi_{\max}$. The decompression process for each embedding vector involves 4 and 6 matrix multiplications respectively, which differ by only two matrix multiplications — a relatively small gap compared to other cases (e.g. 3-order for PPL_α and 10-order for $\varphi_{\max}$). Thus, for CerebrasGPT-256M, the on-device inference latencies of PPL_α and $\varphi_{\max}$ are similar, as shown in Tab. 5.10.

Table 5.9: Parameters, number of floating-point operations (FLOPs) of the compressed and uncompressed sub-billion models, and latency on Raspberry Pi CPU. For FLOPs, the token number of the input texts is 100, while for latency on Raspberry Pi, the token number is 50.

GPT Models		GPT2				CerebrasGPT		
		DistilGPT2	GPT-2	GPT-2-M	GPT-2-L	111M	256M	590M
# Params (M)	original	81.9	124.44	354.82	774.03	111.05	255.98	590.31
	PPL $_{\alpha}$	67.06	106.36	326.45	734.28	101.78	226.69	543.45
	$\varphi_{\max}$	43.45	85.99	303.88	710.83	71.87	200.59	511.07
FLOPs ($10^6/\text{text}$)	original	20250	40490	142250	330980	14470	40400	103060
	PPL $_{\alpha}$	+1.65	+1.88	+3.11	+2.30	+0.38	+1.63	+2.30
	$\varphi_{\max}$	+0.13 -92%	+0.13 -93%	+0.20 -94%	+0.25 -89%	+0.13 -66%	+0.12 -93%	+0.26 -84%
Latency on Raspberry Pi (s/text)	original	0.19 $\pm$ 0.02	0.50 $\pm$ 0.19	1.23 $\pm$ 0.12	3.01 $\pm$ 0.47	0.47 $\pm$ 0.21	0.71 $\pm$ 0.02	1.81 $\pm$ 0.25
	PPL $_{\alpha}$	0.36 $\pm$ 0.19	0.50 $\pm$ 0.16	1.26 $\pm$ 0.22	3.01 $\pm$ 0.29	0.48 $\pm$ 0.23	1.01 $\pm$ 0.29	1.89 $\pm$ 0.28
	$\varphi_{\max}$	0.19 $\pm$ 0.03 ↘	0.71 $\pm$ 0.38 ↘	1.55 $\pm$ 0.36 ↘	3.52 $\pm$ 0.44 ↘	0.72 $\pm$ 0.42 ↘	0.95 $\pm$ 0.27 ↘	1.91 $\pm$ 0.24 ↘
		-47%	+42%	+23%	+16%	+50%	-6%	+1%

Though this compression approach does not provide latency reduction benefits, it does offer advantages in the reduction of memory usage and energy consumption.

DistilGPT2 exhibits different FLOPs-latency trends from the others. In Tab. 5.10 the compression for DistilGPT2 has significantly less latency for $\varphi_{\max}$ than PPL $_{\alpha}$, which contradicts the analysis in the preceding paragraph. A possible reason is the different memory scheduling processes of embedding layers and non-embedding layers. DistilGPT has the same embedding layer weight matrix size (50257×768) as GPT-2, yet has significantly fewer non-embedding layer parameters (and hence $\sim 50\%$ fewer non-embedding memory pages during inference). This difference may lead to distinct memory management dynamics.

Table 5.10: Parameters, number of floating point operations (FLOPs) of the compressed and un-compressed sub-billion models, and latency on Raspberry Pi CPU. For FLOPs, the token number of the input texts is 100, while for latency on Raspberry Pi, the token number is 50.

Series	Model	Variant	Params (M)	FLOPs (10^6)	Latency(s)
GPT2	DistilGPT2	original	81.9	20250	$0.19_{\pm 0.02}$
		PPL $_{\alpha}$	67.06	+1.65	$0.36_{\pm 0.19}$
		$\varphi_{\max}$	43.45	+0.13	$0.19_{\pm 0.03}$
	GPT-2	original	124.44	40490	$0.50_{\pm 0.19}$
		PPL $_{\alpha}$	106.36	+1.88	$0.50_{\pm 0.16}$
		$\varphi_{\max}$	85.99	+0.13	$0.71_{\pm 0.38}$
	GPT-2-M	original	354.82	142250	$1.23_{\pm 0.12}$
		PPL $_{\alpha}$	326.45	+3.11	$1.26_{\pm 0.22}$
		$\varphi_{\max}$	303.88	+0.20	$1.55_{\pm 0.36}$
	GPT-2-L	original	774.03	330980	$3.01_{\pm 0.47}$
		PPL $_{\alpha}$	734.28	+2.30	$3.01_{\pm 0.29}$
		$\varphi_{\max}$	710.83	+0.25	$3.52_{\pm 0.44}$
CerebrasGPT	111M	original	111.05	14470	$0.47_{\pm 0.21}$
		PPL $_{\alpha}$	101.78	+0.38	$0.48_{\pm 0.23}$
		$\varphi_{\max}$	71.87	+0.13	$0.72_{\pm 0.42}$
	256M	original	255.98	40400	$0.71_{\pm 0.02}$
		PPL $_{\alpha}$	226.69	+1.63	$1.01_{\pm 0.29}$
		$\varphi_{\max}$	200.59	+0.12	$0.95_{\pm 0.27}$
	590M	original	590.31	103060	$1.81_{\pm 0.25}$
		PPL $_{\alpha}$	543.45	+2.30	$1.89_{\pm 0.28}$
		$\varphi_{\max}$	511.07	+0.26	$1.91_{\pm 0.24}$

5.7 Chapter Summary

In the context of generative language models with a parameter count of fewer than one billion, the embedding layers take significant proportions of the total model parameters. The low-rank approach is a promising technique for parameter reduction, however, the existing low-rank approaches to compressing language models all involve an extra training process. Based on the Tensor-Train Decomposition, this work aims to find the low-rank structure *without the need for extra training*. The experimental evaluation covers GPT-2/CerebrasGPT and OPT series models, more advanced architectures like Qwen (2.5-0.5B, 3-0.6B) and MiniCPM4-0.5B, as well as both simple language modelling/classification and more complex zero-shot reasoning tasks. The experimental results of embedding and extensional dense layer compression point toward a practical takeaway for TensorSLM: effective compression with this approach should be layer-aware and task-adaptive. Embedding layers can be safely compressed, while dense layers often hide excess capacity. The best results come from tuning compression levels to the sweet spots that balance efficiency and performance.

This section has found that for the larger models, the language task performance was maintained and

even improved while reducing the model size. Additionally, this section has obtained an empirical compression bound for the proposed approach, achieving a compression ratio of $0.5\times$ to $2.0\times$ in the embedding layers, without compromising language task performance. This section measured the latency of the proposed approach on a typical low-end device (i.e. Raspberry Pi). With a significant parameter reduction, the on-device compression latency was typically no more than 2 ms for each token, highly competent for edge applications.

There are both design tradeoffs inherent to the proposed approach and a broad range of future work following this line of work. Future directions include: **(1) hybrid compression**: combining tensor decomposition with pruning; **(2) accelerated tensor operations**: hardware-optimised kernels; **(3) architecture extensions**: applying the approach to other model families (LLaMA, Mistral) and modalities; **(4) adaptive compression**: dynamically adjusting compression based on deployment constraints.

Chapter 6

Conclusion

This chapter summarises the main theoretical and practical contributions of this thesis and their implications for controllable compression of language models, before outlining promising directions for future research and deployment.

6.1 Contribution

Overall contributions. This thesis develops a unified algebraic perspective on generative language model compression, connecting interpretability, software design, and deployment on resource constrained devices. It shows that transformer based language models admit low dimensional geometric structure that can be exploited for controllable, training free compression, and that these abstractions can be realised in a reusable tooling framework. Across [Chapters 3 to 5](#), the work demonstrates that tensor based factorisation is not only theoretically justified, but also practically effective for sub billion parameter models in realistic runtime environments.

Interpretation and analysis of LLM mechanism. [Chapter 3](#) reframes transformer based language models as compositions of linear operators acting on structured subspaces, rather than opaque stacks of parameters. The analysis in [Sec. 3.1](#) quantifies how different model families distribute parameters across embeddings, attention and feed forward layers, and relates these choices to memorisation oriented versus reasoning oriented behaviour. Building on this, the geometric formalism in [Sec. 3.2](#) introduces a subspace based taxonomy that treats matrix and tensor factorisations as first class modelling tools for compression. Finally, [Sec. 3.3](#) establishes log-linear relationships between representation sparsity, perplexity and accuracy, providing explicit targets for how far one can compress before model functionality shifts.

The significance of [Chapter 3](#) is twofold. Conceptually, it connects interpretability, geometric deep learning and compression into a single language based on subspaces and information flow, moving

beyond purely empirical pruning heuristics. Practically, it offers a way to reason about compression budgets in terms of representation dimension rather than ad hoc hyperparameters. These results invite further work on linking the introduced algebraic structure with safety critical behaviours, for example, by studying how subspace reallocation under compression affects factuality, robustness and alignment. The highlights for this part include:

- [Sec. 3.1](#) shows that larger models devote more capacity to feed forward layers, clarifying why their reasoning abilities improve under increased depth and width.
- [Sec. 3.2](#) reinterprets attention and MLP blocks as projections between subspaces, revealing where low rank structure is most compatible with model semantics.
- [Sec. 3.3](#) derives and validates analytic laws that tie dimension sparsity to perplexity and accuracy, turning compression into a controllable design choice.

Implementation with object oriented programming. [Chapter 4](#) operationalises the algebraic framework as a flexible software system for compression and evaluation. After surveying existing inference engines, evaluation harnesses and representation metrics in [Sec. 4.1](#), the requirements in [Sec. 4.2](#) distil concrete design principles: avoid duplicated logic across pipelines, separate model management from compression algorithms, and treat evaluation and profiling as first class citizens. The factory based architecture in [Secs. 4.3 to 4.5](#) realises these principles through registries and plugins that assemble compression, evaluation and profiling workflows on demand.

[Chapter 4](#) is significant because it turns abstract algebraic ideas into reproducible infrastructure that other researchers can extend. By encapsulating compressors, profilers and evaluators behind stable interfaces, it enables systematic comparisons between different low rank and pruning methods, rather than one off experimental scripts. The case studies in [Sec. 4.6](#) illustrate this point: they show that the same pipeline can support multiple model families and benchmarks while exposing nuanced trade offs between compression ratio, reasoning accuracy and engineering complexity. Future work can build on this framework to incorporate new modalities, hardware specific kernels and online adaptation mechanisms without redesigning the core system. The highlights for this part include:

- [Sec. 4.2](#) frames compression tooling as a software engineering problem, emphasising extensibility, reproducibility and separation of concerns.
- [Secs. 4.3 to 4.5](#) introduce registry based factories and model managers that allow new models, algorithms or third-parties to be added with minimal boilerplate.
- [Sec. 4.6](#) demonstrates end to end pipelines on Qwen and MiniCPM models, quantifying how algebraic compression interacts with real benchmarks.

Deployment and runtime analysis. [Chapter 5](#) addresses the practical question of whether algebraic compression can make small language models viable on resource constrained hardware. The

analysis in [Sec. 5.1](#) derives fine grained FLOPs counts for core PyTorch operators and transformer components, providing a hardware agnostic baseline for reasoning about latency. On top of this, [Secs. 5.2](#) and [5.3](#) introduce TensorSLM as a training free tensor train based compression scheme that targets embedding and dense layers while respecting the structure of the forward pass.

The empirical study in [Sec. 5.4](#) shows that embedding layer compression with TensorSLM consistently preserves, and sometimes improves, performance on language modelling and reasoning tasks across GPT-2, OPT and MiniCPM models. Dense layer compression exhibits task specific sweet spots, highlighting that compression policies must be adapted to evaluation objectives. Runtime measurements in [Secs. 5.5](#) and [5.6](#) demonstrate that on a Raspberry Pi 5, TensorSLM reduces inference energy by roughly half with negligible per token latency overhead, suggesting a concrete deployment path for edge applications. These results raise further research questions about co-designing algebraic structure, hardware and scheduling policies to unlock even more efficient on device reasoning. The highlights for this part include:

- [Sec. 5.1](#) provides a reusable FLOPs accounting toolkit, allowing compression schemes to be compared on computational as well as statistical grounds.
- [Secs. 5.2](#) and [5.3](#) clarifies the specification of resource-constraint deployment, [Sec. 5.4](#) show that tensor based embedding compression outperforms matrix-based SVD and structured pruning on several reasoning benchmarks at comparable parameter budgets.
- [Secs. 5.5](#) and [5.6](#) quantify latency and energy on real hardware, demonstrating that algebraic compression can achieve meaningful efficiency gains for small language models in edge settings.

6.2 Future Work

Overall future directions. Looking beyond the specific contributions of this thesis, several broader directions emerge for advancing controllable compression of language models. The algebraic viewpoint developed here suggests treating model design, compression algorithms and deployment strategies as a single co-designed system. Future work should therefore link subspace-level understanding, analytic performance laws and software infrastructure into an iterative loop that supports principled, data-driven compression decisions across model families and hardware platforms. The main cross-cutting research directions include:

- Closing the gap between algebraic taxonomy and practical model design by systematically mapping subspace constructions to concrete matrix and tensor factorisation schemes used in generative language models.
- Extending analytic laws that relate representation dimension, sparsity and performance to more diverse tasks, architectures and compression regimes, especially in low-data and safety-critical

settings.

- Integrating the compression framework, profiling tools and deployment workflows into an end-to-end system that can recommend compression policies conditioned on hardware, latency and energy constraints.

Interpretation and theory. From the perspective of interpretation, the subspace-oriented view of transformers and the taxonomy developed in [Sec. 3.2](#) can be deepened in several ways. A natural extension is to expand the coverage of existing compression and interpretability literature, positioning each method within the same geometric language and clarifying which subspace manipulations it implicitly implements. More ambitious directions include linking these algebraic structures to phenomena such as in-context learning, robustness under distribution shift and safety-relevant behaviours by analysing how subspace reallocation under compression affects factuality, calibration and failure modes. Finally, the linear laws studied in [Sec. 3.3](#) invite theoretical work that derives them from optimisation dynamics and generalisation theory rather than treating them as purely empirical regularities. Key future research topics in this area include:

- Building systematic benchmarks that test whether subspace-based predictions about which layers are compressible match observed changes in behaviour across diverse tasks, languages and prompting setups.
- Relating subspace structure to mechanistic circuits by combining the algebraic framework with causal intervention methods and sparse probes, aiming to distinguish genuinely mechanistic compression from mere parameter removal.
- Exploring task- and user-conditioned subspace allocation, where compression budgets are dynamically reallocated to components most relevant for factuality, alignment or domain-specific reasoning.

Tooling, deployment and runtime. On the systems side, the compression framework in [Chapter 4](#) and the runtime analysis in [Chapter 5](#) open several avenues for practical and methodological work. One direction is to broaden the range of model families, tasks and hardware targets supported by the tooling, turning it into a general-purpose platform for evaluating compression policies under realistic deployment constraints. Another is to push tensor-based methods such as TensorSLM beyond embeddings, exploring tensorisation of dense layers, attention blocks and even optimisers in ways that remain compatible with modern inference engines and compilation stacks. The FLOPs accounting in [Sec. 5.1](#) also invites closer collaboration with hardware and compiler communities to co-design kernels, scheduling strategies and quantisation schemes that exploit the algebraic structure exposed by factorisation. Finally, deploying compressed models in user-facing applications creates opportunities for longitudinal studies of reliability, calibration and user trust in resource-constrained settings. Promising directions for future systems work include:

- Turning the current research code into a robust, open-source framework with configuration, logging and experiment-tracking capabilities that make compression studies reproducible for other researchers and practitioners.
- Designing and evaluating tensor-friendly kernels and compilation strategies for tensor-train and related factorisations on CPUs, GPUs and emerging accelerators, guided by the FLOPs models developed in [Sec. 5.1](#).
- Running end-to-end deployment case studies on diverse edge platforms, measuring not only latency and energy but also user experience, failure recovery and maintainability of compressed model pipelines.

Bibliography

- [1] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL <https://aclanthology.org/P19-1472/>.
- [2] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.
- [3] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. 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 2924–2936, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1300. URL <https://aclanthology.org/N19-1300/>.
- [4] Yonatan Bisk, Rowan Zellers, Ronan Le bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7432–7439, April 2020. doi: 10.1609/aaai.v34i05.6239. URL <https://ojs.aaai.org/index.php/AAAI/article/view/6239>.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008, 2017.

- [6] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.
- [7] Ethan Goh, Robert Gallo, Jason Hom, Eric Strong, Yingjie Weng, Hannah Kerman, Joséphine A Cool, Zahir Kanjee, Andrew S Parsons, Neera Ahuja, et al. Large language model influence on diagnostic reasoning: a randomized clinical trial. *JAMA network open*, 7(10):e2440969–e2440969, 2024.
- [8] 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.
- [9] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.
- [10] Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL <https://qwenlm.github.io/blog/qwen2.5/>.
- [11] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, et al. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [12] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.
- [13] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021.
- [14] Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In Jennifer Dy and Andreas Krause,

- editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 254–263. PMLR, 10–15 Jul 2018. URL <https://proceedings.mlr.press/v80/arora18b.html>.
- [15] Gintare Karolina Dziugaite and Daniel M Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*, pages 1–10, 2017.
- [16] Wenda Zhou, Victor Veitch, Morgane Austern, Ryan P. Adams, and Peter Orbanz. Non-vacuous generalization bounds at the imagenet scale: a PAC-bayesian compression approach. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=BJgqqqsAct7>.
- [17] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Gutttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf.
- [18] 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.
- [19] Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7319–7328, 2021.
- [20] Kiho Park, Yo Joong Choe, Yibo Jiang, and Victor Veitch. The geometry of categorical and hierarchical concepts in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=bVTM2QKYuA>.
- [21] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. Toy mod-

- els of superposition. *Transformer Circuits Thread*, Anthropic, 2022. URL https://transformer-circuits.pub/2022/toy_model/index.html.
- [22] Zeyuan Allen-Zhu and Yuanzhi Li. Physics of language models: Part 3.3, knowledge capacity scaling laws. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=FxNNiUgtfa>.
- [23] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper_files/paper/2014/file/109d2dd3608f669ca17920c511c2a41e-Paper.pdf.
- [24] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2847–2854. PMLR, 06–11 Aug 2017. URL <https://proceedings.mlr.press/v70/raghu17a.html>.
- [25] James E Gentle. *Matrix Algebra: Theory, Computations and Applications in Statistics*. Springer Nature, 2023.
- [26] Wanli Yang, Fei Sun, Jiajun Tan, Xinyu Ma, Du Su, Dawei Yin, and Huawei Shen. The fall of ROME: Understanding the collapse of LLMs in model editing. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4079–4087, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.236. URL <https://aclanthology.org/2024.findings-emnlp.236/>.
- [27] Hao Wang, Shangwei Guo, Jialing He, Hangcheng Liu, Tianwei Zhang, and Tao Xiang. Model supply chain poisoning: Backdooring pre-trained models via embedding indistinguishability. In *Proceedings of the ACM on Web Conference 2025*, WWW ’25, page 840–851, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712746. doi: 10.1145/3696410.3714624. URL <https://doi.org/10.1145/3696410.3714624>.

- [28] Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. A modern self-referential weight matrix that learns to modify itself. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9660–9677. PMLR, 17–23 Jul 2022. URL <https://proceedings.mlr.press/v162/irie22b.html>.
- [29] Anton Razzhigaev, Matvey Mikhalechuk, Elizaveta Goncharova, Ivan Oseledets, Denis Dimetrov, and Andrey Kuznetsov. The shape of learning: Anisotropy and intrinsic dimensions in transformer-based models. In Yvette Graham and Matthew Purver, editors, *Findings of the Association for Computational Linguistics: EACL 2024*, pages 868–874, St. Julian’s, Malta, March 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.findings-eacl.58/>.
- [30] Mingxue Xu, Sadia Sharmin, and Danilo P Mandic. Geometry is all you need: A unified taxonomy of matrix and tensor factorization for compression of generative language models. *arXiv preprint arXiv:2410.03040*, 2024.
- [31] Mingxue Xu, Lisa Alazraki, and Danilo Mandic. How can representation dimension dominate structurally pruned LLMs? In *Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference Workshop at ICLR*, 2025. URL <https://openreview.net/forum?id=gQpgqngjqK>.
- [32] Mingxue Xu, Yao Lei Xu, and Danilo Mandic. TensorSLM: Energy-efficient embedding compression of sub-billion parameter language models on low-end devices. In *Tiny Titans: The next wave of On-Device Learning for Foundational Models (TTODLer-FM) Workshop at ICML*, 2025. URL <https://openreview.net/forum?id=Xrq8ISC6o1>.
- [33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [34] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhramil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyang Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhui Chen. Mmlu-pro: A more robust and challeng-

- ing multi-task language understanding benchmark. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 95266–95290. Curran Associates, Inc., 2024. doi: 10.52202/079017-3018. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ad236edc564f3e3156e1b2feafb99a24-Paper-Datasets_and_Benchmarks_Track.pdf.
- [35] 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>.
- [36] Lintang Sutawika, Hailey Schoelkopf, Leo Gao, Baber Abbasi, Stella Biderman, Jonathan Tow, ben fattori, Charles Lovering, farzanehnakhaee70, Jason Phang, Anish Thite, Fazz, Aflah, Niklas, Thomas Wang, sdtblck, nopperl, gakada, tttyuntian, researcher2, Julen Etxaniz, Chris, Hanwool Albert Lee, Leonid Sinev, Zdeněk Kasner, Kiersten Stokes, Khalid, KonradSzafer, Jeffrey Hsu, and Anjor Kanekar. Eleutherai/lm-evaluation-harness: v0.4.9.1, August 2025. URL <https://doi.org/10.5281/zenodo.16737642>.
- [37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318. Association for Computational Linguistics, 2002.
- [38] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81. Association for Computational Linguistics, 2004.
- [39] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=SkeHuCVFDr>.
- [40] Thibault Sellam, Dipanjan Das, and Ankur Parikh. BLEURT: Learning robust metrics for text generation. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*,

- pages 7881–7892, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.704. URL <https://aclanthology.org/2020.acl-main.704/>.
- [41] Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie. COMET: A neural framework for MT evaluation. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2685–2702, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.213. URL <https://aclanthology.org/2020.emnlp-main.213/>.
- [42] Ricardo Rei, Marcos Treviso, Nuno M. Guerreiro, Chrysoula Zerva, Ana C Farinha, Christine Maroti, José G. C. de Souza, Taisiya Glushkova, Duarte Alves, Luisa Coheur, Alon Lavie, and André F. T. Martins. CometKiwi: IST-unbabel 2022 submission for the quality estimation shared task. In Philipp Koehn, Loïc Barrault, Ondřej Bojar, Fethi Bougares, Rajen Chatterjee, Marta R. Costa-jussà, Christian Federmann, Mark Fishel, Alexander Fraser, Markus Freitag, Yvette Graham, Roman Grundkiewicz, Paco Guzman, Barry Hadrow, Matthias Huck, Antonio Jimeno Yepes, Tom Kocmi, André Martins, Makoto Morishita, Christof Monz, Masaaki Nagata, Toshiaki Nakazawa, Matteo Negri, Aurélie Névéal, Mariana Neves, Martin Popel, Marco Turchi, and Marcos Zampieri, editors, *Proceedings of the Seventh Conference on Machine Translation (WMT)*, pages 634–645, Abu Dhabi, United Arab Emirates (Hybrid), December 2022. Association for Computational Linguistics. URL <https://aclanthology.org/2022.wmt-1.60/>.
- [43] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023. URL <https://arxiv.org/abs/2311.07911>.
- [44] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL <https://openreview.net/forum?id=Byj72udxe>.
- [45] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

- [46] Tom Kocmi, Eleftherios Avramidis, Rachel Bawden, Ondřej Bojar, Anton Dvorkovich, Christian Federmann, Mark Fishel, Markus Freitag, Thamme Gowda, Roman Grundkiewicz, Barry Haddow, Marzena Karpinska, Philipp Koehn, Benjamin Marie, Christof Monz, Kenton Murray, Masaaki Nagata, Martin Popel, Maja Popović, Mariya Shmatova, Steinhórf Steingrímsson, and Vilém Zouhar. Findings of the WMT24 general machine translation shared task: The LLM era is here but MT is not solved yet. In *Proceedings of the Ninth Conference on Machine Translation*, pages 1–46. Association for Computational Linguistics, November 2024. doi: 10.18653/v1/2024.wmt-1.1. URL <https://aclanthology.org/2024.wmt-1.1/>.
- [47] Patrick Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. Groupreduce: Block-wise low-rank approximation for neural language model shrinking. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [48] Matan Ben Noach and Yoav Goldberg. Compressing pre-trained language models by matrix decomposition. In Kam-Fai Wong, Kevin Knight, and Hua Wu, editors, *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, December 2020. Association for Computational Linguistics. URL <https://aclanthology.org/2020.aacl-main.88>.
- [49] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu 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>.
- [50] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4847–4860, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.436.
- [51] Peiyu Liu, Ze-Feng Gao, Wayne Xin Zhao, Zhi-Yuan Xie, Zhong-Yi Lu, and Ji-Rong Wen. Enabling lightweight fine-tuning for pre-trained language model compression based on ma-

- trix product operators. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5388–5398, 2021.
- [52] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Re. Monarch: Expressive structured matrices for efficient and accurate training. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 4690–4721. PMLR, 17–23 Jul 2022. URL <https://proceedings.mlr.press/v162/dao22a.html>.
- [53] Patrick Chen, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Drone: Data-aware low-rank compression for large nlp models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 29321–29334. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/f56de5ef149cf0aedcc8f4797031e229-Paper.pdf.
- [54] Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. LoSparse: Structured compression of large language models based on low-rank and sparse approximation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 20336–20350. PMLR, 23–29 Jul 2023. URL <https://proceedings.mlr.press/v202/li23ap.html>.
- [55] Ali Edalati, Marzieh Tahaei, Ahmad Rashid, Vahid Nia, James Clark, and Mehdi Rezagholizadeh. Kronecker decomposition for GPT compression. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *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>.
- [56] Marzieh Tahaei, Ella Charlaix, Vahid Nia, Ali Ghodsi, and Mehdi Rezagholizadeh. KroneckerBERT: Significant compression of pre-trained language models through Kronecker de-

- composition and knowledge distillation. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *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>.
- [57] Benyou Wang, Yuxin Ren, Lifeng Shang, Xin Jiang, and Qun Liu. Exploring extreme parameter compression for pre-trained language models. In *International Conference on Learning Representations*, 2022.
- [58] Shikai Qiu, Andres Potapczynski, Marc Anton Finzi, Micah Goldblum, and Andrew Gordon Wilson. Compute better spent: Replacing dense layers with structured matrices. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 41698–41716. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/qiu24f.html>.
- [59] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [60] Sunzhu Li, Peng Zhang, Guobing Gan, Xiuqing Lv, Benyou Wang, Junqiu Wei, and Xin Jiang. Hypoformer: Hybrid decomposition transformer for edge-friendly neural machine translation. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 7056–7068, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.475. URL <https://aclanthology.org/2022.emnlp-main.475>.
- [61] Xiaoyu Liu, Jiahao Su, and Furong Huang. Tuformer: Data-driven design of transformers for improved generalization or efficiency. In *The Tenth International Conference on Learning Representations*, 2022.

- [62] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.
- [63] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016. URL <http://arxiv.org/abs/1510.00149>.
- [64] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1580. URL <https://aclanthology.org/P19-1580/>.
- [65] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/2c601ad9d2ff9bc8b282670cdd54f69f-Paper.pdf.
- [66] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=Syl02yStDr>.
- [67] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- [68] 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.
- [69] J Shane McCarley, Rishav Chakravarti, and Vinod Kumar Sridhar. Structured pruning of a bert-based question answering model. 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 1834–1845, 2019.

- [70] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=K9bw7vqp_s.
- [71] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=H1Y8hhg0b>.
- [72] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/41bfd20a38bb1b0bec75acf0845530a7-Paper.pdf.
- [73] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan 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>.
- [74] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 4475–4488. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/1caf09c9f4e6b0150b06a07e77f2710c-Paper-Conference.pdf.
- [75] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=rJl-b3RcF7>.
- [76] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Toward more

- efficient training of deep networks. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=BJxsrgStvr>.
- [77] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017. URL <https://arxiv.org/abs/1710.01878>.
- [78] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [79] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
- [80] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*, pages 291–326. Chapman and Hall/CRC, 2022.
- [81] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Christos Louizos. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.
- [82] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 30318–30332. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/c3ba4962c05c49636d4c6206a97e9c8a-Paper-Conference.pdf.
- [83] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In A. Oh, T. Naumann, A. Ghiber, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 10088–10115. Curran Associates,

- Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1feb87871436031bdc0f2beaa62a049b-Paper-Conference.pdf.
- [84] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27168–27183. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf.
- [85] Gunho Park, Baeseong park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. LUT-GEMM: Quantized matrix multiplication based on LUTs for efficient inference in large-scale generative language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=gLARhFLEOF>.
- [86] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 17402–17414. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/6f6db140de9c9f111b12ef8a216320a9-Paper-Conference.pdf.
- [87] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and overcoming the challenges of efficient transformer quantization. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7947–7969, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.627. URL <https://aclanthology.org/2021.emnlp-main.627/>.
- [88] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=ryup8-WCW>.

- [89] Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7319–7328, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.568. URL <https://aclanthology.org/2021.acl-long.568>.
- [90] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 192–204, San Diego, California, USA, 09–12 May 2015. PMLR. URL <https://proceedings.mlr.press/v38/choromanska15.html>.
- [91] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1019–1028. PMLR, 06–11 Aug 2017. URL <https://proceedings.mlr.press/v70/dinh17b.html>.
- [92] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velivcković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *ArXiv*, abs/2104.13478, 2021.
- [93] Andrew Gordon Wilson. Position: Deep learning is not so mysterious or different. In *Forty-second International Conference on Machine Learning Position Paper Track*, 2025. URL <https://openreview.net/forum?id=42Au7FoD8F>.
- [94] Oscar Skean, Md Rifat Arefin, Dan Zhao, Niket Nikul Patel, Jalal Naghiyev, Yann LeCun, and Ravid Shwartz-Ziv. Layer by layer: Uncovering hidden representations in language models. In *Forty-second International Conference on Machine Learning*, 2025. URL <https://openreview.net/forum?id=WGXb7UdvTX>.
- [95] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann,

- Amanda Aspell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. <https://transformer-circuits.pub/2021/framework/index.html>.
- [96] Chris Olah, Nick Cammarata, Michael Petrov, Ludwig Schubert, and Gabriel Goh. A mechanistic interpretability analysis of grokking, 2022. URL <https://transformer-circuits.pub/2022/mech-interp-essay/index.html>. Transformer Circuits Thread.
- [97] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. *arXiv preprint arXiv:2209.10652*, 2022. URL <https://arxiv.org/abs/2209.10652>.
- [98] Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermy, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Aspell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. Towards monosemanticity: Decomposing language models with dictionary learning, October 2023. URL <https://transformer-circuits.pub/2023/monosemantic-features>.
- [99] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermy, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet, May 2024. URL <https://transformer-circuits.pub/2024/scaling-monosemanticity>.
- [100] Tom Lieberum, Senthoran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on

- gemma 2, November 2024. URL <https://deepmind.google/discover/blog/gemma-scope-helping-the-safety-community-shed-light-on-the-inner-workings-of-language-models>
- [101] Emmanuel Ameisen, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, Brian Chen, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermy, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits Thread*, March 2025. URL <https://transformer-circuits.pub/2025/attribution-graphs/methods.html>.
- [102] Jack Lindsey, Adly Templeton, Jonathan Marcus, Thomas Conerly, Joshua Batson, and Christopher Olah. Sparse crosscoders for cross-layer features and model diffing. *Transformer Circuits Thread*, October 2024. URL <https://transformer-circuits.pub/2024/crosscoders/index.html>.
- [103] Clémentine Fourrier, Nathan Habib, Alina Lozovskaya, Konrad Szafer, and Thomas Wolf. Open llm leaderboard v2. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard, 2024.
- [104] Viktoriia Chekalina, Georgiy Novikov, Julia Gusak, Alexander Panchenko, and Ivan Oseledets. Efficient GPT model pre-training using tensor train matrix representation. In Chu-Ren Huang, Yasunari Harada, Jong-Bok Kim, Si Chen, Yu-Yin Hsu, Emmanuele Chersoni, Pranav A, Winnie Huiheng Zeng, Bo Peng, Yuxi Li, and Junlin Li, editors, *Proceedings of the 37th Pacific Asia Conference on Language, Information and Computation*, pages 600–608, Hong Kong, China, December 2023. Association for Computational Linguistics. URL <https://aclanthology.org/2023.paclic-1.60/>.
- [105] Tri Dao, Nimit Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=BkgrBgSYDS>.

- [106] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. *Proceedings of machine learning research*, 97:1517–1527, 2019.
- [107] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, mar 2003. ISSN 1532-4435.
- [108] Gregoire Deletang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=jznbginyus>.
- [109] Sheldon Axler. *Linear Algebra Done Right*. Springer, 4 edition, 2024.
- [110] Christopher De Sa, Albert Cu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A Two-pronged Progress in Structured Dense Matrix Vector Multiplication, pages 1060–1079. 2018. doi: 10.1137/1.9781611975031.69. URL <https://epubs.siam.org/doi/abs/10.1137/1.9781611975031.69>.
- [111] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5): 2295–2317, 2011. doi: 10.1137/090752286. URL <https://doi.org/10.1137/090752286>.
- [112] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. doi: 10.1137/07070111X. URL <https://doi.org/10.1137/07070111X>.
- [113] Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari do Nascimento, Torsten Hoefer, and James Hensman. Slicept: Compress large language models by deleting rows and columns. *arXiv preprint arXiv:2401.15024*, 2024.
- [114] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *arXiv preprint arXiv:1910.07467*, 2019.
- [115] Federico Barbero, Alex Vitvitskyi, Christos Perivolaropoulos, Razvan Pascanu, and Petar Veličković. Round and round we go! what makes rotary positional encodings useful? In

- The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=GtvuNrK58a>.
- [116] Hugo Touvron et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971, 2023.
 - [117] Microsoft Research. Phi-3 technical report. arXiv preprint, 2024.
 - [118] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
 - [119] Quentin Garrido, Randall Balestrieri, Laurent Najman, and Yann Lecun. RankMe: Assessing the downstream performance of pretrained self-supervised representations by their rank. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10929–10974. PMLR, 23–29 Jul 2023. URL <https://proceedings.mlr.press/v202/garrido23a.html>.
 - [120] Vimal Thilak, Chen Huang, Omid Saremi, Laurent Dinh, Hanlin Goh, Preetum Nakkiran, Joshua M Susskind, and Etai Littwin. LiDAR: Sensing linear probing performance in joint embedding SSL architectures. *International Conference on Learning Representations*, 2024.
 - [121] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *International Conference on Learning Representations*, 2019.
 - [122] Oscar Skean, Jhoan Keider Hoyos Osorio, Austin J Brockmeier, and Luis Gonzalo Sanchez Giraldo. DiME: Maximizing mutual information by a difference of matrix-based entropies. *arXiv preprint arXiv:2301.08164*, 2023.
 - [123] Zhihang Yuan, Yuzhang Shang, Yue Song, Qiang Wu, Yan Yan, and Guangyu Sun. Asvd: Activation-aware singular value decomposition for compressing large language models. *arXiv preprint arXiv:2312.05821*, 2023.

- [124] Mingxue Xu, Lisa Alazraki, and Danilo P Mandic. How can representation dimension dominate structurally pruned llms? *CoRR*, 2025.
- [125] Arm Architecture Team. Part 3: Matrix-matrix multiplication. Neon, SVE, and SME compared — community.arm.com. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/matrix-matrix-multiplication-neon-sve-and-sme-compared>, 2023. [Accessed 2025-05-08].
- [126] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuan-dong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 61121–61143. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/zhao24s.html>.
- [127] Fred Hohman, Mary Beth Kery, Donghao Ren, and Dominik Moritz. Model compression in practice: Lessons learned from practitioners creating on-device machine learning experiences. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI ’24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703300. doi: 10.1145/3613904.3642109. URL <https://doi.org/10.1145/3613904.3642109>.
- [128] Defu Liu, Yixiao Zhu, Zhe Liu, Yi Liu, Changlin Han, Jinkai Tian, Ruihao Li, and Wei Yi. A survey of model compression techniques: Past, present, and future. *Frontiers in Robotics and AI*, 12, 2025. ISSN 2296-9144. doi: 10.3389/frobt.2025.1518965. URL <https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2025.1518965>.
- [129] Yuzhang Shang, Zhihang Yuan, Bin Xie, Bingzhe Wu, and Yan Yan. Post-training quantization on diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1972–1981, June 2023.
- [130] Shikai Qiu, Andres Potapczynski, Marc Anton Finzi, Micah Goldblum, and Andrew Gordon Wilson. Compute better spent: Replacing dense layers with structured matrices. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scar-

- lett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 41698–41716. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/qiu24f.html>.
- [131] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4847–4860, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.436. URL <https://aclanthology.org/2020.findings-emnlp.436>.
- [132] Haoyu Wang, Ruirui Li, Haoming Jiang, Zhengyang Wang, Xianfeng Tang, Bin Bi, Monica Cheng, Bing Yin, Yaqing Wang, Tuo Zhao, and Jing Gao. Lighttoken: A task and model-agnostic lightweight token embedding framework for pre-trained language models. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD ’23, page 2302–2313, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701030. doi: 10.1145/3580305.3599416. URL <https://doi.org/10.1145/3580305.3599416>.
- [133] Klaudia Bałazy, Mohammadreza Banaei, Rémi Lebrete, Jacek Tabor, and Karl Aberer. Direction is what you need: Improving word embedding compression in large language models. In Anna Rogers, Iacer Calixto, Ivan Vulić, Naomi Saphra, Nora Kassner, Oana-Maria Camburu, Trapit Bansal, and Vered Shwartz, editors, *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*, pages 322–330, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.repl4nlp-1.32. URL <https://aclanthology.org/2021.repl4nlp-1.32/>.
- [134] Shicong Liu, Hongtao Lu, and Junru Shao. Improved residual vector quantization for high-dimensional approximate nearest neighbor search. *arXiv preprint arXiv:1509.05195*, 2015.
- [135] Ting Chen, Martin Renqiang Min, and Yizhou Sun. Learning k-way d-dimensional discrete codes for compact embedding representations. In *International Conference on Machine Learning*, pages 854–863. PMLR, 2018.
- [136] Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. Language

- model compression with weighted low-rank factorization. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=uPv9Y3gmAI5>.
- [137] Chi-Heng Lin, Shangqian Gao, James Seale Smith, Abhishek Patel, Shikhar Tuli, Yilin Shen, Hongxia Jin, and Yen-Chang Hsu. Modegpt: Modular decomposition for large language model compression. *arXiv preprint arXiv:2408.09632*, 2024.
- [138] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 32431–32454. PMLR, 21–27 Jul 2024.
- [139] Stefanos Laskaridis, Kleomenis Katevas, Lorenzo Minto, and Hamed Haddadi. Melting point: Mobile evaluation of language transformers. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom ’24, page 890–907, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704895. doi: 10.1145/3636534.3690668. URL <https://doi.org/10.1145/3636534.3690668>.
- [140] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Saxena, Sandhini Sharma, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [141] Nolan Dey, Gurpreet Gosal, Hemant Khachane, William Marshall, Ribhu Pathria, Marvin Tom, Joel Hestness, et al. Cerebras-gpt: Open compute-optimal language models trained on the cerebras wafer-scale cluster. *arXiv preprint arXiv:2304.03208*, 2023.
- [142] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

- [143] MiniCPM Team. MiniCPM4: Ultra-efficient llms on end devices. Tech. report, 2025. URL <https://github.com/OpenBMB/MiniCPM>.
- [144] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.
- [145] Anonymous. A practical methodology for measuring inference power on edge devices. Technical report, 2023.