LLM Size Reduction & Carbon Footprint

Auteur : Dosquet, Pierre
Promoteur(s) : Ittoo, Ashwin
Faculté : HEC-Ecole de gestion de l'Université de Liège
Diplôme : Master en ingénieur de gestion, à finalité spécialisée en digital business
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/22785

LLM Size Reduction & Carbon Footprint

Jury:

Supervisor:

Ashwin ITTOO

Reader(s):

Joseph SMITZ

Master thesis presented by
Pierre DOSQUET
To obtain the degree of
MASTER IN MANAGEMENT
BUSINESS ENGINEERING
with a specialization in
Digital Business
Academic year 2024/2025

Acknowledgements[1]

I would first like to express my sincere gratitude to my academic supervisor, **Ashwin Ittoo**, for his invaluable guidance, availability, and consistent support throughout the supervision of this work. His insightful advice and thoughtful feedback greatly contributed to the quality and direction of this research.

My appreciation also goes to **Nelie Laura Makenne**, who provided me with the initial scientific resources that enabled me to begin this work with greater serenity.

I am equally thankful to **the coordinators and faculty members of the Digital Business program**. The technical and analytical skills I developed throughout the curriculum have been essential in meeting the requirements of this research and in strengthening my ability to carry it out independently.

Finally, I would like to thank my parents and friends for their support throughout my studies.

The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247.

[1] For reasons of transparency, it should be noted that generative artificial intelligence was used as a support tool during the redaction of this master thesis.

Table of contents

Glossary

Terms	Definitions
carbon dioxide equivalent	A carbon dioxide equivalent or CO2 equivalent, abbreviated as CO2-eq is a metric measure used to compare the emissions from various greenhouse gases on the basis of their global-warming potential (GWP), by converting amounts of other gases to the equivalent amount of carbon dioxide with the same global warming potential (eurostat, 2025)
carbon footprint	The total amount of greenhouse gases that are emitted into the atmosphere each year by a person, family, building, organization, or company. A person's carbon footprint includes greenhouse gas emissions from fuel that an individual burns directly, such as by heating a home or riding in a car. It also includes greenhouse gases that come from producing the goods or services that the individual uses, including emissions from power plants that make electricity, factories that make products, and landfills where trash gets sent (United States Environmental Protection Agency, 2025)
greenhouse gas	A group of gases contributing to global warming and climate change. As defined under the Kyoto Protocol (1997), these include three non-fluorinated gases (i.e. carbon dioxide (CO_2), methane (CH_4), and nitrous oxide (N_2O) and four fluorinated gases (i.e. hydrofluorocarbons (HFCs), perfluorocarbons (PFCs), sulphur hexafluoride (SF_6), and nitrogen trifluoride (NF_3). Their impact is commonly measured in terms of CO_2 equivalent (CO_2e), which expresses the global warming potential of each gas relative to that of carbon dioxide (eurostat, 2025)
large language model	A type of neural network designed to predict sequences of words based on input data. Large language models (LLMs) have significantly advanced in recent years and are now capable of engaging in dialogue, generating coherent prose, and analyzing vast amounts of textual data (MIT Sloan Teaching & Learning Technologies, 2025)
parameters	Parameters are categorized into two types: construction parameters, which define the model's architecture (e.g., the organization and interconnection of layers), and behavior parameters, which govern how the model responds to input, adapts to data, and produces outputs. In this work, the term parameter specifically refers to behavior parameters, understood as the dynamic elements that shape the model's responses and operational behavior in interaction with user input or external systems (MIT Sloan Teaching & Learning Technologies, 2025)

Introduction

In recent years, model compression techniques have proven to be highly effective at reducing large language models' storage footprint and accelerating inference. By lowering memory requirements and increasing computational speed, these methods offer a promising pathway to more energy-efficient large language models. However, since model parameters encode learned knowledge, the alteration of their number or numerical representation inherently risks degrading performance. This trade-off between efficiency gains and potential quality loss remains a central concern when adopting compression strategies.

While the environmental impact of training large uncompressed models has received increasing attention, the effects of compression on hardware energy consumption and associated greenhouse gas (GHG) emissions during inference remain largely unexplored. To my knowledge, no prior work has investigated how compression influences emissions or energy use.

To address these gaps, I carried out an empirical study investigating the effects of compression on model performance and on the energy consumption of hardware components, specifically CPU, GPU, and RAM. For this purpose, four decoder-only transformer models were selected to reflect a balance of academic relevance and technological maturity: LLaMA-7B, LLaMA-30B, Mistral-7B-v0.3, and Mistral Small 3. Each model was compressed using the OPTQ method at 4-bit precision, so that both the full-precision and quantized versions could be compared.

To evaluate model performance, I use three commonly adopted metrics, namely WikiText-2 perplexity, and MMLU and IFEval accuracy. Hardware energy consumption was measured using the CodeCarbon package, running inference on two distinct hardware configurations. To contextualize these energy measurements in environmental terms, we carry out a comparative carbon footprint analysis using four distinct electricity mixes, enabling a grid-aware perspective on compression-related emissions measured in $CO2eq$.

My results yield three key findings. First, the performance degradation introduced by quantization is marginal: compressed models retain nearly all their capabilities across benchmarks, including language modeling, reasoning, and instruction following.

Second, compression can substantially reduce direct energy consumption, but the effect is highly sensitive to the hardware configuration. In our experiments, both hardware setups were high-end, yet I observed that a seemingly minor difference between them led to sharply divergent outcomes: one configuration resulted in energy savings of up to 39%, while the other caused an increase in hardware energy consumption of up to 26%. This highlights that even subtle hardware variations can significantly affect the energy efficiency of compressed models.

Third, our carbon footprint analysis highlights that environmental impact depends not only on model architecture or energy use, but also on the geographic context in which the model is deployed. I computed CO_2eq emissions by applying country-specific carbon intensity data to the measured energy consumption of both compressed and uncompressed models. Strikingly, I observed that a compressed model deployed in a carbon-intensive grid can emit up to six times more emissions than a full-sized model deployed in a low-carbon region such as France. Conversely, relocating a full-sized model from a high-intensity grid to a cleaner one can reduce emissions by an order of magnitude.

Taken together, our findings underscore that the sustainability benefits of model compression cannot be evaluated in isolation but must be understood in relation to both the hardware used and the geographic context of deployment.

Developments

Literature review

Transformer Networks

In this section, I introduce the Transformer architecture, which forms the basis of most modern large language models. The objective is to provide only the background necessary to facilitate the understanding of the remainder of this work. While it is important to acknowledge that some models use variations of the Transformer architecture, this overview is limited to the fundamental structure and functioning of the architecture. The purpose is not to provide an exhaustive review, but rather to offer a general understanding. I used the description of Transformer Architecture in (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023).

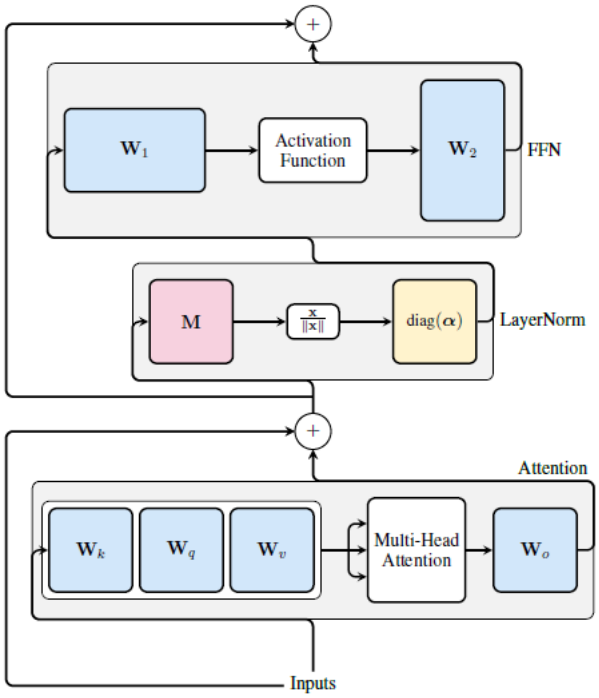


Figure 1: Diagram of a single layer in a Transformer network from (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023) Inputs are first processed through the attention block (including multi-head attention and associated linear projections), followed by layer normalization and a feed-forward network (FFN). The linear operations in both the attention and FFN blocks are shown in blue, and the key operations of the layer normalization are highlighted. Biases are omitted for clarity.

Transformer networks are composed of a stack of layers, each containing a multi-head self-attention block followed by a feed-forward network (FFN) block, interleaved with normalization and residual connections.

Embeddings

Let D, be the embedding dimension and N the sequence length. The input to the model is a sequence of token IDs and position IDs, which are used to index into the embedding matrix W_{emb}. For an input sequence s, this produces the initial signal matrix:

$$X = W_{emb}[s, :] \text{ where } X \in \mathbb{R}^{N \times D}.$$

Layer Normalization

After embedding lookup, the signal matrix X undergoes normalization which is expressed as:

$$LayerNorm(X) = RMSNorm(XM)diag(\alpha)\sqrt{D} + 1_N B^T$$

Equation 1

where $RMSNorm$ divides each row by its norm, $M = I - \frac{1}{D} * 11^T$ subtracts the mean for each row, and α,β are learnable parameters.

Multi-Head Attention Blocks

The core of the Transformer is the multi-head self-attention mechanism[2]. The input signal matrix X is linearly projected into three matrices:

$$Q = XW_q,\ K = XW_k \text{ and } V = XW_v \text{ where } W_q,\ W_k \text{ and } W_v \in \mathbb{R}^{D \times D}$$

The multi-head attention divides the embedding dimension D into h separate heads, with each head having dimension $d_k = D/h$. For each head i, we extract $Q_i, K_i, V_i \in \mathbb{R}^{N \times d_k}$.

For each head, the self-attention is computed as:

$$head_i = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

Equation 2

This softmax is applied along the sequence length, meaning each token computes a distribution over the input positions, weighted by the similarity of its query and all keys, scaled by $1/\sqrt{d_k}$.

Once the attention outputs for all h heads are computed, they are concatenated along the last dimension to form a matrix of shape $N \times D$, then projected with the output matrix W_o:

$$Multihead(X) = Concat(head_1, \ldots, head_h)W_o$$

Equation 3

This mechanism enables each head to capture different types of dependencies in parallel.

In (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023), this entire attention block is written in a compact, unified form as:

$$\sigma(XW_{in} + b_{in})W_{out} + b_{out}$$

Equation 4

where σ denotes the multi-head attention operation.

[2] *Where necessary, particularly for the details of the multi-head attention mechanism, the original "Attention Is All You Need" paper* (Vaswani, et al., 2017) *was consulted for clarification.*

Feed Forward Network (FFN) Blocks

After the attention block, the signal passes through a feed-forward network, typically a two-layer MLP:

$$FFN(X) = \sigma(XW_1 + b_1)W_2 + b_2$$

Equation 5

where σ is a is a non-linear activation (often GELU or ReLU).

Residual Connections and Normalization

After each attention and FFN block, a residual connection adds the input to the output of the block, and normalization (LayerNorm or RMSNorm) is applied to stabilize training and accelerate convergence.

Language Modeling Head

Following L alternating layers of attention and FFN, a final head block computes logits for the vocabulary:

$$Output = XW_{head} + b_{head}$$

Equation 6

where X is the output of the last Transformer block.

Compression methods

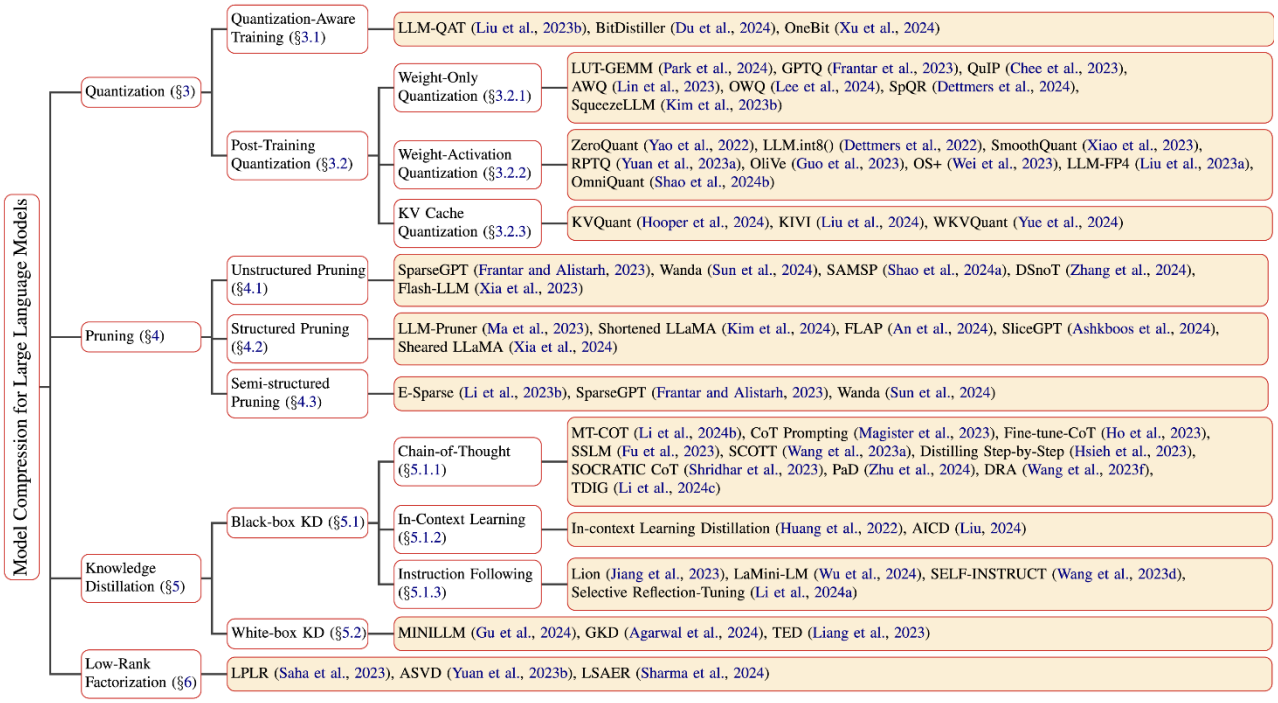


Figure 2: Taxonomy of model compression methods for large language models from (Zhu, Li, Liu, & Ma, 2024)

As a starting point for covering the wide range of compression methods for large language models (LLMs), I base my review on the taxonomy provided in (Zhu, Li, Liu, & Ma, 2024), which offers a representative classification of current compression techniques. I have slightly adapted this taxonomy to reflect recent developments in the field.

For each category, I describe at least one compression method, with the choice guided by the relevance and importance of the method in the research literature. The results presented correspond to those reported in the original papers introducing these compression methods. For the selection of evaluation metrics, I primarily refer to performance on WikiText-2 perplexity, given its widespread use in language modeling. If this latter metric was not available, I report the metric identified as most important by the respective authors. Additionally, information on inference speedup is included only when provided in the original papers.

Quantization

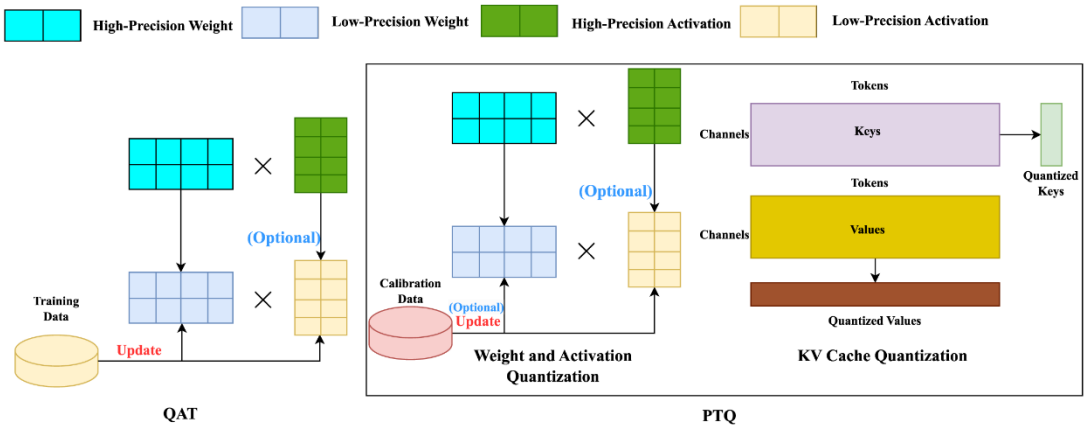


Figure 3: Overview of quantization methods in Transformer networks. The left panel illustrates quantization-aware training (QAT), while the right panel shows post-training quantization (PTQ), including both weight and activation quantization as well as key-value (KV) cache quantization. Different color blocks represent high- and low-precision weights and activations. Figure taken from (Zhu, Li, Liu, & Ma, 2024).

Quantization is a well-established technique that has been applied to various neural network architectures, including convolutional networks, fully connected networks, and, more recently, large language models. The primary objective of quantization is to reduce the bit-width required for numerical representations, transitioning from full-precision formats to lower precision formats such as INT8 or INT4. Recent advances have even achieved binarization, but at a cost of a lower accuracy. This process of quantization can be applied to different components of a neural network, including weights, activations, and the key-value cache, either individually or in combination. (Zhu, Li, Liu, & Ma, 2024; Cheng, Wang, Zhou, & Zhang, 2020)

As described by (Zhu, Li, Liu, & Ma, 2024), quantization methods can be broadly divided into two main sub-categories: **Quantization-Aware Training (QAT)** and **Post-Training Quantization (PTQ)**. QAT methods can be further divided into two subcategories depending on the level of model adaptation involved: one that integrates quantization during full retraining, and another that leverages Parameter-Efficient Fine-Tuning (PEFT) techniques for post-quantization error recovery. PTQ methods, on the other hand, can be classified based on the components being quantized: *Weight-only Quantization*, *Weight-Activation Quantization*, *KV-Cache Quantization*, and *Weight-Activation-KV Cache Quantization*.

Quantization-Aware Training (QAT)

PEFT-Based Error Recovery after Quantization

To address the computational cost associated with retraining large language models, several methods have been proposed by leveraging Parameter-Efficient Fine-Tuning (PEFT) techniques (Zhu, Li, Liu, & Ma, 2024).

***P**arameter-**E**fficient and **Q**uantization-**A**ware Adaptation (**PEQA**) - (Kim, et al., 2023)*

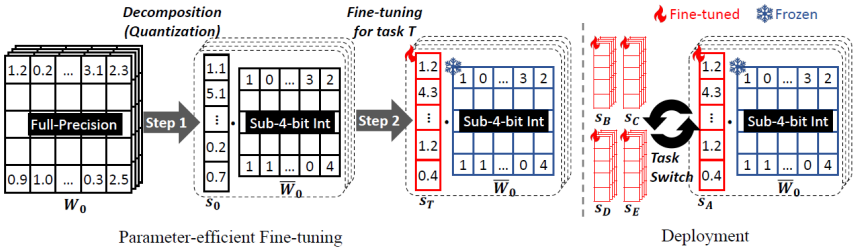


Figure 4: Overview of PEQA. Figure from (Kim, et al., 2023)

In **PEQA**, the parameter matrix is first decomposed into two components: a matrix of low-bit integers and a matrix of quantization scales. During fine-tuning, the low-bit integer matrix remains frozen, while only the quantization scale matrix is updated for each downstream task.

Method	Weights Bits	**Wikitext-2 Perplexity**
		LLaMA 65B
LoRA	16	3.82
LoRA + OPTQ	4	4.1
PEQA	4	**4.02**
LoRA + OPTQ	3	5.32
PEQA	3	**4.27**

Table 1: Main results of PEQA. Lower is better.

Using the PEQA method, the perplexity on WikiText-2 remains very close to the baseline, where the baseline is defined as LoRA with 16-bit weights. While PEQA results in slightly higher perplexity than this baseline, it still outperforms the approach that applies OPTQ quantization to LoRA weights.

***Q**uantization **Lo**w **R**ank **A**daptation (**QLoRA**) - (Dettmers, Pagnoni, Holtzman, & Zettlemoyer, 2023)*

The QLoRA method achieves results that match or surpass those of larger models, while requiring fewer resources by fine-tuning quantized models.

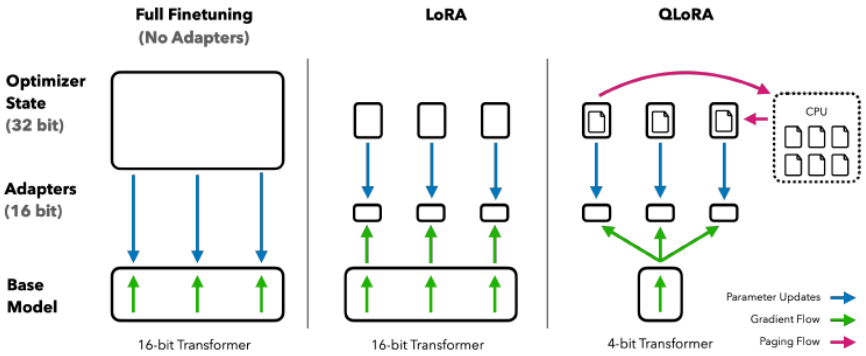


Figure 5: Comparison of finetuning methods. LoRA reduces memory by using adapters, while QLoRA combines adapters with 4-bit quantization for maximal efficiency. Figure from (Dettmers, Pagnoni, Holtzman, & Zettlemoyer, 2023)

The authors introduce three key innovations: 4-bit NormalFloat (NF4), a data type optimized for normally distributed weights; double quantization, which compresses both weights and quantization constants; and paged optimizers, which reduce memory spikes using NVIDIA's unified memory. The method quantizes a pre-trained model to 4-bit precision and adds trainable low-rank adapter weights (LoRA) on top of frozen quantized weights. During training, gradients are backpropagated through the quantized model, allowing the adapters to learn effectively in context without updating the full model. This enables efficient fine-tuning while minimizing accuracy loss.

Model	Weights Bits	ChatGPT vs Sys	Sys vs ChatGPT	Mean
GPT-4		119.40%	110.10%	114.50%
Guacano 65B	4	96.70%	110.90%	99.30%

Table 2: Main results of QLoRA. Zero-shot Vicuna benchmark scores (as a percentage of ChatGPT, evaluated by GPT-4) for models finetuned on OASST1.

The Vicuna benchmark assesses open domain chatbot models with a diverse set of prompts. Guanaco, a family of models finetuned with QLoRA on a variant of the OASST1 dataset, is designed for efficient instruction tuning with limited computational resources.

Model performance is measured in a zero-shot setting, where GPT-4 is used to compare each model's responses against those of ChatGPT. To control for potential order bias in automated evaluation, results are reported both when ChatGPT's answer is presented first ("ChatGPT vs Sys") and when the system's answer is presented first ("Sys vs ChatGPT"); the final score is averaged across both scenarios. Guanaco models achieve scores that closely match ChatGPT, with Guanaco 65B reaching 99.3% of ChatGPT's performance, despite using significantly less memory and training data. These findings highlight the efficiency and competitiveness of QLoRA finetuning for conversational agents.

A similar strategy to QLoRA, was later adopted by Meta for the lightweight versions of the LLaMA 3.2 models (1B and 3B parameters), by also combining Quantization-Aware Training (QAT) with LoRA adaptation Starting from BF16 checkpoints obtained after supervised fine-tuning, Meta performed an additional round of QAT to simulate low-precision behaviour during training. This step allowed the model to adapt to quantization before freezing the backbone. A second round of supervised fine-tuning was then conducted using LoRA adapters, applied to all layers within the transformer block. Finally, both the backbone and adapters were jointly fine-tuned using Direct Preference Optimization (DPO), an alignment technique designed to make model outputs more consistent with human

preferences. This staged approach helps mitigate the quantization-induced performance drop and supports efficient deployment of small, high-quality models. (Meta AI, 2025)

LoRA-Fine-Tuning-aware Quantization (LoftQ) - (Li, et al., 2023)

Similarly to QLoRA, the LoftQ method addresses the issue of quantization discrepancy by explicitly incorporating the LoRA fine-tuning objective into the quantization process itself.

Algorithm 1 LoftQ

input Pre-trained weight W, target rank r, N-bit quantization function $q_N(\cdot)$, alternating step T

Initialize $A_0 \leftarrow 0, B_0 \leftarrow 0$

for t = 1 to T **do**

  Obtain quantized weight $Q_t \leftarrow q_N(W - A_{t-1}B_{t-1}^\top)$

  Obtain low-rank approximation $A_t, B_t \leftarrow \text{SVD}(W - Q_t)$ by (9)

end for

output Q_T, A_T, B_T

Figure 6: LoftQ – Iterative joint low-rank adaptation and N-bit quantization for pre-trained weights. Figure from (Li, et al., 2023)

It approximates the original pre-trained weight matrix using three components: Q, an N-bit quantized matrix, and A and B, two low-rank matrices forming the approximation AB^T.

These components are obtained through an alternating optimization procedure consisting of two main steps: quantization and singular value decomposition (SVD).

First, the quantized matrix Q is derived by quantizing the residual between the original weights and the current low-rank approximation AB^T. Then, SVD is applied to the updated residual to refine the low-rank matrices A and B By optimizing this decomposition jointly, LoftQ provides a more robust initialization for LoRA-based fine-tuning, which has shown improved performance, especially under extreme quantization settings, such as 2-bit precision.

Method	Weights Bits	**Wikitext-2 Perplexity**	
		LLAMA-2-7b	LLAMA-2-13b
LoRA	16	5.08	5.12
QLoRA	4	5.7	5.22
LoftQ	4	5.24	5.16
QLoRA	3	5.73	5.22
LoftQ	3	5.63	5.13
QLoRA	2	N.A	N.A
LoftQ	2	7.85	7.69

Table 3: Main results for LoftQ. Lower is better. N.A indicates values that are not available.

Across all bit-widths, LoftQ consistently achieves lower perplexity than QLoRA and LoRA. This demonstrates that LoftQ provides better language modeling performance regardless of the quantization level, while QLoRA lags behind.

While QLoRA applies LoRA after quantization without modifying the quantized weights, both Meta's and LoftQ's approaches emphasize preparing the quantized backbone to support effective fine-tuning.

Quantization-Aware Low-Rank Adaptation (QA-LoRA) - (Xu, et al., 2023)

Although the previously mentioned methods significantly reduce the memory footprint required during QAT, they do not inherently provide improvements during inference. Specifically, the weights must still be dequantized back to their higher-precision formats before inference, limiting the overall efficiency gains. QA-LoRA addresses this limitation by proposing a training framework that enables effective inference directly on quantized weights, thus eliminating the need for costly dequantization steps and further enhancing both memory efficiency and inference speed.

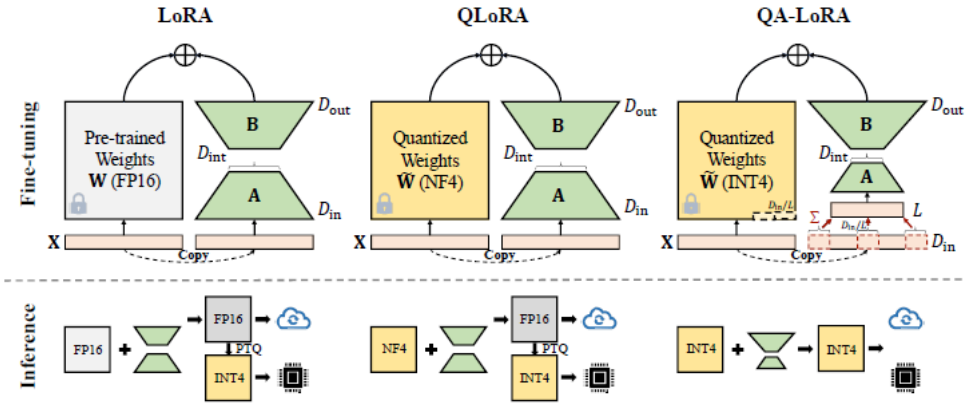


Figure 7: Compared to LoRA and QLoRA, QA-LoRA allows inference on quantized weights. Figure from (Xu, et al., 2023)

QA-LoRA addresses this limitation by merging the quantized weights with the low-rank adaptation AB^T. A key insight is that simply adding the low-rank adapters $s \cdot AB^T$ to the quantized weights $\widetilde{W}$ can break the uniform arithmetic structure essential for quantization. Typically, a single pair of scaling and zero-point factors (α_j, β_j) is applied to each column j, so all elements w_{ij} must fit into a consistent quantized range. Incorporating a fully flexible AB^T can violate that requirement unless A is overly constrained (e.g., all rows identical), which severely limits adaptation capacity.

To overcome this, QA-LoRA employs group-wise quantization, partitioning each column of W into multiple groups. Each group has its own scaling and zero-point factors, allowing different segments of the column to be quantized independently. Correspondingly, only the rows of A that lie within the same group need to share identical values, preserving a higher effective rank and thus maintaining strong adaptation capabilities.

Practically, QA-LoRA implements a parameter-free average pooling on the input vector, reducing its dimension from D_{in} to L, which becomes the number of quantization groups per column. This approach balances the parameter budgets for quantization and adaptation, allowing QA-LoRA to achieve better accuracy while avoiding dequantization overhead, thereby improving both speed and memory efficiency at inference.

Method	Weights Bits	**MMLU (0-shot)**
		LLaMA-65B
QLoRA	4 + 16	62.8
QLoRA + OPTQ	4	62.3
QA-LoRA	4	65.1
QLoRA + OPTQ	3	61.5
QA-LoRA	3	61.2
QLoRA + OPTQ	2	51.4
QA-LoRA	2	57.1

Table 4: Main Results for QA-LoRA. Higher is better.

QA-LoRA achieves higher accuracy than QLoRA and QLoRA + OPTQ at 4 bits and 2 bits, including when QLoRA uses full-precision adapters. At 3 bits, QLoRA + OPTQ slightly outperforms QA-LoRA, but the difference remains small.

Post-Training Quantization (PTQ)

While the Quantization-Aware Training (QAT) methods discussed above help to reduce the computational resources required for retraining, they can still pose challenges for practitioners with limited resources. Post-Training Quantization (PTQ) methods, on the other hand, quantize models without retraining, thereby demanding fewer resources and significantly lowering the overall computational cost (Zhu, Li, Liu, & Ma, 2024).

Weight-only Quantization

Weight-only quantization methods target only the quantization of model weights, an approach that is generally less sensitive and therefore easier to implement than quantizing activations. However, not all hardware supports mixed-precision matrix multiplications; consequently, weights must often be dequantized prior to multiplication with higher-precision activations. This requirement can introduce additional computational overhead, partially offsetting the benefits of weight-only quantization (Zhu, Li, Liu, & Ma, 2024).

OPTQ - (Frantar, Ashkboos, Hoefler, & Alistarh, 2023)

Previously referred as GPTQ, OPTQ is a one-shot weight-only quantization method capable of efficiently quantizing models with more than 100 billion parameters while incurring only minimal performance loss. This approach can shorten inference time by up to 4.53 ×. For instance, they successfully quantized GPT-3 175B to 3- and 4-bit precision using a single NVIDIA A100 GPU (80 GB) in roughly four hours.

Algorithm 1 Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ *// quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ *// Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ *// quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ *// quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update all remaining weights*
end for

Figure 8: Pseudo Code of OPTQ. Figure from (Frantar, Ashkboos, Hoefler, & Alistarh, 2023)

In order to achieve such results, the authors base their approach on the Optimal Brain Quantization (OBQ) method, which was originally designed to solve the layer-wise quantization problem for smaller models (e.g., ResNet-50 with ~50M parameters). OBQ selects the order of weight quantization greedily, quantizing at each step the weight that leads to the least quantization error. This results in a different quantization order for each row, which becomes computationally expensive and does not scale to models with billions of parameters.

$$argmin_{\widehat{W}} = \left\|WX - \widehat{W}X\right\|_2^2$$

Equation 7: The layer-wise quantization problem.

$$w_q = argmin_{w_q} \frac{\left(quant(w_q) - w_q\right)^2}{[H_F^{-1}]_{qq}}$$

Equation 8: w_q being the greedy-optimal weight to quantize next.

$$\delta_F = -\frac{w_q - quant(w_q)}{[H_F^{-1}]_{qq}} \cdot (H_F^{-1})_{:,q}.$$

Equation 9: δ_F being the corresponding optimal update of all weights in F.

$$H_{-q}^{-1} = \left(H^{-1} - \frac{1}{[H^{-1}]_{qq}} H_{:,q}^{-1} H_{q,:}^{-1}\right)_{-p}$$

Equation 10: H_{-q}^{-1}, the updated inverse of the Hessian matrix using one-step Gaussian elimination by removing the q^{th} column and row of H

where:

- $H_F = 2X_F X^T$, the Hessian matrix of the quantization problem and F the set of remaining full-precision weights.
- $quant(w_q)$ rounds w_q to the nearest value on the quantization grid.

Conversely, OPTQ simplifies this process by using a fixed, arbitrarily chosen weight order that is shared across all rows, significantly reducing the computational burden. Specifically, the time complexity is reduced from $O\left(d_{row} \cdot d_{col}^3\right)$ for OBQ to $O\left(max\left\{d_{row} \cdot d_{col}^2, d_{col}^3\right\}\right)$ for OPTQ.

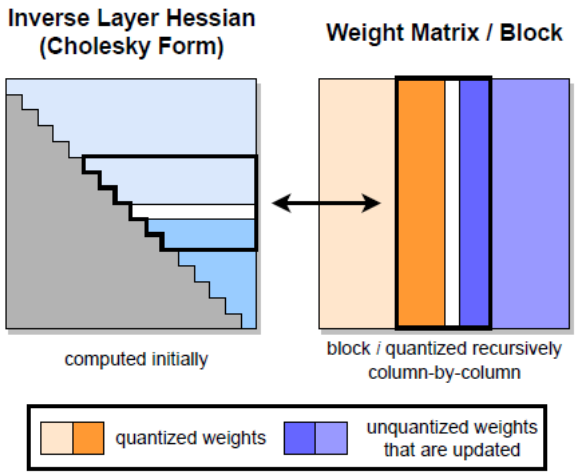


Figure 9: OPTQ procedure. Blocks of consecutive columns are quantized at each step i using the inverse Hessian information, and the remaining weights are updated at the end of each step. The white column is the column being quantized.

To further improve efficiency and fully utilize the compute capacity of modern GPUs, OPTQ introduces Lazy Batch Updates. Since the final rounding decision for each column is independent of others, columns are grouped and processed in blocks, enabling parallel updates and better GPU utilization. This optimization significantly reduces quantization time without compromising accuracy.

Additionally, to address numerical inaccuracies that may cause H_F^{-1} to become indefinite, OPTQ reformulates the update step using a Cholesky-based decomposition. By precomputing only the necessary rows of H^{-1} with stable Cholesky kernels and applying mild dampening, the method ensures numerical stability even on very large models. This reformulation not only improves robustness but also leads to faster quantization. The aforementioned adaptations and improvements result in the following equations:

$$\delta_F = -\left(w_Q - quant(w_Q)\right)\left([H_F^{-1}]_{QQ}\right)^{-1}(H_F^{-1})_{:,Q}$$

Equation 11: Adapted equation for δ_F

$$H_{-Q}^{-1} = \left(H^{-1} - H_{:,Q}^{-1}\left([H^{-1}]_{QQ}\right)^{-1}H_{Q,:}^{-1}\right)_{-Q}$$

Equation 12: Adapted equation for the updated inverse of the Hessian matrix

where Q is the set of indices at a step i

Method	Bit Width			**Wikitext-2 Perplexity**
	Weights	Activations	KV Cache	OPT-175B
full	16	16	16	8.34
LLM.int8()	4	16	16	10.54
OPTQ	4	16	16	8.37
LLM.int8()	3	16	16	7.03E+03
OPTQ	3	16	16	8.68

Table 5: Main results for OPTQ. Lower is better.

OPTQ delivers perplexity scores that are nearly identical to the full 16-bit baseline, both at 4-bit and 3-bit quantization. While LLM.int8() remains reasonable at 4 bits, OPTQ still outperforms it. At 3 bits, LLM.int8() fails completely, whereas OPTQ maintains strong performance.

Model	GPU	Speedup
OPT-175B	1 A100-80GB	3.24

Table 6: Inference speedup for OPTQ quantized OPT-175B on a single GPU

OPTQ enables OPT-175B to run on a single GPU with an inference speedup of 3.24× compared to full-precision inference. This demonstrates that quantization with OPTQ significantly accelerates inference while maintaining model accessibility on limited hardware. However, the inference speedups enabled by OPTQ do not result from reduced computation, but rather from decreased memory movement. Moreover, this method still requires resource-intensive dequantization during inference, limiting its efficiency gains. These limitations are addressed by the next method.

LUT-GEMM - (Park, et al., 2024)

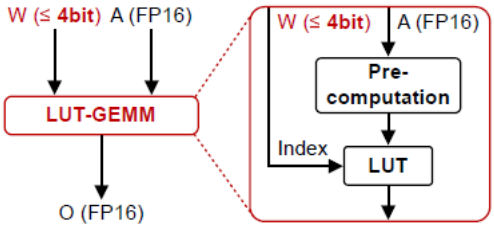


Figure 10: Matrix multiplication scheme for LUT-GEMM: quantized weights (≤ 4 bits) and FP16 activations are multiplied using a precomputed lookup table, producing FP16 outputs. Figure from (Park, et al., 2024)

In (Park, et al., 2024), the authors introduce LUT-GEMM, an efficient kernel for quantized matrix multiplication that not only eliminates the dequantization step but also reduces computational cost compared to previous weight-only quantization methods. Furthermore, they also leverage group-wise quantization, which offers a flexible trade-off between compression ratio and accuracy.

The authors build on Binary-Coding Quantization (BCQ), which approximates symmetrically quantized weights centred around zero using a sum of binary vectors and associated scaling factors, expressed as:

$$\hat{w} = \sum_{i=1}^{q} \alpha_i b_i$$

Equation 13: Binary-Coding Quantization weight representation

where:

- $\hat{w}$ is the approximated weight
- q, the number of bits
- $\alpha_i \in \mathbb{R}^+$is a scaling factor
- $b_i \in \{-1, +1\}^n$ with n the size of the vector w

They extend this method by introducing a bias term, enabling support for asymmetric quantization, which can be expressed as:

$$\hat{w} = \sum_{i=1}^{q} (\alpha_i \cdot b_i) + z$$

Equation 14: Extended BCQ weight representation

where:

- z, a bias term
- $b_i \in [-1, +1]^n$

This adaptation allows the representation of both uniform and non-uniform quantized weights within an extended BCQ format. This extension enables the LUT-GEMM kernel to support both symmetric and asymmetric quantization methods under a unified representation.

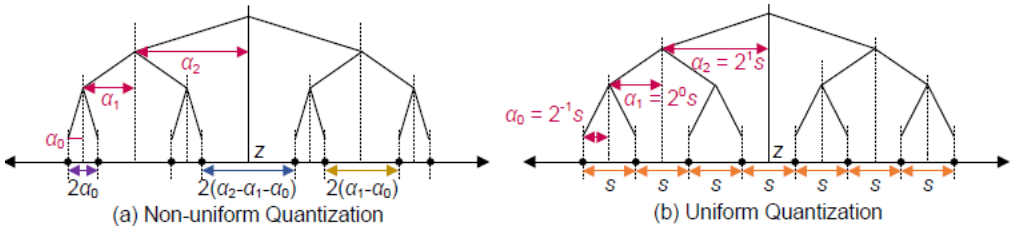


Figure 11: Illustration of non-uniform and uniform quantization schemes. Figure from (Park, et al., 2024).

To further improve efficiency, the authors avoid bit-level memory access by precomputing all possible combinations of full-precision activations and binary patterns in a lookup table (LUT).

Finally, to improve scalability and maintain performance, the method uses a common scaling factor shared across groups of weights of size g, implementing group-wise quantization. This design allows a flexible trade-off between compression ratio, memory footprint, and accuracy.

Model	GPU	Bit Width			Speedup
		Weights	Activations	KV Cache	
LLaMA-65B	1 A100-80GB	4	16	16	2.02

Table 7: Inference speedup for LUT-GEMM on LLaMA-65B (4-bit weights) using a single A100-80GB GPU.

LUT-GEMM enables a 2.02× inference speedup compared to OPTQ for LLaMA-65B with 4-bit quantized weights on a single GPU.

Weight-Activation Quantization

Building on weight-only quantization, Weight-Activation Quantization extends the approach by applying quantization to both weights and activations, resulting in further reductions in model size and memory footprint.

LLM.int8() - (Dettmers, Lewis, Belkada, & Zettlemoyer, 2022)

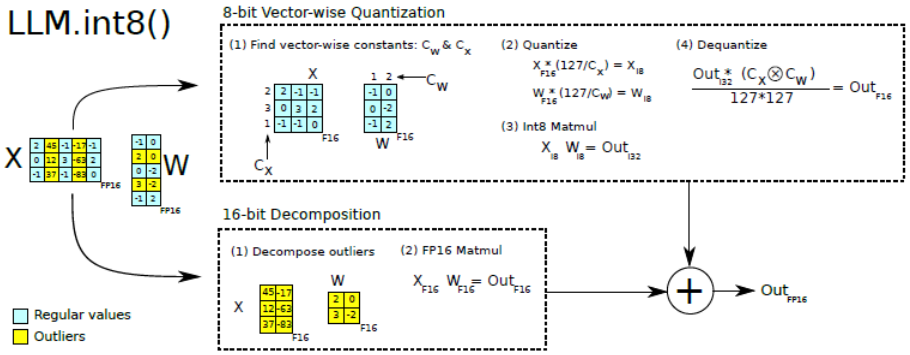


Figure 12: LLM.int8() quantization scheme: most values are quantized to 8 bits, while outliers are processed separately in FP16 for accurate matrix multiplication. Figure from (Dettmers, Lewis, Belkada, & Zettlemoyer, 2022)

LLM.int8() is one of the earliest effective methods for weight–activation quantization in large language models. This method decomposes the weight and activation matrices into sub-matrices, separating them based on magnitude. Specifically, it isolates outlier features, rare but high-magnitude activation values and pairs them with a corresponding sub-matrix of weights. The remaining features, with lower magnitudes, are grouped separately. This results in two parallel matrix multiplication paths: one operating on high-precision outliers, and the other on low-bit quantized values.

The key insight of the work is the observation that outlier features begin to consistently emerge in models of 6.7B parameters and above, and naively quantizing these outliers leads to a significant degradation in performance. By treating outliers separately, LLM.int8() preserves model accuracy while enabling the benefits of activation quantization.

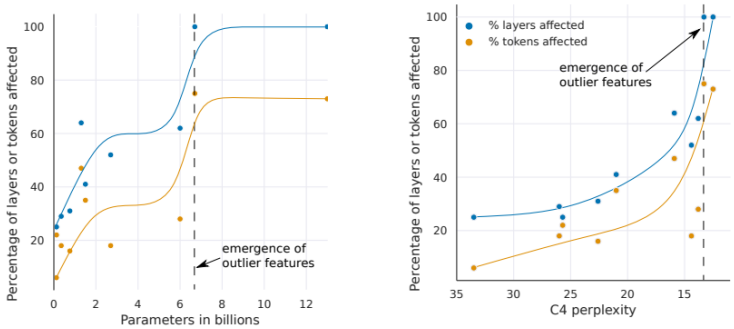


Figure 13: Outlier features emerge at 6.7B parameters. Figure from (Xiao, et al., 2023).

Notably, activation outliers tend to appear in fixed channels, and activations within those channels vary little. In theory, per-channel quantization could effectively address these outliers without significantly impacting performance. However, this approach requires rescaling each individual activation, which is not feasible in practice. Current hardware-accelerated GEMM kernels (e.g., INT8 matrix multiplications) only allow scaling along the outer dimensions of the matrices. Applying different scales across the inner loop breaks parallelism and efficiency (Xiao, et al., 2023).

SmoothQuant - (Xiao, et al., 2023)

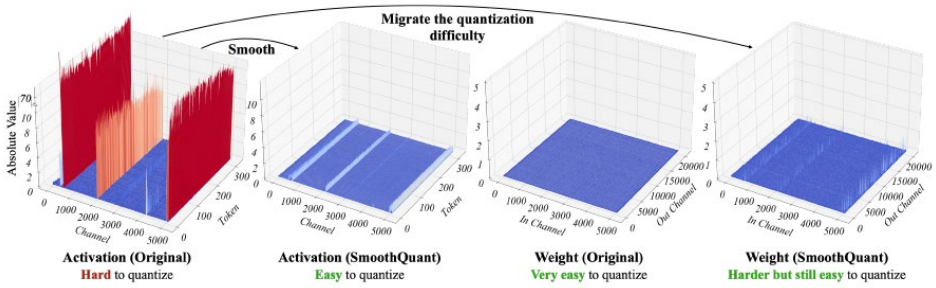


Figure 14:SmoothQuant migrates quantization difficulty from activations to weights, making activations easier and weights only slightly harder to quantize, thereby improving overall quantization efficiency. Figure from (Xiao, et al., 2023)

SmoothQuant was introduced as a solution that avoids altering the computational structure. Instead of applying scaling to activations, this method transfers the quantization difficulty to the weights. This is achieved by dividing the input activations by a per-channel scaling factor while inversely scaling the weights, preserving mathematical equivalence. This scaling factor is defined as:

$$s_j = max(|X_j|)^{\alpha} / max(|W_j|)^{1-\alpha}$$

Equation 15

where:

- α is the hyperparameter that controls the migration strength
- j corresponds to j-th input channel

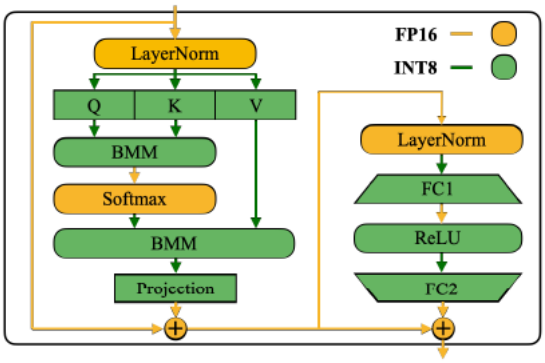


Figure 15: SmoothQuant method applied to a layer. Only LayerNorm and Softmax activations are kept in full precision (FP16), while all other operations are quantized to INT8. Figure from (Xiao, et al., 2023)

In transformer blocks, scale smoothing is applied to the input activations of self-attention and feed-forward layers, and all linear layers are quantized using W8A8. Batched matrix multiplication (BMM) operations within the attention mechanism are also quantized in INT8. Meanwhile, lightweight element-wise operations such as ReLU, Softmax, and LayerNorm retain their activations in FP16.

Method	Bit Width			**Wikitext-2 Perplexity**
	Weights	Activations	KV Cache	OPT-175B
Full	16	16	16	10.99
SmoothQuant-O3	8	8	16	11.17

Table 8: Main results for SmoothQuant. Lower is better.

SmoothQuant and its O3 variant achieve perplexity scores on Wikitext-2 that are nearly identical to the full-precision baseline, indicating minimal performance loss despite quantization of both weights and activations.

KV Cache Quantization

While quantizing both weights and activations can significantly reduce memory usage and improve inference speed, the Key-Value (KV) cache remains a major bottleneck when processing long input sequences, particularly as modern language models adopt increasingly long context windows. As the KV cache grows proportionally with sequence length, its memory footprint can dominate runtime costs. Standard 8-bit quantization methods are insufficient to address this challenge. Therefore, methods specifically designed to quantize the KV cache at lower bit-widths (sub 4-bits) are becoming increasingly important for enabling efficient long-context inference without sacrificing performance (Zhu, Li, Liu, & Ma, 2024) (Hooper, et al., 2024).

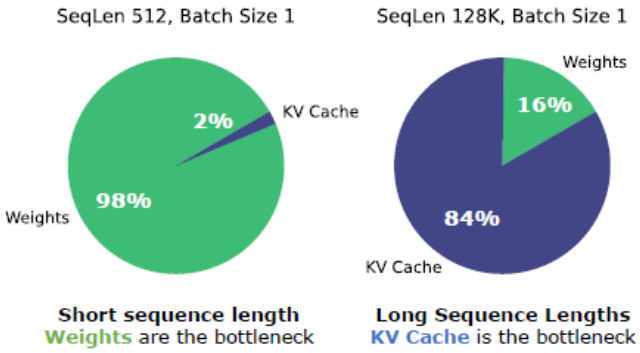


Figure 16: Memory bottleneck shifts from weights at short sequence lengths to KV cache at long sequence lengths. Figure from (Hooper, et al., 2024).

KVQuant - (Hooper, et al., 2024)

In (Hooper, et al., 2024), the authors introduce KVQuant, a method designed to maintain performance while quantizing the KV cache to sub-4-bit precision. KVQuant combines four complementary techniques to improve accuracy and efficiency under aggressive compression.

First, they apply per-channel Key quantization, based on the observation that Key outliers tend to persist within specific channels and vary little across tokens. This allows for effective scaling per channel. In contrast, since Value outliers vary both across tokens and channels, per-token quantization is retained for Values.

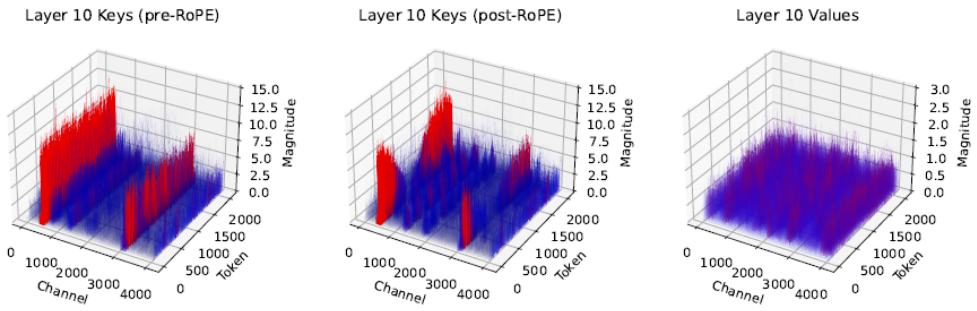


Figure 17: Distribution of Keys and Values magnitudes: pre-RoPE, post-RoPE, and values. Figure from (Hooper, et al., 2024)

Second, they propose Pre-RoPE Key quantization. Applying quantization after Rotary Positional Embeddings (RoPE) mixes channels and disrupts outlier structure, making quantization less effective. By quantizing Keys before RoPE and applying RoPE post-dequantization via a fused kernel, the original channel-wise structure is preserved.

Third, to address the non-uniform distribution of activations, they introduce nuqX, an X-bit per-layer sensitivity-weighted non-uniform datatype. This datatype is computed offline using a calibration set, minimizing quantization error weighted by sensitivity from the Fisher Information Matrix. Activations are normalized to a common range before signpost derivation, enabling per-vector quantization at inference. The signposts are then rescaled per-channel (Keys) or per-token (Values) to match runtime activation ranges.

Fourth, they implement per-vector dense-and-sparse quantization, isolating outlier values within each vector and storing them in full precision. The remaining dense portion is quantized using the nuqX datatype, tightening the quantization range and improving accuracy.

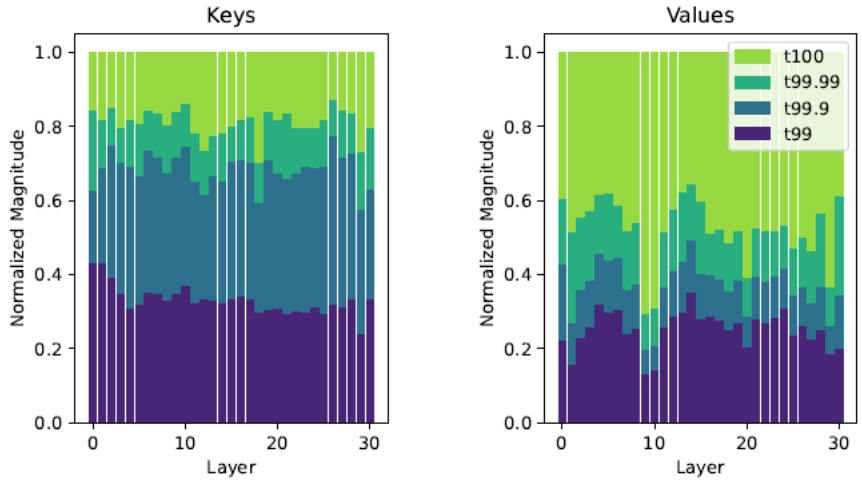


Figure 18: Distribution of normalized key and value magnitudes across layers; only a small fraction of values contributes significantly to the overall magnitude, motivating dense-and-sparse per-vector quantization. Figure from (Hooper, et al., 2024)

Finally, to account for the attention sink effect, where the first token consistently receives high attention scores, they exclude it during nuqX calibration to avoid skewing the quantization statistics.

These techniques are supported by custom GPU kernels for 4-bit quantization, sparse outlier handling, and efficient matrix–vector operations. Quantized KV entries are stored as 4-bit indices into lookup tables, with sparse values stored in Compressed-Sparse Row or Compressed-Sparse Column format. RoPE is applied on-the-fly within the Key matrix kernel, maintaining compatibility with pre-RoPE quantization and ensuring fast, memory-efficient inference.

Method	Bit Width			**Wikitext-2 Perplexity**
	Weights	Activations	KV Cache	LLaMA-65B
Full	16	16	16	3.53
KVQuant-1%	16	16	4	3.54
	16	16	3	3.57
	16	16	2	3.7

Table 9: Main results for KVQuant. Lower is better.

With KVQuant-1%, removing the largest 1% of outliers allows the KV cache to be quantized down to 2 bits with minimal loss in perplexity compared to the full-precision baseline, demonstrating the effectiveness of this approach for aggressive quantization.

Weight-Activation-KV Cache Quantization

Whereas earlier schemes methods quantize only one or two components of a large language model (e g., weights alone, or weights + activations), Weight–Activation–KV-Cache (WAK) Quantization targets all three memory-dominant elements simultaneously. By unifying the memory savings of weight quantization, the compute-speed benefits of activation quantization, and the context-length scalability gained from KV-cache quantization.

SpinQuant - (Liu, et al., 2024)

Prior methods aiming for compressing all three memory components used random rotations to reduce quantization error, but in (Liu, et al., 2024), they find that performance can vary significantly depending on the choice of rotation matrices. This observation motivates the optimization of rotation matrices to directly minimize the final loss of the quantized network.

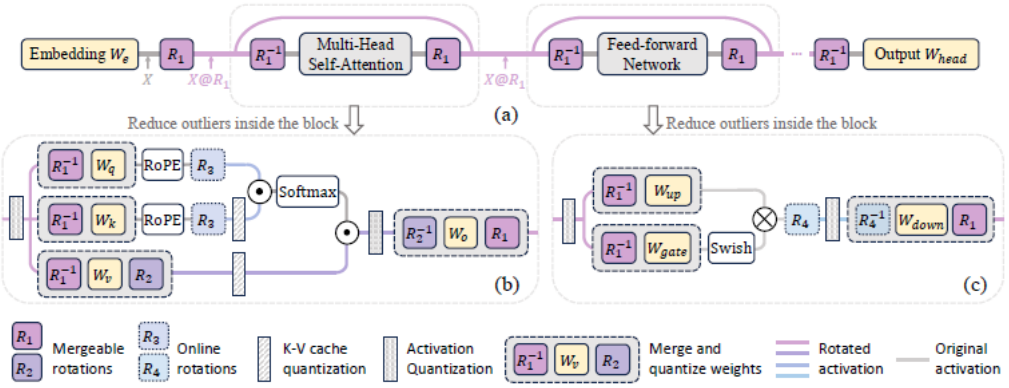


Figure 19: Overview of the SpinQuant framework. Figure from (Liu, et al., 2024)

To achieve this, (Liu, et al., 2024) propose the SpinQuant framework which introduces two strategies: $SpinQuant_{no\ had}$ and $SpinQuant_{had}$.

The first, $SpinQuant_{no\ had}$, involves two learned rotation matrices, R_1 and R_2.

- R_1 rotates the residual activations before they enter transformer blocks, and its inverse is applied afterward, ensuring that the full-precision output remains unchanged.
- R_2 is applied within the attention block to rotate the value matrix and the activations entering the output projection, followed by a head-wise inverse rotation R_2^T to preserve functional equivalence.

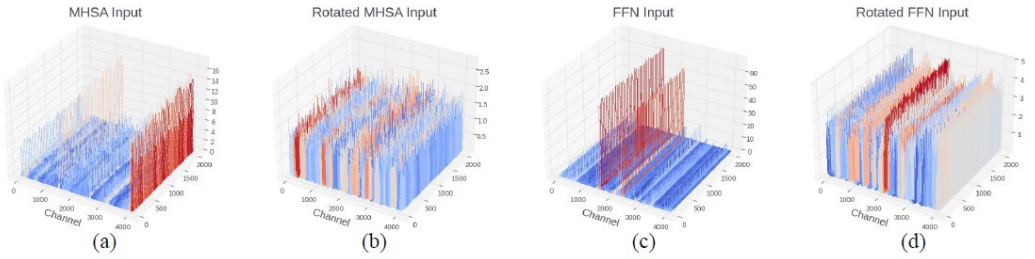


Figure 20: Distribution of MHSA and FFN activations before and after input rotation, illustrating the smoothing effect of rotation on activation values. Figure from (Liu, et al., 2024)

These rotations can be merged directly into the model's weight matrices, as they are orthonormal and applied symmetrically before and after linear operations. This preserves the full-precision model behavior and allows efficient inference without runtime changes. $SpinQuant_{no\ had}$, demonstrates strong performance, particularly in the W4A8 setting.

To further reduce activation bit-widths to 4 bits and support KV cache quantization, the authors introduce a second strategy: $SpinQuant_{had}$. This variant incorporates two additional rotations, R_3 and R_4, implemented using Hadamard transforms.

- R_3 is inserted before KV cache quantization to smooth outliers and improve low-bit accuracy.
- R_4 is applied when reducing activations to sub-4-bit precision, enhancing outlier suppression.

Hadamard matrices are orthonormal and composed of ±1 entries, allowing them to be computed efficiently online via the Fast Hadamard Transform, with only marginal overhead to inference latency.

Unlike R_3 and R_4, which are applied online, R_1 and R_2 are optimized offline using the following quantization-aware objective:

$$\arg\min_{R \in \mathcal{M}} \mathcal{L}_Q(R_1, R_2 | W, X)$$

Equation 16

where R_1 and R_2 lie on the Stiefel manifold, the space of all orthonormal matrices. To solve this, the authors apply the Cayley SGD method, a specialized optimization algorithm that preserves orthonormality throughout training. Importantly, this optimization is done while the underlying weight parameters remain frozen. The rotation matrices $\{R_1, R_2\}$ account for only ~0.26% of the total weight size, and are strictly constrained to be orthonormal. As a result, the original floating-point network remains unchanged, and the rotations influence only the quantization performance, not the functional output of the model.

In their setup, R_1 and R_2 are optimized with respect to the activation-quantized network, while the weights remain in full-precision (16-bit) during optimization. Once the optimal rotations are found and absorbed into the weights, the rotated weights are quantized using OPTQ, combining SpinQuant's rotation-based activation smoothing with OPTQ's weight quantization technique.

Method	Bit Width			**Wikitext-2 Perplexity**
	Weights	Activations	KV Cache	LLaMA-70B
Full	16	16	16	3.3
SpinQuant (no had)	4	8	16	3.7
SpinQuant (had)	4	8	16	3.5
SpinQuant (no had)	4	8	8	3.7
SpinQuant (had)	4	8	8	3.5
SpinQuant (no had)	4	4	4	7.4
SpinQuant (had)	4	4	4	3.8

Table 10: Main results for SpinQuant. Lower is better.

SpinQuant maintains very low perplexity on Wikitext-2, with minimal difference compared to the full-precision baseline. The Hadamard variant consistently outperforms the strategy without it, and this benefit is especially pronounced when all components (weights, activations, KVCache) are quantized to 4 bits.

Model	Hardware	Speedup
LLaMA-3 8B	MAC Book Pro CPU	3

Table 11:Inference speedup for LLaMA-3 8B quantized (W4A8) on a MacBook Pro CPU.

Quantizing LLaMA-3 8B to W4A8 yields a 3× speedup compared to the full-precision model on a standard MacBook Pro CPU, demonstrating significant efficiency gains on consumer hardware.

Pruning

In comparison with quantization, pruning aims to reduce the size or complexity of a model by removing entire redundant components. Pruning methods are divided into three distinct categories: unstructured pruning, structured pruning, and semi-structured pruning (Zhu, Li, Liu, & Ma, 2024).

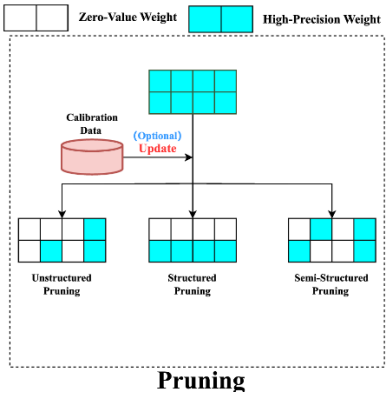


Figure 21: Overview of the pruning methods. Figure from (Zhu, Li, Liu, & Ma, 2024)

Unstructured Pruning

Unstructured pruning involves the removal of individual weights from the model. This results in a sparse and irregular structure that enables fine-grained elimination of redundant parameters, often without compromising performance. However, the irregularity introduced by this method requires specialized processing or software optimizations to achieve efficient inference (Zhu, Li, Liu, & Ma, 2024).

SparseGPT - (Frantar & Alistarh, SparseGPT, 2023)

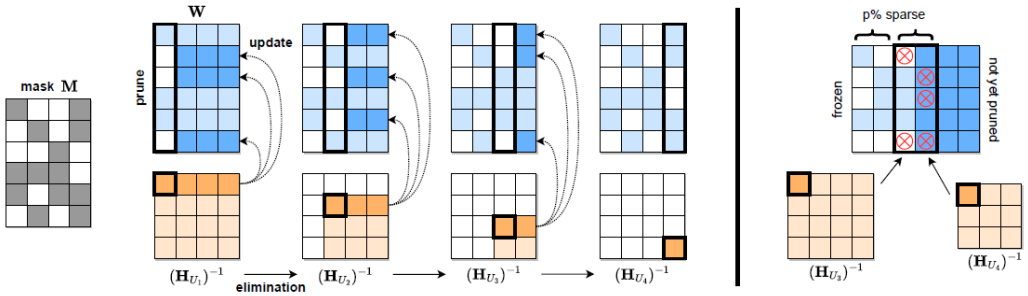


Figure 22: Overview of the SparseGPT method. Figure from (Frantar & Alistarh, SparseGPT, 2023)

SparseGPT is a layer-wise pruning method designed to scale to extremely large language models without requiring retraining. Traditional approaches formulate pruning as a joint optimization over a pruning mask and reconstructed weights. Since this problem is NP-hard, prior methods approximate it by selecting a mask (e.g., using weight magnitude) and applying least-squares reconstruction, an approach that becomes infeasible at GPT-scale.

To overcome this, SparseGPT introduces a fast approximate reconstruction method. Instead of computing a separate inverse Hessian per row (i.e. $O(d_{col}^3)$ for all rows), it shares a common inverse Hessian across all rows (i.e. $O(d_{col}^2)$). This is achieved by synchronizing pruning decisions through a sequence of subsets U_j, allowing all necessary inverse Hessians to be generated recursively with just one initial matrix inversion.

$$U_{j+1} = U_j - \{ j \}$$

Equation 17: U_j, the sequence of subsets being defined recursively

with $U_1 = \{1, \dots, d_{col}\}$

SparseGPT builds on the Optimal Brain Surgery (OBS) framework to provide optimal updates for remaining weights after pruning. To reduce computation further, it restricts updates to a subset U included in M, preserving optimality within that subset while drastically lowering the cost.

To avoid non-uniform sparsity and over-pruning due to outlier weights in large language models, adaptive mask selection is used. SparseGPT employs iterative blocking, updating weights block by block while refining the mask based on error impact.

The method is also easily extended to semi-structured sparsity (e.g., 2:4 or 4:8 patterns), by setting the block size to match the hardware pattern and selecting the best n weights per row. This allows SparseGPT to benefit from optimized hardware execution, such as on NVIDIA Ampere GPUs.

To further enhance performance, SparseGPT is compatible with the OPTQ greedy framework, where weights are compressed column by column and frozen after update. This design allows for seamless integration of quantization, with quantized weights fixed during pruning and the associated quantization error included in subsequent compensation steps. The result is a unified, efficient algorithm for joint sparsification and quantization, delivering substantial compression with minimal loss in model quality.

Method	Compression Rate	**Wikitext-2 Perplexity**
		OPT-175B
SparseGPT	Dense	8.35
	50%	8.21
	4:8	8.45
	2:4	8.74

Table 12: Main results for SparseGPT. Lower is better.

SparseGPT achieves minimal perplexity degradation on Wikitext-2. The 50% sparsity setting (removing half of the weights) performs slightly better than the dense baseline, while semi-structured sparsity (4:8, 2:4) leads to only a modest increase in perplexity.

Structured Pruning

Contrary to unstructured pruning, structured pruning removes entire components of the model. As a result, it does not require specific handling of the resulting model and is generally more compatible with standard hardware. However, it can lead to worse performance degradation than unstructured pruning. Structured pruning can be divided into *three types* based on the pruning metrics used to identify and remove redundant components (Zhu, Li, Liu, & Ma, 2024).

Loss-Based Pruning

This type of structured pruning removes units based on whether their removal has an impact on the loss or its gradient. The decision relies on measuring the sensitivity of the model's performance to the removal of each component (Zhu, Li, Liu, & Ma, 2024)

LLM-Pruner - (Ma & Fang, 2023)

LLM-Pruner is a task-agnostic pruning framework for large language models. It automatically identifies dependent structures within the model and assesses which of them can be pruned without significantly compromising performance. This framework is designed to operate with only a limited amount of data and completes the compression process within a short time.

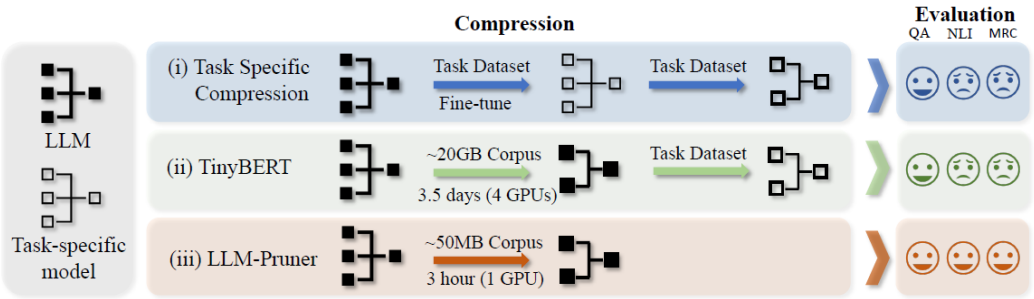


Figure 23: Overview of the LLM-Pruner framework. Figure from (Ma & Fang, 2023)

First, they define dependency between neurons based on their input-output relationships. If a neuron N_j receives input solely from a neuron N_i (or if N_i outputs only to N_j), then N_j is considered dependent on N_i. This simple rule allows the framework to automatically analyse the structure of LLMs without manual intervention.

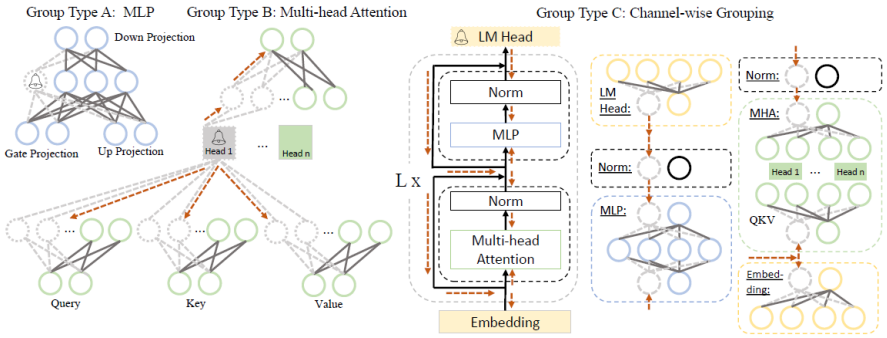


Figure 24: Illustration of the grouped structure in LLaMA. Figure from (Ma & Fang, 2023)

By applying this dependency rule iteratively, LLM-Pruner constructs a dependency graph, where each neuron can trigger the discovery of other dependent neurons. The process continues until no new dependent neurons are detected, forming groups of coupled structures.

To measure the importance of each group, LLM-Pruner evaluates the entire group collectively, in line with the principle of structured pruning (as opposed to unstructured pruning, which removes individual weights). Given that access to the original training data can be limited, they use publicly available datasets or manually created samples to estimate importance.

LLM-Pruner introduces two importance metrics: vector-wise importance (i.e. which estimates the impact at the whole vector level) and element-wise importance, which provides a more fine-grained evaluation at the individual weight level. The importance of each group is computed by aggregating either the vector-wise or element-wise importance scores.

$$I_{W_i} = |\Delta\mathcal{L}(D)| = \left|\mathcal{L}_{W_i}(D) - \mathcal{L}_{W_i=0}(D)\right| = \left|\frac{\partial\mathcal{L}^T(D)}{\partial W_i}W_i - \frac{1}{2}{W_i}^T H W_i + O(\|W_i\|^3)\right|$$

Equation 18: Vector-wise importance

where:

- $D = \{x_i, y_i\}_{i=1}^N$, a given dataset
- $\mathcal{L}$, the next-token prediction loss
- H, the Hessian matrix

They propose four aggregation strategies to compute group importance:

I. Summation of individual importance scores

II. Product of scores

III. Maximum value across scores

IV. Last-only, which corresponds to the importance of the last executed structure in the group (analogous to erasing all previously computed results).

Groups are then ranked according to their importance, and pruning is performed based on a predefined pruning ratio.

The final step in LLM-Pruner is the recovery phase. To enable fast and efficient recovery, they leverage LoRA (Low-Rank Adaptation) to post-train the pruned model.

$$f(x) = (W + \Delta W)X + b = (WX + b) + (PQ)X$$

Equation 19: Recovery equation for LLM-Pruner

where:

- $\Delta W = PQ \in \mathbb{R}^{d^- \times d^+}$ with $P \in \mathbb{R}^{d^- \times d}$ and $Q \in \mathbb{R}^{d \times d^+}$

Importantly, after fine-tuning, the additional parameters P and Q are merged back into W, so that no additional inference latency is introduced. This approach ensures that the final compressed model remains lightweight and efficient for deployment.

Method	Pruning ratio	**Wikitext-2 Perplexity**
		LLaMa-13B
Dense	0%	11.58
Channel	20% without recovery	49.03
Block		16.01
Channel	20% with recovery	17.58
Block		15.18

Table 13: Main results for LLM-Pruner. Lower is better.

LLMPruner introduces a moderate increase in perplexity compared to the dense baseline. While the rise in perplexity is not dramatic, it is still noticeable, highlighting a trade-off between model sparsity and performance. Notably, the 'Channel' strategy only prunes Type C groups, whereas 'Block' strategies prune both Type A and Type B.

Magnitude-Based Pruning

Magnitude-based pruning uses a heuristic metric based on the magnitude of the pruning units. This metric is used to select which units to prune, by identifying those whose values fall below a certain threshold (Zhu, Li, Liu, & Ma, 2024)

SliceGPT (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023)

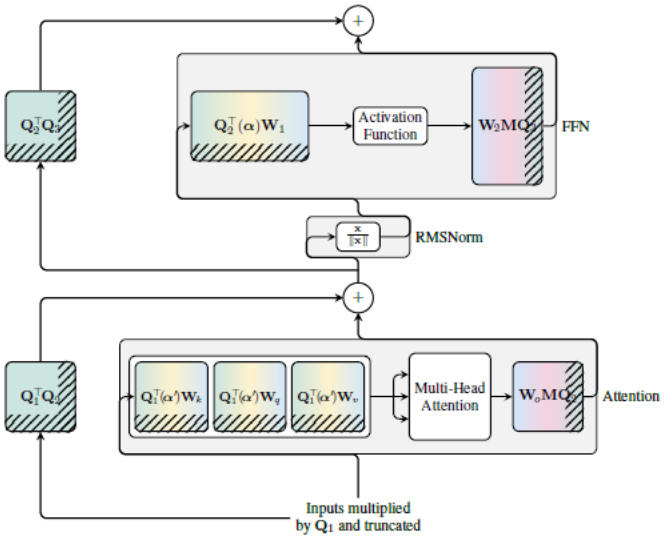


Figure 25: Overview of SliceGPT for a layer in the Transformer Architecture. Figure from (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023)

SliceGPT is a structured pruning method that extends the principle of magnitude pruning to large language models. Instead of removing individual small-magnitude weights, SliceGPT removes entire rows and columns from the weight matrices, based on the relative importance of the corresponding directions in the model's internal representations. This operation directly reduces the embedding dimension of the network, achieving compression and speedup. To enable such structured pruning without altering model behavior, SliceGPT leverages the computational invariance property of the Transformer architecture.

An invariant function is characterized by the property that a transformation of its input does not alter its output. In the case of the Transformer architecture, applying an orthogonal transformation Q to the weights preserves the model's output. This is possible because RMSNorm and orthogonal transformations commute (i.e. applying Q before RMS normalization and then Q^T afterward produces the same result as simply applying RMSNorm alone.

This invariance relies on a key mathematical property: an orthogonal matrix Q satisfies:

$$Q^T Q = Q Q^T = I$$

Equation 20

meaning it preserves the norm of any input vector X. Specifically, multiplying by Q does not change the vector's norm, as $\| QX \| = \| X \|$. Because many nonlinear operations within the transformer, such as RMSNorm, softmax, and activation functions, depend on vector norms or relative scaling, this property ensures that the network's behavior remains consistent even after inserting Q and Q^T.

Moreover, since the main computational blocks of the Transformer, namely the Attention and Feed-Forward layers, consist of linear transformations at their inputs and outputs, the orthogonal transformations can be fully absorbed into the weight matrices of these layers. Although these blocks include nonlinear operations, SliceGPT modifies only the surrounding linear mappings, leaving the

nonlinearities untouched. As a result, the transformations introduced by SliceGPT preserve the overall computation of the model, enabling safe and efficient structured pruning.

While orthogonal transformations can be applied directly when the normalization block is an RMSNorm, this is not immediately possible with LayerNorm.

The main difficulty lies in the mean subtraction step of LayerNorm, which is not preserved under orthogonal transformations. Additionally, LayerNorm normalizes by the standard deviation, while RMSNorm only normalizes by the root mean square.

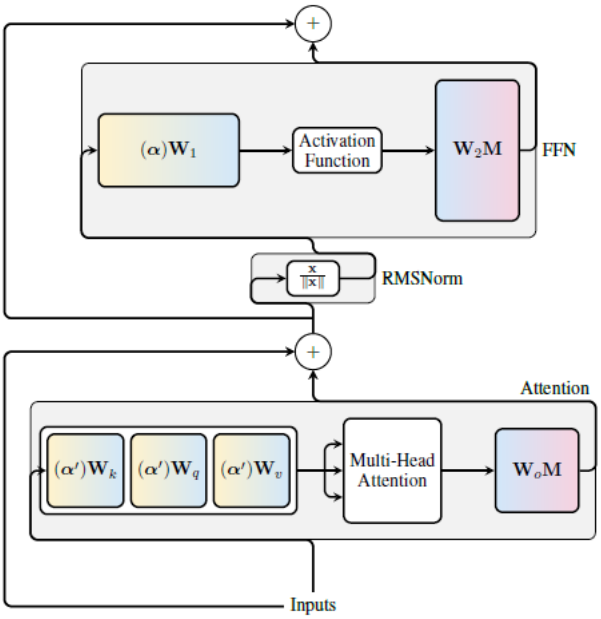


Figure 26: Converting LayerNorm to RMSNorm. Figure from (Ashkboos, Croci, Nascimento, Hoefler, & Hensman, 2023)

To enable orthogonal transformations, SliceGPT converts the network from LayerNorm to RMSNorm by adjusting the surrounding linear operations. The scaling vector γ (denoted as α) from the LayerNorm is applied directly to the input of each block, before the input weight matrices. In the modified structure, matrices such as W_1, W_q, W_k, and W_v are pre-multiplied by the corresponding scaling factors α or α'. Meanwhile, the mean subtraction is handled by post-multiplying the output weight matrices W_2 and W_o with the mean-subtraction matrix M.

After this transformation, the normalization block simplifies to a pure RMSNorm, allowing SliceGPT to safely apply orthogonal transformations across the network. The residual connections and the propagation of scaling between blocks are maintained by consistently applying the scaling adjustments during the transformation.

In order to determine the orthogonal matrix Q_ℓ for each block, SliceGPT relies on Principal Component Analysis (PCA). PCA is a statistical technique used to find the principal directions (principal components) that capture the most variance in a dataset. For each block, SliceGPT collects the activation outputs over a calibration dataset, computes the covariance matrix C_ℓ, and determines the orthogonal matrix as the set of eigenvectors of C_ℓ, sorted by descending eigenvalues. This orthogonal matrix defines a rotation that aligns the block's output signals along the directions of maximum variance.

$$C_\ell = \sum_i X_{\ell,i}^T X_{\ell,i}$$

Equation 21: Covariance matrix for a layer ℓ

Because a different orthogonal matrix C_ℓ is computed for each block ℓ, the residual connections between blocks must also be adapted. SliceGPT addresses this by inserting a corrective linear transformation $Q_{\ell-1}^T Q_\ell$ into the residual paths. This adjustment ensures that the rotated residual matches the rotated main signal, preserving the consistency required by the Transformer's structure.

Once the orthogonal transformations Q_ℓ have been applied, SliceGPT proceeds to the slicing phase. The aim is to reduce the embedding dimension by removing the least important principal components. After PCA, the components are ordered by importance based on the variance they explain. SliceGPT removes the components that contribute the least to the signal's variance.

Instead of explicitly materializing the full signal matrix, SliceGPT modifies the network's weights directly: it deletes rows from the transformed input matrices W_{in} and columns from the output matrices W_{out}, corresponding to the pruned components. The mean-subtraction matrices used in the residual paths are similarly sliced to match the reduced dimensions. This structured deletion reduces the model size and computational complexity while maintaining most of the original model's performance.

Importantly, the final model remains dense and operates with a smaller embedding size, achieving real-world speedups without requiring specialized sparse computation hardware.

Method	Pruning Ratio	**Wikitext-2 Perplexity**
		LLaMA-2 70B
Dense		3.32
SliceGPT	10%	3.69
	20%	4.25
	25%	4.6
	30%	5.05

Table 14: Main results for SliceGPT. Lower is better.

SliceGPT maintains good perplexity scores at moderate pruning ratios, with only limited performance loss. However, as the pruning ratio increases, perplexity rises accordingly.

Model	GPUs	Speedup
LLaMA-2 70B	5 Quadro RTX6000 (24GB)	1.65

Table 15: Inference speedup using SliceGPT

Using enterprise-grade GPUs (5× Quadro RTX6000), LLaMA-2 70B achieves a 1.65× speedup compared to the full-precision model, illustrating substantial efficiency gains in a professional hardware setup.

Regularization-Based Pruning

This type of structured pruning introduces a regularization term into the loss function to promote sparsity. By encouraging certain components to shrink during training, the model naturally learns which units can be removed with minimal impact on performance (Zhu, Li, Liu, & Ma, 2024)

Sheared-LLaMA - (Xia, Gao, Zeng, & Chen, 2024)

Sheared-LLaMA is a series of models, derived by pruning larger pre-trained language models into smaller, efficient target architectures using a method referred to as targeted structured pruning. This process removes entire layers, attention heads, hidden dimensions, and intermediate MLP dimensions, while preserving a regular, dense architecture. The resulting model, called the targeted model, matches the structure of an existing, well-optimized model (e.g., 1.3B or 2.7B), ensuring a good balance between expressiveness and inference efficiency.

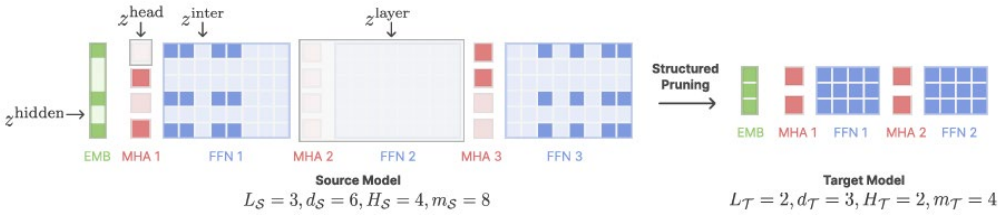


Figure 27: Overview of Shear for LLaMA. The numbers below each model indicate the remaining structural components after structured pruning. Figure from (Xia, Gao, Zeng, & Chen, 2024)

To obtain the pruned model, the authors frame the pruning process as a constrained optimization problem. A set of learned pruning masks are introduced at multiple levels of granularity to control which components (layers, heads, dimensions) are retained. The optimization objective seeks to minimize the model's loss while satisfying architectural constraints that ensure the final model matches the chosen target configuration.

$$\tilde{\mathcal{L}}^{head}(\lambda, \phi, z) = \lambda^{head} \cdot \left(\sum z^{head} - H_{\mathrm{T}}\right) + \phi^{head} \cdot \left(\sum z^{head} - H_T\right)^2$$

Equation 22: Equation to obtain the target number of heads H_T. Others sub-structures are obtained in a similar way.

To make this problem differentiable and suitable for standard gradient-based optimization, the pruning masks are modeled using hard concrete distributions, which allow for smooth approximations of binary decisions.

$$\mathcal{L}_{prune}(\theta, z, \lambda, \phi) = \mathcal{L}(\theta, z) + \sum_{j=1}^{L_S} \tilde{\mathcal{L}}_j^{head} + \sum_{j=1}^{L_S} \tilde{\mathcal{L}}_j^{int} + \tilde{\mathcal{L}}^{layer} + \tilde{\mathcal{L}}^{hidden}$$

Equation 23: min-max objective function to optimize jointly the model weights and pruning masks

This approach is directly inspired by the ℓ_0 regularization approach, which encourages sparsity by penalizing the number of active units, and makes it possible to train both model parameters and pruning decisions jointly.

Once pruning is complete, the model undergoes a continued pre-training phase to recover performance. However, the authors observe that when training on a multi-domain dataset such as RedPajama, different data domains recover at varying rates. Some domains, like GitHub, improve quickly, while others, such as C4, lag behind. This discrepancy arises because the pruning process preserves unequal amounts of information across domains. To address this inefficiency and guide the model more effectively toward balanced recovery, the authors introduce Dynamic Batch Loading.

This method dynamically adjusts the proportion of training data drawn from each domain k at each training step t. For each domain, a reference loss is defined as the target performance level. At regular intervals, the model's validation loss ℓ_t for each domain is measured and compared against the

reference. If a domain's current loss remains above its reference, its sampling weight w is increased; if the domain has already reached or surpassed its target, its weight is reduced.

This adjustment process is applied during both the pruning and recovery phases, enabling the model to continually focus training on domains that require more attention. As a result, Dynamic Batch Loading improves the efficiency of data usage, accelerates convergence toward desired performance levels, and ensures more uniform recovery across heterogeneous domains.

Models	Tokens	**MMLU (5-shot)**
LLaMA2-7B	2T	46.6
Pythia-1.4B	300B	25.7
Sheared-LLaMA-1.3B	50B	25.7
Open-LLaMA-3B-v1	1T	27
Sheared-LLaMA-2.7B	50B	26.4

Table 16: Main results for Sheared-LLaMA. Higher is better.

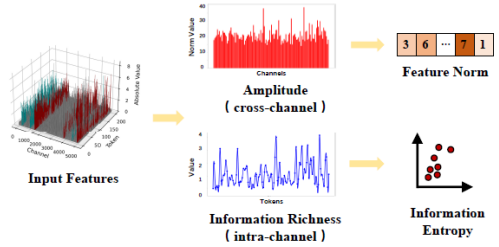
Sheared-LLaMA models achieve comparable accuracy to standard baselines on the MMLU 5-shot benchmark while requiring significantly fewer training tokens. For both 1.3B and 1.4B parameter models, Sheared-LLaMA matches baseline performance with much less data. The same trend holds for larger models (e.g., OpenLLaMA 3B v1 and Sheared-LLaMA 2.7B trained with 50B tokens), where the accuracy difference is negligible, underscoring the efficiency of Sheared-LLaMA in terms of token usage.

Semi-Structured Pruning

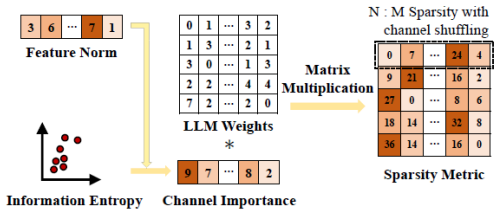
In semi-structured pruning, the goal is to remove parameters based on predefined patterns. This method is typically represented using an N:M format, where N is the number of weights retained, and M is the number of contiguous parameters considered. For example, in a 4:8 pattern, only 4 out of every 8 contiguous parameters are kept (Zhu, Li, Liu, & Ma, 2024).

E-Sparse - (Li, et al., 2024)

E-Sparse is a 2:4 pruning method based on entropy-guided sparsity. The method leverages Sparse Tensor Core technology (i.e. specialized hardware units in modern GPUs designed to efficiently execute sparse matrix operations). By aligning the pruning strategy with this hardware-friendly 2:4 sparsity pattern, E-Sparse significantly reduces inference time while maintaining competitive model accuracy.



(a) Evaluate the input features from both cross-channel and intra-channel dimensions.



(b) The entropy-based sparsity metric of E-Sparse.

Figure 28: Overview of E-Sparse. Figure from (Li, et al., 2024)

$$IR_c = -\sum_{k=1}^{K} p_k^c \log(p_k^c)$$

Equation 24: Information Richness metric

Entropy is used as a metric for information richness, allowing the evaluation of each channel's importance within the hidden feature states of the model. Channels with high entropy are considered to carry more information, while those with low entropy are viewed as less informative. The authors build this method on two key observations.

First, the information richness varies significantly across channels, even when channels have similar activation magnitudes. This makes entropy a more fine-grained and discriminative metric for pruning compared to magnitude alone. Second, channels with similar entropy values, often those with high information content, tend to appear in close proximity. This poses a challenge for semi-structured pruning methods like 2:4 sparsity, which prune weights from small, consecutive groups. To address this issue and avoid pruning multiple informative channels at once, the authors introduce a channel shuffle mechanism, which redistributes channels more evenly across groups.

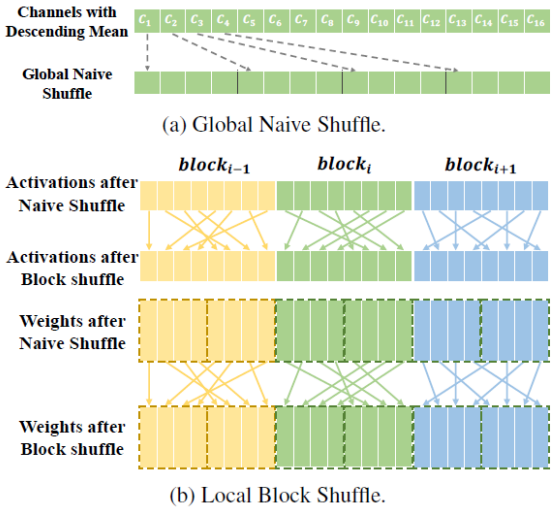


Figure 29: Channel Shuffle of E-Sparse. Figure from (Li, et al., 2024)

The first step is a global naive shuffle, where the mean of the entropy-based importance metric is computed for each channel. The channels are then sorted in descending order, effectively ensuring that channels with similar mean importance are distributed across different sparsity groups.

The second step is the local block shuffle. In this step, the entropy matrix is divided into n blocks, each containing m channels. By setting $m = 256$, the search space becomes tractable, allowing the use of a greedy search algorithm to optimize the channel ordering within each block. This two-level reordering strategy enables E-Sparse to maintain high information diversity across pruning groups, making it well-suited for structured sparsity patterns like 2:4.

As a result, E-Sparse can perform effective pruning without retraining and preserves the remaining weights in their original form.

Method	N:M Sparsity	Wikitext Perplexity
		LLaMA-65B
Dense		3.56
SparseGPT	2:4	6.28
E-Sparse		5.69
SparseGPT	4:8	6.28
E-Sparse		5.69

Table 17: Main results for E-Sparse. Lower is better.

E-Sparse achieves lower perplexity than SparseGPT across all tested settings, indicating more effective structured pruning. Although there is a slight increase in perplexity compared to the dense baseline, the degradation remains minimal.

Knowledge Distillation (KD)

Knowledge Distillation (KD) refers to the process of transferring knowledge from a large, often complex, *teacher model* to a smaller, more efficient *student model*. This approach allows the student model to approximate the performance of the teacher while benefiting from reduced computational requirements. KD methods are generally categorized into two main types based on the nature of the information transferred: Black-Box and White-Box distillation techniques (Zhu, Li, Liu, & Ma, 2024).

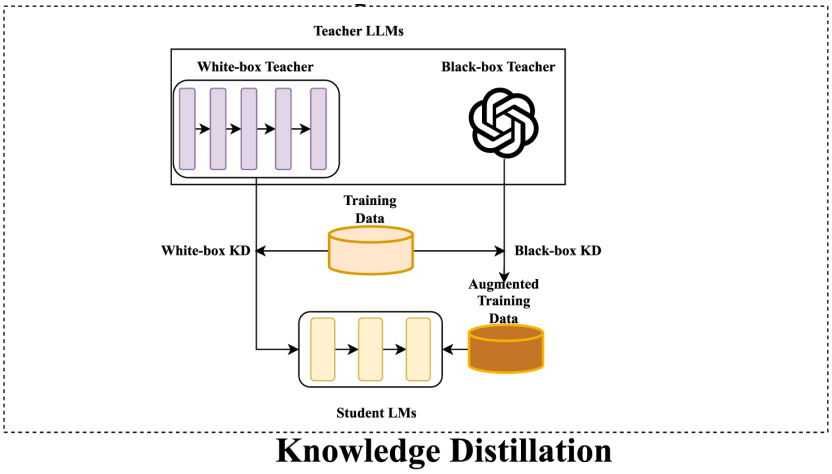


Figure 30: Overview of Knowledge Distillation. Figure from (Zhu, Li, Liu, & Ma, 2024)

Black-box KD

Black-Box Knowledge Distillation involves prompting a teacher model to generate a *distillation dataset*, which is subsequently used to fine-tune the student model. In this setting, only the outputs of the teacher are accessible (i.e. its internal architecture and parameters remain hidden). This approach is particularly well-suited for scenarios involving *closed-source models*, where access to internal representations is restricted, and interaction is limited to input-output behavior.

More specifically, Black-Box Knowledge Distillation methods often aim to transfer *specific and noteworthy capabilities* that typically emerge in models exceeding a certain number of parameters These capabilities, known as emergent abilities, are not explicitly programmed but rather arise during large-scale training. Among the most well-known emergent abilities are Chain-of-Thought reasoning, In-Context Learning, and Instruction Following, which significantly enhance the model's capacity to solve complex tasks, generalize from limited examples, and follow human instructions effectively ((Zhu, Li, Liu, & Ma, 2024).

MT-CoT - (Li, et al.)

MT-CoT is a framework designed to facilitate the generation of a distillation dataset that can be used to transfer knowledge from a teacher model to a student model. The primary goal of MT-CoT is to enable the student model not only to acquire strong reasoning capabilities but also to develop the ability to generate coherent and interpretable explanations. By incorporating chain-of-thought style supervision into a multi-task learning setup, MT-CoT promotes the simultaneous learning of answer prediction and explanation generation.

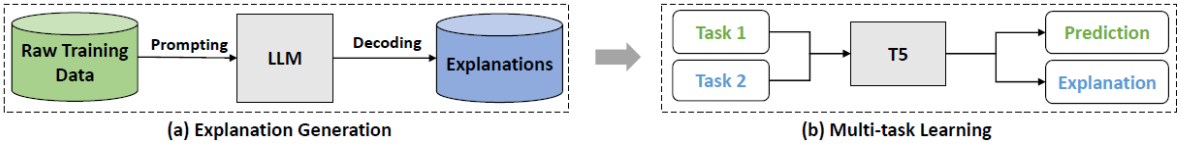


Figure 31: Overview of COT Framework. Figure from (Li, et al.)

To generate the distillation dataset, the authors introduce CROP, a hybrid prompting strategy that combines the strengths of COTE and RP. COTE accepts only explanations where the teacher model's predicted answer matches the ground truth, ensuring high quality but leaving gaps in the dataset. RP, on the other hand, always produces explanations by conditioning on the correct answer, guaranteeing full coverage but without quality filtering.

CROP resolves this trade-off by using COTE as the primary method and falling back on RP when COTE fails. This way, CROP produces a complete set of explanations with a strong emphasis on correctness.

Once the distillation dataset is built using CROP, the MT-CoT framework trains the student model through multi-task learning. It defines two tasks: (1) answer prediction (qta) and (2) explanation generation (qtr), where the model is trained to produce both the correct answer and a chain-of-thought explanation. Unlike earlier approaches (i.e. MT-Re and MT-Ra), MT-CoT explicitly models the causal reasoning process by requiring the model to generate structured reasoning steps *before* producing the answer. This setup helps the student model internalize both *what* the answer is and *why*.

Method	Tasks	
	CSQA	OBQA
GPT-3 Direct Prompting (175B)	80.59	83
GPT-3 Chain of Thought Prompting (175B)	73.71	72.6
GPT-3 Explain. after Answers Prompting (175B)	80.84	83.4
T5-MT (3B)	82.47	78.6

Table 18: Main results for MT-CoT. Results in %. Higher is better.

By training a T5 model with the MT-CoT framework, competitive accuracy is achieved compared to much larger models. Notably, a 3B parameter model with MT-CoT matches the performance of models with 175B parameters on both benchmarks, demonstrating the efficiency of this approach.

White-box KD

Conversely to black-box distillation, which relies solely on the teacher model's outputs, white-box knowledge distillation leverages the internal structure and intermediate representations of the teacher model. By aligning the internal knowledge representations between teacher and student, white-box distillation can provide more fine-grained supervision, potentially leading to more effective knowledge transfer (Zhu, Li, Liu, & Ma, 2024).

While this approach cannot be directly applied to closed-source models, a possible workaround involves using a two-step distillation process: first transferring knowledge from the proprietary teacher model to an intermediary open-source large model and then applying white-box distillation from this intermediary to the final student model (Zhu, Li, Liu, & Ma, 2024).

MiniLLM models - (Gu, Dong, Wei, & Huang, 2023)

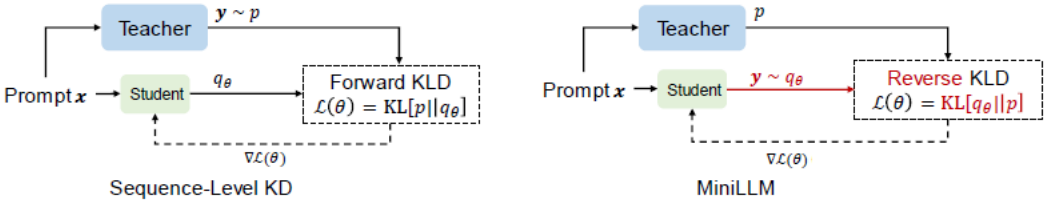


Figure 32: Overview of MiniLLM framework. Figure from (Gu, Dong, Wei, & Huang, 2023)

The authors propose a knowledge distillation framework that leverages the output distribution of a teacher model to train smaller, more efficient student models. Models trained with their approach are referred to as MINILLM. Unlike traditional methods that minimize the forward Kullback-Leibler divergence (KLD), which performs well for classification tasks with limited output space, MINILLM instead minimizes the reverse KLD.

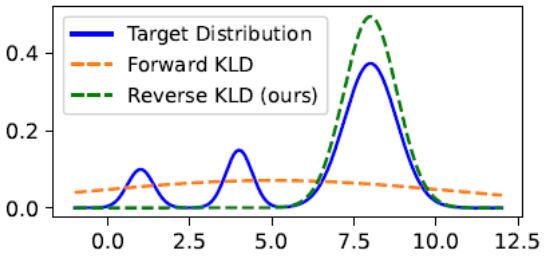


Figure 33: Illustration of the distributions. Figure from (Gu, Dong, Wei, & Huang, 2023)

This design choice is motivated by the observation that forward KLD tends to overrepresent low-probability regions of the teacher distribution in the student model, leading to noisy or low-quality generations. In contrast, reverse KLD suppresses those low-probability regions, encouraging the student to focus on the most probable and meaningful outputs. This mode-seeking behavior is particularly important for text generation, where maintaining factuality and coherence is critical.

To optimize the reverse KLD objective, the authors adopt a Policy Gradient optimization strategy. The rationale is that text generation can be framed as a sequential decision-making process, where each token prediction corresponds to an action taken by a policy (i.e., the student model) conditioned on the previously generated tokens.

To make this optimization stable and effective, the authors propose three key strategies:

1. Single-Step Decomposition: instead of relying solely on cumulative rewards, they decompose the gradient to separate immediate token-level rewards from long-term cumulative rewards. This significantly reduces variance and improves training stability and convergence.
2. Teacher-Mixed Sampling: to mitigate issues like reward hacking, where the student learns to exploit the reward function by generating degenerate but high-scoring outputs, they propose sampling tokens from a mixture of the teacher and student distributions. This helps guide the student while preserving diversity and discouraging repetitive or nonsensical outputs.
3. Length Normalization: Since reward accumulation can unfairly penalize longer sequences, they normalize the cumulative reward. This correction prevents the model from preferring shorter, incomplete outputs, which are easier to optimize.

Together, these strategies enable effective optimization of the reverse KLD objective in the context of open-ended text generation, resulting in student models that produce high-quality, factual, and coherent responses.

The training process begins with a teacher model that has been fine-tuned on a task-specific dataset D, or that already performs well in generalization, and a student model pre-trained on a large unsupervised corpus DPTD. Training proceeds in two main phases:

1. Supervised Fine-Tuning: the student is first fine-tuned on D using the ground-truth responses. The checkpoint with the lowest validation loss is selected as initialization for the next phase.
2. Policy Gradient Optimization: the student is further trained to minimize the reverse KLD objective using the previously described strategies (single-step decomposition, teacher-mixed sampling, and length normalization). Additionally, a language modeling loss on DPTD is added to preserve general language capabilities.

This two-phase training enables the student model to align closely with the teacher while retaining its general language skills.

Model	Method	Size	**Instruction Following Task**
			Vicuna
OPT	Teacher	13B	17.8
	MiniLLM	1.3B	17.9
		2.7B	19.1
		6.7B	18.7

Table 19: Main results for MiniLLM. Higher is better.

MiniLLM models achieves comparable or even better results with much smaller models. Most notably, when the parameter count is reduced tenfold (down to 1.3B), performance remains equivalent to larger models, underscoring MiniLLM‘s models efficiency at smaller scales.

Low Rank Factorization

Low-rank factorization is a technique used to reduce a large weight matrix into smaller matrices. For example, a weight matrix of dimension $m \times n$ can be factorized into two smaller matrices of dimensions $m \times k$ and $k \times n$, where k is much smaller than $min(m, n)$. This decomposition can be used to compress large language models (Zhu, Li, Liu, & Ma, 2024)

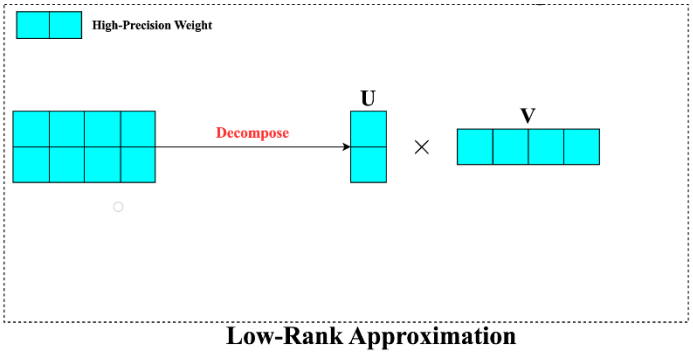


Figure 34: Overview of Low Rank Factorization. Figure from (Zhu, Li, Liu, & Ma, 2024)

SVD-LLM - (Wang, Zheng, Wan, & Zhang, 2024)

In (Wang, Zheng, Wan, & Zhang, 2024), the authors introduce a post-training SVD-based compression method, SVD-LLM, which addresses two major limitations of prior approaches.

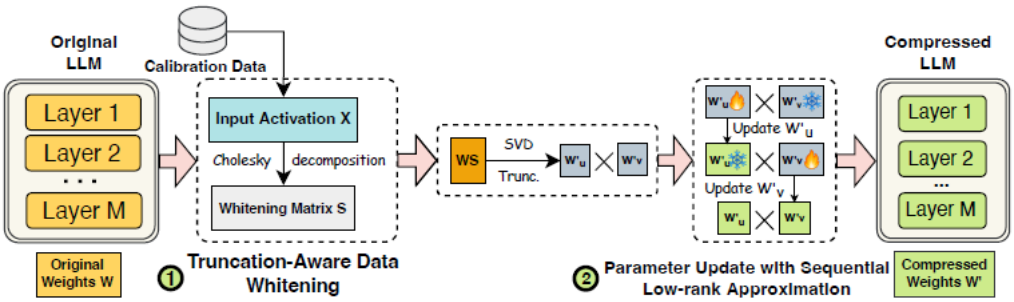


Figure 35: Overview of SVD-LLM. Figure from (Wang, Zheng, Wan, & Zhang, 2024)

First, truncating smaller singular values does not necessarily result in minimal compression loss, due to the absence of a direct mapping between singular values and the actual loss. Second, existing methods do not perform any parameter updates after truncation, which is particularly detrimental at high compression ratios where larger singular values are progressively removed, leading to substantial performance degradation.

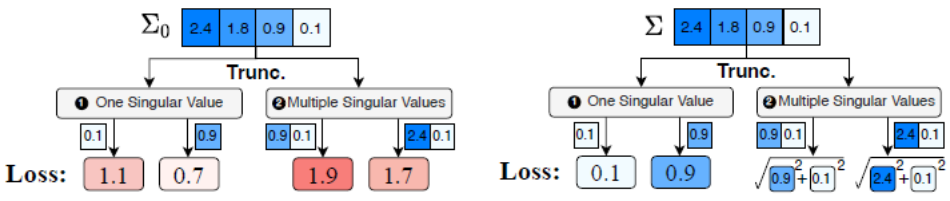


Figure 36: Data normalization (left) vs Truncation-Aware Data Whitening (right). Figure from (Wang, Zheng, Wan, & Zhang, 2024)

To enable a direct mapping between singular values and compression loss, the authors introduce a truncation-aware data whitening technique. Input activations are first generated using a set of randomly. The whitening process enforces the transformed activations $S^{-1}X$ to be orthonormal, ensuring that each channel is statistically independent. The whitening matrix S is computed via

Cholesky decomposition of the activation covariance matrix. Singular Value Decomposition (SVD) is then applied to the product WS, yielding the matrices U, Σ, and V. The smallest singular values in the resulting diagonal matrix Σ are truncated. Crucially, the compression loss is equal to the truncated singular value (in the case of one), or to the square root of the sum of squared truncated singular values, when multiple are removed.

While the aforementioned truncation-aware whitening technique mitigates accuracy loss, parameter updates become necessary at high compression ratios, where even larger singular values are truncated. To address this, the authors propose a sequential low-rank approximation strategy to fine-tune the decomposed matrices W_u, and $W_{v'}$. Specifically, the method involves iteratively fine-tuning one matrix while keeping the other frozen (i.e. first optimizing W_u, with W_v, fixed, and then vice versa. Updates are thus performed in a stable manner while accounting for interdependencies between the two. After fine-tuning, the learned low-rank updates $B_u A_u$ and $B_v A_v$ are added to W_u, and $W_{v'}$, respectively, to form the final compressed weight matrices.

Ratio	Method	**Wikitext-2 Perplexity**
		LLaMA-7B
0%	Original	5.68
20%	SVD-LLM (W)	7.94
	SVD-LLM	7.73
40%	SVD-LLM (W)	13.73
	SVD-LLM	9.27
60%	SVD-LLM (W)	66.62
	SVD-LLM	15
80%	SVD-LLM (W)	1349
	SVD-LLM	31.79

Table 20: Main results for SVD-LLM. Lower is better

Perplexity increases as the compression ratio rises. The SVD-LLM model with the recovery phase consistently maintains lower perplexity. At high compression (e.g., 80%), SVD-LLM without the recovery phase fails, showing a dramatic increase in perplexity which highlights the importance of the recovery phase for maintaining performance at high compression ratio.

Carbon Footprint

Categorization Framework of Machine Learning Impacts on GHG Emissions

In (Kaack, et al., 2022) ,the authors introduce a systematic framework in which they define three *primary categories* to categorize the impacts of machine learning (ML) on greenhouse gas (GHG) emissions.

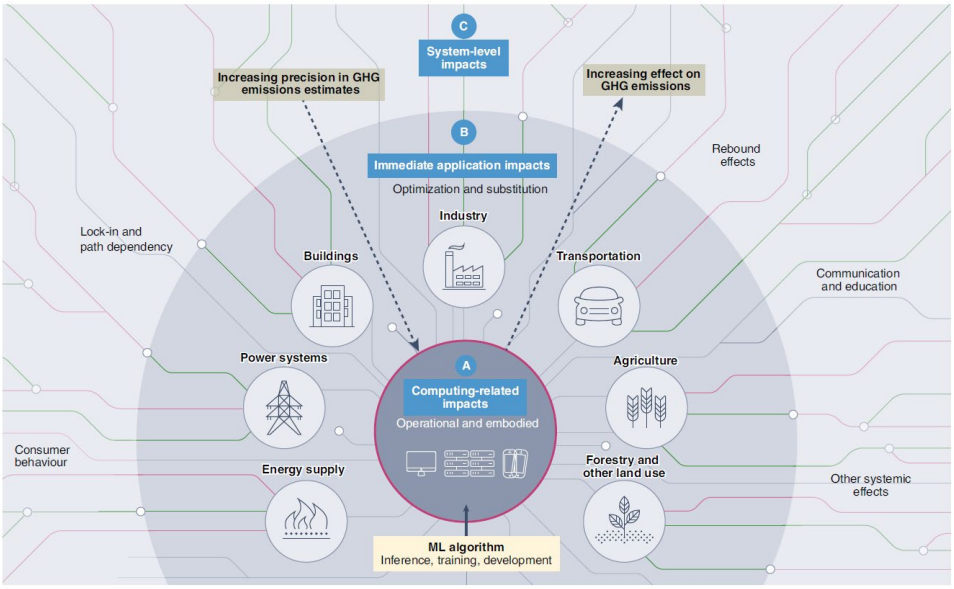


Figure 37: Framework for assessing GHG emissions impacts for ML from (Kaack, et al., 2022)

Computing-related impacts

In the computing-related impacts category, they develop two different points of view: the **Use-Phase view** and the **Top-Down view**. The Use-Phase view focuses on estimating the energy consumption of individual ML model instances across different stages. In contrast, the Top-Down view aims to assess the total global GHG emissions attributable to ML workloads by considering both operational energy use and embodied emissions from computing infrastructure.

The **Use-Phase view** breaks down the ML model life cycle into three stages: inference, training, and development. Inference refers to the stage where a trained model is applied to new inputs; it is the most frequent operation but typically the least energy-intensive per run. Training involves adjusting model parameters using a dataset through multiple iterations (epochs); it requires significantly more energy than inference but occurs less frequently. Development encompasses the experimental phase, during which multiple models are trained and evaluated to identify the most effective configuration. This stage is the most energy-intensive and least frequent, due to the extensive trial-and-error process involved.

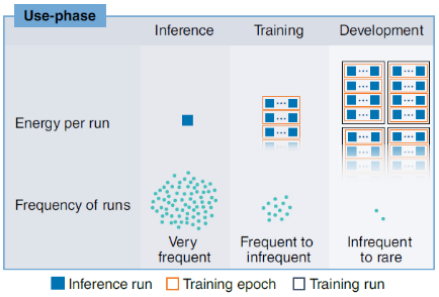


Figure 38: Use-Phase view from (Kaack, et al., 2022)

To mitigate emissions within this view, several strategies have been proposed. In some cases, increasing training effort can reduce the long-term energy footprint by producing more efficient models that require less computation during inference. Other approaches include model compression, which reduces model size and computation without significantly impacting performance, and the selective use of lightweight models for simpler tasks.

The **Top-Down view** aims to estimate the total global GHG emissions attributable to ML workloads by considering both the operational energy use and the embodied emissions of computing infrastructure. This approach does not focus on individual model runs but rather on aggregated data from data centers and devices running ML systems. It includes emissions from electricity consumption (Scope 1 and 2 under the Greenhouse Gas Protocol) and from hardware manufacturing, transport, and disposal (Scope 3). The rapid growth in model size and usage frequency raises concerns about the increasing climate impact of ML over time.

Mitigation strategies under this view focus on system-wide efficiency improvements and infrastructure decisions. These include shifting workloads to data centers powered by low-carbon energy sources, maximizing hardware utilization and virtualization, reusing or repurposing existing hardware (especially GPUs), and investing in energy-efficient processors such as AI accelerators. Additionally, delaying hardware replacement cycles (i.e. when energy efficiency gains are marginal) can reduce embodied emissions over time.

Immediate applications impact

The **immediate application impacts** category refers to the short-term effects that result directly from the deployment of ML systems in specific use cases. These impacts can be either positive or negative, depending on the application context. On the one hand, ML can enable GHG reductions by improving efficiency in sectors such as energy, transportation, and agriculture (e.g. forecasting renewable energy production, optimizing industrial operations, or enabling predictive maintenance). On the other hand, ML can also contribute to increased emissions when applied to carbon-intensive activities, such as enhancing fossil fuel exploration or scaling industrial livestock management. Estimating these impacts remains challenging due to the diversity of application areas, a lack of consistent deployment data, and the difficulty of attributing changes in emissions directly to ML usage.

System-Level Impacts

The **system-level impacts** category captures broader, indirect effects of ML on GHG emissions that arise from structural or behavioral changes. These include rebound effects, where efficiency gains reduce costs and unintentionally drive higher consumption, and lock-in effects, where ML accelerates the adoption of technologies that may hinder low-carbon alternatives. ML can also influence lifestyles, such as through recommendation systems that encourage increased consumption. Although difficult to quantify, these impacts can be significant and long-lasting.

CO2e Measurements Methods

Reporting method

In (Strubell, Ganesh, & McCallum, 2019), the authors use the following equation to estimate the energy consumption required to train deep neural networks:

$$p_t = \frac{1.58t(p_c + p_r + gp_g)}{1000}$$

where p_t is the total power required (in kilowatt-hours), p_c is the average power draw from all CPUs, p_r is the average power draw of a single GPU, g is the number of GPUs used, and p_r is the average power draw from all DRAM slots. The factor 1.58 corresponds to the Power Usage Effectiveness (PUE), which accounts for additional energy costs associated with cooling and infrastructure. This total energy consumption is then converted into estimated carbon emissions by multiplying p_t by the average CO_2 produced per kilowatt-hour, which depends on the local energy mix, influenced by location, time, and infrastructure.

Direct Computing Impacts of BLOOM Model

In (Lucioni, Viguier, & Ligozat, 2022), the authors adopt a Life Cycle Assessment (LCA) approach to quantify the carbon footprint of the BLOOM model. However, due to limited data availability, they focus on the direct computing impacts using the top-down perspective introduced by (Kaack, et al., 2022).

Embodied Emissions

To estimate embodied emissions, the emissions associated with the materials and processes involved in manufacturing computing equipment, they used proxy values from available figures for comparable hardware. Specifically, they reference data for HPE servers and publicly estimated values for NVIDIA A100 GPUs. Additionally, they assume a hardware lifetime of six years and an average usage rate of 85%, as reported by the administrators of the Jean Zay computing cluster. These assumptions allow them to derive an estimated CO_2eq per hour of server usage. Notably, their estimate includes only servers and GPUs, excluding other infrastructure components such as cooling systems, power distribution units, and network switches, which were not accounted for due to data limitations.

Dynamic Power Consumption

Dynamic power consumption refers to the electricity required to actively run the training process, and has been the primary focus of much of the existing work on estimating CO_2 emissions in machine learning. It is calculated as the product of GPU hours, the Thermal Design Power (TDP) of the GPUs used, and the carbon intensity of the energy grid powering the data center.

While TDP represents the maximum power draw of a GPU and is therefore a conservative estimate, it is often used as a reliable proxy when real-time energy measurements are not available. In highly optimized training configurations, such as those used for training BLOOM, GPU utilization is close to 100%, making TDP a reasonable approximation for actual energy use.

Idle Power Consumption

However, dynamic power consumption alone does not account for the energy required to operate the broader data center infrastructure. While metrics like Power Usage Effectiveness (PUE) are commonly used to approximate this overhead, they do not capture all sources of energy consumption. To improve accuracy, they conducted empirical experiments to measure the actual energy usage of the Jean Zay computing cluster in different operational states. They found that infrastructure consumption, which includes networking systems, cooling, and general data center maintenance when servers are turned off, accounts for 13.5% of total energy usage. Idle consumption, where servers are powered on but not actively used, represents 32%, while dynamic consumption, corresponding to active training,

comprises only 54.5% of the total. These results highlight the significant contribution of non-computational components to the overall energy and carbon footprint of model training.

Deployment and Inference

To assess the carbon footprint of deployment and inference, a stage that has received relatively little attention in existing research. They deployed the BLOOM model on a Google Cloud Platform (GCP) instance equipped with 16 NVIDIA A100 40GB GPUs. Over an 18-day period, they simulated real-world usage by sending an average of 558 API requests per hour, with traffic varying over time to reflect the unpredictable nature of real-world inference workloads. The CodeCarbon package was used to monitor real-time energy consumption of the instance during this period.

Their measurements revealed that 75.3% of the total energy was consumed by the GPUs, primarily just to keep them running in memory, even when not actively processing requests. RAM usage accounted for 22.7%, and the CPU for only 2% of total energy. Notably, the average GPU power draw was significantly below the TDP, indicating that the GPUs were not operating at full capacity which is expected in inference settings, where requests arrive intermittently and batching is typically not feasible.

Predictive method

In (Faiz, et al., 2023), the authors develop an end-to-end projection model called LLMCarbon, designed to accurately estimate the carbon footprint of large language models (LLMs) across their entire life cycle, including the experimentation, training, inference, and storage phases. The primary objective is to assist LLM designers in evaluating the trade-offs between carbon emissions, training duration, and model performance, early in the design process.

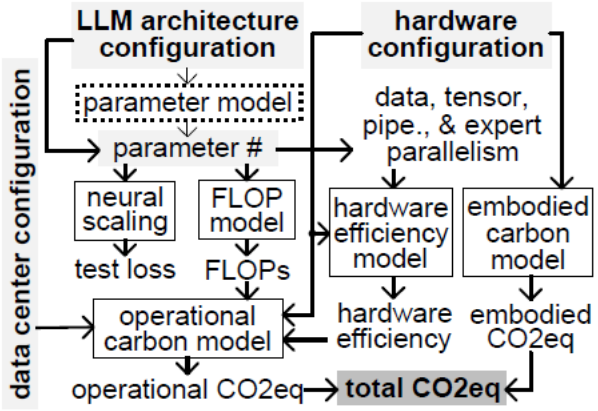


Figure 39: Overview of LLMCarbon from (Faiz, et al., 2023)

To predict the carbon footprint of an LLM, LLMCarbon takes as input the model's architectural description (or alternatively, the number of parameters), the number of training tokens, the hardware configuration, and the data center specifications.

Input variables

To estimate the number of parameters, LLMCarbon uses two formulations. Equation 25 refers to a dense decoder-only LLM architecture, where the total number of parameters depends on the hidden size h, number of layers l, and vocabulary size V:

$$P_d \approx 12lh^2 + Vh$$

Equation 25

Equation 26 extends this to Mixture-of-Experts (MoE) models, where a proportion $\rho \in [0,1]$ of the feedforward layers in the dense model is replaced by MoE layers. The MoE parameter count where N_e the number of experts is then given by:

$$P_e \approx (1-\rho)P_e + \rho(4h^2 + 8h^2N_e)l$$

Equation 26

Based on the number of parameters and the size of the training dataset (i.e., the number of tokens), LLMCarbon derives the expected test loss using the Chinchilla scaling law, which ensures comparability across different model sizes and training configurations. The scaling law is defined as:

$$L(P,D) = \frac{A}{P^{\alpha}} + \frac{B}{D^{\beta}} + E$$

Equation 27: Chinchilla scaling law

where P is the number of parameters, D is the number of training tokens, and A, B, α, β, and E are empirically fitted constants.

The FLOP model in LLMCarbon estimates the total number of floating-point operations (FLOPs) required for both training and inference. It takes as input the number of model parameters P and the number of training tokens D. Based on these, the number of FLOPs for training is computed as:

$$TC \approx 6PD$$

Equation 28: Training cost

and for inference as:

$$IC \approx 2PD$$

Equation 29: Inference cost

These formulations reflect the dominant cost of multiply-accumulate operations during forward and backward passes (for training) and forward-only passes (for inference).

Hardware efficiency

In order to achieve high hardware efficiency, defined as the ratio between actual computational throughput and the peak theoretical throughput, parallelism strategies, which LLMs rely on to scale across multiple devices, must be optimized.

LLMCarbon considers four types of parallelism. In data parallelism, the full model is replicated across all devices, while the training dataset is evenly partitioned among them; periodic gradient aggregation ensures consistent model weights across replicas. Tensor parallelism distributes components of each LLM layer (such as attention heads and feed-forward units) across multiple devices, requiring coordinated computation and communication. In pipeline parallelism, model layers are divided evenly across devices, and data flows sequentially through them in microbatches. Expert parallelism, specific to Mixture-of-Experts (MoE) models, distributes different expert modules across devices, with tokens dynamically routed to a subset of experts during execution.

The optimal parallelism setting is represented as a tuple (p, t, d, e), where each variable corresponds to the degrees of pipeline, tensor, data, and expert parallelism, respectively. For dense models, the process begins by increasing tensor parallelism t up to the number of interconnected devices per server (e.g., $z = 8$), constrained by bandwidth limits. Once this threshold is reached, pipeline parallelism p is scaled further, ensuring that the product $p * t$ remains within memory constraints. Expert parallelism is not used in dense models and thus set to $e = 1$. For MoE models, expert parallelism e is determined by the number of experts (e.g., e=64), and data parallelism d is fixed at 1 to reduce inter-device communication overhead. MoE models scale primarily through pipeline parallelism.

The total number of required computing devices is calculated as $n = t \cdot p \cdot d$. However, hardware efficiency decreases if the number of devices used re deviates from this optimal value n. This degradation is modeled by:

$$eff_{re} = \begin{cases} \gamma_0 \cdot \frac{re}{n} \cdot eff_n & re < n \\ \gamma_1 \cdot \frac{re}{n} \cdot eff_n + \gamma_2 & re > n \end{cases}$$

Equation 30

where γ_0, γ_1, and γ_2 are fitting constants, eff_n is the maximum hardware efficiency achievable at the optimal device count n.

Execution time

The resulting device-level execution time is given by:

$$t_{dev} = \frac{TFLOP}{n_{dev} \cdot FLOP_{peak} \cdot eff}$$

Equation 31: Device-level execution time

Energy consumption

LLMCarbon estimates the total energy consumed by all hardware components involved in LLM processing, including CPUs, GPUs or TPUs, memory (DRAM), SSDs, and other system components:

$$energy_{hard} = \sum_{i \in hardware\_set} (P_i \cdot eff_i \cdot n_i \cdot t_i)$$

Equation 32: Hardware energy consumption

where for each hardware unit i:

- P_i is the peak power i,
- eff_i is the efficiency
- n_i is the number of units
- t_i the execution time

This estimation is grounded in the observation that once parallelism is optimized, the execution time is primarily governed by the performance of the hardware accelerators (such as GPUs or TPUs).

The operational energy consumption is then adjusted using the Power Usage Effectiveness (PUE), an industry-standard metric that accounts for the additional energy overhead required to support data center infrastructure, such as cooling, power distribution, and networking:

$$energy_{oper} = energy_{hard} \cdot PUE$$

Equation 33

Carbon Emissions Equivalent

To estimate the resulting carbon emissions, the operational energy is multiplied by the carbon intensity of the data center:

$$CO2e_{oper} = energy_{oper} \cdot carb\_inten$$

Equation 34

Carbon intensity depends on the energy mix used by the data center, ranging from low-carbon sources like nuclear and hydropower to high-carbon sources like coal and natural gas. This factor allows LLMCarbon to incorporate location-specific environmental impact into its carbon footprint estimates.

Embodied Carbon Emissions

To compute the embodied carbon emissions, LLMCarbon uses the concept of Carbon emitted Per Unit Area (CPA). The CPA captures emissions associated with the entire manufacturing process, including fabrication yields, energy consumption per unit area, chemical processing emissions, and emissions linked to raw material extraction. The total embodied carbon footprint is then calculated by summing, across all hardware components, the product of each device's chip area and its corresponding CPA, scaled by the fraction of time the device is used and normalized by its expected lifetime:

$$CO2e_{emb} = \sum_{i \in hardware\_set} \frac{t_i \cdot CO2e_{chip_i}}{lifetime_i}$$

Equation 35

with $CO2e_{chip_i} = area \cdot CPA$

Equation 36

Total Carbon Emissions

The total carbon footprint of an LLM is then obtained by summing the operational and embodied emissions:

$$CO2e = CO2e_{oper} + CO2e_{emb}$$

Equation 37: Total CO2 equivalent

Methodology

The preceding literature review reveals that, despite growing interest in model compression, existing research has focused almost exclusively on downstream performance. Compressed models are typically evaluated on WikiText-2, sometimes accompanied by a few additional benchmarks, but rarely in ways that capture the full range of their capabilities, such as reasoning, instruction following, or multitask learning. Inference speedup is occasionally reported, but it is not systematically measured nor linked to actual energy consumption.

To date, and to the best of our knowledge, there is no empirical study examining how model compression affects hardware energy consumption, how this consumption varies across hardware configurations, or how deployment context, particularly grid carbon intensity, influences the carbon footprint of compressed versus uncompressed models. The potential carbon advantage of compression remains entirely unquantified and potentially misleading when considered in isolation from real-world infrastructure.

To address these limitations, we propose the following methodology. We compare the performance of a set of compressed and uncompressed models using standardized benchmarks. We then measure their hardware energy consumption. Finally, we analyze the resulting carbon footprint across different deployment scenarios.

Model selection

The selection of large language models for this study was guided by practical considerations. First, only open-source models were considered, as access to model weights is essential for applying compression techniques. Furthermore, models based on a decoder-only dense architecture were only considered, as this design dominates current language modeling practices and benefits from broad community and tooling support. Finally, due to limited computational resources, models had to be compatible with the available hardware. In particular, it was not feasible to load or evaluate models exceeding 30 billion parameters.

In addition to these technical constraints, the selection aimed to ensure relevance and diversity. The goal was to include models with strong academic grounding, such as LLaMA-7B and LLaMA-30B, which are widely cited and part of a well-established family. Alongside them, Mistral-7B-v0.3 was selected as a more recent alternative with fewer citations but increasing visibility. Lastly, Mistral Small 3 was included for its strong efficiency and is currently considered one of the most performant models within its size category. This diversity was intentionally sought to include models that differ in scale and technological maturity.

Model	# Parameters (in billions)	Release Date
LLaMA-7B	7	2023
LLaMA-30B	30	2023
Mistral-7B-v0.3	7	2024
Mistral Small 3	24	2025

Table 21: Overview of models

Evaluation Metrics and Protocol

Metrics selection

To assess model performance across complementary dimensions, three evaluation tasks were selected based on their prevalence in the literature and their alignment with common use cases.

Perplexity is a standard metric for evaluating language modeling performance. Given a model and an input text sequence, it measures how likely the model is to generate that sequence. In practice, it reflects the model's ability to predict the next word in a sequence with confidence and accuracy. Lower perplexity indicates that the model assigns higher probabilities to the correct next words, suggesting it has better internalized the statistical distribution of natural language. In this work, perplexity was computed using the WikiText-2 dataset , a benchmark widely adopted in the literature for evaluating autoregressive language models.

The *Massive Multitask Language Understanding (MMLU) benchmark* was used to evaluate the models' world knowledge and problem-solving capabilities. It includes 57 tasks from diverse domains such as mathematics, history, computer science, and law.

Finally, *IFEval* was included to assess instruction-following capabilities. This benchmark evaluates a model's ability to follow explicit and verifiable instructions contained within structured plans.

Together, these three evaluation tasks offer a complementary view of model performance across linguistic prediction, general reasoning, and instruction adherence.

Evaluation Protocol

All performance evaluations were conducted using the open-source framework lm-evaluation-harness, which is widely used in academic research on large language models. All models were evaluated in a zero-shot setting. Generation parameters such as temperature, top-k, and top-p were left at their default values. All models, including both the original and compressed versions, were evaluated under the same conditions using this framework.

Compression method

The selected models were quantized using the OPTQ method. OPTQ was chosen for its scalability and compatibility with commonly used tooling in both academic and industrial settings, particularly with packages such as Transformers and VLLM. A 4-bit quantization format was selected, as it is the most widely supported and easiest to integrate into current inference pipelines.

This method can be applied efficiently even to models with hundreds of billions of parameters while requiring minimal computational resources. Although this work focuses on models up to 30 billion parameters, the method was also selected for its potential applicability to larger-scale models in future settings.

Other compression techniques were not retained, as they do not meet the same criteria in terms of compatibility and ease of use. They are not commonly supported by existing tooling and require significant engineering effort to be usable in practice.

Inference Carbon Footprint Estimation

To estimate the carbon emissions associated with model inference, a methodology similar to the one presented in (Lucioni, Viguier, & Ligozat, 2022) was followed.

The first step consisted in generating a fixed sequence of prompts to be sent to the models. These prompts were extracted from the DailyDialog dataset (Li, et al., 2017) by randomly selecting a predefined number of samples and then taking the first sentence of each selected sample. This dataset contains multi-turn conversations that are human-written, relatively noise-free, and cover a wide range of daily life topics. It was selected for its relevance to realistic, everyday user interactions. The total number of prompts was chosen to simulate an average interaction rate of approximately one prompt every six seconds, in line with the scenario described in (Lucioni, Viguier, & Ligozat, 2022).

Since, to my knowledge, no real-world user requests were available, in terms of prompts, content, or frequency, the use of a standardized dataset was necessary to ensure comparability and repeatability across experiments. Each prompt was associated with a predefined randomly set timestamp to ensure that all models received the same inputs at the exact same moments. Inference was executed continuously for four hours per model, including both baseline and quantized versions. This duration, shorter than the 18-day window used in (Lucioni, Viguier, & Ligozat, 2022), was constrained by the limited access to computational resources, which prevented a longer and more realistic deployment phase.

Additionally, unlike (Lucioni, Viguier, & Ligozat, 2022), which relied on an API deployment for inference, our experiments loaded the models directly within the evaluation script. This approach was chosen due to resource constraints, as deploying a dedicated API server was not feasible with the available infrastructure.

Hardware energy consumption was measured using the CodeCarbon Python package (Courty, et al., 2024), also used in (Lucioni, Viguier, & Ligozat, 2022), which tracks power usage from CPU, GPU, and RAM during execution. Power usage was integrated over time to obtain the hardware energy consumption in kilowatt-hours (kWh).

CodeCarbon assumes a memory configuration consisting of only two DIMM slots (2 × 5 W), resulting in a reported RAM power consumption of 10 W. However, the HPE XL645d nodes used in our experiments may be configured with up to 16 DIMM slots. As the exact memory layout was not available, I conservatively assumed that all 16 slots were populated. Applying CodeCarbon's per-slot efficiency scaling (100% for slots 1–4, 90% for slots 5–8, and 80% for slots 9–16) yields an estimated RAM power consumption of approximately 70 W per node. To reflect this in our energy calculations, we multiplied the default RAM consumption figure by a factor of 7, aligning it with the maximum possible DIMM slot usage. Thus, RAM energy consumption is expressed as:

$$energy_{RAM} = 7 \times energy_{RAM,CodeCarbon}$$

Equation 38: Ram Energy consumption

where $energy_{RAM,CodeCarbon}$ is the RAM energy consumption reported by CodeCarbon.

It is important to note that CodeCarbon estimates RAM energy consumption based on the number of populated DIMM slots, rather than on the total memory volume.

The hardware energy consumption is thus computed using:

$$energy_{hardware} = energy_{CPU,CodeCarbon} + energy_{GPU,CodeCarbon} + energy_{RAM}$$

Equation 39: Hardware Energy Consumption

The resulting energy values were then converted into CO_2 emissions equivalents using carbon intensity data obtained from (Electricity Maps ApS, 2025). This website was selected because it provides, to the best of my knowledge, the most comprehensive publicly available estimates. It is based on a life cycle assessment (LCA) approach, which accounts for the entire process of electricity production, from generation to delivery, and provides country-specific carbon intensity values. The following equation is used to compute the CO2 equivalent ($CO2e$) for each country:

$$CO2e = energy_{hardware} \times carb\_inten$$

Equation 40: Hardware CO2 equivalents

where $CO2e$ is expressed in grams and $carb\_inten$ represents the carbon intensity (gCO2/kWh)

Since no detailed information was available regarding the precise data center locations where the models would eventually be deployed, the yearly national average grid carbon intensity was adopted as the most reasonable proxy. To reflect a likely European deployment scenario, carbon intensity figures for France, Germany, and the Netherlands, countries geographically close to Belgium, were considered. This choice was guided by the assumption that, in the absence of real-world latency data, proximity would be favored to minimize latency.

No Power Usage Effectiveness (PUE) adjustment was applied, as accurate and up-to-date PUE values were not available, neither for the HPC cluster used in Belgium, nor for hypothetical deployments in the surrounding countries.

Predictive estimation methods, such as LLMCarbon, were not used, as they rely on theoretical calculations based on the number of floating-point operations (FLOPs) required during inference. However, this approach becomes unreliable when applied to compressed models, as compression techniques, modify the effective number of FLOPs executed due to changes in model architecture, data precision, and execution pathways. Adapting the original estimation formula to account for these modifications would require adjustments and empirical validation, which were outside the scope of this study.

Practical implementation

Experimental Setup

All experiments were conducted on the *Lucia* high-performance computing (HPC) cluster. I used the two following hardware configurations[3].

Hardware Setup

Setup 1

GPU	CPU	RAM
Nvidia A100 SXM4 80GB	AMD EPYC 7513 32-core (only 2 active cores)	240GB

Table 22: Hardware configuration used in Setup 1

This configuration was used for all performance evaluations, as well as the first round of carbon footprint measurements. It allowed the execution of all selected models, including the largest ones with up to 30 billion parameters. This configuration provided sufficient computational and memory resources to run both baseline and quantized versions without constraint.

Setup 2

GPU	CPU	RAM
Nvidia A100 40GB	AMD EPYC 7513 32-core (only 2 active cores)	60GB

Table 23: Hardware configuration used in Setup 2

This configuration was used exclusively for the second round of energy measurements. The main difference compared to the primary setup lies in the reduced memory resources, both in terms of system RAM and GPU memory (VRAM). The limited VRAM capacity made it impossible to run larger models such as LLaMA-30B and Mistral Small 3 in their original form. However, their quantized versions were able to execute without issue.

Software Setup

The software environment was organized according to the specific requirements of each experimental phase. For performance evaluation, all models were tested using the *lm-evaluation-harness* framework (version 0.4.8), with the original, uncompressed models[4] retrieved from the Hugging Face model hub. For energy consumption measurements, the models were executed using the *Transformers* library (version 4.47.0).

Models were quantized[5] using the GPTQ-Model package (version 1.7.4), applying 4-bit quantization. The weights were grouped using the default group size of 128. These models were then loaded using the GPTQ model loader included in the same package during the energy measurement phase.

All experiments were conducted within a Jupyter Notebook environment. Energy consumption data was collected using *CodeCarbon* (version 3), configured with measure_power_sec set to 5 to record

[3] *I used AI nodes for the Setup 1 and GPU nodes for the Setup 2. System details can be found at https://doc.lucia.cenaero.be/system_details/compute_nodes/*

[4] *In the results section, uncompressed models are referred to as "full," indicating their use of full precision, while quantized models are referred to as "compressed."*

[5] As a reminder, GPTQ is now referenced as OPTQ.

power usage at five-second intervals. The tracking mode was set to *process*, aiming to isolate the energy consumption of the inference process from other concurrent system activities.

Finally, all models were fully loaded onto the GPU to prevent unintended offloading behavior that could result from automatic memory balancing. Moreover, the maximum output length during inference was capped at 100 tokens in order to avoid out-of-memory (OOM) errors, particularly for the largest model (LLaMA 30B). On the hardware side, inference computations utilized the *Marlin* backend kernel, which is the default configuration for A100 GPUs.

Experiments

Performance Evaluation

Wikitext-2 Perplexity

Version	Wikitext-2 Perplexity			
	LLaMA-7B	LLaMA-30B	Mistral-7B-v0.3	Mistral Small 3
Full	9.38	6.09	7.90	6.63
Compressed	9.72	6.35	8.14	6.82
PPL Difference	0.34	0.27	0.24	0.19

Table 24: Wikitext-2 perplexity scores for full and compressed models. Lower values indicate better performance. "PPL difference" refers to the perplexity of the full model minus that of the compressed model.

When applying 4-bit quantization with a group size of 128 weights, performance degradation remains minimal, as shown by the results obtained on the Wikitext-2 benchmark. The difference in perplexity between full and compressed versions remains low across all models, ranging from 0.19 to 0.34.

Older models such as LLaMA-7B and LLaMA-30B show slightly higher increases in perplexity, whereas more recent architectures like Mistral-7B-v0.3 and Mistral Small 3 are less affected. This suggests that newer models may be inherently more robust to quantization. In addition, smaller models tend to show slightly greater degradation than larger ones.

MMLU Accuracy

Version	MMLU Accuracy			
	LLaMA-7B	LLaMA-30B	Mistral-7B-v0.3	Mistral Small 3
Full	0.3204	0.5461	0.5907	0.7698
Compressed	0.2785	0.5357	0.5447	0.7652
Acc. Difference	0.0419	0.0104	0.0460	0.0046

Table 25: MMLU accuracy scores for full and compressed models. Higher values indicate better performance. "Accuracy difference" refers to the accuracy of the full model minus that of the compressed model.

Across all models, the accuracy drop observed after quantization remains very limited. The differences between full and compressed versions range from 0.0046 to 0.0460.

As in the perplexity analysis, older models such as LLaMA-7B and LLaMA-30B show slightly higher performance degradation than newer ones. In particular, Mistral Small 3 exhibits the smallest drop (only 0.0046), while LLaMA-7B shows the largest (0.0419).

IFEval Accuracy

Version	IFEval Accuracy			
	LLaMA-7B	LLaMA-30B	Mistral-7B-v0.3	Mistral Small 3
Full	0.1867	0.1774	0.1608	0.1091
Compressed	0.1830	0.1627	0.1608	0.1109
Acc. Difference	0.0037	0.0148	0.0000	-0.0018

Table 26: IFEval accuracy scores for full and compressed models. Higher values indicate better performance. "Accuracy difference" refers to the accuracy of the full model minus that of the compressed model.

Instruction-following performance, as measured by IFEval, is largely preserved across all models after quantization. The observed differences are marginal, with some models showing no change at all (e.g., Mistral-7B-v0.3) and others exhibiting minimal variation. Notably, Mistral Small shows a slight improvement post-quantization (−0.0018), which remains within expected statistical noise margins.

Among all models, LLaMA-30B presents the largest drop (0.0148), although this remains minor in absolute terms.

Hardware Energy and Carbon Footprint Analysis

Hardware Energy Consumption

LLaMA-7B	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7022	0.4520	0.2792	1.4335
Compressed	1	1.1617	0.3720	0.2792	1.8129
Relative Change (%)		65%	-18%	0%	**26%**

LLaMA-30B	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7142	0.9674	0.2790	1.9606
Compressed	1	1.1262	0.5697	0.2790	1.9749
Relative Change (%)		58%	-41%	0%	**1%**

Mistral-7B-v0.3	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7040	0.4741	0.2782	1.4563
Compressed	1	1.1026	0.3899	0.2782	1.7707
Relative Change (%)		57%	-18%	0%	**22%**

Mistral Small 3	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7093	0.7990	0.2794	1.7877
Compressed	1	1.1206	0.5341	0.2794	1.9341
Relative Change (%)		58%	-33%	0%	**8%**

Table 27: Hardware energy consumption and relative change using only Setup 1. The compressed version uses the same hardware setup. Relative change is defined as full version minus compressed version.

Despite the systematic reduction in GPU energy consumption introduced by 4-bit quantization, hardware energy consumption does not decrease when evaluated on Setup 1. Instead, the energy savings on the GPU side are consistently offset by a notable increase in CPU usage. This effect is particularly pronounced in smaller models such as LLaMA-7B and Mistral-7B-v0.3, where the increase in CPU energy consumption outweighs the GPU savings, leading to an overall rise in total energy usage of 26% and 22%, respectively.

For larger models, this trade-off remains present: although GPU energy consumption decreases, it is offset by increased CPU usage, ultimately leading to a net increase in total energy consumption as well, though the magnitude is smaller. Specifically, the rise reaches only 1% for LLaMA-30B and 8% for Mistral Small 3.

These findings suggest that quantization, while reducing GPU load, introduces additional overhead on the CPU that can significantly undermine its energy efficiency; particularly in smaller models, where GPU reductions are less impactful and CPU fallback operations become more costly in relative terms.

LLaMA-7B	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7022	0.4520	0.2792	1.4335
Compressed	2	0.4219	0.3276	0.2792	1.0288
Relative Change (%)		-40%	-28%	0%	**-28%**

LLaMA-30B	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7142	0.9674	0.2790	1.9606
Compressed	2	0.4176	0.5045	0.2790	1.2011
Relative Change (%)		-42%	-48%	0%	**-39%**

Mistral-7B-v0.3	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7040	0.4741	0.2782	1.4563
Compressed	2	0.3813	0.3481	0.2782	1.0077
Relative Change (%)		-46%	-27%	0%	**-31%**

Mistral Small 3	Setup	CPU Energy	GPU Energy	RAM Energy	**Hardware Energy**
Full	1	0.7093	0.7990	0.2794	1.7877
Compressed	2	0.4127	0.4656	0.2794	1.1578
Relative Change (%)		-42%	-42%	0%	**-35%**

Table 28: Hardware energy consumption and relative change, with the full version evaluated on Setup 1 and the compressed version on Setup 2. Relative change is defined as the energy consumption of the full version (Setup 1) minus that of the compressed version (Setup 2).

However, when these quantized models are deployed using Setup 2, a significant drop in hardware energy consumption is observed across all cases. Compared to their full-precision counterparts evaluated under Setup 1, energy usage decreases by 28% for LLaMA-7B, 39% for LLaMA-30B, 31% for Mistral-7B-v0.3, and 35% for Mistral Small 3. These results illustrate the dual advantage of quantization: it not only reduces energy consumption but also enables the deployment of larger models on more constrained hardware. For instance, while the full-precision version of LLaMA-30B could not be executed on the 40GB GPU available in Setup 2 due to memory limitations, its quantized variant ran without issue, highlighting the role of compression in expanding deployment possibilities.

While these results underscore the substantial energy savings enabled by quantization, it is important to note that both GPUs used in the experiments remain high-end components. Further investigation is required to determine whether comparable benefits can be achieved on more cost-effective or consumer-grade GPUs.

Moreover, the precise reason behind the increased CPU energy consumption observed in Setup 1 remains uncertain. One hypothesis relates to the difference in GPU specifications between the two setups, specifically the memory bandwidth (NVIDIA, 2025). Setup 1 employs an A100 SXM4 80GB GPU, which offers higher memory bandwidth compared to the A100 40GB GPU used in Setup 2. Since both configurations rely on the same Marlin kernel for inference, this hardware-level difference may influence the overall compute behavior. In particular, a higher memory bandwidth in Setup 1 could allow data to be transferred from the GPU to the CPU more quickly, potentially preventing the CPU from entering an idle (low energy) state between operations. As a result, the CPU could remain active for longer periods, contributing to higher energy consumption. To investigate this hypothesis, future work could involve reducing the sampling interval from 5 seconds to 1 second, or even lower, depending on the CPU's idle transition latency. This would allow for finer-grained tracking of short-term CPU activity spikes that may currently be obscured by the chosen measurement resolution.

Component-Wise Energy Shift Analysis

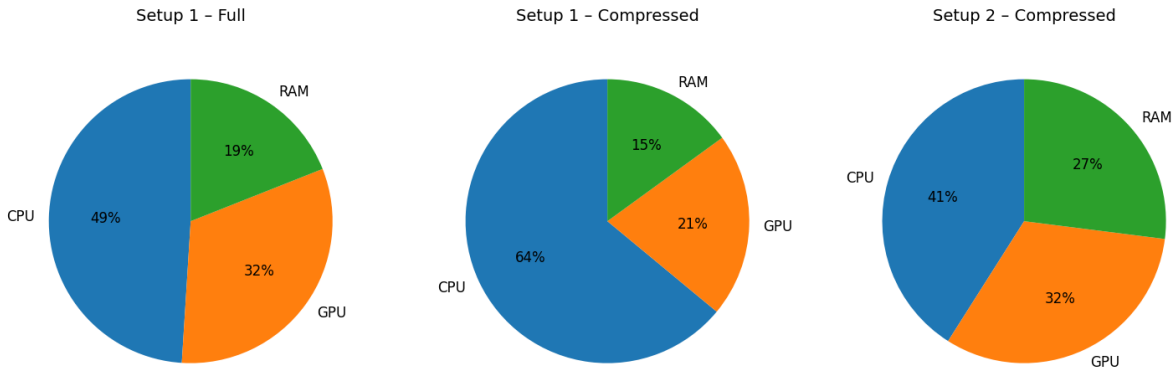


Figure 40: Energy usage breakdown (CPU, GPU, RAM) for LLaMA-7B: left refers full model using Setup 1; middle refers to the compressed model using Setup 1; right refers to the compressed model using Setup 2.

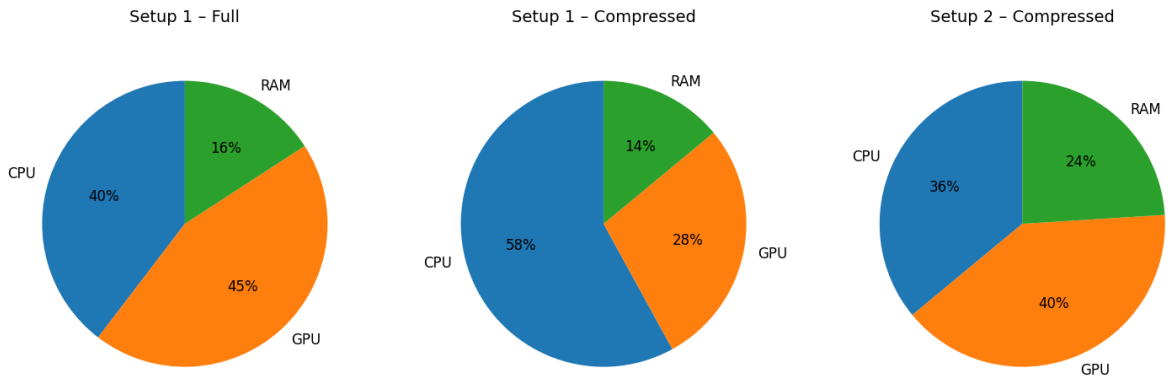


Figure 41: Energy usage breakdown (CPU, GPU, RAM) for Mistral Small 3: left refers full model using Setup 1; middle refers to the compressed model using Setup 1; right refers to the compressed model using Setup 2.

A breakdown of energy usage shares by component (CPU, GPU, and RAM) across different setups reveals consistent patterns in how computational load shifts following quantization. This analysis focuses on LLaMA-7B, and Mistral Small, chosen to represent contrasting model profiles in terms of size and recency. LLaMA-7B is among the smallest and oldest models in this study, whereas Mistral Small is both larger and more recent. Other models exhibited intermediate or redundant patterns and were therefore omitted to maintain clarity of interpretation.

In Setup 1, transitioning from full-precision to 4-bit quantized models leads to a marked reallocation of energy usage across system components. Initially, LLaMA-7B consumes 49 % of its total energy on the CPU, 32 % on the GPU, and 19 % on RAM; after compression, these shares shift to 64 % for the CPU, 21 % for the GPU, and 15 % for RAM. Likewise, Mistral Small's energy distribution moves from 40 % CPU, 45 % GPU, and 16 % RAM in the uncompressed state to 58 % CPU, 28 % GPU, and 14 % RAM when quantized. This pronounced increase in CPU share, coupled with a corresponding decrease in GPU share, confirms that quantization under Setup 1 drives computational load toward the CPU, fully offsetting any GPU energy savings and ultimately raising overall consumption, particularly for smaller models that already underutilize high-performance GPUs, as shown in the previous section.

In Setup 2, compressed models of both LLaMA-7B and Mistral Small exhibit a more balanced distribution of energy usage shares. For LLaMA-7B, 41 % of total energy is consumed by the CPU, 32 % by the GPU, and 27 % by RAM; for Mistral Small, the CPU accounts for 36 %, the GPU for 40 %, and

RAM for 24 %. Due to this increase in RAM's share of energy usage, it would be valuable, as part of future work, to explore more accurate methods for measuring RAM energy consumption, beyond the current assumption of constant power draw.

Taken together, these results highlight that quantization alters the compute distribution profile in non-trivial ways. While CPU usage increases post-quantization in Setup 1, this effect is mitigated in Setup 2, underscoring the importance of evaluating compression impacts on energy consumption under realistic deployment conditions.

Inference Carbon Footprint

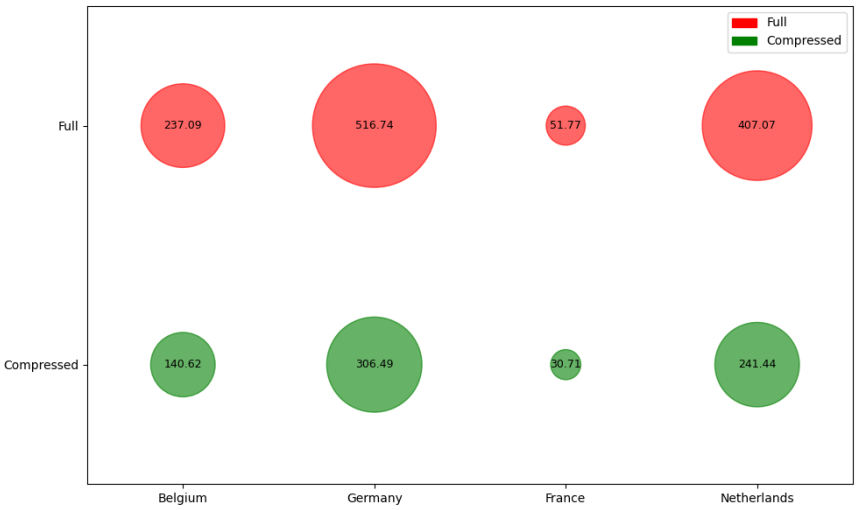


Figure 42: Bubble chart showing estimated CO_2 equivalent emissions (gCO_2e) for Mistral Small in full-precision (Setup 1) and compressed (Setup 2) configurations across Belgium, Germany, France, and the Netherlands. Bubble size reflects the magnitude of emissions, illustrating the combined effects of model efficiency and national grid carbon intensity.

To assess the carbon footprint associated with inference, we estimated the CO_2 equivalent emissions (gCO_2e) for the Mistral Small 3 model in two configurations: full-precision executed on Setup 1, and 4-bit quantized executed on Setup 2. The carbon intensity data grid used corresponds to the 2024 yearly country averages. The analysis was conducted specifically on Mistral Small 3, the most recent and largest model in our benchmark. Similar patterns would likely be observed across the other evaluated models, based on the results presented in the previous section.

Country	gCO2e/kwh (LCA)
Belgium	153.14
Germany	333.77
France	33.44
Netherlands	262.93

Table 29: Yearly average carbon intensity values (gCO_2e/kWh) for national electricity grids in 2024, as used for computing CO_2eq emissions in Figure 12 .

Results were computed across four European countries (i.e. Belgium, Germany, France, and the Netherlands) to highlight the effect of deployment location on emissions. While quantization consistently leads to energy savings, these savings do not necessarily translate into proportional reductions in CO_2 emissions. The carbon footprint is primarily driven by the carbon intensity of the local electricity grid, meaning that even energy-efficient models can produce high emissions when deployed in carbon-intensive regions.

For example, running the full-precision model in Germany results in emissions of nearly 597 gCO_2e, more than ten times higher than the same model run in France (60 gCO_2e), despite identical energy usage. Similarly, the compressed model deployed in the Netherlands (304 gCO_2e) emits almost as much as the full-precision version executed in Belgium (274 gCO_2e). Even in compressed form, running the model in Belgium (177 gCO_2e) generates significantly more emissions than running the uncompressed model in France. These results underscore that reducing energy consumption through compression is necessary but not sufficient: the environmental impact ultimately depends on the carbon intensity of the electricity powering the system.

Conclusions

This work has presented a comprehensive evaluation of the impact of model compression on both performance and environmental cost. Through a systematic comparison of compressed and uncompressed large language models, we have shown that compression, when applied using 4-bit quantization, leads to only marginal performance degradation across a set of benchmarks designed to better capture the full range of model capabilities. While it can enable substantial reductions in direct energy consumption during inference, this effect is not guaranteed: depending on the hardware configuration, compression can in some cases lead to an *increase* in energy usage. Moreover, the magnitude of energy savings, also depends on the size of the model, with larger models generally exhibiting greater energy reductions when compressed. This underscores the importance of evaluating compression strategies in the context of specific deployment environments rather than assuming universal efficiency gains.

Our findings also demonstrate that the environmental impact of deploying language models cannot be evaluated independently of deployment context. In particular, we observed that a compressed model deployed in a carbon-intensive grid can emit significantly more $CO2eq$ than a full model running in a low-carbon region. This highlights the need to consider not just the efficiency of the model, but also the carbon intensity of the underlying infrastructure.

Further Work

While the current evaluation provides meaningful insights, several areas could be improved or extended in future work. The choice of models and evaluation protocol was reasonably comprehensive, and the results obtained were consistent and informative. However, exploring alternative hardware configurations or sampling settings, and additional compression techniques could help determine to which extent our findings in terms of hardware energy consumption applies.

More importantly, future studies should consider using datasets that more realistically reflect real-world inference workloads when estimating energy consumption. The use of a small or synthetic dataset may not fully capture the complexity and variability of actual user queries. In parallel, employing more robust and fine-grained energy measurement techniques, (e.g. longer-term usage sampling) could provide more accurate assessments of hardware energy consumption under realistic operational conditions.

Finally, incorporating time-dependent grid dynamics into carbon footprint analysis would offer a more realistic and nuanced understanding of how the timing of deployment affects environmental impact.

List of resource persons

Name	Position	Role
Ashwin Ittoo	Full professor at HEC Liège	Academic supervisor
Nelie Laura Makenne	PhD candidate at HEC Liège	Assisted in identifying key literature on model compression methods

Bibliography and references

Ashkboos, S., Croci, M. L., Nascimento, M. G., Hoefler, T., & Hensman, J. (2023). SliceGPT: Compress Large Language Models by Deleting Rows and Columns. *ICLR 2023*. Retrieved October 13, 2025, from https://openreview.net/forum?id=vXxardq6db

Cheng, Y., Wang, D., Zhou, P., & Zhang, P. (2020). *A Survey of Model Compression and Acceleration for Deep Neural Networks*. arXiv. doi:10.48550/arXiv.1710.09282

Courty, B., Schmidt, V., Luccioni, S., Kamal, G., Coutarel, M., Feld, B., . . . MinervaBooks. (2024, May). mlco2/codecarbon: v3.0.2. doi:10.5281/zenodo.11171501

Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *35*, pp. 30318-30332. NeurIPS 2022. Retrieved April 20, 2025, from https://proceedings.neurips.cc/paper_files/paper/2022/hash/c3ba4962c05c49636d4c6206a97e9c8a-Abstract-Conference.html

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv. doi:10.48550/arXiv.2305.14314

Electricity Maps ApS. (2025). Retrieved from Electricity Maps: https://ww2.electricitymaps.com/

Faiz, A., Kaneda, S., Wang, R., Osi, R. C., Sharma, P., Chen, F., & Jiang, L. (2023). LLMCarbon: Modeling the End-to-End Carbon Footprint of Large Language Models. *ICLR 2025*. Retrieved May 5, 2025, from https://openreview.net/forum?id=aIok3ZD9to

Frantar, E., & Alistarh, D. (2022). Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning. *Advances in Neural Information Processing Systems*, *35*, pp. 4475-4488. Retrieved April 17, 2025, from https://proceedings.neurips.cc/paper_files/paper/2022/hash/1caf09c9f4e6b0150b06a07e77f2710c-Abstract-Conference.html

Frantar, E., & Alistarh, D. (2023, March 22). SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. *Proceedings of the 40th International Conference on Machine Learning*, *202*, pp. 10323–10337. Retrieved May 27, 2025

Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2023). OPTQ: Accurate Quantization for Generative Pre-trained Transformers. *ICLR 2023*. Retrieved April 15, 2025, from https://openreview.net/forum?id=tcbBPnfwxS

Gu, Y., Dong, L., Wei, F., & Huang, M. (2023). MiniLLM: Knowledge Distillation of Large Language Models. *ICLR 2025*. Retrieved May 1, 2025, from https://openreview.net/forum?id=5h0qf7IBZZ

Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M. V., Shao, Y. S., Keutzer, K., & Gholami, A. (2024, December 12). KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *Advances in Neural Information Processing Systems, 37*, 1270-1303. Retrieved April 21, 2025, from https://proceedings.neurips.cc/paper_files/paper/2024/hash/028fcbcf85435d39a40c4d61b42c99a4-Abstract-Conference.html

Kaack, L. H., Donti, P. L., Strubell, E., Kamiya, G., Creutzig, F., & Rolnick, D. (2022, June). Aligning artificial intelligence with climate change mitigation. *Nature Climate Change, 12*(6), 518-527. doi:10.1038/s41558-022-01377-7

Kim, J., Lee, J. H., Kim, S., Park, J., Yoo, K. M., Kwon, S. J., & Lee, D. (2023). Memory-Efficient Fine-Tuning of Compressed Large Language Models via sub-4-bit Integer Quantization. *NeurIPS 2023*. Retrieved May 13, 2025, from https://openreview.net/pdf?id=2jUKhUrBxP

Li, S., Chen, J., Chen, Z., Zhang, X., Li, Z., Wang, H., . . . Yan, X. (n.d.). Explanations from Large Language Models Make Small Reasoners Better. *2nd Workshop on Sustainable AI*. Retrieved May 1, 2025, from https://openreview.net/forum?id=rH8ZUcfL9r

Li, Y., Niu, L., Zhang, X., Liu, K., Zhu, J., & Kang, Z. (2024, March 22). *E-Sparse: Boosting the Large Language Model Inference through Entropy-based N:M Sparsity*. arXiv. doi:10.48550/arXiv.2310.15929

Li, Y., Su, H., Shen, X., Li, W., Cao, Z., & Niu, S. (2017, November 11). DailyDialog: A Manually Labelled Multi-turn Dialogue Dataset. doi:10.48550/arXiv.1710.03957

Li, Y., Yu, Y., Liang, C., He, P., Karampatziakis, N., Chen, W., & Zhao, T. (2023). *LoftQ: LoRA-Fine-Tuning-Aware Quantization for Large Language Models*. arXiv. doi:10.48550/arXiv.2310.08659

Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhavy, D., Krishnamoorthi, R., . . . Blankevoort, T. (2024). SpinQuant: LLM Quantization with Learned Rotations. *ICLR 2025*. Retrieved from https://openreview.net/forum?id=ogO6DGE6FZ

Lucioni, A. S., Viguier, S., & Ligozat, A.-L. (2022). *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. arXiv. doi:10.48550/arXiv.2211.02001

Ma, X., & Fang, G. (2023). LLM-Pruner: On the Structural Pruning of Large Language Models. *NeurIPS 2023*. Retrieved April 24, 2024, from https://openreview.net/forum?id=J8Ajf9WfXP

Meta AI. (2025, April 15). *Introducing quantized Llama models with increased speed and a reduced memory footprint*. Retrieved from Meta AI: https://ai.meta.com/blog/meta-llama-quantized-lightweight-models/

ModelCloud.ai and qubitium@modelcloud.ai. (2024). GPTQModel. *GitHub repository*. Retrieved from https://github.com/modelcloud/gptqmodel

NVIDIA. (2025, May 25). *NVIDIA A100 | Tensor Core GPU*. Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

Park, G., Park, B., Kim, M., Lee, S., Kim, J., Kwon, B., . . . Lee, D. (2024). LUT-GEMM: Quantized Matrix Multiplication based on LUTs for Efficient Inference in Large-Scale Generative Language Models. *ICLR 2024*. Retrieved April 15, 2025, from https://openreview.net/forum?id=gLARhFLE0F

Strubell, E., Ganesh, A., & McCallum, A. (2019). *Energy and Policy Considerations for Deep Learning in NLP*. arXiv. doi:10.48550/arXiv.1906.02243

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems*. *30*. Curran Associates, Inc. Retrieved May 5, 2025, from https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Wang, X., Zheng, Y., Wan, Z., & Zhang, W. (2024). SVD-LLM: Truncation-aware Singular Value Decomposition for Large Language Model Compression. *ICLR 2025*. Retrieved May 2, 2025, from https://openreview.net/forum?id=LNYIUouhdt

Xia, M., Gao, T., Zeng, Z., & Chen, D. (2024). *Sheared LLaMA: Accelerating Language Model Pre-training via Structured Pruning*. arXiv. doi:10.48550/arXiv.2310.06694

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., & Han, S. (2023, June 15). SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. *Proceedings of the 40th International Conference on Machine Learning*. *202*, pp. 38087–38099. (ICLM 23). Retrieved April 15, 2025

Xu, Y., Xie, L., Gu, X., Chen, X., Chang, H., Zhang, H., . . . Tian, Q. (2023). *QA-LoRA: Quantization-Aware Low-Rank Adaptation of Large Language Models*. arXiv. doi:10.48550/arXiv.2309.14717

Zhu, X., Li, J., Liu, Y., & Ma, C. (2024, November 27). A Survey on Model Compression for Large Language Models. *Transactions of the Association for Computational Linguistics, 12*, 1556-1577. doi:10.1162/tacl_a_00704

EXECUTIVE SUMMARY

Maximum 300 mots/words

In recent years, model compression techniques have proven highly effective at reducing large language models' storage footprint and accelerating inference. By lowering memory requirements and increasing computational speed, these methods offer a pathway to more energy-efficient large models. However, altering model parameters through compression inherently risks degrading performance. Thus, the trade-off between efficiency gains and potential quality loss remains central when adopting compression strategies.

While the environmental impact of training large models has received growing attention, the effects of compression on hardware energy consumption and related GHG emissions during inference remain largely unexplored.

To address this gap, I conducted an empirical study investigating the effects of compression on both model performance and energy consumption across hardware components—specifically CPU, GPU, and RAM. Four decoder-only transformer models were selected for their academic relevance and maturity: LLaMA-7B, LLaMA-30B, Mistral-7B-v0.3, and Mistral Small 3. Each model was compressed using the OPTQ method at 4-bit precision.

Model performance was evaluated using WikiText-2 perplexity, and MMLU and IFEval accuracy. Energy consumption was measured using CodeCarbon during inference on two high-end hardware configurations. To contextualize these results, I conducted a comparative carbon footprint analysis using four electricity mixes, offering a grid-aware perspective on compression-related emissions in CO_2eq.

The findings show that (1) quantization-induced performance degradation is marginal; compressed models retain nearly all capabilities across benchmarks. (2) Compression can substantially reduce energy use, but the magnitude depends on hardware—one configuration yielded up to 39% savings, while another saw a 26% increase. (3) Environmental impact hinges not only on model and energy use but also on deployment geography: a compressed model on a carbon-intensive grid can emit up to six times more than a full-sized model on a clean grid. Thus, sustainability benefits of compression must be assessed in relation to hardware and geographic context.

MOTS-CLÉS/KEYWORDS: large language models, compression methods, carbon footprint, energy consumption, inference

NOMBRE DE MOTS/WORD COUNT: 15 612

Ecole de Gestion de l'Université de Liège