Quantization Methods for Matrix Multiplication and Efficient Transformers

by

Semyon Savkin

Bachelor of Science in Mathematics and Computer Science and Engineering, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by:	Semyon Savkin Department of Electrical Engineering and Computer Science August 15, 2025
Certified by:	Yury Polyanskiy Professor, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Quantization Methods for Matrix Multiplication and Efficient Transformers

by

Semyon Savkin

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

We study quantization in Machine Learning. First, we introduce NestQuant — a technique for quantization of matrix products and post-training quantization of LLMs. Beyond reducing the memory footprint, quantization accelerates inference, as the primary bottleneck during autoregressive generation is often the memory bandwidth. NestQuant leverages two nested lattices to construct an efficient vector codebook for quantization, along with practical encoding and decoding algorithms. The approach is grounded in recent theoretical work that characterizes the optimal rate–distortion trade-off for matrix products. Empirically, on Llama-3-8B, it reduces the perplexity gap between full-precision and quantized models by more than 55% relative to the current state-of-the-art technique (SpinQuant).

Second, we investigate data-domain quantization for RF signals. We propose a tokenized transformer for source separation that discretizes RF waveforms into learned tokens and operates directly on the resulting sequences, outperforming strong convolutional baselines.

Together, these contributions connect information-theoretic limits with deployable systems: structured vector quantizers accelerate LLM inference and enable competitive discrete representations for RF tasks.

Thesis supervisor: Yury Polyanskiy
Title: Professor

Acknowledgments

First, I am very grateful to my advisor, Prof. Yury Polyanskiy. His help and guidance, as well as the ideas that he proposed, were indispensable for my research progress. I am incredibly lucky to be advised by someone with such deep expertise in the field.

I also want to thank my collaborators: Eitan Porat and Prof. Or Ordentlich in the NestQuant project, and Egor Lifar, Dr. Tejas Jayashankar, Rachana Madhukara, and Prof. Gregory Wornell in the RF transformer project. It was a pleasure to work with them.

In the NestQuant project, I significantly benefited from discussions with Nikita Lazarev, Shang Yang, and Guangxuan Xiao about GPU kernels and the overall quantization algorithm.

Finally, I want to thank my family, my girlfriend, and my friends for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantization is the process of mapping a large set of possible values to a smaller, finite set of discrete values. Quantization techniques have been used for many years in signal processing and data compression. Recently, quantization has also become important in machine learning, where it is needed for fast and low-memory inference of Large Language Models (LLMs).

The performance of LLMs is highly influenced by their size. Having a larger model size results in lower throughput and a need for an accelerator with more available memory. Therefore, being able to quantize models makes it possible to run a more performant model under resource constraints.

In this work, we will focus on post-training quantization (PTQ). PTQ techniques allow storing weights and doing computations in lower precision. The model is trained in full precision, while quantization is applied afterward, with no or little calibration data.

Since memory bandwidth is a bottleneck in LLM inference, reducing precision not only decreases memory consumption but also accelerates inference. During the generation phase, the main operation is the matrix–vector product, where loading the weights from DRAM to compute cores is often the most time-consuming operation. If the weights are quantized, loading them takes less time.

We present a new LLM quantization framework named NestQuant. The NestQuant algorithm is described in Chapters 4-6. NestQuant is a generic drop-in replacement for any matrix multiplication. Its performance for synthetic random Gaussian matrices approaches information-theoretic limits (see Fig. 7.1) and significantly outperforms uniform quantization. Switching from scalar (uniform) quantization to vector quantization introduces some computational overhead (see Chapter 8); however, among vector quantizers, NestQuant is rather economical as it builds upon the Voronoi Codes framework of [1]. We mention that in the presence of activation quantization, it is important to quantize weights properly — an innovation we call QA-LDLQ (see Chapter 5).

In Chapter 9, we consider a problem related to quantization — continuous data tokenization in generative modeling. We demonstrate the benefits of the tokenized representation in the domain of source separation of RF signals.

1.1 Thesis structure

Chapter 2 covers the necessary background for the domain of LLM quantization, and Chapter 3 surveys the current works in this domain, grouping them by the area of proposed contribution.

Chapters 4-8 are primarily based on the paper "NestQuant: Nested Lattice Quantization for Matrix Products and LLMs" [2] published at the 42nd International Conference on Machine Learning (ICML 2025). This was one of the two projects I worked on. Chapter 8 also includes preliminary work on hardware simulation of the vector quantizer used in NestQuant.

Chapter 9 discusses tokenized representations for source separation in the RF domain. It is based on a paper that is currently under review at The Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025), which was the second project in my Master's degree.

Chapter 2

Background

In this chapter, we describe the foundational concepts in the field of LLM and matrix multiplication quantization, as well as core techniques that are used across many algorithms.

2.1 Quantization overview

In general, the goal of quantization is to compress an object into a low-bit representation, minimizing the distortion between the original and reconstructed objects. Let X be the set of possible objects with some associated distribution p_X. For a maximum codebook size s, we aim to find the following components:

$$\begin{array}{ccc} \text{Codebook} & \mathcal{C} & |\mathcal{C}| \leq s \\ \text{Encoder} & f & X \to \mathcal{C} \\ \text{Decoder} & g & \mathcal{C} \to X \end{array}$$

Given a metric $c\colon X \times X \to \mathbb{R}$, we are minimizing the distortion D, which is the expectation of the metric between the original object x and the reconstructed object $g \circ f(x)$:

$$\mathcal{D}(X, c, \mathcal{C}, f, g) := \int_{x \in X} c(x, (g \circ f)(x))\, dP_X$$

Often, we are interested in compressing multiple i.i.d. samples from the distribution. For a finite number of samples n, we can define the average distortion as the mean of the metric values across individual samples:

$$c_n\colon X^n \times X^n \to \mathbb{R}, \quad c_n(\{x_1, \ldots, x_n\}, \{x'_1, \ldots, x'_n\}) = \frac{1}{n} \sum_{i=1}^{n} c(x_i, x'_i)$$

We also parametrize the codebook size through the *rate* R — the number of bits needed to send the compressed representation of one sample: $s = 2^{Rn}$. Note that now our encoder and decoder operate on n-vectors of samples (X^n).

We note that having a larger rate (thus, larger codebook size) enables us to pack codebook elements more densely and achieve a smaller distortion. For a given distribution, there exists the optimal trade-off curve of (s, D) pairs. The function that describes this curve for the

case of encoding i.i.d. samples as their number goes to infinity is called the *rate-distortion function*.

Definition 2.1.1 (adapted from [3])**.** Given a set X with a distribution P_X, as well as a metric c, the rate-distortion function $R(D)$ is defined as:

$$R(D) = \lim_{n\to\infty} \inf_{R\in\mathbb{R}} \{R\colon \exists\, f, g, \mathcal{C}\colon |\mathcal{C}| \le 2^{nR}, \mathcal{D}(X, c_n, \mathcal{C}, f, g) \le D\}$$

The rate-distortion function characterizes the fundamental limits of lossy compression. It serves as a rate lower bound for the evaluation of any practical quantization algorithm: no code can achieve a better rate than the rate given by the rate-distortion function.

2.1.1 Gaussian data

As we will see later, the case of quantizing Gaussian data with quadratic loss is crucial for compressing LLMs, since the inputs to the quantizer can be made approximately Gaussian. In this section, we set $X = \mathbb{R}$, $P_X = \mathcal{N}(0, 1)$, and $c(x, y) = |x - y|^2$, and describe algorithms applicable to such data.

Lloyd's algorithm

One way of quantizing n independent Gaussian inputs is to use a scalar quantizer: a real-valued codebook applied independently to each input. If the size of this codebook is q, we achieve rate $R = \log_2 q$. For each value of q, an optimal codebook for our metric c can be found using Lloyd's algorithm [4], described below.

1. Initialize the codebook elements randomly, $c_1 \le c_2 \le \ldots \le c_q$.
2. Define Voronoi boundaries:

   $$l_i = \begin{cases} -\infty, & i = 1, \\ \frac{c_i + c_{i-1}}{2}, & i > 1, \end{cases} \qquad r_i = \begin{cases} \frac{c_i + c_{i+1}}{2}, & i < q, \\ \infty, & i = q. \end{cases}$$

   Assign all points x such that $l_i \le x < r_i$ to cluster i.
3. Set new $c'_i := \mathbb{E}[x \mid x \in [l_i, r_i]]$.
4. Repeat steps 2 and 3 until convergence.

In practice, this algorithm often converges to the optimal codebook. However, while quantizing each entry independently is intuitive, the performance of such scalar quantizers is suboptimal compared to the rate-distortion frontier, which can be expressed in closed form.

Theoretical bounds

The following theorem connects the rate-distortion function with mutual information:

Theorem 2.1.2 (Rate–Distortion Theorem)**.** *For the setup above,*

$$R(D) = \min_{p(\hat{x}|x):\, E[c(x,\hat{x})]\leq D} I(x;\hat{x})$$

Using this theorem, we obtain a closed-form solution for the rate-distortion function of a Gaussian random variable: $R(D) = -\frac{1}{2}\log_2(D)$. The inverse function relating optimal D to R is $D = 2^{-2R}$. Figure 2.1 compares the optimal rate-distortion curve with the performance of Lloyd's algorithm.

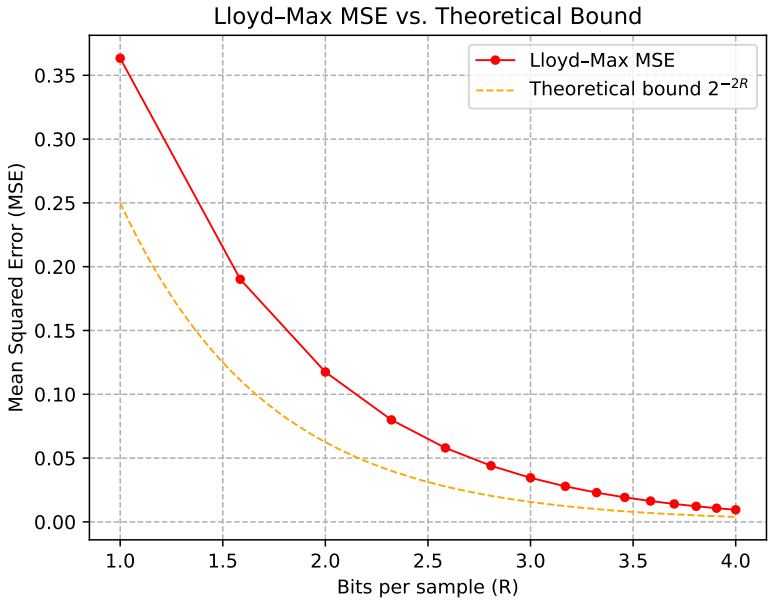


Figure 2.1: Comparison between the theoretical rate-distortion function and the results achieved by Lloyd's algorithm for scalar quantization

Many practical vector quantization methods come close to the theoretical bound. One is trellis quantization [5]; another is lattice quantization [6]. None of the finite-dimensional algorithms can achieve the theoretical bounds, but as the dimensionality of a good lattice or trellis grows to infinity, the achieved rate-distortion function approaches the theoretical optimum.

2.1.2 Matrix multiplication quantization

In the problem of matrix multiplication quantization, we compress two matrices $A \in \mathbb{R}^{a\times n}$ and $B \in \mathbb{R}^{n\times b}$ with the goal of reconstructing only their product, rather than the matrices themselves. Formally, a rate-R quantization scheme consists of two encoders $f_1 \colon \mathbb{R}^{a\times n} \to$

$[2^{Ran}]$ and $f_2 \colon \mathbb{R}^{n\times b} \to [2^{Rnb}]$, and a decoder $g \colon [2^{Ran}] \times [2^{Rnb}] \to \mathbb{R}^{a\times b}$ (where $[k]$ is the set of integers between 0 and $k-1$). The reconstructed matrix product is obtained as:

$$\widehat{AB} = g(f_1(A), f_2(B))$$

The authors of [7] show that if the entries in matrices A and B are independent unit Gaussians, then for any rate-R algorithm we must have:

$$\forall_{i,j} \mathbb{E}\left[\left((\widehat{AB})_{ij} - (AB)_{ij}\right)^2\right] \geq n\Gamma(R), \tag{2.1}$$

where

$$\Gamma(R) = \begin{cases} 1 - \left(1 - (2\cdot 2^{-2R^*} - 2^{-4R^*})\right)\frac{R}{R^*}, & R < R^* \\ 2\cdot 2^{-2R} - 2^{-4R}, & R \geq R^* \end{cases}, \tag{2.2}$$

and $R^* \approx 0.906$ is a solution to the equation $R = \frac{1}{2}\log_2(1 + 4R\ln 2)$.

Since matrix multiplication dominates the computational cost in LLM inference, quantization of matrix multiplication is an important subproblem of LLM quantization.

2.2 Language models overview

The goal of this section is to provide background on large language models and the transformer architecture, and to define what it means to "quantize" a language model across different regimes. We define the framework of autoregressive text generation and describe (a variant of) the transformer architecture.

2.2.1 Autoregressive text generation

Modern LLMs operate on sequences of *tokens* rather than directly on raw text. Let $\{t_1, \ldots, t_m\}$ be a fixed dictionary. Each text s can be decomposed into a token sequence $a_1, \ldots, a_k$, such that concatenating the strings t_{a_i} for $i = 1, 2, \ldots, k$ yields s. The decomposition is deterministic and is obtained by running a certain algorithm (typically Byte-Pair Encoding [8]). The model learns the distribution of token sequences induced by the training corpus by modeling the conditional distribution:

$$p_{a_{k+1}|a_1,a_2,\ldots,a_k}(\cdot|\cdot) \tag{2.3}$$

Given such a model, one can sample new tokens one by one conditioned on all the previous tokens in the sequence. The sampling process starts with some user prompt (that is decomposed into tokens) and ends when a special "end-of-sequence" token is drawn.

In this setting, the user and the LLM alternate turns, with the model generating tokens autoregressively until a special end-of-response token is produced.

2.2.2 Transformer model architecture

The transformer architecture is a parametrized function mapping a sequence of tokens to the conditional distribution of the next token from equation (2.3). We describe the Llama architecture [9]; other LLM transformer architectures are structurally similar to this one.

The transformer has an integer parameter d — the token dimension. First, each token in the sequence is mapped into a d-dimensional vector with an arbitrary embedding function $[m] \to \mathbb{R}^d$. Then, some number of transformer layers is applied to the tokens, where a transformer layer is a parametrized function that takes a sequence of d-dimensional vectors and returns the same number of d-dimensional vectors. The transformer layers have independent parameters. Finally, each of the resulting vectors is projected by the same linear mapping $\mathbb{R}^d \to \mathbb{R}^m$. The last vector (among k vectors corresponding to tokens) contains logits of the probability distribution of the new token, and the probabilities can be obtained by applying the softmax function to the logits.

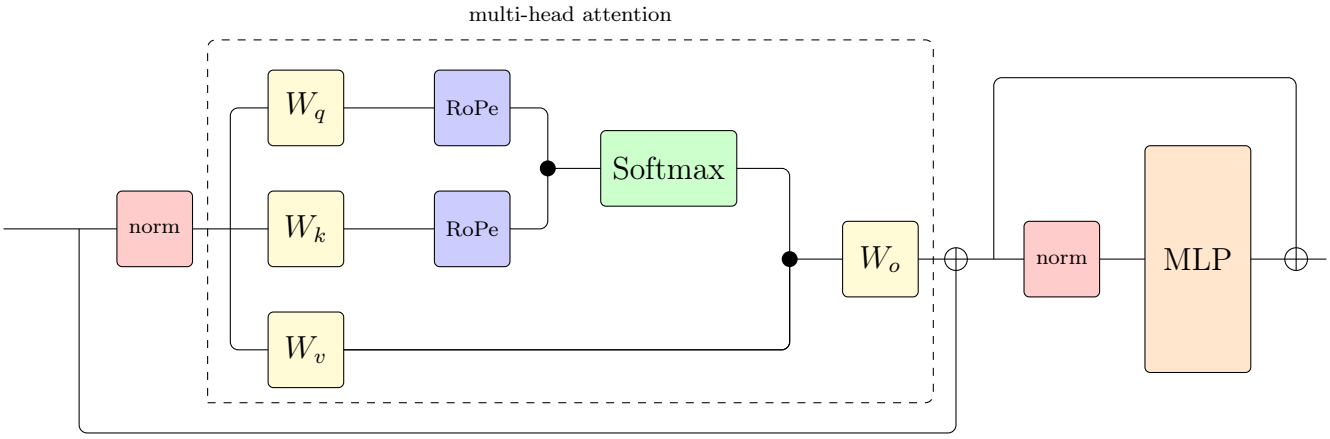


Figure 2.2: The internal architecture of the transformer layer

Now we describe the transformer layer. Its primary component is *multi-head attention*, which is the only place where the vectors corresponding to different tokens interact. The multi-head attention consists of heads, whose outputs are summed to produce the final result. Let d_h be the head dimension. Each head consists of linear projections W_q, W_k, and W_v mapping $\mathbb{R}^d \to \mathbb{R}^{d_h}$; the outputs are referred to as queries, keys, and values. Then, we apply *positional embeddings* to keys and queries — a function on the token vector that depends on the position of the token in the sequence. This is the only component that makes the transformer not permutation-invariant. For each query q, we compute the dot products with the keys that are earlier or equal in the sequence, and apply softmax on the results, obtaining *attention scores* with respect to the previous tokens. The result for each query is the linear combination of values with coefficients taken from the attention scores. We apply the final projection W_o ($\mathbb{R}^{d_h} \to \mathbb{R}^d$) to get the head's result for each token.

Besides the attention, transformer layers also have an MLP layer (consisting of several linear layers and non-linearity), per-token norms, and residual connections. In total, each transformer layer has 7 linear layers: W_q, W_k, W_v, W_o from attention, and W_{up}, W_{gate}, and W_{down} from the MLP.

From an efficiency perspective, we note:

1. In order to evaluate the next-token prediction for a sequence, we need the final vector for the last token. But we cannot just run the transformer only on the last token, since the attention result depends on the previous tokens. However, we do not need to do a full pass on the whole sequence either. The only information we need from other tokens is the key and value vectors at each of the transformer layers. We store them (this technique is called KV caching). With the keys and values of previous tokens cached, we can run inference only on newer tokens.

2. There are two different modes of operation in LLMs. When processing a token sequence from the user, we can run all linear sublayers on all the tokens together, thus obtaining dense matrix multiplication operations. However, when the model generates the response, it generates tokens one by one; thus, the primary operation will be matrix-vector multiplication. We refer to the first mode as the "prefill" stage and to the second mode as the "generation" stage.

2.2.3 Quantization of transformer LLMs

Unquantized LLM inference is typically performed in 16-bit precision (the numbers are stored in either fp16 or bf16 format). However, for some layers that require higher precision (norms, softmax, MLP nonlinearity), the vectors may be upcast to fp32.

The same is true for low-bit quantization: quantizing the model to R bits does not mean that all computations are done in R-bit numbers. In fact, there are three most popular quantization regimes where most methods are compared:

1. **Weight-only quantization.** For some $R < 16$, all weights of linear layers in the transformers are stored in R-bit format. Note that this does not affect the initial embedding layer and the final linear projection. Key properties of this regime include:

   - The weights are stored in compressed format in DRAM, thus reducing memory usage of the model.
   - When running the inference, the weights need to be dequantized.
   - Reduces the memory transfer between DRAM and compute cores (where the weights are dequantized), thus achieves speedup.
   - The encoding algorithm in the quantizer can be very slow, because the quantization is done offline. The decoding needs to be fast, because it's done during inference.
   - Especially beneficial in the generation stage, where loading weights from DRAM is the primary bottleneck.

2. **KV-cache quantization.** We store the vectors in the KV-cache in low-bit format.

   - Has similar implications as weight quantization: less memory usage, faster queries to the KV-cache, especially in the generation phase.

- Unlike weight quantization, requires reasonably fast encoding, since we need to apply it to each token during inference.

3. **Activation quantization.** We refer to the inputs to linear layers as "activations." Activation quantization means that every input to every linear layer (aside from the last projection) needs to be compressed into fewer bits. Note that the output can be computed in full precision.

   - Has an impact on the runtime (since fewer bits will need to be moved between memory buffers) during the prefill stage.
   - Minimal impact on memory usage.
   - Generally, activations are the hardest to quantize to the given number of bits without a significant accuracy drop.

We note that the attention scores are not quantized in any of these regimes. Typically, evaluations are conducted in three settings: weight-only, weight+KV-cache ("W+KV"), and full quantization of weights, KV-cache, and activations ("W+KV+A").

2.3 LLM quantization techniques

2.3.1 Uniform scalar quantization

Uniform scaling quantization is the simplest quantization algorithm. In this algorithm, the quantization is independent per entry, and the codebook points form an arithmetic sequence.

We will use the following convention to describe a linear layer. Let n be the number of input features, m be the number of output features, W be the matrix describing the weight $(m \times n)$, and X be the matrix describing the input $(b \times n)$, where b is the batch size. The output of the layer is XW^T (size $b \times m$).

The uniform scalar quantization works independently row-by-row. Suppose we want to quantize a row with maximum absolute value C into R bits. We use a quantization range of integers in $[-A, A]$ for $A = 2^{R-1} - 1$ (we sacrifice one value for convenience). Then, the quantized representation of a coordinate x in this row will be $\text{round}\left(\frac{xA}{C}\right)$, while the dequantization function maps $y \to \frac{yC}{A}$.

Note that we can multiply the quantized representations directly to get an estimate for the matrix product. Thus, we can use integer multipliers, which work faster on current GPUs. The adjustment for the scaling coefficients in the dequantization function is asymptotically less significant.

While this algorithm does not achieve the best rate-distortion function, it is used in practice due to its efficiency.

2.3.2 Outliers and random rotation

One common problem when quantizing activations arises from outliers. When we scale the values to put them in the quantizable range, the outliers in a token make the corresponding

scaling coefficient very large. After we scale down by a large coefficient, the quantization error for non-outlier values becomes large relative to the values themselves. This makes naive 4-bit quantization of LLMs not feasible.

To mitigate the issue with outliers, we choose an orthogonal transformation U and apply it to both the weight and the activations (i.e., $X \to XU$, $W \to WU$). This transformation of a linear layer is fully equivalent:

$$(XU)(WU^T) = XUU^TW^T = XW^T$$

The goal is to reduce the impact of outliers. In particular, if we sample U uniformly from the Haar measure on orthogonal matrices, the distribution of each row of X will be approximately Gaussian. The exact mathematical fact is:

Lemma 2.3.1. *Let x_n be any sequence of deterministic vectors of dimension n, and U_n be a random orthogonal $n \times n$ matrix. Then, for any fixed k, the distribution of the first k coordinates of $\sqrt{n}\frac{U_n x_n}{|x_n|}$ converges to $\mathcal{N}(0, I_k)$.*

Proof. Note that the distribution of $\sqrt{n}\frac{U_n x_n}{|x_n|}$ is uniform over the n-dimensional sphere of radius $\sqrt{n}$. Let $Z_n \sim \mathcal{N}(0, I_n)$. The distribution of $\frac{\sqrt{n}Z_n}{|Z_n|}$ is also uniform over the same sphere, so we can prove convergence in distribution for the first k coordinates of $\frac{\sqrt{n}Z_n}{|Z_n|} = \frac{Z_n}{|Z_n|/\sqrt{n}}$. Note that $\frac{|Z_n|^2}{n} = \sum_{i=1}^{n} \frac{(Z_n)_i}{n}$ converges in probability to 1, thus the denominator converges in probability to 1. The first k coordinates of Z_n are always distributed as $\mathcal{N}(0, I_k)$, and since the denominator converges in probability to 1, the fraction converges in distribution to $\mathcal{N}(0, I_k)$ by Slutsky's theorem. □

If the data we quantize is Gaussian, the issue of outliers is no longer a problem and we can use the theory for Gaussian quantization (see subsection 2.1.1). However, we cannot generate an arbitrary matrix, since the activations X need to be multiplied by U during inference, which takes just as much as multiplying by the original weight. There are two possible approaches to tackle this problem:

- For some layers, it is possible to incorporate the multiplication by U into the previous weight. There is a set of layer transformations that make activations before most of the linear layers randomly rotated; however, it is not possible to achieve this for all layers.
- Instead of multiplying by a random orthogonal matrix, we multiply by a random Hadamard matrix, which can be done efficiently. While it is harder to explain why the Gaussian assumption is applicable in this case, using a Hadamard matrix removes outliers quite efficiently in practice. In particular, when n is a power of 2, multiplication of a vector by an $n \times n$ Hadamard matrix obtained by a construction can be done in $O(n \log n)$.

2.3.3 GPTQ

Here we describe a weight quantization idea from [10] that became popular in recent LLM quantization works. Consider weight W, and let the reconstructed weight be W'. A natural

objective to minimize is the MSE between the quantized and original weight: $|W - W'|^2$. However, note that the weight encodes a linear transform. If we model the input x of this linear transform as a random vector, we can choose the objective to be the MSE between outputs of this transform:

$$\mathcal{L}(W') = \mathbb{E}[|W'x - Wx|^2]$$

Let $\Delta W = W - W'$ and $H = E[xx^T]$. Then:

$$\mathcal{L}(W') = \mathbb{E}[|W'x - Wx|^2] = \mathbb{E}[\text{tr}(\Delta W)x((\Delta W)x)^T] = \\ \mathbb{E}[\text{tr}(\Delta W)xx^T(\Delta W)^T] = \mathbb{E}[\text{tr}(\Delta W)H(\Delta W)^T]$$

It is unclear how to optimize this objective. The authors of GPTQ propose the following heuristic: we quantize the weight column-by-column, letting U be an upper triangular matrix of *linear feedback* from already quantized columns to new columns. Specifically, for $i < j$, $U_{i,j}$ is the feedback from the i-th to j-th column: when quantizing the weight at position (r, j), instead of quantizing W_{rj}, we quantize $W_{rj} + \sum_{i=1}^{j-1}(\Delta W)_{ri}U_{ij}$. If $\mathcal{Q}$ is the quantization operator, the following equation holds:

$$W' = \mathcal{Q}(W + (\Delta W)U) \Rightarrow -\Delta W = \mathcal{Q}(W + (\Delta W)U) - W$$

Assume now that quantization just adds quantization error to the data, modeled as random Gaussian with mean 0 and variance τ (thus, quantization error is $\sqrt{\tau}Z$ where $Z_{ij} \sim \mathcal{N}(0, 1)$). Then:

$$-\Delta W = W + (\Delta W)U + \sqrt{\tau}Z - W = (\Delta W)U + \sqrt{\tau}Z \Rightarrow \Delta W = -\sqrt{\tau}Z(U + I)^{-1}$$

We decompose $H = PDP^T$, where P is unit upper triangular and D is diagonal (analogous to LDL decomposition). Then, the expectation of our loss is:

$$\mathcal{L} = \mathbb{E}[\text{tr}\,\sqrt{\tau}Z(U + I)^{-1}PDP^T(\sqrt{\tau}Z(U + I)^{-1})^T] = \\ \tau\mathbb{E}[\text{tr}\,Z(U + I)^{-1}PDP^T(U + I)^{-T}Z^T] = \tau\,\text{tr}(U + I)^{-1}PDP^T(U + I)^{-T}$$

H is positive semidefinite, so D has non-negative entries. Therefore, to minimize the objective we want $(U + I)^{-1}P = I$ (since it must be a unit upper triangular matrix, and $\text{tr}\,ADA^T$ is minimized at $A = I$ for unit upper triangular A). We can achieve this by setting $U = P - I$. The GPTQ algorithm is:

- Compute the matrix H. To do this, run the model on a certain *calibration dataset* and get the statistics $E[xx^T]$.
- Compute the decomposition $H = PDP^T$, set $U = P - I$.
- Run the quantization with column feedback given by this matrix U.

The GPTQ algorithm significantly optimizes its objective compared to naively optimizing MSE. While expected, optimizing the GPTQ objective results in better downstream performance than optimizing the MSE between unquantized and reconstructed weights.

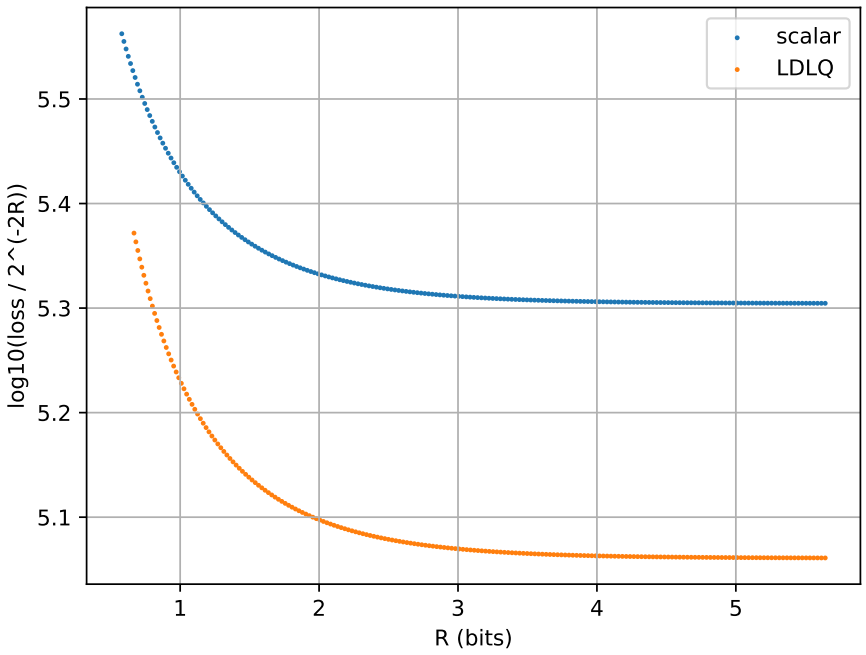


Figure 2.3: Relative performance of GPTQ (LDLQ) with pointwise quantization for the w1 layer of the 15th transformer block of Llama-3-8B. The loss is proportional to the expected MSE between layer outputs. The x-axis is the rate R in bits, and the y-axis shows $\log_{10} \frac{\mathcal{L}}{2^{-2R}}$. We see that GPTQ consistently outperforms pointwise quantization by a factor of ~ 1.5.

Chapter 3

Related works

3.1 Outlier avoidance

3.1.1 LLM.int8()

Before the rotational methods described in section 2.3.2 became popular, LLM quantization methods were designed to deal with outliers in different ways. One such method is described in [11]. The authors note that in linear layers of LLMs the weights do not contain many outliers, while the activation tokens contain outliers only in a small subset of channels.

Recall that in our setup the linear layer has n input units, m output units, the weight matrix is of shape $m \times n$, and the input matrix has shape $b \times n$, where b is the batch/sequence dimension. By rearranging the input dimension, we can get the decomposition $X = (X_l\ X_h)$, $W = (W_l\ W_h)$, where X_l does not have outliers and W_h has a much smaller number of columns than X_l. Then, the output to the layer XW^T can be decomposed as:

$$XW^T = (X_l\ X_h)\begin{pmatrix} W_l \\ W_h \end{pmatrix} = X_l W_l + X_h W_h$$

The key idea of the method is to compute $X_l W_l$ in low precision (8 bits), but compute $X_h W_h$ in high precision (16 bits). The authors claim that the fraction of channels with outliers is no more than 0.1%, so the overhead for keeping the outliers in high precision is not large. This method enables 8-bit quantization of weights and activations with negligible performance drop.

3.1.2 SmoothQuant

Another method of dealing with outliers is per-channel scaling coefficients, utilized by SmoothQuant [12]. Let s be a sequence of length n. Then, we can rewrite:

$$XW^T = (X\mathrm{diag}(s)^{-1})(W\mathrm{diag}(s))^T \tag{3.1}$$

We note that weights W generally do not have outliers. To reduce the significance of outliers in X, we set:

$$s_i = \sqrt{\frac{\max_r X_{ri}}{\max_r W_{ri}}}$$

These coefficients "balance" the outliers in X with the smoothness of W, making these matrices similarly challenging to quantize. However, this method can mostly be applied for 8-bit quantization and has noticeable performance drop when doing lower bitrate compression.

3.1.3 AWQ

The per-channel scaling can also benefit weight-only quantization, as shown by [13]. The authors develop a heuristic for the importance of the weight (input) channel, which is based on the average magnitude of the input activation in this channel. By setting $s_i > 1$ for important channels and $s_i = 1$ on other channels in decomposition 3.1, we reduce the error on these important channels at the cost of potentially increasing the error on other channels. This is because when quantizing row r of weight, the vector we are quantizing (using uniform scalar quantization) contains numbers $W_{r1}s_1, \ldots, W_{rn}s_n$. The quantization error depends on the bit rate and the maximum of the magnitudes among row elements. If $s_i \neq 1$, this error will be scaled down by s_i; however, setting large s_i can make per-row maximums larger and jeopardize the quantization of other channels.

The authors of [13] propose to calculate the average magnitude of per-channel activations $b_1, \ldots, b_n$, and set $s_i = b_i^\alpha$, and perform a grid search over α with respect to end-to-end performance. This technique enables extreme low-bit compression of weights, with a perplexity gap from unquantized Llama-2-7B of 0.13 for 4-bit compression and 0.77 for 3-bit compression.

3.2 Optimized rotations

The idea of using random/orthogonal rotations for suppressing outliers (described in subsection 2.3.2) has been used in QuIP [14] for weight-only quantization and QuaRot [15] for W+KV+A quantization. QuIP uses random orthogonal matrices that are merged with the weights, while QuaRot uses random Hadamard matrices applied online. However, some approaches go further, using the rotation matrices as one of the optimization parameters. We describe these methods in this section.

3.2.1 SpinQuant

In SpinQuant [16], there are four rotation matrices that parametrize the transformation of the network: R_1, R_2, R_3, and R_4. The matrices R_1 and R_2 are fused into weights, so they can be arbitrary orthogonal matrices. The matrices R_3 and R_4 need to be applied to the activations online, so they need to admit special structure (the authors use Hadamard matrices).

The main idea of the paper is to use gradient descent to optimize R_1 and R_2. However, the optimization needs to be constrained to orthogonal R_1 and R_2. To do this, the authors employ Cayley SGD.

The gradient descent operates on full-precision weights and computes $\operatorname{argmin}_{R_1,R_2} \mathcal{L}_Q(R_1, R_2)$, where $\mathcal{L}_Q$ is the end-to-end cross-entropy loss of the entire network. The question is how to perform gradient descent over the Stiefel manifold of orthogonal matrices. The following

formulas describe one update step from an old matrix R to a new matrix R':

$$
\begin{aligned}
G &= \nabla_R \mathcal{L}_Q \\
\hat{G} &= GR^T - \frac{1}{2} RR^T GR^T \\
Y &= \hat{G} - \hat{G}^T \\
R &\to \left(I - \frac{\alpha}{2}\right)^{-1} \left(I + \frac{\alpha}{2} Y\right) R
\end{aligned}
$$

After the rotations are optimized, the method also applies GPTQ for the quantization of the weights.

3.2.2 FlatQuant

FlatQuant [17] optimizes the transformation but drops the condition of it being orthogonal. If P is an arbitrary transformation, observe that $XW^T = (XP)(WP^{-T})^T$. So, the MSE between outputs $Y = XW^T$ becomes:

$$|Y - \hat{Y}|^2 = |Y - \mathcal{Q}(XP)\mathcal{Q}(WP^{-T})|^2 \tag{3.2}$$

We can optimize this objective directly over arbitrary P using a calibration dataset of X. However, an arbitrary P would be impractical: while P^{-T} can be merged with W offline, multiplying the activation X by P will not be efficient at runtime. So, the authors choose P to be represented as a Kronecker product of two matrices P_1 and P_2 of size $O(\sqrt{n})$, and P_1 and P_2 are optimized via gradient descent with loss from equation 3.2. The multiplication of a vector by a matrix $P = P_1 \otimes P_2$ can be done at runtime in $O(n\sqrt{n})$, which is asymptotically faster than multiplication by the (quantized) weight.

Both FlatQuant and SpinQuant target 4-bit quantization of weights, KV cache, and activations. FlatQuant achieves perplexity 6.90 on Llama-3-8B, while SpinQuant achieves 7.3. The perplexity of the unquantized model is 6.14.

3.3 Using vector quantization

The works that we have mentioned so far use uniform scalar quantization (see Section 2.3.1), which is a suboptimal method for compressing Gaussian data. Some LLM quantization works use more sophisticated vector quantization methods. Most such methods have complex encoding procedures but simple decoding. Because of this, these methods are used mostly for weight-only quantization.

3.3.1 QuIP#

QuIP# [18] is a weight-only quantization method targeting 2-, 3-, and 4-bit regimes. One of its innovations is a codebook named E8P. This codebook quantizes 8-dimensional vectors and has size 2^{16} (so the bitrate is 2 bits per dimension). The codebook is constructed in the following way:

- Let $D_8 = \{z \in \mathbb{Z}^8 \mid z_1 + \ldots + z_8 \text{ is even}\}$.
- Construct a set S of size $2^8 = 256$ from elements of $D_8 + \frac{1}{2}$ with positive signs and norm $\leq \sqrt{10}$, as well as 29 extra elements with norm $\sqrt{12}$.
- Let S' contain all elements of S with arbitrary $\pm$ signs, such that the number of $-$ signs is even.
- The final codebook is $S' + \frac{1}{4} \cup S' - \frac{1}{4}$.

Thus, the size of the codebook is 2^{16}. The authors describe an efficient decoding algorithm from 16 bits to the corresponding codebook element, requiring only a lookup table of size 2^8: the first 8 bits encode an element of S, the next 7 bits encode the signs (the last sign is inferred from parity), and the final bit encodes whether to add or subtract $\frac{1}{4}$.

This codebook achieves good distortion and outperforms uniform scalar quantization. It can be extended to 3- and 4-bit rates via Residual Vector Quantization (RVQ) [19].

Another idea in QuIP# is a fine-tuning method that works in two stages:

- Before quantizing a layer, fine-tune it to compensate for the error introduced by previously quantized layers.
- After all layers are quantized, fine-tune the set of non-quantized parameters.

3.3.2 AQLM

AQLM is another work concurrent to QuIP# that quantizes vectors as sums of codebook entries. It also targets weight-only quantization and achieves better performance than QuIP#. AQLM uses a vector quantization method that represents vectors as sums of codebook elements.

Let us have a set of d-dimensional vectors to quantize, and m codebooks $C_1, \ldots, C_m$, each containing 2^B vectors. Using Bm bits in the representation p, we can represent a vector as $v = \sum_i C_{i,p_i}$ — i.e., our vector is encoded as a sum of codebook entries. A natural question is how to find good codebooks for a given set of vectors.

A good initialization strategy is as follows: First, run the K-means algorithm on the vectors to obtain codebook C_1. Then, subtract the cluster center from each vector and run K-means on the residuals to obtain C_2. Repeating this process on residuals yields the remaining codebooks. However, we can further optimize them.

For encoding, we can generalize RVQ to beam search: we keep the k best candidates for approximating our vector as a sum of elements from the first l codebooks, then add a new codebook by considering all possible new candidates and selecting the k best again. This produces the code for each vector.

After finding the codes, we can update the codebooks using simple least-squares optimization to minimize the reconstruction MSE loss. We alternate between (beam search) and (codebook update) until convergence.

3.3.3 QTIP

QTIP [20] uses a high-dimensional trellis code. This code has very slow encoding but fast decoding, and as the dimension of trellis vector quantizers grows, the performance approaches the theoretical rate–distortion bound.

A trellis is a directed graph on 2^L nodes, where each node has exactly 2^R incoming edges and 2^R outgoing edges. Each node corresponds to a value. We define a mapping between $L + (n-1)R$-bit sequences and sequences of n real numbers: the first L bits encode the initial node, and the next $(n-1)R$ bits encode the edge IDs for transitions. Then, a sequence of n nodes is mapped to a sequence of n real numbers.

The closest sequence can be found via dynamic programming (too slow for runtime but fine for weights). Decoding can be done in $O(n)$ by following the edges. However, storing the trellis explicitly is infeasible, as it would not fit in cache.

The authors propose two tricks for efficient decoding:

- The trellis structure is based on bit shifts: if the current node ID is represented by L bits $v_0 v_1 \dots v_{L-1}$ and the edge is represented by k bits $e_0 \dots e_{k-1}$, then the new node is $v_k v_{k+1} \dots v_{L-1} e_0 \dots e_{k-1}$.
- The value function of a node is a pseudorandom function of its ID, computable in just a few instructions.

QTIP outperforms QuIP# and AQLM due to its highly efficient, high-dimensional codebook.

3.4 A better objective function

The works mentioned above use the objective from GPTQ (see subsection 2.3.3). Some quantization methods gain an advantage by using a better objective function. In particular, the GPTQ objective is just MSE between layer outputs, so it's constrained to only one layer, while some objective functions can leverage inter-layer connections.

In theory, we are interested in using a second-order approximation of our final loss with respect to the weights. However, the full information would have to contain the second partial derivative with respect to all pairs of weights. Since the number of weights is nm, this would be an $nm \times nm$ matrix, which is infeasible to store for real LLMs. If this matrix was computed (we call it F), the objective becomes:

$$\operatorname{vec}(\Delta W) F \ \operatorname{vec}(\Delta W)^T \tag{3.3}$$

The following two works explore the possibility of using some approximations of this matrix which are more computationally feasible.

3.4.1 GuidedQuant

GuidedQuant [21] proposes to use weighted MSE between layer outputs as the objective. The weight depends on the squared gradient of the final loss with respect to the output neuron. The loss formula is:

$$\mathcal{L} = \mathbb{E}\left[\sum_{i=1}^{m}(X(\Delta W_i)^T)^2\left(\frac{\partial l}{\partial z_i}\right)^2\right]$$

In this formula, ΔW_i is the error in the i-th row of the weight matrix, $\frac{\partial l}{\partial z_i}$ is the derivative of the final loss with respect to the i-th output unit.

One can show that minimizing this loss is equivalent to minimizing loss 3.3, but with elements of F corresponding to pairs of weights not in the same row set to zero.

Note that minimizing this loss is still computationally infeasible. When we are minimizing the regular GPTQ loss, the only information we need from the activation distribution is the $n \times n$ matrix $H = E[xx^T]$. However, since the coefficients for each x are different in each output channel, we now need an $n \times n$-size H matrix for each of the rows of W, which is of size $O(n^2 m)$, which is too large to store. Instead, the authors group the output units into g groups (by naively placing consecutive channels into each group). Then, we average the squared gradient of the loss with respect to the output in each group, thus requiring only g $n \times n$ matrices to be stored. If the decomposition of the weight matrix into groups is $W = \begin{pmatrix} W_1 \\ W_2 \\ \dots \\ W_g \end{pmatrix}$, the loss can be expressed as:

$$\sum_{i=1}^{g}(\Delta W)_i H_i (\Delta W)_i^T$$

This method can be paired with any codebook, as well as an optimization algorithm for the loss of the form $(\Delta W)H(\Delta W)^T$ (such as GPTQ). The authors found that this per-group loss benefits the final performance of quantization.

3.4.2 YAQA

In YAQA [22], another approach to approximating F is chosen. The authors decompose F as $H_r \otimes H_c$, where H_r is an $m \times m$ matrix capturing mixed partial derivatives with respect to pairs of rows, and H_c is an $n \times n$ matrix capturing mixed partial derivatives with respect to pairs of columns.

To optimize the objective induced by this approximation of F, the authors use a two-dimensional generalization of GPTQ. In GPTQ, for every pair of columns (i, j) $(i < j)$ we have a feedback matrix U from the error in column i to the value we are quantizing in column j. In the version of GPTQ from [22], there are two feedback matrices: U^r for rows and U^c for columns. When quantizing the value at position (i, j), all the other values in the rectangle $(0, 0) \dots (i, j)$ are already quantized, and the feedback from the error at position (i', j') in this rectangle is $U^r_{i'i} U^c_{j'j}$ (if $i = i'$ or $j = j'$, we set the corresponding value of U to be 1, even though the matrices are upper triangular). In other words, the following equation holds ($\mathcal{Q}$ is the quantization operator):

$$\hat{W} = \mathcal{Q}(W + (U^r)^T(\Delta W)U^c + (\Delta W)U^c + (U^r)^T(\Delta W))$$

Similar to GuidedQuant, this objective outperforms GPTQ for various design choices of the quantizer.

3.5 Our work

In the next chapters, we will present our quantization scheme named NestQuant. It has contributions across two dimensions. Its primary component is a vector quantization codebook based on Voronoi codes [1]. Our codebook is competitive with other weight-only quantization methods presented in 3.3. But it has an additional property: both the encoding and decoding algorithms are simple and can be done at runtime. Therefore, NestQuant can be used for KV cache quantization and activation quantization. In these setups, our method achieves state-of-the-art results for 4-bit quantization. NestQuant's codebook can also be built in a systematic manner for any rate $R = \log_2(q)$ where q is an integer.

Another contribution of NestQuant is a modified GPTQ objective function that takes into account the quantization noise of activations (QA-LDLQ). Using this objective function is particularly important for quantizing LLMs that contain layers with a significant *amplification ratio*, such as Llama-3-70B.

The description of NestQuant is structured as follows:

- Chapter 4 describes the NestQuant codebook.
- Chapter 5 contains the motivation and details of the QA-LDLQ objective and optimization method.
- Chapter 6 gives an overview of the NestQuant method with all the pieces put together.
- Chapter 7 summarizes the experimental results of NestQuant, as well as ablation studies.
- Chapter 8 shows the steps necessary to make NestQuant run fast (GPU kernels and hardware simulations).

Chapter 4

Codebook design

In this chapter, we describe a vector codebook that performs well on Gaussian source data and has efficient encoding and decoding algorithms. We can get near-Gaussian data by applying rotations to the weights and activations, as described in subsection 2.3.2.

4.1 Motivation for the codebook choice

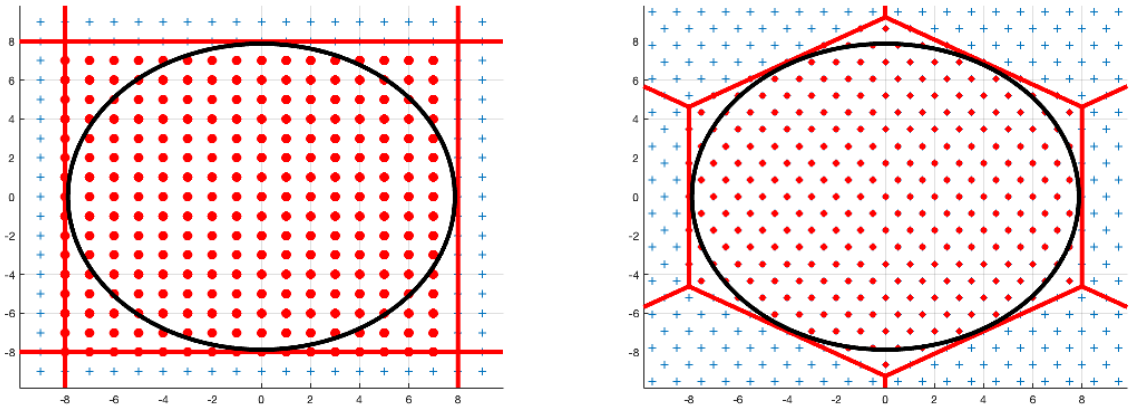


Figure 4.1: Demonstrating the advantage of NestQuant in 2D. Typical weights and activations are vectors inside the black circle. Uniform quantization wastes about 32% of allocated bitstrings for vectors outside of the circle, while nested hexagonal lattices only waste 15% (explicitly enumerating points inside the circle to avoid the waste is too slow to do at runtime). This allows NestQuant to use a finer grid while quantizing to the same rate R. The gain becomes much more dramatic in higher dimensions.

Uniform scalar quantization with L_∞ scaling, as described in section 2.3.1, is suboptimal for two reasons. First, that uniform quantization induces error that is distributed uniformly on the small cube, which is suboptimal from the MSE point of view. The second reason, much more serious, is known as the shaping gain and is demonstrated in Fig. 4.1. When

entries of the vector are Gaussian, it will typically lie inside the black circle. Thus those grid elements outside of it will almost never be used, wasting bitspace.

Instead, we use normalization by the L_2-norm (see Algorithm 4) and then use points inside the Voronoi region of a Gosset lattice, as shown in Fig. 4.1 (right), which results in far fewer wasted bitstrings for rare vectors, thus allowing us to use finer grids.

Lattices

A lattice $\Lambda \subset \mathbb{R}^d$ is a discrete subgroup of $\mathbb{R}^d$. Any lattice $\Lambda \subset \mathbb{R}^d$ has a (non-unique) generating matrix $G \in \mathbb{R}^{d\times d}$, such that $\Lambda = G\mathbb{Z}^d$ (that is, Λ is the integer span of the columns of G). We define the nearest neighbor quantizer $Q_\Lambda : \mathbb{R}^d \to \Lambda$ as

$$Q_\Lambda(x) = \operatorname*{argmin}_{\lambda\in\Lambda} \|x - \lambda\|, \tag{4.1}$$

where ties are broken arbitrarily, but in a systematic manner. The Voronoi region $\mathcal{V}_\Lambda$ is defined as the set of all points in $\mathbb{R}^d$ that are closer to 0 than to any other lattice point:

$$\mathcal{V}_\Lambda = \left\{x \in \mathbb{R}^d \ : \ Q_\Lambda(x) = 0\right\}. \tag{4.2}$$

The covolume of a lattice is defined as $\operatorname{covol}(\Lambda) = |\det G| = \operatorname{vol}(\mathcal{V}_\Lambda)$. We say that a lattice $\Lambda_c \subset \mathbb{R}^d$ is *nested* in the lattice $\Lambda_f \subset \mathbb{R}^d$ if $\Lambda_c \subset \Lambda_f$. Note that for any integer $q \geq 2$ we have that $q\Lambda \subset \Lambda$, and that $\Lambda/q\Lambda \cong (\mathbb{Z}/q\mathbb{Z})^d$. For an introduction to lattices and their role in quantization, see [23].

Lattice quantizer design

In a lattice quantizer, the codebook elements are placed at the lattice points. We cannot choose the entire lattice Λ as a codebook, since its size is infinite. So, we need to choose a region S, and the codebook will be the intersection $\Lambda \cap S$.

Granular and overload error

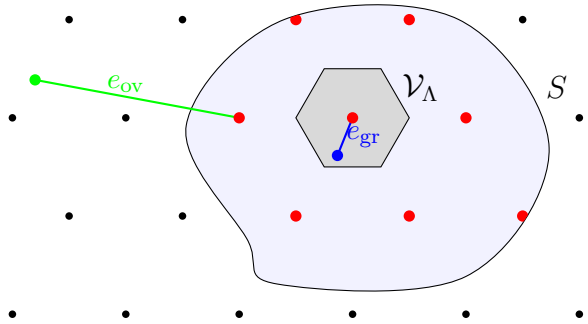


Figure 4.2: The blue point experiences granular quantization error, while the green point has an overload error

There are two different sources of errors in lattice quantizers. The first is called *granular* quantization error e_{gr}, and it occurs when the closest lattice element to the point we are quantizing is in S. Otherwise, we encounter an *overload* error e_{ov}, which is typically larger in magnitude than granular errors. These two errors are illustrated in figure 4.2.

The granular error is related to the second moment of the lattice Voronoi region. A common way to measure the granular error corresponding to a lattice $\Lambda \subset \mathbb{R}^d$ is via the normalized second moment (NSM) defined as

$$G(\Lambda) = \frac{1}{\mathrm{vol}(\mathcal{V}_\Lambda)^{1+\frac{2}{d}}} \frac{1}{d} \int_{x \in \mathcal{V}_\Lambda} \|x\|^2 dx. \tag{4.3}$$

This quantity corresponds to the MSE when Λ is normalized to have unit covolume and is then used as a quantizer. It is well known that the optimal (smallest) NSM among all lattices in $\mathbb{R}^d$ approaches $\frac{1}{2\pi e}$ from above as d increases [23]. Furthermore, for $d = 1$ we get $G(\mathbb{Z}) = \frac{1}{12}$. Consequently, in terms of granular error, by replacing the simple scalar quantizer based on $\mathbb{Z}$ with high-dimensional lattices, we can already gain a factor of $\frac{2\pi e}{12} \approx 1.42329$ in performance (the Gosset lattice achieves 1.16 gain).

Generally speaking, in order to achieve a small quantization error, one must keep the probability of overload very small. This can be achieved by scaling up the codebook to $\beta\mathcal{C} = \beta\Lambda \cap \beta\mathcal{S}$ with a large enough $\beta > 0$ such that overload becomes very rare. However, increasing β also increases the squared granular error by a factor of β^2. Thus, one would like to use the smallest possible β for which overload is rare. In order to allow for smaller β, we would like to choose $\mathcal{S} \subset \mathbb{R}^n$ such that $\beta\mathcal{S}$ captures as much Gaussian mass as possible.

Denote by $\mu = \mathcal{N}(0, I_d)$ the standard Gaussian measure. Since we need $2^{dR} = |\Lambda \cap \mathcal{S}| \approx \frac{\mathrm{vol}(\mathcal{S})}{\mathrm{covol}(\Lambda)}$, a good shaping region $\mathcal{S}$ maximizes $\mu(\mathcal{S})$, which in turn minimizes the overload probability that is approximated by $1 - \mu(\mathcal{S})$, under a volume constraint. Clearly, the optimal $\mathcal{S}$ under this criterion is $r\mathcal{B}$ where $\mathcal{B} = \{x \in \mathbb{R}^d \ : \ \|x\| \leq r_{\mathrm{eff}}(1)\}$ is a Euclidean ball with radius $r_{\mathrm{eff}}(1)$ chosen such that $\mathrm{vol}(\mathcal{B}) = 1$, and r is chosen such that $\mathrm{vol}(r\mathcal{B}) = r^d$ satisfies the required volume constraint. Unfortunately, for $d > 1$ the codebook $\mathcal{C} = \Lambda \cap r\mathcal{B}$ loses much of the lattice structure, and does not allow efficient enumeration, and consequently encoding and decoding require using a lookup table (LUT).

Voronoi codes

In Voronoi codes [1] the same lattice Λ is used for both quantization and shaping. In particular, the shaping region is taken as $\mathcal{S} = q\mathcal{V}_\Lambda$, where $q = 2^R$ is an integer. As elaborated below, if $Q_\Lambda(x)$ admits an efficient implementation, one can efficiently perform encoding and decoding to the codebook $\mathcal{C} = \Lambda \cap (2^R\mathcal{V}_\Lambda) \cong \Lambda/2^R\Lambda$. Moreover, in stark contrast to ball-based shaping, the encoding and decoding complexity does not depend on R.

Lattice choice

A good choice of lattice Λ should satisfy: 1) an efficient lattice decoding algorithm; 2) small NSM; 3) large $\mu(\mathcal{V}_\Lambda)$; 4) be a subset of the standard integer lattice $\mathbb{Z}^d$.

In this work, we use the Gosset lattice (E_8) that satisfies all these properties. It has a fast decoding algorithm (Algorithm 6), its NSM is $\approx 0.0716821 \approx 1.2243\frac{1}{2\pi e}$ [24], and

its Gaussian mass $\mu(r\mathcal{V}_{E_8})$ is very close to $\mu(r\mathcal{B})$ (the Gosset lattice has unit covolume, so $\mathrm{vol}(r\mathcal{V}_{E_8}) = \mathrm{vol}(r\mathcal{B})$). The last point is illustrated in Figure 4.3, where the large loss for cubic shaping with respect to lattice shaping is also evident.

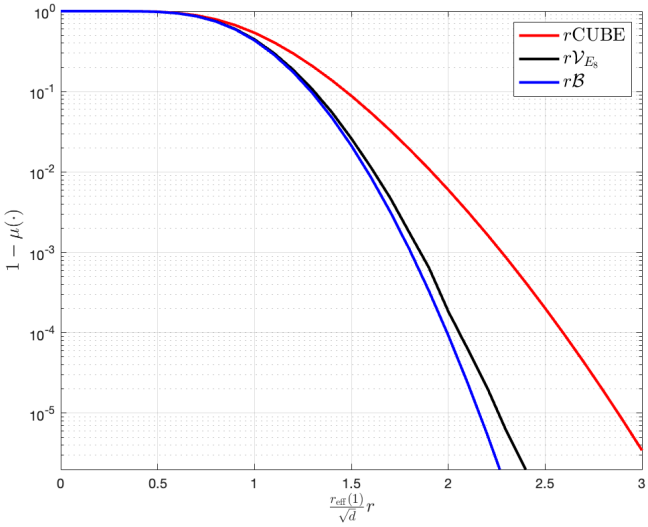


Figure 4.3: Complement Gaussian measure of an 8-dimensional cube (corresponding to shaping using an ℓ_∞ ball), a Voronoi region of the Gosset lattice E_8 (corresponding to shaping using Voronoi codes with base lattice E_8), and a Euclidean ball (corresponding to shaping with a ball, which does not admit efficient implementation)

Overload avoidance via union of Voronoi codes

Because we rely on lattice quantizers of relatively small dimension ($d = 8$), even if $\mu(r\mathcal{V}_\Lambda)$ is very close to $\mu(r\mathcal{B})$, overload events are unavoidable. This follows because in small dimensions the norm of an i.i.d. Gaussian vector is not sufficiently concentrated. Thus, if one is restricted to $\mathcal{C} = \beta(\Lambda \cap (2^R\mathcal{V}_\Lambda))$, the parameter β must be taken quite large in order to keep the overload probability small. This, in turn, incurs a significant penalty in the obtained distortion. As a remedy, rather than using a Voronoi code, we take $\mathcal{C}$ as a union of (a small number of) Voronoi codes in different scales. Namely, we take $\mathcal{C} = \cup_{t=1}^{k}\beta_t(\Lambda \cap (2^R\mathcal{V}_\Lambda))$, where $\beta_1 < \cdots < \beta_k$. The smallest values of β_t are set such that overload is not too common but not extremely rare, such that for most realizations of a Gaussian vector $X \in \mathbb{R}^d$ the distortion is close to the fundamental limit $D(R)$. Whenever X is atypically large, there will be overload in $\beta_t(\Lambda \cap (2^R\mathcal{V}_\Lambda))$ for small t, but not for large t, such that the quantization error will lie in $\beta_t\mathcal{V}_\Lambda$ for one of the larger values of $\{\beta_t\}$.

4.2 Codebook details

In this section, we describe the construction for a Vector Quantization (VQ) codebook of size q^d for quantizing a d-dimensional vector, where q is an integer parameter. This construction is

based on Voronoi codes [1] and admits efficient encoding and decoding algorithms, whenever the base lattice has an efficient closest lattice vector algorithm. Another appealing feature of Voronoi codes is that the encoding/decoding complexity is independent of the quantization rate $R = \log_2(q)$.

Definition 4.2.1 (Voronoi code [1])**.** The Voronoi codebook with base lattice $\Lambda \subset \mathbb{R}^d$ and nesting ratio $q \in \mathbb{N}$, corresponding to rate $R = \log_2(q)$ bits/entry, is defined as $C = \Lambda \cap \mathcal{V}_{q\Lambda} = \Lambda \cap (q\mathcal{V}_\Lambda) \subset \mathbb{R}^d$. In particular, $\lambda \in \Lambda$ belongs to codebook C iff $\lambda \in \mathcal{V}_{q\Lambda}$, and $|C| = q^d$.

The Voronoi code consists of the minimum energy representative of each coset in $\Lambda/q\Lambda \cong (\mathbb{Z}/q\mathbb{Z})^d$. Consequently, we can represent each coset, and hence each codeword in C, as an element of $\mathbb{Z}_q^d$ [1].

Assuming we have access to an oracle $Q_\Lambda(x)$, which maps $x \in \mathbb{R}^d$ to its closest point in Λ, quantization (encoding) and dequantization (decoding) for a Voronoi code are described in Algorithm 1 and Algorithm 2, respectively. Here, $G \in \mathbb{R}^{d\times d}$ is a generator matrix of Λ. The encoder first maps $x \in \mathbb{R}^d$ to its nearest lattice point $p = Q_\Lambda(x)$. Since it only has a budget of dR bits for describing p, it only describes the coset of $\Lambda/q\Lambda$ it belongs to. This is done by sending $v \bmod q$, where $v \in \mathbb{Z}^d$ is the integer vector for which $p = Gv$, referred to as p's coordinates. Upon receiving $v \bmod q$, the decoder knows that $Q_\Lambda(x) \in p + q\Lambda$, and must choose one point in this coset as the reconstruction $\hat{x} \in \mathbb{R}^d$. It chooses $\hat{x}$ as the minimum energy vector in $p + q\Lambda$, corresponding to the unique point in $p + q\Lambda \cap \mathcal{V}_{q\Lambda}$. We have that $\hat{x} = Q_\Lambda(x)$ iff $Q_\Lambda(x) \in \mathcal{V}_{q\Lambda} = q\mathcal{V}_\Lambda$. When $Q_\Lambda(x) \notin q\mathcal{V}_\Lambda$, the quantizer is in *overload*.

Algorithm 1 Encode

Input: $x \in \mathbb{R}^d$, Q_Λ
$p \leftarrow Q_\Lambda(x)$
$v \leftarrow G^{-1}p$ ▷ coordinates of p
return $v \bmod q$ ▷ quantized representation of p

Algorithm 2 Decode

Input: $c \in \mathbb{Z}_q^d$, Q_Λ
$p \leftarrow Gc$ ▷ equivalent to answer modulo $q\Lambda$
return $p - q\,Q_\Lambda\left(\frac{p}{q}\right)$

In our experiments for this paper, we used the Gosset (E_8) lattice as Λ with $d = 8$. This lattice is a union of D_8 and $D_8 + \frac{1}{2}$, where D_8 contains elements of $\mathbb{Z}^8$ with an even sum of coordinates. There is a simple algorithm for finding the closest point in the Gosset lattice, first described in [25]. We provide the pseudocode for this algorithm in section 8.1.

4.3 Optimal scaling coefficients

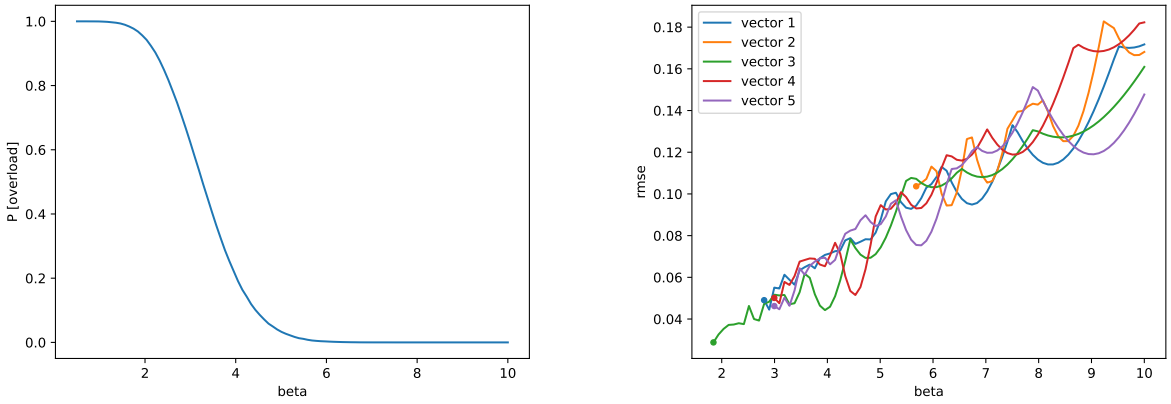
One of the important parts of the algorithm is finding the optimal set of β_i. Given the distribution of d-dimensional vectors quantized via a Voronoi codebook, we can determine the optimal set of a given size exactly using dynamic programming.

Recall that instead of using one instance of a lattice codebook C, we use a union of codebooks C, each scaled by a different coefficient. Specifically, our final codebook $\mathcal{C}$ is parameterized by coefficients $\beta_1 \leq \beta_2 \leq \ldots \leq \beta_k$, and is equal to:

$$\mathcal{C} = \beta_1 C \cup \beta_2 C \cup \ldots \cup \beta_k C$$

When quantizing a vector to the i-th scaled codebook, we could either get a small granular error when the vector is in $V_{\beta_i \Lambda}(0)$, or a large overload error otherwise. If we use a codebook with smaller β, the probability of an overload error increases, but the expected magnitude of the granular error decreases due to the volume of the Voronoi region being smaller (Figure 4.4). We can have two strategies for encoding:

1. **First-β:** Use the smallest β which does not result in an overflow error.
2. **Opt-β:** Try all the values of β, and choose the one that has the smallest reconstruction MSE.



(a) Overload error probability

(b) Granular RMSE for five sampled vectors

Figure 4.4: Granular and overload error for standard Gaussian vectors, $q = 16$

K	2	4	6	8	10
Opt-β	0.0878	0.0795	0.0708	0.0669	0.0646
First-β	0.0878	0.0798	0.0712	0.0676	0.0656

Table 4.1: Mean RMSE for reconstructed i.i.d. standard Gaussian 8-vectors, $q = 16$, k betas are uniform on $[0, 10]$.

Even though Opt-β should provide smaller error, the definition of First-β will be useful for us. The difference in error between Opt-β and First-β is small (Table 4.1). Moreover, First-β can be used to determine the optimal set of β_i to use.

Let us have n samples $v_1, v_2, \ldots, v_n$ from the distribution of vectors we are quantizing, and a large set of betas B, containing $\beta_1 < \beta_2 < \ldots < \beta_m$, from which we want to take the optimal subset of size k which minimizes the loss under the First-β strategy. For each vector v_i and beta β_j we compute mse_{ij} — the MSE if we use β_j to quantize v_i — and $overload_{ij}$ — whether an overload error occurs in this scenario.

We solve this optimization problem using dynamic programming. Let's define dp_{ij} to be the minimum sum of MSE we can get if we have to quantize all the vectors which do not yield an overload error for β_i, using β_i and $j-1$ smaller betas and the First-β strategy. If i is large enough so that no vector has an overflow error on β_i, dp_{ik} has the answer to the problem. To compute the value of dp_{ij}, we can iterate over s — the index of the second-largest beta in the set (the largest being β_i). Then, the recalculation works in the following way:

$$dp_{ij} \leftarrow \min\left(dp_{ij}, dp_{s,j-1} + \sum_{p,cond_p} mse_{pi}\right)$$

where $cond_p = overload_{ps} \wedge \neg overload_{pi}$

Following the DP transitions yields the optimal set of β.

Algorithm 3 Dynamic programming for finding the set of β

Input: vectors v_i, beta set B, mse_{ij}, $overload_{ij}$
$dp_{i,j} = \infty$ for i in $0 \ldots m$, j in $0 \ldots k$
$from_{i,j} =$ null for i in $0 \ldots m$, j in $0 \ldots k$
$dp_{0,0} = 0$
for $i = 1$ **to** m **do**
    for $j = 1$ **to** k **do**
        for $s = 0$ **to** $i - 1$ **do**
            $cond_p = overload_{ps} \wedge \neg overload_{pi}$ for $p \in 1 \ldots n$
            $cost = \sum_p cond_p \cdot mse_{pi}$
            if $dp_{ij} > dp_{s,j-1} + cost$ **then**
                $dp_{ij} \leftarrow dp_{s,j-1} + cost$
                $from_{ij} \leftarrow s$
            end if
        end for
    end for
end for
Let pos be chosen so that β_{pos} has no overflow errors
result = []
for $j = k$ **downto** 1 **do**
    result.append(pos)
    $pos \leftarrow from_{pos,j}$
end for

Chapter 5

QA-LDLQ

5.1 Summary

In this section, "GPTQ" and "LDLQ" are used interchangeably to refer to the adaptive rounding algorithm described in Section 2.3.3.

We note that in the presence of activation quantization, the LDLQ and GPTQ algorithms for weight quantization become significantly suboptimal [1], thus necessitating a correction (QA-LDLQ) that we describe here.

Let W be the weight (shape $a \times n$), X be a random original (unquantized) activation vector (shape $n \times 1$), and Z be the quantization error of X, modeled as zero-mean random noise (shape $n \times 1$) independent of X. Then, if U is the quantized weight, the output quantization error becomes $\delta(U) := WX - U(X + Z)$, so that the loss to be minimized is $\mathbb{E}[\|\delta(U)\|^2]$ instead of $\mathbb{E}[\|(W - U)X\|^2]$ that LDLQ and GPTQ minimize.

Lemma 5.1.1. *Suppose Z is independent from X, $\mathbb{E}[Z] = 0$, and let $H = \mathbb{E}[XX^\top]$ and $J = \mathbb{E}[ZZ^\top]$. Then for any set $\mathcal{C}_Q \subset \mathbb{R}^{a \times n}$*

$$\begin{aligned} U^* &= \operatorname*{argmin}_{U \in \mathcal{C}_Q} \mathbb{E}[\|\delta(U)\|^2] \\ &= \operatorname*{argmin}_{U \in \mathcal{C}_Q} (\tilde{W} - U)(H + J)(\tilde{W} - U)^\top, \end{aligned} \tag{5.1}$$

where $\tilde{W} = WH(H + J)^{-1}$.

Thus, QA-LDLQ involves: (a) computing $\tilde{W}$, and (b) running the standard LDLQ using $\tilde{W}$ as input with the Hessian set to $H + J$.

Proof of Lemma 5.1.1. Recalling the definition of $\tilde{W} = WH(H + J)^{-1}$, that X and Z are statistically independent, and Z has zero mean, and that H, J are positive semi-definite symmetric matrices, we have

$$\mathbb{E}\|\delta(U)\|^2 = \mathbb{E}\|(W - U)X - UZ\|^2 = (W - U)H(W - U)^\top + UJU^\top \tag{5.2}$$

$$= (\tilde{W} - U)(H + J)(\tilde{W} - U) + C(W, H, J), \tag{5.3}$$

[1] In fact, using original LDLQ on Llama-3-70B produces $\sim 10^4$ perplexity at W4A4 setting due to significant outliers in layer 0.

where

$$C(W, H, J) = W(H - H(H + J)^{-1}H)W^\top, \tag{5.4}$$

is independent of U, and therefore does not affect the minimization. □

5.2 Motivation

When quantizing Llama-3-70B with original LDLQ, we have noticed very poor (ppl $\sim 10^4$) performance. We have observed that the reason for this is quantization of activations for a small subset of linear layers (4 out of 560), and if they are left in full precision, the perplexity becomes more reasonable (less than 4). We have found them by choosing the layers with the largest *amplification ratio* — a concept we define next.

Consider a weight W (shape $a \times n$), where n is the number of in-features, and a is the number of out-features. We define *amplification* of a random vector X ($n \times 1$) by W as $\alpha(W, X) := \frac{\mathbb{E}[\|WX\|_2]}{\mathbb{E}[\|X\|_2]}$. Then, if X is the distribution of input activations, and Z is a random Gaussian vector, we define the amplification ratio for W as $\frac{\alpha(W,Z)}{\alpha(W,X)}$. A large amplification ratio makes a layer's activations harder to quantize because it amplifies quantization noise far more than the activations themselves. One extreme example of this issue is the value projection of the attention of the first transformer block in Llama-3-70B. This layer has an amplification ratio of ~ 157, as computed from 10 wikitext2 sequences of length 2048. This makes naive 4-bit quantization of activations nearly impossible, as even small perturbations of the input of the layer are greatly amplified in the output.

We have developed QA-LDLQ to mitigate the issue of large amplification ratio. We model quantization noise as a random Gaussian vector with mean 0 and covariance matrix $J = \varepsilon^2 I$, where ε^2 depends on the quantization rate and the statistics of X. Larger ε^2 makes $\tilde{W} = WH(H + \varepsilon^2 I)^{-1}$ more robust to input perturbations (reducing the amplification ratio) but also increases bias, as expressed in $C(W, H, J)$. Figure 5.1 demonstrates this tradeoff for the value projection layer mentioned earlier.

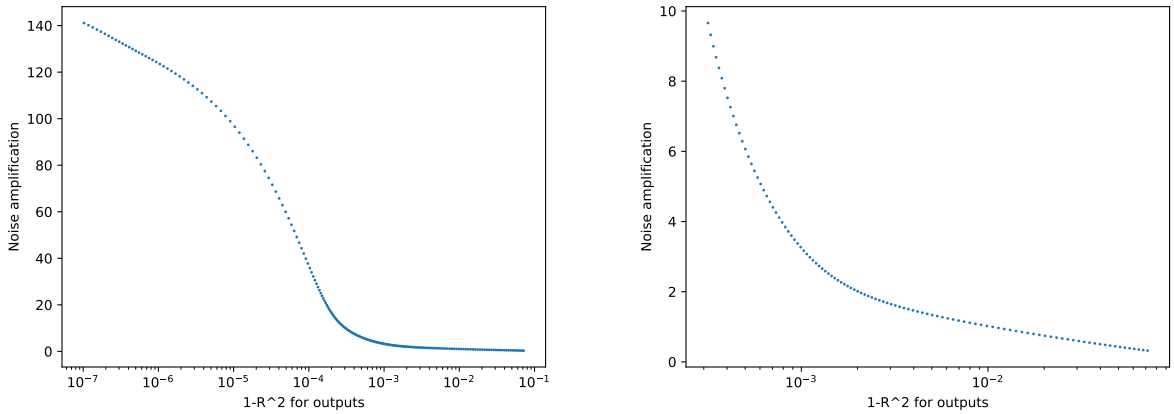


Figure 5.1: We run QA-LDLQ for value projection layer of the first transformer block of Llama-3-70B. We try different values of ε on logarithmic scale from 10^{-5} to 1. For each ε, we find modified weight $\tilde{W}$, and plot the amplification ratio for $\tilde{W}$ in y-axis, as well as how close the outputs of the weight $\tilde{W}$ to the outputs of weight W. The value on x axis is defined as $1 - R^2 := \frac{\mathbb{E}\|WX - \tilde{W}X\|^2}{\mathrm{Var}(WX)}$, where X contains activation inputs from 10 sequences of length 2048 from wikitext2. The right plot is bottom right corner of the left plot, zoomed in. We note that by paying a small price in the accuracy of the weight, we can reduce the amplification ratio dramatically.

Chapter 6

NestQuant overview

6.1 Matrix quantization

When quantizing a matrix, we normalize its rows and quantize each block of d entries using the codebook. Algorithm 4 describes the quantization procedure for each row of the matrix.

Algorithm 4 NestQuant

Input: A — a vector of size $n = db$, q, array of k scaling coefficients $\beta_1, \dots, \beta_k$
QA — n integers in $\{0, 1, \dots, q-1\}$ ▷ quantized representation
B — b integers in $\{1, \dots, k\}$ ▷ scaling coefficient indices
$s \leftarrow \|A_i\|_2$ ▷ normalization coefficient
$A \leftarrow \frac{A\sqrt{n}}{s}$
for $j = 0$ **to** $b - 1$ **do**
  $err = \infty$
  for $p = 1$ **to** k **do**
    $v \leftarrow A[dj + 1..dj + d]$
    $enc \leftarrow \text{Encode}\left(\frac{v}{\beta_p}\right)$
    $recon \leftarrow \text{Decode}(enc) \cdot \beta_p$
    if $err > |recon - v|_2^2$ **then**
      $err \leftarrow |recon - v|_2^2$
      $QA[dj + 1..dj + d] \leftarrow enc$
      $B_j \leftarrow p$
    end if
  end for
end for
Output: QA, B, s

We can take dot products of quantized vectors without complete dequantization using Algorithm 5. We use it in the generation stage on linear layers and for querying the KV cache.

Algorithm 5 Dot product

Input: QA_1, B_1, s_1 and QA_2, B_2, s_2 — representations of two vectors of size $n = db$ from Algorithm 4, array β
$ans \leftarrow 0$
for $j = 0$ **to** $b - 1$ **do**
  $p_1 \leftarrow \text{Decode}(QA_1[dj + 1..dj + d])$
  $p_2 \leftarrow \text{Decode}(QA_2[dj + 1..dj + d])$
  $ans \leftarrow ans + (p_1 \cdot p_2)\beta_{B_1[j]}\beta_{B_2[j]}$
end for
return ans

6.2 LLM quantization

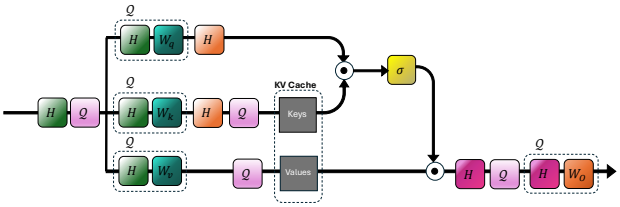


Figure 6.1: The quantization scheme of multi-head attention. H is the Hadamard rotation described in 6.2. $\mathcal{Q}$ is the quantization function described in 6.1

Recall that we apply a rotation matrix H to every weight–activation pair of a linear layer without changing the output of the network. Let n be the number of input features to the layer. Following [16, 18, 26]:

- If $n = 2^k$, we set H to be the Hadamard matrix obtained by Sylvester's construction.
- Otherwise, we decompose $n = 2^k m$, such that m is small and there exists a Hadamard matrix H_1 of size m. We construct a Hadamard matrix H_2 of size 2^k using Sylvester's construction and set $U = H_1 \otimes H_2$.

Note that it is possible to multiply an $r \times n$ matrix by H in $O(rn \log n)$ in the first case and $O(rn(\log n + m))$ in the second case, which is negligible compared to other computational costs and can be done online.

In NestQuant, we quantize all weights, activations, keys, and values using Algorithm 4. We merge the Hadamard rotation with the weights and quantize them. We also apply the Hadamard rotation and quantization to the activations before linear layers. We also apply rotation to keys and queries since it does not change the attention scores, and we quantize keys and values before putting them in the KV cache. Figure 6.1 illustrates the procedure for multi-head attention layers.

When quantizing a weight, we modify the NestQuant algorithm by introducing corrections to unquantized weights when a certain vector piece is quantized via the QA-LDLQ mechanism, described in Chapter 5. It is based on the LDLQ algorithm described in Section 4.1 of [18].

6.3 Algorithm summary

Here we describe the main steps of NestQuant.

1. Collect the statistics for LDLQ via calibration data. For each linear layer with in-dimension d, we compute a $d \times d$ "Hessian" matrix H.
2. We choose an initial set of scaling coefficients $\hat{\beta}$, and for each weight we simulate LDLQ quantization with these coefficients, obtaining a set of 8-dimensional vectors to quantize.
3. We run a dynamic programming algorithm described in Section 4.3 on the 8-vectors to find the optimal β values for each weight matrix.
4. We also run the dynamic programming algorithm for activations, keys, and values for each layer. To get the distribution of 8-vectors, we run the model on a small set of examples.
5. We quantize the weights using QA-LDLQ and precomputed β.
6. During inference, we quantize any activation before it is passed to the linear layer and any KV cache entry before it is saved.

Note that we do not undertake any expensive (but surely useful) fine-tuning, such as optimizing rotation matrices or post-quantization training, as described in [16] and [18], since our goal is demonstrating the basic primitive, not obtaining the absolute SOTA numbers.

Chapter 7

Experimental results

7.1 Simulated Data

We compared the mean L_2 loss per entry of SpinQuant to the uniform L_∞-scaling quantizer (used in SpinQuant and other methods). The mean L_2 loss per entry for the product of two matrices $A \in \mathbb{R}^{n\times k}, B \in \mathbb{R}^{m\times k}$ is computed as $\frac{\|AB^T - \hat{A}\hat{B}^T\|_2}{nm}$. We set $n = k = m = 4096$ and sampled two matrices A, B from the standard normal distribution $A_{ij}, B_{ij} \sim \mathcal{N}(0,1)$. We compare to the lower bound from (2.1).

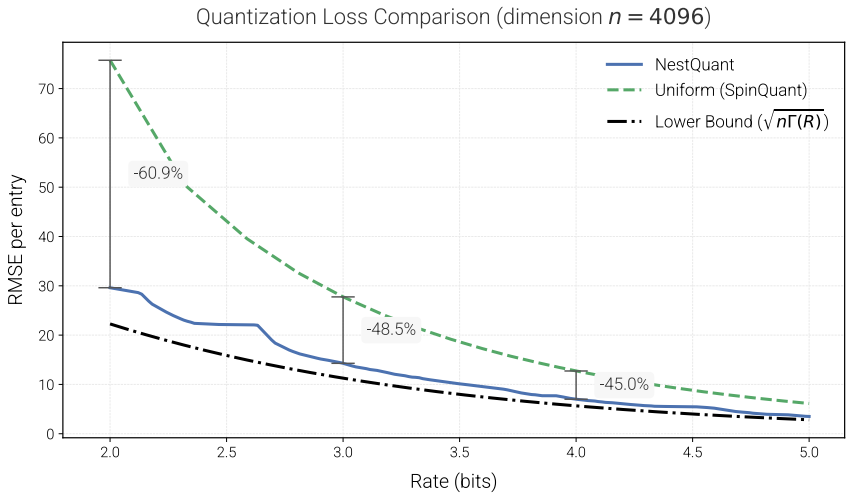


Figure 7.1: RMSE for quantized matrix multiplication for i.i.d. $\mathcal{N}(0,1)$ matrices. The NestQuant algorithm is optimized over q and multiple β's. Also shown is the information-theoretic lower bound from (2.1).

For NestQuant, we do a grid search over (q, k). For a given q and k, we find the best subset in $\frac{1}{2} \cdot \{1, 2, \ldots, 50\}$ of scaling coefficients β of size k using the algorithm from 4.3. Then we calculate the expected bits per entry computed as $\log_2 q + \frac{1}{8}\sum_{i=1}^{k} p(\beta_i)\log_2 p(\beta_i)$ where $p(\beta_i)$ is the probability that the ith β value is used in quantization. In Figure 7.1, we plot the efficient frontier of bits per entry vs. root mean L_2 loss.

Bits (W-A-KV)	Method	Llama-2-7B	Llama-2-13B	Llama-2-70B	Llama-3-8B	Llama-3-70B
16-16-16	Floating point	5.47	4.88	3.32	6.14	2.86
4-16-16	QuaRot	5.60	5.00	3.41	-	-
	QuIP#	5.56	4.95	**3.38**	-	-
	OstQuant	5.64	4.94	3.41	6.53	3.19
	NestQuant	**5.53**	**4.93**	**3.38**	**6.31**	**3.14**
	NestQuantM	5.55	4.95	-	6.35	-
4-16-4	NestQuant	5.57	4.96	3.39	6.37	3.19
	NestQuantM	5.59	4.99	-	6.49	-
4-4-16	SpinQuant	5.9	5.2	3.8	7.1	-
	OstQuant	5.60	5.14	3.57	7.24	3.97
	DuQuant	6.08	5.33	3.76	8.06	-
4-4-4	QuaRot	6.10	5.40	3.79	8.16	6.66
	SpinQuant	5.9	5.3	3.8	7.3	-
	OstQuant	5.91	5.25	3.59	7.29	4.01
	NestQuant	**5.67**	**5.03**	**3.49**	**6.63**	**3.61**
	NestQuantM	5.73	5.07	-	6.82	-

Table 7.1: The wikitext2 perplexity with a context window of 2048 for various quantization methods of Llama models.

7.2 Llama results

7.2.1 Experimental design

We choose the train split of the Wikitext2 [27] dataset as a calibration dataset for computing H, and evaluate the model on the validation split, computing the perplexity metric. For step 2 in the algorithm (Section 6.3), we select $\hat{\beta} = [3.5, 4.5, 6.0, 14.5, 25.0]/q$, because it is the β we get when optimizing them for weight quantization without consideration of LDLQ. The overall universe of β values contains values from 1 to 40 with spacing ranging from 0.25 to 2. For running DP on activations, keys, and values, we run the model on a batch of 6 full-length sequences, which is sufficient for this low-dimensional hyperparameter.

When choosing the maximum β for a given distribution, we add a margin of $\frac{3.0}{q}$ for weights and $\frac{4.0}{q}$ to the maximum β needed to have 0 overload errors on known data to account for potential overload errors in unknown data. While a small number of overload errors does not affect perplexity significantly, we still aim to minimize their probability.

When computing perplexity for Wikitext2 with a given context length, we average the perplexities for all positions, which is standard practice in other works on LLM quantization.

7.2.2 Results for 4-bit quantization

In comparisons to other methods, we focus on the 4-bit setup, choosing $q = 14$ and $k = 4$. We show the wikitext2 perplexity comparisons for multiple Llama models in Table 7.1, and other benchmark comparisons (for Llama-3-8B) in Table 7.4. The methods included in these tables are SpinQuant [16], QuIP# [18], QuaRot [15], DuQuant [26], and OstQuant [28]. On the wikitext2 dataset, we computed the perplexity scores of the quantized models with a context size of 2048.

NestQuant consistently achieves better perplexity metrics across different models for both

q	Bits	Bits (no zstd)	W	W + KV	W + KV + A
14	3.99	4.06	6.308	6.379	6.633
12	3.76	3.83	6.376	6.475	6.841
10	3.50	3.57	6.486	6.640	7.251
8	3.18	3.25	6.700	6.968	7.989

Table 7.2: Wikitext2 perplexity of NestQuant quantization of Llama-3-8B at different rates. The "bits" column is the bit rate per entry with zstd compression of scaling coefficients, and "bits (no zstd)" is the bit rate without compression. The "W", "W+KV", and "W+KV+A" describe the quantization regime (whether weights, KV cache, or activations are quantized). The perplexity of non-quantized model is 6.139

the weight-only regime and full quantization. For all models we tested—except Llama-2-7B—NestQuant with W4A4KV4 (4-bit weights, activations, and KV-cache) quantization even outperforms previous works using W4A4KV16. For W4KV4A4 (4-bit weights, KV-cache, and activations) quantization of Llama-3-8B, we achieve a perplexity score of 6.6, compared to approximately 7.3 for SpinQuant and OstQuant. Even without LDLQ, we achieve a perplexity score of 6.8, which is still better. Finally, NestQuant outperforms QuIP# for weight-only Llama-2 quantization.

In addition, we evaluate a simpler version of the algorithm, easier to implement in hardware, called NestQuantM. More details on this variant can be found in Section 8.2.

7.2.3 LLM quantization scaling

We quantize the Llama-3-8B model [9] using different values of q. We choose the number of scaling coefficients (k) to be 4; Section 7.3.1 explains the rationale behind this choice. More details on the hyperparameter choice of the experiments are in Section 7.2.1. For each experiment, we compute the number of bits per entry similar to Section 7.1, but for the configuration with compressed β indices, we apply the Zstandard (zstd) compression algorithm instead of using the entropy of the distribution. As our evaluation metric, we use the perplexity on the validation split of wikitext2 with context length 2048.

7.2.4 Results for Llama3.2-1B

Here, we show the results of NestQuant on the newer 1B-parameter Llama3.2-1B model. We do experiments in the same configurations as the Llama-3-8B model, computing the wikitext2 perplexity.

q	Bits	Bits (no zstd)	W	W + KV	W + KV + A
14	3.99	4.06	10.061	10.529	11.197
12	3.76	3.837	10.178	10.862	11.910
10	3.50	3.57	10.377	11.552	14.191
8	3.18	3.25	10.850	13.309	18.710

Table 7.3: Wikitext2 perplexity of NestQuant quantization of Llama-3.2-1B. The format of the table is the same as in Table 7.2. The perplexity of non-quantized model is 9.749

7.2.5 Results for 3-bit model quantization

We present the results for 3-bit quantization of weights and activations on small models (Llama-3-8B and Llama-2-7B). We use $q = 7$ and $k = 4$, which results in 2.98 bits per entry.

Bits (W-A-KV)	Method	Llama-2-7B	Llama-3-8B
16-16-16	Floating point	5.47	6.14
4-4-16	NestQuant	5.64	6.56
3-3-16	NestQuant	6.33	8.25

7.2.6 Zero shot benchmarks

We also perform the evaluation of NestQuant on various zero-shot benchmarks: ARC-Easy and ARC-Challenge [29], Hellaswag [30], [31], and Winogrande [32]. The results on 4-bit models with comparisons to other models are summarized in Table 7.4.

Model	Bits ↓	Bits (no zstd) ↓	ARC-C ↑	ARC-E ↑	Hellaswag ↑	PIQA ↑	Winogrande ↑	Zero-shot Avg ↑	Wikitext2 ppl ↓
Baseline (FP16)	16	16	0.54	0.78	0.79	0.81	0.74	0.73	6.1
Weights only									
LLM-QAT	4.00	-	0.51	0.77	0.48	0.79	0.72	0.65	7.7
GPTQ	4.00	-	0.47	0.72	0.74	0.77	0.71	0.68	7.2
SpinQuant	4.00	-	**0.54**	0.77	0.78	**0.80**	0.72	**0.72**	6.5
NestQuant $q = 14, k = 4$ (ours)	**3.99**	4.06	0.53	**0.78**	**0.79**	**0.80**	**0.73**	**0.72**	**6.3**
Weights + KV cache									
SpinQuant	4.00	-	0.51	0.77	0.77	0.78	0.69	0.70	6.6
NestQuant $q = 14, k = 4$ (ours)	**3.99**	4.06	**0.53**	**0.78**	**0.79**	**0.79**	**0.74**	**0.72**	**6.4**
Weights, KV cache, activations									
LLM-QAT	4.00	-	0.27	0.41	0.38	0.60	0.53	0.44	52.5
Quarot	4.00	-	0.44	0.67	0.75	0.75	0.66	0.67	8.4
SpinQuant	4.00	-	**0.51**	**0.75**	0.75	0.77	0.66	0.68	7.3
NestQuant $q = 14, k = 4$ (ours)	**3.99**	4.06	**0.51**	**0.75**	**0.78**	**0.79**	**0.72**	**0.71**	**6.6**

Table 7.4: 4-bit quantization of Llama-3-8B. The bits column for NestQuant corresponds to actually measured average number of bits per entry (when a vector of auxiliary scaling coefficients β is compressed via zstd) and the second column shows quantization rate when no compression step is used.

7.3 Ablation studies

We found LDLQ to be useful in improving the quality of quantized model. In table 7.5, we compare the wikitext2 perplexity of models with and without LDLQ.

Algorithm	W	W + KV	W + KV + A
NestQuant	6.308	6.379	6.633
NestQuant (no LDLQ)	6.528	6.605	6.849

Table 7.5: Effect of LDLQ on NestQuant ($q = 14$ and $k = 4$) wikitext2 perplexity

While Hadamard matrices from Sylvester construction are commonly used in other works (QuIP#, Quarot), there are multiple ways to construct a fast rotation for the case when dimension is not a power of 2 (such as the down projection in MLP of Llama-3). We tested three possible options for rotation on $q = 14$, $k = 4$, W + KV + A quantization.

Algorithm	**W + KV + A**
Fourier	6.773
$S \otimes H$, S — orthogonal, H — Sylvester Hadamard	6.770
$H_1 \otimes H$, H_1 — hardcoded Hadamard, H — Sylvester Hadamard	**6.663**

Table 7.6: Effect of rotation on NestQuant ($q = 14$ and $k = 4$) wikitext2 perplexity

7.3.1 Choosing the number of scaling coefficients

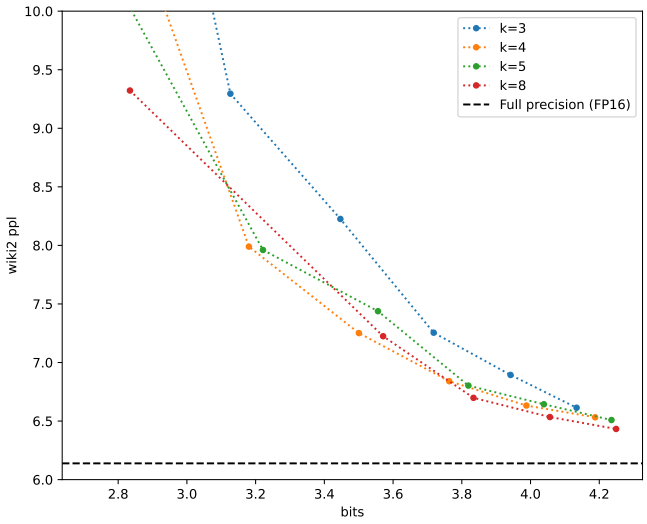


Figure 7.2: The perplexity-bitrate scaling of NestQuant with different values of k, all components of the model (weights, KV cache, activations) are quantized

The value of k, i.e., the number of scaling coefficients, is an important hyperparameter of the algorithm. Increasing k decreases the quantization error by allowing each vector to be quantized to the lattice point with a more suitable scaling. However, it also increases the bitrate and makes the encoding slower, since we need to try a larger number of scaling coefficients.

We used $k \in \{3, 4, 5, 8\}$ to quantize Llama-3-8B across different values of q, plotting the resulting perplexity against bitrate in Figure 7.2. We can see that using $k = 3$ leads to suboptimal performance of the quantization scheme, while the performances of $k = 4, 5, 8$ are comparable. In our experiments, we use $k = 4$ because a lower k results in faster encoding.

Chapter 8

Algorithm efficiency

8.1 Gosset oracle

In this section, we discuss the algorithm for finding the nearest neighbour in the E_8 lattice and estimate its performance in FLOPs (Floating Point Operations). We note that

$$E_8 = D_8 \cup \left(D_8 + \frac{1}{2}\right),$$

where D_8 contains vectors in $\mathbb{Z}_8$ with an even sum of coordinates. To compute $V_{E_8}(x)$, we compute two candidate points: $x_1 = V_{D_8}(x)$ and $x_2 = V_{D_8+\frac{1}{2}}(x)$, and choose the one that has the smaller L^2 distance to x.

To get $V_{D_8}(x)$, we can round each coordinate to the nearest integer. If the sum of rounded coordinates is odd, we need to "flip" the rounding direction of the coordinate for which the flip would cost the least. Note that finding the closest point in $V_{D_8+\frac{1}{2}}$ works the same, but the rounding grid now contains half-integers, not integers.

In Algorithm 6, we first round our vector down (getting d) and compute the mask (g) of whether it's optimal to round up for D_8. We note that the optimal rounding for $D_8 + \frac{1}{2}$ is $d + 0.5$, while the optimal rounding for D_8 is $d + g$.

We want to understand whether rounding to D_8 or $D_8 + \frac{1}{2}$ is better. Let $dist_i$ be the absolute distance from the i-th entry $x_i \in [d_i, d_i + 1]$ to the middle of this integer segment $d_i + 0.5 = x_{2,i}$. We note that the contribution of this point to the MSE for D_8 is $(0.5 - dist_i)^2$, while for $D_8 + \frac{1}{2}$ it is $dist_i^2$. The difference is:

$$0.25 - dist_i + \cancel{dist_i^2} - \cancel{dist_i^2} = 0.25 - dist_i.$$

If the sum of this value over i is negative (i.e. $\sum dist_i > 2$), it's optimal to quantize to D_8, otherwise to $D_8 + \frac{1}{2}$. In pseudocode, we store $\sum dist_i$ as Δ.

We must check the constraint that the sum of coordinates in D_8 is even, and if it is not, "flip" one of the rounding directions. The optimal coordinate to flip can be determined through $dist$. The new value of the flipped coordinate can then be determined through g. We also need to update Δ given that the MSE difference changes.

The full pseudocode of the algorithm is in Algorithm 6.

Algorithm 6 Oracle for the Gosset lattice

1: **Input:** $x \in \mathbb{R}^8$
2: $d \leftarrow \text{floor}(x)$
3: $x_2 \leftarrow d + 0.5$
4: $g \leftarrow (x > x_2)$
5: $s \leftarrow 2 \cdot g - 1$
6: $x_1 \leftarrow d + g$
7: $dist \leftarrow (x - x_2) \cdot s$
8: $\Delta \leftarrow \sum_i dist_i$
9: **if** $\sum_i x_{1,i}$ is odd **then**
10: $\quad pos = \text{argmin}\, dist$
11: $\quad x_{1,pos} \leftarrow x_{1,pos} - s_{1,pos}$
12: $\quad \Delta \leftarrow \Delta + 2 \cdot dist_{pos} - 1$
13: **end if**
14: **if** $\sum_i x_{2,i}$ is odd **then**
15: $\quad pos = \text{argmax}\, dist$
16: $\quad x_{2,pos} \leftarrow x_{2,pos} + g_{2,pos}$
17: $\quad \Delta \leftarrow \Delta + 1 - 2 \cdot dist_{pos}$
18: **end if**
19: **if** $\Delta > 2$ **then**
20: $\quad$ **return** x_1
21: **else**
22: $\quad$ **return** x_2
23: **end if**

8.2 NestQuantM algorithm

Due to the high complexity of argmin and argmax operations in Algorithm 6 for fast hardware implementation, we propose a different, simpler version of NestQuant decoding. Instead of using argmax and argmin on lines 10 and 15, we always assign *pos* to be equal to 1 (thus, indicating that we always "flip" the rounding direction of the first coordinate to fix the parity). Note that this change is only applied in the decoding stage, while during encoding we use the full version of the algorithm, which keeps the granular error the same. Let's denote our modified Gosset oracle as $f\colon \mathbb{R}^8 \to \mathbb{R}^8$.

Lemma 8.2.1. *For a vector $x \in \mathbb{R}^8$ and a vector $v \in E_8$, $f(x + v) = f(x) + v$.*

Proof. Recall that $D_8 \subset \mathbb{Z}^8$ contains all the integer vectors with an even sum of coordinates. The modified Gosset oracle f uses the modified D_8 oracle g, and at input x constructs two candidate points $c_1(x) \in D_8$ and $c_2(x) \in D_8 + \frac{1}{2}$. Then, the algorithm chooses the closest point among these candidates to x. Note that $c_1(x) = g(x)$ and $c_2(x) = g\left(x - \frac{1}{2}\right) + \frac{1}{2}$.

We will prove that for $u \in D_8$, $g(x + u) = g(x) + u$ for any $x \in \mathbb{R}^8$. Now, let's show the original lemma. Note that since we are choosing the closest candidate point, if the condition on candidate sets $\{c_1(x + v), c_2(x + v)\} = \{c_1(x), c_2(x)\} + v$ holds, then $f(x + v) = f(x) + v$. Now, consider two cases:

1. $v \in D_8$. Then:

$$c_1(x+v) = g(x+v) = g(x) + v = c_1(x) + v$$
$$c_2(x+v) = g\left(x + v - \frac{1}{2}\right) + \frac{1}{2} = g\left(x - \frac{1}{2}\right) + \frac{1}{2} + v = c_2(x) + v$$

2. $v \in D_8 + \frac{1}{2}$. Then, we say that

$$v = \left(u - \frac{1}{2}\right) = \left(w + \frac{1}{2}\right)$$

for $u, w \in D_8$.

$$c_1(x+v) = g(x+v) = g\left(x + u - \frac{1}{2}\right) = g\left(x - \frac{1}{2}\right) + u = g\left(x - \frac{1}{2}\right) + \frac{1}{2} + v = c_2(x) + v$$
$$c_2(x+v) = g\left(x - \frac{1}{2} + v\right) + \frac{1}{2} = g(x+w) + \frac{1}{2} = g(x) + w + \frac{1}{2} = g(x) + v = c_1(x) + v$$

Thus, in both cases the condition on candidate sets holds, and we get $f(x+v) = f(x) + v$.

Now we show that if $u \in D_8$, $g(x+u) = g(x) + u$. Note that when evaluating $g(x+u)$, we will get the same rounding directions as in $g(x)$, since u is an integer vector. Since u has an even sum of coordinates, the parity will also be the same. Then, our decision to flip the rounding of the first coordinate will also match. Therefore, the vector between the original and rounded coordinates will be the same:

$$g(x) - x = g(x+u) - x - u \Rightarrow g(x+u) = g(x) + u$$

□

Let v be the vector we obtain after rounding to E_8, c be the lattice coordinates of v, and G be the generating matrix of the lattice. The compressed representation is $c \bmod q$, which corresponds to a point $v' = G(c \bmod q)$. The reconstructed point $\hat{v}$ is defined to be $v' - qf(v'/q)$ by the decoding algorithm. Note that $v - v' \in qE_8$. Let's assume that $f(v/q) = 0$. Then:

$$\hat{v} = v' - qf\left(\frac{v'}{q}\right) = v' - qf\left(\frac{v}{q} + \frac{v'-v}{q}\right) = v' - q\left(f\left(\frac{v}{q}\right) + \frac{v'-v}{q}\right) = v' - q \cdot \frac{v'-v}{q} = v$$

We have used Lemma 8.2.1 in the third equality and our assumption in the fourth equality. Given this fact, we conclude that when using NestQuantM for decoding, the composition of encoding and decoding functions (for a fixed β) is similar to the original NestQuant, except the shaping region has changed to the set of points $v \in E_8$ such that $f(v/q) = 0$. Since f is close to the original Gosset oracle, we expect this region to still capture Gaussian probability density well. In the case of overload errors, we are still able to choose a larger value of β due to the multi-β strategy.

8.3 CUDA Kernel Implementation

The decoding algorithm 2 consists of a basis change $p \leftarrow Gc$ and a subsequent computation of the coset of $\frac{p}{q}$,

$$\frac{p}{q} - Q_\Lambda\left(\frac{p}{q}\right).$$

We leverage the asymmetry between encoding and decoding choosing G which is fast to decode. In particular we use

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

G is not a basis for E_8 but for $2E_8$. We use G because we want to work in integer-lattice as half-integers cannot be represented with integers. Moreover, we represent 8-vector $\vec{x}$ as two 32 bit integers each representing the two parts of the vector x_0, x_1, x_2, x_3 and x_4, x_5, x_6, x_7.

For reference, the CUDA implementation of the implementation of $p \leftarrow Gc$.

```
__device__ void G_q_fast(
    const uint32_t enc, uint32_t* out0, uint32_t* out1
)
{
    uint32_t even = enc & 0x0F0F0F0F;
    uint32_t odd = (enc & 0xF0F0F0F0) >> 4;

    uint32_t two_even = even << 1;
    uint32_t two_odd = odd << 1;
    uint32_t x0 = even & 0xFF;
    uint32_t concatenator = (1 << 24) | (1 << 16) | (1 << 8) | 1;
    uint32_t x0_concat = x0 * concatenator;

    uint32_t temp_even = __vadd4(two_even & 0xFFFFFF00, x0_concat);
    uint32_t temp_odd = __vadd4(two_odd, x0_concat);

    uint32_t s1 = __dp4a(two_even, (uint32_t)0x01010100, (uint32_t)0);
    uint32_t s2 = __dp4a(two_odd, (uint32_t)0x01010101, s1);

    *out0 = temp_even;
    *out1 = temp_odd + s2;
}
```

The computation is done by decomposing G into a sum of different matrices.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$+ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The next step is to compute $\frac{p}{q}$ and the coset of $\frac{p}{q}$ we merge both operations into one function.

```
__device__ void decode_nestquant(
    uint32_t *enc, int *B_local_decode, const int N = 8
)
{
    uint32_t inp0, inp1;
    unsigned int dist[2];

    const uint32_t MASK_FRACPART = 0x1F1F1F1F;
    const uint32_t MASK_INTPART = 0xE0E0E0E0;
    const int HALF_CONCAT = 0x10101010;
    const int TWO = 0x40;
    const int ONE = 0x20;
    const int HALF = 0x10;

    G_q_fast(enc[0], &inp0, &inp1);

    int32_t fracPart0 = inp0 & MASK_FRACPART;
    int32_t fracPart1 = inp1 & MASK_FRACPART;

    int32_t integerPart0 = inp0 & MASK_INTPART;
    int32_t integerPart1 = inp1 & MASK_INTPART;

    int g0 = (fracPart0 & HALF_CONCAT) << 1;
    int g1 = (fracPart1 & HALF_CONCAT) << 1;

```

```
26      int32_t change = ((g0 & ONE) - HALF);
27      int32_t sum1 = get_sum(integerPart0, integerPart1);
28      int32_t two_parity_1 = (sum1 & ONE) >> 4;
29      int32_t fracPart0_1 = __vsub4(fracPart0, (int32_t)(two_parity_1 * change) & 0xFF);
30
31      int32_t sum2 = get_sum(g0, g1) + sum1;
32      int32_t two_parity_2 = (sum2 & ONE) >> 4;
33      int32_t fracPart0_2 = __vadd4(fracPart0, (int32_t)(two_parity_2 * change) & 0xFF);
34
35      dist[0] = __vabsdiffs4(fracPart0, HALF_CONCAT);
36      dist[1] = __vabsdiffs4(fracPart1, HALF_CONCAT);
37      int Delta = get_sum(dist[0], dist[1]);
38
39      int32_t dist00 = (dist[0] & 0xFF);
40      Delta -= (two_parity_1 + two_parity_2) * dist00;
41      Delta += two_parity_1 << 4;
42
43      if (Delta <= TWO) {
44          B_local_decode[0] = __vsub4(fracPart0_1, HALF_CONCAT);
45          B_local_decode[1] = __vsub4(fracPart1, HALF_CONCAT);
46      } else {
47          B_local_decode[0] = __vsub4(fracPart0_2, g0);
48          B_local_decode[1] = __vsub4(fracPart1, g1);
49      }
50  }
```

For $q = 16$, p must be divided by 16. We used a basis for $2E_8$, and not E_8, so every element is doubled. This is why in our code $\frac{1}{2}$ is represented by 16, 1 is represented by 32, and 2 is represented by 64. To avoid using the round function explicitly (which is not parallelized in hardware), we compute the integer and fractional parts of the vector using masks (as can be seen in lines 17–21), and then determine g in two parts, g_0 and g_1 (see lines 23 and 24). This is done by checking that the fractional part is greater than or equal to $\frac{1}{2}$ (equivalently, in our scaled basis, that the fractional part has the fourth LSB set to one). We leverage the fact that

$$x - (\lfloor x \rfloor + 0.5) = \{x\} - 0.5$$

(lines 44 and 45) and

$$x - \text{round}(x) = \lfloor x \rfloor + \{x\} - (\lfloor x \rfloor + g) = \{x\} - g$$

(lines 47 and 48). To compute the bit flip, we compute $2g_1 - 1$ (corresponding to the first element in the vector; see line 26 for reference). We add (or subtract) it from the fractional part based on the parity check we compute in lines 28 and 32, respectively.

To decode β parameters in the kernel, we encode them as indices to a predefined dictionary (of size 4). Thus, each β can be represented using 2 bits.

```
1  int beta1 = (beta_packed >> shift) & 0x3;
2  int beta2 = (beta_packed >> (shift + 2)) & 0x3;
3  int decoded_beta1 = beta_dict[beta1];
```

```
int decoded_beta2 = beta_dict[beta2];
```

Encoding and decoding kernels share the same logical structure so that the complex mapping defined for β is shared (this allows fast contiguous reads of multiple βs at once).

8.3.1 Runtime comparison of GEMV

Table 8.1: Runtime comparison of GEMV kernels on an 8192×8192 matrix using an NVIDIA A100 GPU.

Method	**Time (µs)**
Baseline (16 bits)	97
NestQuantM (4.25 bits)	60
QuIP# (2 bits)	38
QuIP# (4 bits)	∼75
int4 uniform	31

The QuIP# computation involves invoking two calls to QuIP# (2 bits), so we extrapolate the running time based on QuIP# (2 bits).

8.4 Dequantization circuit

In this section, we develop a pipelined, efficient circuit for NestQuant dequantization. Dequantization consists of two stages: first, perform a matrix multiplication between the generating matrix G and the coordinates. Then, find the closest point in the E_8 lattice scaled by q and subtract it.

We make the circuit for 4-bit quantization with $q = 16$. The matrix G is equal to the following:

$$G = \begin{pmatrix} 2 & 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}$$

Instead of multiplying by G, we will multiply by $2G$ to keep integer values; then we need to subtract the closest value in $32E_8$. Note that if $f(x) = x - \mathcal{V}_{32E_8}(x)$, then $f(x+64t) = f(x)$ for all $t \in \mathbb{Z}^8$. This means that we can keep only 6 bits of the matrix multiplication result, interpreting it as an unsigned number. We don't need to do any multiplications due to the sparsity of the matrix; the fixed bit shifts for multiplying by 2 and 4, as well as 6-bit sums, are sufficient.

Now, let's describe the circuit for computing $f(x)$. To round the point x to the closest element of $32E_8$, we consider two candidates: the closest x_1 in $32D_8$ (recall that this set contains all integer 8-vectors with coordinates divisible by 32 such that their sum is divisible by 64) and the closest x_2 in $32D_8 + 16$ (equal to $32D_8$ with 16 added to every coordinate). Then, the answer is x minus the best candidate among x_1 and x_2 by Euclidean distance to x.

We describe the procedure for obtaining x_1. If we drop the requirement for the sum of coordinates being divisible by 64, we simply have to round each coordinate to the closest value divisible by 32. There are two ways to round each number: floor and ceil. We should choose the best rounding for each component. However, if the sum of coordinates of the optimal rounding fails the divisibility by 64, we need to "flip" one of the rounding directions. In the full algorithm, we need to choose the optimal component to flip; however, always flipping the first component is close to optimal in the end-to-end quantization benchmarks (see Section 8.2), and we will use this approach.

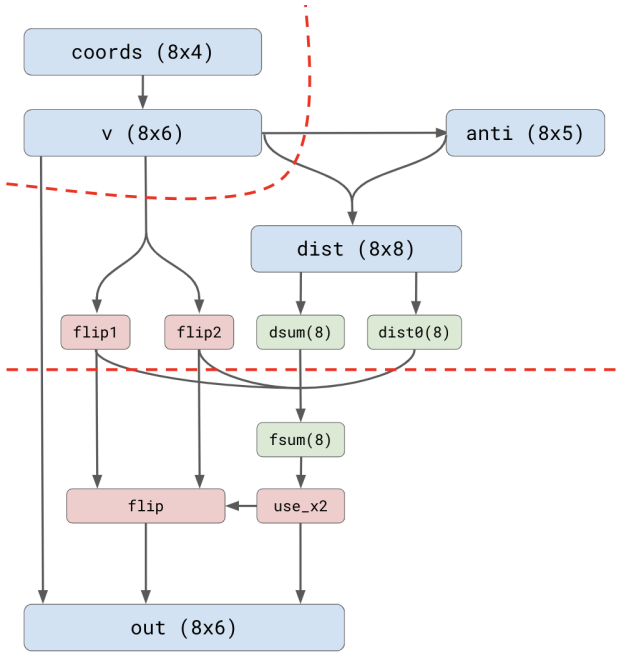


Figure 8.1: Circuit diagram of NestQuant dequantization circuit

Let x^i denote the i^{th} component of vector x and $y[k]$ denote the k^{th} least significant bit of y. Let L^i be x^i with the last 5 bits set to 0. We denote $\tilde{x}_1$ and $\tilde{x}_2$ to be x_1 and x_2 before flipping rounding. Then, $\tilde{x}_1^i$ is equal to L^i or $L^i + 32$ depending on $x^i[4]$, and $\tilde{x}_2^i$ is $L^i + 16$. Let d_i be the absolute distance between x^i and $L^i + 16$. The MSE between $\tilde{x}_1$ and x is $\sum_i (16 - d_i)^2$, and the MSE between $\tilde{x}_2$ and x is $\sum_i d_i^2$.

$$\tilde{x}_1 \text{ is better } \Leftrightarrow \sum d_i^2 \geq \sum (16 - d_i)^2 \Leftrightarrow$$
$$\sum d_i^2 - \left(\cancel{d_i^2} + 256 - 32d_i + \cancel{d_i^2}\right) \geq 0 \Leftrightarrow \left(32 \sum d_i\right) - 2048 \geq 0 \Leftrightarrow \sum d_i \geq 64$$

use_x2	$v^i[4]$	flip	value	target	out	rep
0	0	0	p	0	p	$00p$
1	0	0	p	16	$p-16$	$11p$
0	1	0	$16+p$	32	$p-16$	$11p$
1	1	0	$16+p$	16	p	$00p$
0	0	1	p	32	$p-32$	$10p$
1	0	1	p	-16	$p+16$	$01p$
0	1	1	$16+p$	0	$p+16$	$01p$
1	1	1	$16+p$	48	$p-32$	$10p$

Table 8.2: 6-bit two complement representation of the result depending on the parameters. We denote 4 least significant bits of v^i as p

We set $\Delta = \sum d_i$, and we compare Δ with 64 to select the candidate. Note that we are choosing between x_1 and x_2, not between $\tilde{x}_1$ and $\tilde{x}_2$, so we need to update Δ accordingly. One can show that if we flip the first coordinate of $\tilde{x}_1$, we need to subtract $2d_1$, and if we flip $\tilde{x}_2$, we add $32 - 2d_1$, and the comparison between Δ and 64 will become the valid criterion.

Using this information, we design the dequantization circuit with the following variables:

- **coords:** The input coordinates.
- **v:** The coordinates of the point after matmul.
- **anti:** The i-th value is equal to $16 - v^i[3:0]$.
- **d:** d_i is computed with a MUX between v^i and $anti^i$, based on $v^i[4]$, padded to 8 bits.
- **dsum:** The sum of d_i; **dist0** is d_1.
- **flip1** and **flip2:** Indicate whether the rounding direction of the first component in $\tilde{x}_1$ and $\tilde{x}_2$ needs to be flipped. One can show that $\text{flip}_2 = \bigoplus_i v^i[5]$, $\text{flip}_1 = \text{flip}_2 \oplus \bigoplus_i v^i[4]$.
- **fsum:** The updated Δ to account for flips.
- **use_x2:** True when we choose x_2; otherwise we choose x_1. **flip** is true when the candidate we chose has a flip.
- **out:** The final result. Table 8.2 shows that the 6-bit two's-complement form of out^i is the concatenation of $(\text{flip} \wedge (i = 1)) \oplus \text{use\_x2} \oplus v^i[4]$, $\text{use\_x2} \oplus v^i[4]$, and $v^i[3:0]$.

The dashed red lines in the circuit figure indicate the places where pipelines are separated. We have synthesized two versions of the circuit: with 3 pipeline stages and with 2 pipeline stages (without the second separation). We have implemented the circuit using Minispec. The synthesis results (area, critical path delay, and gate count) are presented in Table 8.3. The area is computed for 45 nm technology.

Pipeline stages	Area	Delay	Gate count
2	1117.47 um^2	549.62 ps	903
3	1350.48 um^2	354.59 ps	963

Table 8.3: Simulation results

Chapter 9

Tokenization

In the previous chapters, we used quantization to compress the components of a transformer model and improve its efficiency. In this chapter, we will look at quantization from a different angle.

Transformer models operate on sequences of discrete tokens. Language data is inherently discrete: each character has a finite number of values. The process of encoding the data into tokens is performed using a combinatorial algorithm (such as Byte Pair Encoding) and is typically lossless. For the case of a continuous-valued data domain, such as images and audio, encoding the data into discrete tokens necessarily loses information and is referred to as *tokenization*. The task of tokenization shares similarities with the task of quantization: in both cases, we are minimizing the distortion between the original data and the data reconstructed from a discrete representation.

In tokenization, the data is typically not quantized directly. This process typically consists of three stages: first, a trained encoder network processes the data and extracts the most important features from it, which are then passed through a vector quantizer. Finally, the decoder reconstructs the data object from the quantized features.

While quantization of model parameters is done primarily for efficiency, discretizing data can enhance the capabilities of generative models for the corresponding data domain. In [33], the vector quantizer is represented by an explicit vector codebook that is trained together with the weights of the encoder and decoder with a VAE objective to produce an image generation model. The authors of [34] use a vector quantizer based on RVQ to create a codec for audio compression. Finally, [35] is a work in the domain of music generation, which trains an autoregressive model on the discretized representations of music.

Our goal is to demonstrate the benefit of tokenization in the task of source separation in RF signals. We present a tokenized transformer architecture trained to remove signal interference by predicting the discrete tokens of its compressed representation. Our architecture outperforms the baselines that only use a continuous representation of data and are trained on an MSE objective.

9.1 Digital systems preliminaries

Digital communications involve transmitting bits by modulating a continuous carrier waveform. Prior to modulation, a digital communication signal can be represented in its complex baseband form as

$$u(t) = \sum_{p=-\infty}^{\infty} \sum_{\ell=0}^{L-1} c_{p,\ell}\, g(t - pT_s, \ell)\, \exp\{j2\pi\ell t/L\}. \tag{9.1}$$

Groups of bits are mapped to symbols $c_p \in \mathbb{C}$ using a *digital constellation*, which assigns bit patterns to a finite set of complex values. These symbols are then combined into a continuous complex-valued waveform via (9.1), using a *pulse shaping filter* $g(\cdot)$ to limit bandwidth and reduce inter-symbol interference [36, Sec 4.4.3]. Although the waveform appears continuous, it still bears *underlying discrete structures* due to the finite constellation and deterministic filtering.

The number of bits per symbol largely determines the constellation. Common schemes include modulating two bits at a time (Quadrature Phase Shift Keying, or QPSK) or one bit at a time (Binary Phase Shift Keying, or BPSK). Additionally, multiple groups of bits can be transmitted in parallel by considering orthogonal sub-carrier waveforms, represented by multiplication with multiple orthogonal complex sinusoids in (9.1). This is representative of Orthogonal Frequency Division Multiplexing (OFDM), inherent to many popular wireless standards such as 5G and Wi-Fi.

To recover the bits at the receiver, one may adopt *matched filtering* (MF) [37, Sec 5.8] before the estimation of the underlying symbols, and thereafter decode them back to bits. For commonly used pulse shaping functions, such as the root-raised cosine (RRC), the matched filter and pulse shaping filter coincide. We refer readers to [36–38] for a more thorough exposition of the topic.

9.1.1 Problem setup

We consider a QPSK signal of interest (SOI) mixed with a single interference source, modeled as

$$y = s + \kappa b, \tag{9.2}$$

where s represents the SOI and b is the interfering signal. In this setting, assuming unit-power signals, we can quantify the relative levels of SOI power to interference power through the signal-to-interference ratio,

$$\mathrm{SIR}(\kappa) := \frac{1}{\kappa^2}. \tag{9.3}$$

9.2 Proposed architecture

Convolutional architectures are the dominant approach for RF source separation, leveraging inductive biases inherent to digital communication signals. While effective, these models rely on large receptive fields and struggle with variable-length mixtures and real-time processing.

Motivated by the success of transformers in language and vision tasks, we propose a transformer-based architecture for RF source separation that enables large-scale learning and autoregressive decoding. We first provide an overview of our architecture, followed by a detailed description of each component.

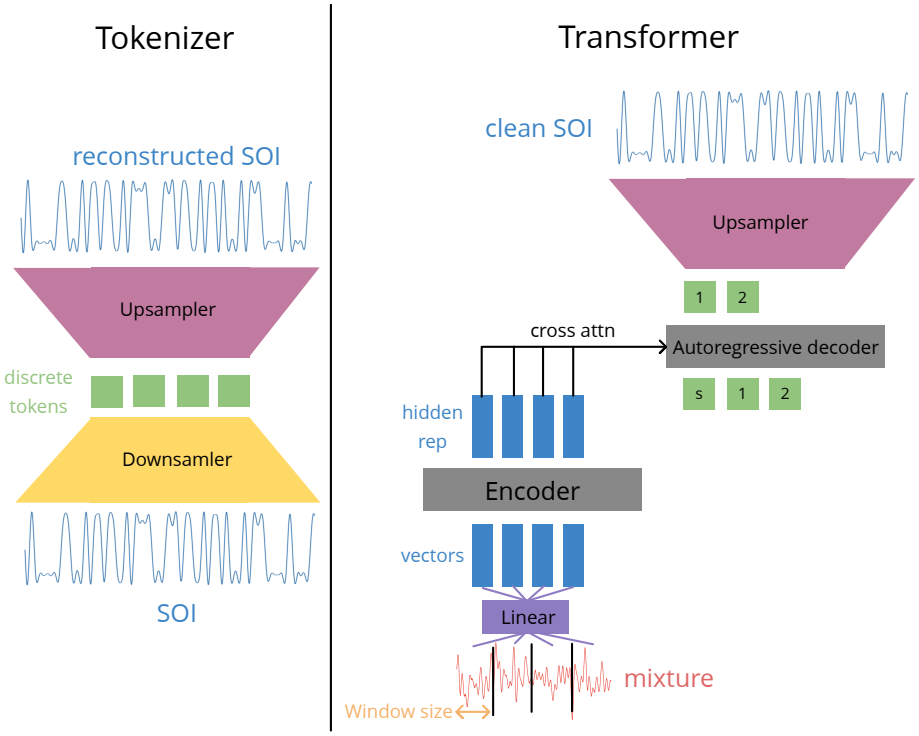


Figure 9.1: Schematic overview of the proposed architecture

9.2.1 Architecture Overview

As shown in Figure 9.1, our architecture consists of two components: a tokenizer that learns discrete representations of the SOI and a transformer that predicts a tokenized encoding of the SOI from a mixture. The tokenizer is implemented with an encoder-decoder architecture, where the encoder maps the SOI $\mathbf{s} \in \mathbb{C}^N$ to a discrete-valued sequence $\mathbf{c} \in \{1, 2, \ldots, k\}^L$ and the decoder learns the reverse mapping back to the SOI. Here k is the alphabet size defined by the total number of possible tokens. The encoded sequence length is $L = \lceil N/w \rceil$, where w is the window size that controls the number of SOI samples that are compressed into one token. The tokenizer is trained by minimizing the MSE loss between the reconstructed and ground-truth SOI waveforms.

For the transformer, we adopt an encoder-decoder architecture [39], where the mixture is processed by the encoder and the decoder predicts the tokenized SOI waveform autoregressively. Following this, the pre-trained tokenizer's decoder converts the SOI tokens into a continuous waveform, from which the underlying bits can be recovered using matched filtering.

Next, we describe these two components in more detail, starting with the tokenizer.

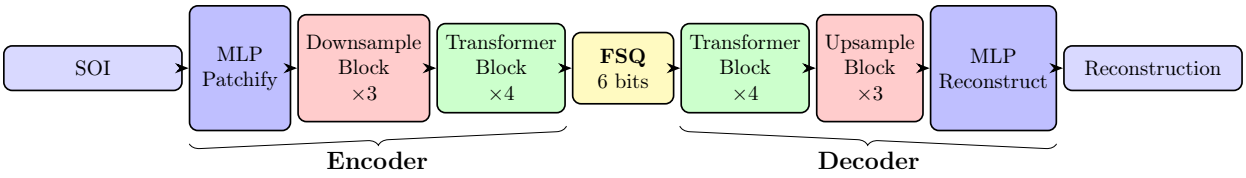


Figure 9.2: Overview of the SOI Tokenizer architecture. The main differences from the SoundStream architecture are: (i) additional transformer blocks after downsampling and before upsampling; (ii) the use of FSQ instead of RVQ for discretization; and (iii) the omission of the discriminator network.

9.2.2 The SOI Tokenizer

Our tokenizer builds on the SoundStream encoder–decoder architecture originally developed for neural audio compression [40], which uses a residual vector quantization (RVQ) module to produce discrete representations of input waveforms. However, directly applying this design to RF signals is suboptimal. To better capture the unique structure and statistical properties of RF data, we introduce several key modifications tailored specifically for RF signal tokenization.

Given the inherent discreteness of RF signals and to aid in training the transformer on practical sequence lengths, we aim to further compress the underlying information and therefore consider an extremely low-bitrate setting for tokenization. To achieve this, we substituted RVQ with finite scalar quantization (FSQ) [41], which we found to work better for this low-bitrate setup. Additionally, we found that for the QPSK SOI, adding extra transformer blocks before and after FSQ in the encoder and decoder, respectively, also leads to better validation loss. The full architecture of our tokenizer is illustrated in Figure 9.2. We train it using an MSE reconstruction loss and backpropagate through the FSQ module as in [41].

9.2.3 The RF Transformer

With a trained tokenizer for the signal of interest (SOI) in place, we can proceed to implement our source separation model. The proposed architecture is an encoder-decoder transformer trained to predict the tokenized representation of the SOI $\mathbf{s}$ from a given input mixture waveform $\mathbf{y}$.

The first step embeds the mixture signal $\mathbf{y} \in \mathbb{C}^N$ into a sequence of continuous-valued vectors. The signal is divided into non-overlapping windows of length w, with additional context of c_L samples to the left and c_R to the right of each window. Each windowed segment is linearly projected into a d-dimensional embedding, resulting in an embedding matrix $\mathsf{Z} \in \mathbb{R}^{L \times d}$, where $L = \lceil N/w \rceil$ is the number of segments. Specifically, the i-th embedding $\mathbf{z}_i$ is computed from the segment spanning positions $w \cdot i - c_L$ to $w \cdot (i+1) + c_R$, with zero padding applied when indices exceed the signal bounds. Real and imaginary components of the complex-valued input are treated as separate input dimensions during projection.

The mixture embeddings are processed by a stack of encoder blocks, while the discrete tokens corresponding to the (partially) decoded SOI are fed through a stack of decoder blocks. Each block follows the standard transformer architecture, comprising self-attention,

Table 9.1: Summary of the interference datasets used in our experiments.

Interference	Dataset Type	Description	# Recordings	Recording Length
CommSignal2	Recorded	Unknown	100	43560
CommSignal3	Recorded	Unknown	139	260000
CommSignal5G1	Synthetic	5G OFDM signal	149	230000
EMISignal1	-	Microwave emission	530	230000

normalization layers, a feedforward network, and residual connections. Additionally, each decoder block includes a cross-attention mechanism that conditions the SOI representation on the encoder's final output. Instead of standard sinusoidal positional embeddings, we adopt rotary positional embeddings [42].

The RF transformer is trained via teacher forcing with cross-entropy loss. The training dataset is composed of mixture-SOI pairs, where the SOI is tokenized. When running inference on a new mixture, we decode the tokens of the SOI autoregressively and then use the SOI tokenizer's decoder to reconstruct the signal in the waveform domain.

9.3 Experimental results

9.3.1 Experimental Setup

We evaluated our proposed architecture using four distinct mixture scenarios. Each mixture includes a QPSK SOI and is corrupted by a different real-world interference signal from the MIT RF Challenge dataset: EMISignal1, CommSignal2, CommSignal3, and CommSignal5G1. Table 9.1 summarizes the characteristics of these interference signals. Our training setup closely followed the protocol outlined in the ICASSP 2024 SP Grand Challenge on RF source separation [43].

Both the tokenizer and transformer were trained on waveform segments of length N_{train}. During training, we randomly sampled independent SOI and interference signals, cropping each to length N_{train}. This is representative of an *unsynchronized setting*, where the start of the SOI waveform may not align with the start of a QPSK symbol. As a result, direct MF decoding without compensating for symbol offset fails. Compared to the synchronized setup used in the ICASSP SP Grand Challenge, this setting is more challenging but also aids in augmenting the training data, which is vital for transformer training.

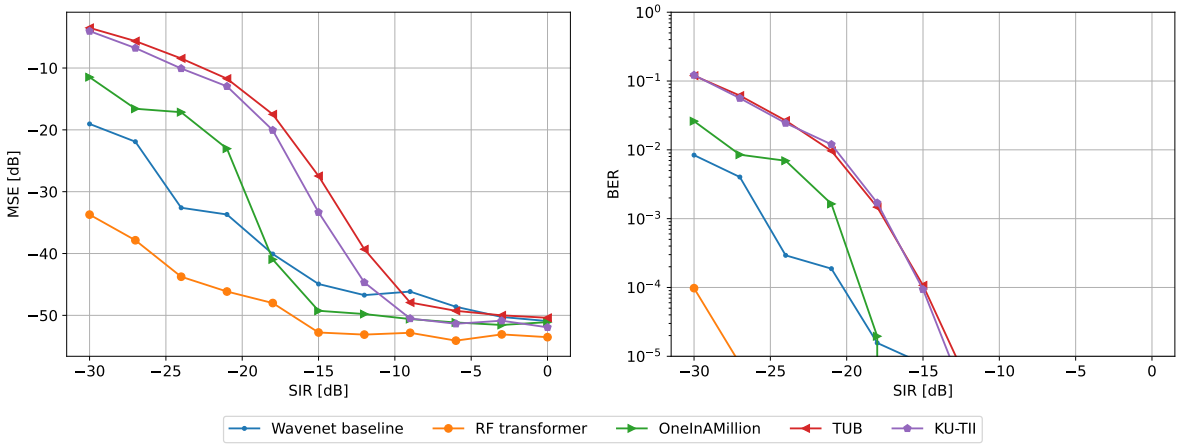
To create the mixture, we selected a random SIR from which we computed κ to define the mixture as in (9.2). In practice, we also augmented the interference signal by multiplying it with a random phase offset. Due to the limited dataset size of CommSignal2, we also applied additional transformations to the interference for this dataset, which we describe in the Appendices.

When testing, we used the signal length $N_{\text{test}} = 40960$, which could be larger than N_{train}. To deal with this scenario, we selected a set of overlapping windows of size N_{train} with stride s. We obtained an SOI estimate after decoding the tokens using the tokenizer's decoder, and the final prediction for each sample in the predicted waveform is the average of all predictions

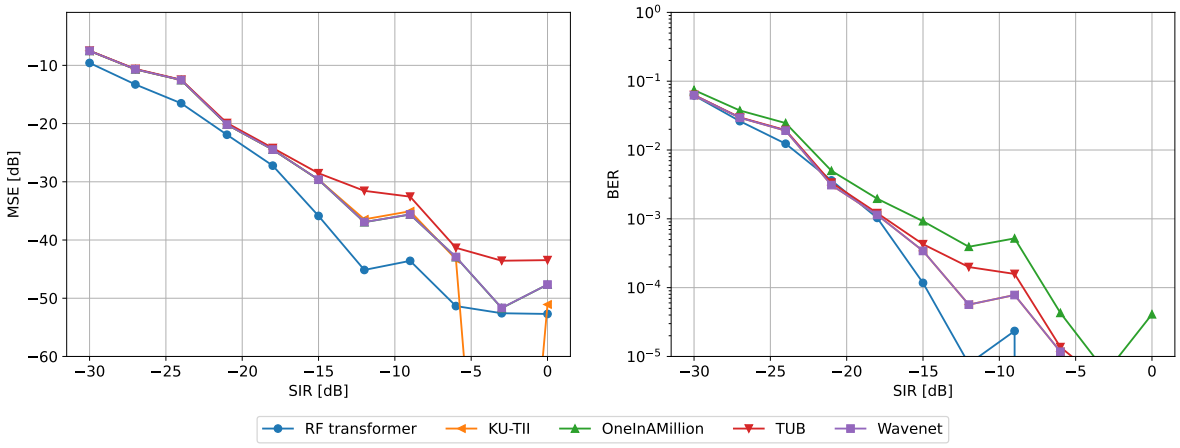
from overlapping windows. In our experiments, we typically choose $N_{\text{train}} = 2560$.

9.3.2 Results

We tested the models on a separate test set with 50 SOI-interference pairs. We evaluated performance at 11 SIR levels, ranging from -30 dB to 0 dB with a step size of 3 dB. For each SIR, we computed the average MSE of model predictions and the BER. We compared against the WaveNet and existing baselines from the ICASSP 2024 SP Grand Challenge.



(a) Performance of various methods for separating QPSK and CommSignal5G1 interference.



(b) Performance of various methods for separating QPSK and EMISignal1 interference.

Figure 9.3: Source separation performance for separating mixtures with CommSignal5G1 and EMISignal1 interference using different methods. In both cases, our proposed architecture is highly competitive and surpasses most baselines across a wide range of SIRs.

Table 9.2 summarizes the average performance of our proposed method and baselines across the datasets. For MSE, we take the average result in dB across SIRs, capping the MSE at -50 dB. We take the geometric mean of BER values, capping BER at 10^{-5}.

Table 9.2: The performance of source separation methods on all datasets

	MSE (dB)				BER ($\log_{10}$)			
Method or team	CS2	CS3	CS5G1	EMI	CS2	CS3	CS5G1	EMI
RF transformer (**ours**)	-27.22	**-6.18**	**-46.32**	**-33.01**	-2.92	-0.83	**-4.91**	**-3.52**
WaveNet	-24.14	-	-39.43	-28.92	-3.05	-	-4.23	-3.33
KU-TII	**-38.44**	-6.04	-30.17	-29.07	**-4.18**	-1.10	-3.41	-3.33
OneInAMillion	-23.54	-4.41	-37.11	-28.92	-3.03	-0.86	-3.94	-2.97
TUB	-25.54	-4.97	-28.85	-26.88	-2.95	-0.92	-3.41	-3.23

Our method delivers strong performance across diverse interference types. As shown in Figure 9.3a, it substantially outperforms baseline models on CommSignal5G1 and achieves state-of-the-art results for EMISignal1 (Figure 9.3b). For mixtures with CommSignals 2 and 3, it remains on par with the best-performing methods. In the 5G interference case, our RF transformer attains an average BER of 9.59×10^{-6}—a 122× reduction compared to 1.17×10^{-3} for the WaveNet baseline. KU-TII achieves the highest score on CommSignal2 using additional synthetic training data, whereas our model, like most others, is trained solely on the provided dataset.

Chapter 10

Conclusion

10.1 RF transformer

We conclude that using a tokenized representation of the target (signal of interest) improves source separation performance. In future work, we could investigate a causal RF transformer, which can perform interference cancellation in real time.

10.2 NestQuant

NestQuant consists of two main innovations: an optimized codebook based on Voronoi codes and a new optimization objective (QA-LDLQ) with an optimization algorithm that reduces it to the regular LDLQ objective. NestQuant outperforms state-of-the-art methods for KV-cache and activation quantization and is competitive with weight-only quantization methods. We also introduce NestQuantM — a simplified algorithm for which it is possible to implement a fast GPU kernel, and we design a circuit that performs NestQuantM decoding efficiently.

For future work, it is important to investigate the following extensions of NestQuant:

- It might be beneficial to apply MMSE scaling and dithering to the codebook to improve robustness
- NestQuant can be integrated with a more advanced objective from Section 3.4 or with optimized rotation matrices from Section 3.2
- More research is needed to understand the performance of GPU kernels with NestQuant, as well as of specialized hardware that supports vector quantization
- Integrating vector quantization into accelerators is a promising direction for Deep Learning hardware, but designing chips for a specific quantization protocol is a significant commitment. Further research is needed to identify a universal and efficient codebook for integration. While the current version of NestQuant is not proposed as the definitive standard, it provides a strong baseline for future work.

References

[1] J. Conway and N. Sloane. "A Fast Encoding Method for Lattice Codes and Quantizers". In: *IEEE Transactions on Information Theory* 29.6 (1983), pp. 820–824. DOI: 10.1109/TIT.1983.1056761.

[2] S. Savkin, E. Porat, O. Ordentlich, and Y. Polyanskiy. "NestQuant: Nested Lattice Quantization for Matrix Products and LLMs". In: *arXiv preprint arXiv:2502.09720* (2025).

[3] C. E. Shannon. "Coding Theorems for a Discrete Source with a Fidelity Criterion". In: *IRE National Convention Record* 4 (1959), pp. 142–163.

[4] S. P. Lloyd. "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982). Originally published as Bell Laboratories Technical Note, 1957, pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[5] M. W. Marcellin and T. R. Fischer. "Trellis Coded Quantization of Memoryless and Gauss–Markov Sources". In: *IEEE Transactions on Communications* 38.1 (Jan. 1990). Presented in part at the 1988 ISIT, Kobe, Japan; NSF Grant NCR-8821764, pp. 82–93. DOI: 10.1109/26.46532.

[6] R. Zamir and M. Feder. "On Universal Quantization by Randomized Uniform/Lattice Quantizers". In: *IEEE Transactions on Information Theory* 38.2 (1992), pp. 428–436. DOI: 10.1109/18.119699.

[7] O. Ordentlich and Y. Polyanskiy. *Optimal Quantization for Matrix Multiplication*. 2024. arXiv: 2410.13780 [cs.IT]. URL: https://arxiv.org/abs/2410.13780.

[8] R. Sennrich, B. Haddow, and A. Birch. "Neural machine translation of rare words with subword units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. 2016, pp. 1715–1725.

[9] Llama Team, AI @ Meta. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[10] E. Frantar, S. P. Singh, and D. Alistarh. "Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning". In: *Advances in Neural Information Processing Systems 36 (NeurIPS 2022)*. 2022.

[11] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. "LLM.int8(): 8-Bit Matrix Multiplication for Transformers at Scale". In: *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*. 2022, pp. 29498–29512.

[12] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models". In: *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. Vol. 202. Proceedings of Machine Learning Research. 2023, pp. 3732–3754.

[13] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. "AWQ: Activation-Aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems* 6 (2024), pp. 87–100.

[14] J. Chee, Y. Cai, V. Kuleshov, and C. D. Sa. *QuIP: 2-Bit Quantization of Large Language Models with Guarantees*. See entry `chee2024` for archival details. 2024. arXiv: 2307.13304 [cs.LG]. URL: https://arxiv.org/abs/2307.13304.

[15] S. Ashkboos, A. Mohtashami, M. L. Croci, B. Li, P. Cameron, M. Jaggi, D. Alistarh, T. Hoefler, and J. Hensman. "QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs". In: *Advances in Neural Information Processing Systems 38 (NeurIPS 2024)*. 2024.

[16] Z. Liu, C. Zhao, I. Fedorov, B. Soran, D. Choudhary, R. Krishnamoorthi, V. Chandra, Y. Tian, and T. Blankevoort. *SpinQuant: LLM Quantization with Learned Rotations*. 2024. arXiv: 2405.16406 [cs.LG]. URL: https://arxiv.org/abs/2405.16406.

[17] Y. Sun et al. "FlatQuant: Flatness Matters for LLM Quantization". In: *arXiv preprint arXiv:2410.09426* (2024). Submitted to ICLR 2025; poster at ICML 2025.

[18] A. Tseng, J. Chee, Q. Sun, V. Kuleshov, and C. D. Sa. "QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks". In: *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*. Vol. 235. Proceedings of Machine Learning Research. 2024, pp. 48630–48656.

[19] A. Défossez, J. Copet, G. Synnaeve, and Y. Adi. "High Fidelity Neural Audio Compression". In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. PMLR, 2023, pp. 7480–7512. URL: https://proceedings.mlr.press/v202/defossez23a.html.

[20] A. Tseng, Q. Sun, D. Hou, and C. D. Sa. "QTIP: Quantization with Trellises and Incoherence Processing". In: *Advances in Neural Information Processing Systems 38 (NeurIPS 2024)*. 2024.

[21] J. Kim, M. El Halabi, W. Park, C. J. Schaefer, D. Lee, Y. Park, J. W. Lee, and H. O. Song. "GuidedQuant: Large Language Model Quantization via Exploiting End Loss Guidance". In: *arXiv preprint arXiv:2505.07004* (May 2025). Submitted May 11, 2025, available at arXiv.

[22] A. Tseng, Z. Sun, and C. De Sa. "Model-Preserving Adaptive Rounding". In: *arXiv preprint arXiv:2505.22988* (2025). Introduces YAQA: Yet Another Quantization Algorithm. DOI: 10.48550/arXiv.2505.22988. URL: https://arxiv.org/abs/2505.22988.

[23] R. Zamir. *Lattice Coding for Signals and Networks: A Structured Coding Approach to Quantization, Modulation, and Multiuser Information Theory*. Cambridge University Press, 2014.

[24] E. Agrell and B. Allen. "On the Best Lattice Quantizers". In: *IEEE Transactions on Information Theory* (2023).

[25] J. Conway and N. Sloane. "Fast Quantizing and Decoding Algorithms for Lattice Quantizers and Codes". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 227–232. DOI: 10.1109/TIT.1982.1056484.

[26] H. Lin, H. Xu, Y. Wu, J. Cui, Y. Zhang, L. Mou, L. Song, Z. Sun, and Y. Wei. "DuQuant: Distributing Outliers via Dual Transformation Makes Stronger Quantized LLMs". In: *Advances in Neural Information Processing Systems 38 (NeurIPS 2024)*. 2024.

[27] S. Merity, C. Xiong, J. Bradbury, and R. Socher. "Pointer Sentinel Mixture Models". In: *International Conference on Learning Representations (ICLR 2017)*. 2017.

[28] X. Hu, Y. Cheng, D. Yang, Z. Xu, Z. Yuan, J. Yu, C. Xu, Z. Jiang, and S. Zhou. "OstQuant: Refining Large Language Model Quantization with Orthogonal and Scaling Transformations for Better Distribution Fitting". In: *International Conference on Learning Representations (ICLR 2025)*. 2025.

[29] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. *Think You Have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge*. 2018. arXiv: 1803.05457 [cs.AI]. URL: https://arxiv.org/abs/1803.05457.

[30] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. "HellaSwag: Can a Machine Really Finish Your Sentence?" In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL 2019)*. 2019, pp. 4791–4800.

[31] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi. "PIQA: Reasoning about Physical Commonsense in Natural Language". In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*. 2020, pp. 7432–7439.

[32] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. "WinoGrande: An Adversarial Winograd Schema Challenge at Scale". In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*. 2020, pp. 8732–8740.

[33] A. van den Oord, O. Vinyals, and K. Kavukcuoglu. "Neural discrete representation learning". In: *Advances in neural information processing systems* 30 (2017).

[34] A. Defossez, J. Copet, G. Synnaeve, and Y. Adi. "High fidelity neural audio compression". In: *Proceedings of the 39th International Conference on Machine Learning*. PMLR. 2022, pp. 5370–5380.

[35] A. Agostinelli et al. "MusicLM: Generating Music From Text". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2023, pp. 1–5.

[36] R. W. Heath Jr. *Introduction to wireless digital communication: a signal processing perspective*. Prentice Hall, 2017.

[37] A. Lapidoth. *A foundation in digital communication*. Cambridge University Press, 2017.

[38] A. Goldsmith. *Wireless communications*. Cambridge University Press, 2005.

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

[40] N. Zeghidour, A. Luebs, A. Omran, J. Skoglund, and M. Tagliasacchi. "Soundstream: An end-to-end neural audio codec". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 30 (2021), pp. 495–507.

[41] F. Mentzer, D. Minnen, E. Agustsson, and M. Tschannen. "Finite scalar quantization: Vq-vae made simple". In: *arXiv preprint arXiv:2309.15505* (2023).

[42] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. "Roformer: Enhanced transformer with rotary position embedding". In: *Neurocomputing* 568 (2024), p. 127063.

[43] T. Jayashankar, B. Kurien, A. Lancho, G. Lee, Y. Polyanskiy, A. Weiss, and G. Wornell. "The Data-Driven Radio Frequency Signal Separation Challenge". In: *2024 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*. 2024, pp. 53–54. DOI: 10.1109/icasspw62465.2024.10627554. URL: https://doi.org/10.1109/icasspw62465.2024.10627554.