HyperChr: QUANTIZATION OF HETEROGENEOUSLY DISTRIBUTED MATRICES THROUGH DISTRIBUTION-AWARE SUBSPACE PARTITIONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Matrix quantization is crucial for reducing the memory footprint of matrices across various applications, including large-scale machine learning models and data compression. We have observed that matrices in different application domains exhibit heterogeneity in the distribution across columns. Leveraging this characteristic, we introduce *HyperChr*, a novel matrix quantization algorithm tailored for heterogeneous data distributions prevalent across different matrix columns. Unlike traditional quantization methods, *HyperChr* capitalizes on the heterogeneous distribution characteristics of each column to optimally partition high-dimensional subspaces and perform compression within each subspace. This technique enhances the compression effectiveness by grouping vectors with similar distribution ranges, enabling more precise quantization. Moreover, *HyperChr* dynamically adjusts the number of centroids in each subspace based on the specific data distribution traits, optimizing both storage efficiency and data fidelity.

We evaluate *HyperChr*'s performance on diverse datasets, demonstrating its superiority in reducing quantization errors compared to existing methods. Our results show that *HyperChr* exhibits significant improvements at lower compression ratios ($\theta = 2 - 8$), reducing MAE by an average of 55.3% and MSE by 75.3% compared to PQ. However, at higher compression ratios ($\theta = 10 - 16$), the improvements are more moderate, with an average reduction of 14.9% in MAE and 25.9% in MSE compared to PQ. In addition, our algorithm reduces the average dequantization time by 62.9%, which is crucial for large language model inference.

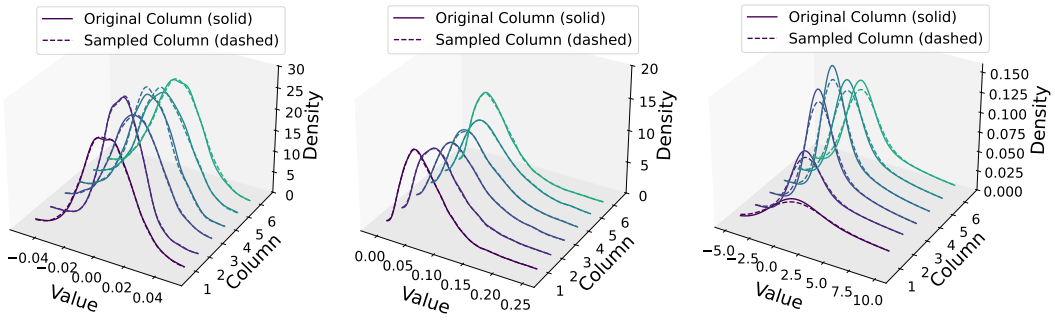
1 INTRODUCTION

Matrix quantization refers to the process of reducing the memory footprint of a matrix while minimizing certain performance metrics, such as mean squared error (MSE) or cross-entropy. The goal is to represent matrix elements in a more compact format without sacrificing too much accuracy. When the quantized matrix needs to be used, it can be dequantized to restore the matrix, enabling further computations with minimal loss in precision.

With the rise of big data and large models, the computation and storage of large matrices have become increasingly critical. For instance, GPT-2 contains 1.5 billion parameters Solaiman et al. (2019), whereas GPT-3 has expanded to 175 billion parameters Brown et al. (2020). Therefore, matrix quantization plays a crucial role in Large Language Models (LLMs) weight quantization Dettmers et al. (2024); Lin et al. (2024); Shao et al. (2023); Xiao et al. (2023). Additionally, the KV cache accounts for over 30% of GPU memory during deployment, compared to the 65% occupied by the parameters Kwon et al. (2023). Both the parameters and KV cache are stored in the form of matrices, and many recent model quantization techniques have been applied to LLM k-v cache quantization Liu et al. (2024); Zhang et al. (2024); Hooper et al. (2024); Kawakibi Zuhri et al. (2024); Duanmu et al. (2024); Yue et al. (2024); Lee et al. (2024); Adnan et al. (2024). In Q4 2023, Facebook reported 3.05 billion monthly active users, with user data typically represented as graphs and matrices fac. Consequently, matrix quantization is widely used in graph compression Brisaboa et al. (2009); Claude & Ladra (2011). Matrix quantization also finds applications in other fields, such

as vector databases Xu et al. (2018); Jegou et al. (2010) and image compression Yu et al. (2018); Ning et al. (2016).

There has been a growing body of related work on matrix quantization, which can be broadly classified into two categories: general-purpose matrix compression algorithms and task-specific matrix compression algorithms. General-purpose matrix compression techniques include product quantization (PQ) Jegou et al. (2010), optimized product quantization (OPQ) Ge et al. (2013), and locally optimized product quantization (LOPQ) Kalantidis & Avrithis (2014). In contrast, task-specific matrix compression algorithms are designed for particular use cases, such as large language models (LLMs) or other machine learning applications. These include methods like LLM.int8 Ge et al. (2013), Optimal Brain Damage LeCun et al. (1989), GPTQ Frantar et al. (2022), AWQ Lin et al. (2023), SmoothQuant Xiao et al. (2023), OmniQuant Shao et al. (2023), and SqueezeLLM Kim et al. (2023). The first category of methods primarily builds on the PQ algorithm, focusing on making various optimizations for general use, while the second category integrates task-specific heuristics into the Round-To-Nearest (RTN) algorithm Gray & Neuhoff (1998), which quantizes elements by rounding each value to its nearest representable level.



(a) LLM weight distribution. (b) GIST distribution. (c) SIFT distribution.

Figure 1: Column distribution of different datasets.

However, many real-world datasets exhibit heterogeneous distributions across different columns. This heterogeneity is particularly prevalent in applications such as Large Language Model (LLM) weights, LLM key-value (KV) caches, image retrieval, and other applications, where certain columns of data display vastly different statistical properties compared to others.

Figures 1a, 1b, and 1c illustrate the column distributions for LLM weights, GIST cache, and SIFT descriptor, respectively. In each figure, the solid line represents the distribution of the entire column, while the dashed line shows the distribution when randomly selecting half of the data from that column. These visualizations highlight that while different columns exhibit unique distributions, the distributions within the same column are quite consistent, pointing towards the necessity for distribution-aware quantization strategies.

Traditional quantization methods, which typically assume a homogeneous distribution across all elements, often yield suboptimal results when applied to these heterogeneous matrices, ignoring the diverse nature of real-world data.

Existing matrix quantization algorithms, such as Product Quantization (PQ), RTN, and its variations, have demonstrated promising performance. Nevertheless, they do not explicitly consider heterogeneity in data distributions. These methods typically apply uniform quantization without adapting to the diverse nature of the data. This uniform approach can lead to suboptimal performance when dealing with matrices that exhibit heterogeneous data distributions across their dimensions.

In this paper, we introduce *HyperChr* (HyperChromosome), a novel algorithm tailored to exploit the heterogeneity in data distributions across matrix columns. Recognizing that data exhibit diverse distributions along each column, *HyperChr* strategically partitions high-dimensional subspaces according to the distribution properties of each column. This approach allows for the grouping of vectors with similar distribution ranges into the same subspace, enhancing the efficacy of the quantization process. Additionally, we propose a method to dynamically determine the compression ratio (number of centroids) for each subspace based on its characteristics.

We validate the effectiveness of *HyperChr* across a range of datasets, demonstrating that it consistently achieves lower quantization errors compared to conventional methods. The results show that the *HyperChr* algorithm exhibits significant improvements at lower compression ratios[1] ($\theta = 2-8$), reducing MAE by an average of 55.3% and MSE by 75.3% compared to PQ. However, for higher compression ratios ($\theta = 10 - 16$), the improvements are more moderate, with an average reduction of 14.9% in MAE and 25.9% in MSE compared to PQ. In addition, our algorithm reduces the average dequantization time by 62.9%. Fast dequantization is crucial for large Language model inference. Our code has been open-sourced [2].

Our contributions in this work are threefold:

- We identify the heterogeneous distribution characteristics of matrices in different application scenarios.
- Leveraging the heterogeneous distribution properties of matrices, we design the *HyperChr* algorithm to improve the performance of matrix compression.
- We dynamically adjust the intervals according to the varying distribution characteristics of the data.

2 PROBLEM SETTING

In this section, we formally define the problem of heterogeneous matrix quantization. We then discuss the feasibility of utilizing these heterogeneous distributions to enhance the effectiveness of matrix quantization.

We define the matrix to be quantized as X, the quantized matrix as X^q, and the matrix recovered through dequantization as X'. Both X and X' are $n \times d$ matrices, while X^q depends on the quantization method. The elements of the matrices are denoted as $x_{i,j}$, $x^q_{i,j}$, and $x'_{i,j}$, respectively.

The memory usage of a matrix is given by memory(), and the memory constraint is denoted as mem_constraint.

Definition 1 (Heterogeneous Matrix Quantization)**.** The goal of Heterogeneous Matrix Quantization is to minimize the following objective function, which balances the Mean Squared Error (MSE) and the memory used by the quantized matrix:

$$\textbf{minimize } \text{MSE}(X, X') + \lambda \cdot \text{memory}(X^q),$$

where the MSE between X and X' is defined as:

$$\text{MSE}(X, X') = \frac{1}{n \cdot d} \sum_{i=1}^{n} \sum_{j=1}^{d} \left(x_{i,j} - x'_{i,j}\right)^2.$$

The quantization process is subject to a memory constraint:

$$\text{memory}(X^q) \leq \text{mem\_constraint},$$

and considers the column-wise heterogeneous distribution of the matrix, such that:

$$X_{:,j} \sim \mathcal{D}_j \quad \text{for} \quad j = 1, 2, \ldots, d,$$

where $\mathcal{D}_j$ represents the distinct distribution governing the values in column j of matrix X.

Here, λ is a regularization parameter that adjusts the trade-off between minimizing MSE and the memory footprint of X^q.

For datasets with column-wise heterogeneous distributions, directly applying quantization algorithms can lead to inefficiencies for two main reasons. On the one hand, the value ranges in different columns can vary significantly. For example, in the RTN (Round-to-Nearest) algorithm, a column a might have values ranging from -10 to 10, while another column b might only span from -1 to 1. Applying a uniform quantization over the range $[-10, 10]$ would lead to wasted quantization

[1] Compression ratio = $\frac{\text{Space occupied by the matrix before quantization}}{\text{Space occupied by the matrix after quantization}}$

[2] https://github.com/HyperChr-release/HyperChr-release

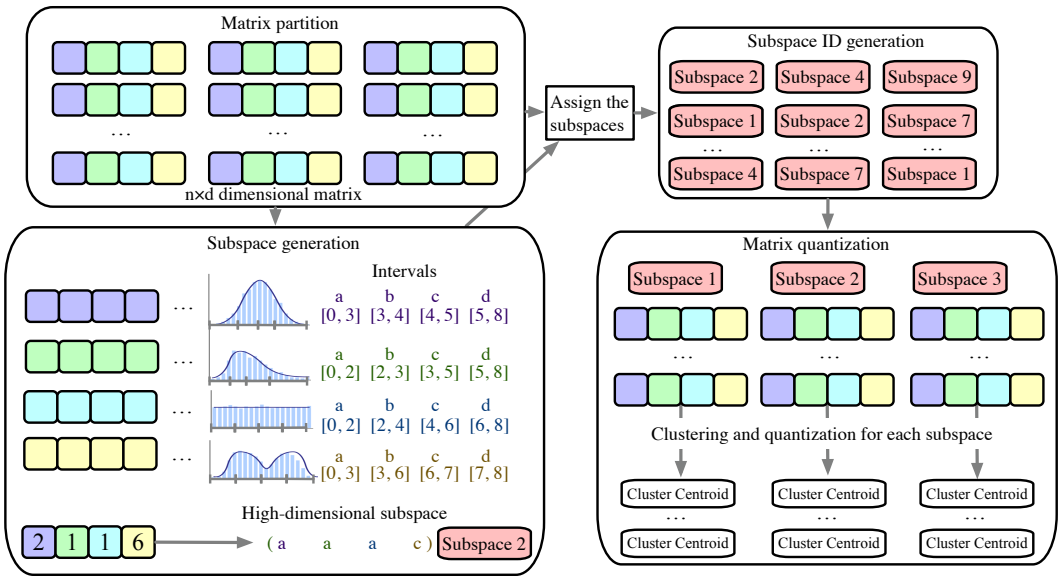


Figure 2: Overview of the *HyperChr* algorithm.

levels for column b, as the regions $[-9, -1]$ and $[1, 9]$ would contain no data points, reducing the efficiency of the quantization process. On the other hand, even when columns c and d have similar ranges, such as $[-10, 10]$, their data densities within that range may differ. For instance, column d may have very few or no data points in the range $[-9, 9]$, which would again result in inefficiencies for quantization algorithms that do not account for these distributional differences. Thus, leveraging the heterogeneous nature of data distributions across columns to optimize matrix compression is both feasible and beneficial. By adapting the quantization strategy to the unique distribution of each column, it is possible to achieve more efficient and accurate compression, preserving memory while minimizing quantization error.

3 METHODS

The core idea of *HyperChr* is to leverage the characteristic that data within the matrix column have similar distributions, while the distributions differ across columns. By partitioning data within each column based on their distribution, different column intervals form high-dimensional subspaces. The vectors within each high-dimensional subspace exhibit similarity, leading to more effective quantization results.

3.1 ALGORITHM LOGIC

The design and operations of the HyperChr algorithm are illustrated in Figure 2, which provides a conceptual overview of the algorithm's workflow and its key components. The algorithm is structured into four main parts: matrix partition, subspace generation, subspace ID generation, and matrix quantization. Each component plays a crucial role in optimizing the quantization process.

Matrix Partition: Firstly, during the matrix partition phase, we pre-partition the columns of the original matrix to reduce the computational complexity of subsequent operations and to avoid the exponential explosion associated with high-dimensional subspaces. Specifically, we divide the matrix's columns into several groups, such as grouping every four columns together as shown in Figure 2 This strategy helps reduce computational and storage overhead when dealing with large-scale data.

Subspace Generation: During the subspace generation phase, we generate intervals for columns in the same position across different groups based on their combined data distribution. For example, as illustrated in Figure 2, the first column of the first group and the first column of the second group are considered together. Considering each column's data distribution separately would increase the

storage space of intervals and lead to suboptimal results; in the example shown, it would triple the storage space. These intervals reflect the range and distribution characteristics of the data in each column. By combining intervals from different columns, we generate high-dimensional subspaces.

Subspace ID Generation: In the subspace ID generation phase, we map each vector in the original matrix using the intervals generated in the subspace generation step. Specifically, we determine the combination of intervals where each vector's elements fall, thereby assigning it a unique subspace identifier. This process maps data from the original high-dimensional space into predefined subspaces, facilitating subsequent quantization operations.

Matrix Quantization: Finally, in the Matrix Quantization phase, we perform independent quantization for the set of vectors within each subspace. Since vectors within a subspace exhibit similar features, the quantization algorithm can improve quantization effectiveness and accuracy. We compress the vectors in each subspace into several centroids using a clustering algorithm, and then use the indices of the centroids to replace the data in the original matrix.

3.2 *HyperChr* QUANTIZATION

Algorithm 1 Matrix Quantization Algorithm

Input: Matrix $X \in \mathbb{R}^{n \times d}$, number of subgroups m, number of intervals l for each dimension
Output: Quantized matrix X^q, Codebook $\mathbf{C}$
Step 1: Matrix Partition
Partition X into m column groups $\{G_1, G_2, \ldots, G_m\}$, where each group has $s = \frac{d}{m}$ columns
Step 2: Subspace Generation
for each dimension $k = 1$ to s **do**
    Collect columns $\{G_1^{(k)}, G_2^{(k)}, \ldots, G_m^{(k)}\}$ ▷ $G_i^{(k)}$ is the k-th column of group G_i
    Combine data from these columns to determine l intervals $\{I_{k,1}, I_{k,2}, \ldots, I_{k,l}\}$ based on the l-quantiles of their distribution
    $\mathbf{I}_k \leftarrow \{I_{k,1}, I_{k,2}, \ldots, I_{k,l}\}$
    $\mathbf{I} \leftarrow \mathbf{I} \cup \mathbf{I}_k$
Step 3: Subspace ID Generation
for each group G_i in $\{G_1, G_2, \ldots, G_m\}$ **do**
    for each data point $g_{i,j}$ in group G_i **do** ▷ $g_{i,j}$ is the j-th row vector of group G_i
        for each dimension $k = 1$ to s **do**
            Assign interval index $idx_{i,j,k}$ such that $g_{i,j,k} \in I_{k,idx_{i,j,k}}$
        Concatenate indices $\mathbf{idx}_{i,j} = [idx_{i,j,1}, idx_{i,j,2}, \ldots, idx_{i,j,s}]$
        Compute subspace ID $S_{j,i} = \text{ComputeSubspaceID}(\mathbf{idx}_{i,j})$ ▷ In matrix S, the j-th row and i-th column correspond to group G_i and row j
Step 4: Matrix Quantization
for each unique subspace identifier s in set S **do**
    Collect the vector $V_s = \{g_{i,j} \mid S_{j,i} = s\}$
    Calculate the number of centroids k_s, $k_s = \text{calcCentroids}(V_s)$
    Apply clustering to V_s to obtain k_s centroids, $\mathbf{C}_s$
    Add centroids $\mathbf{C}_s$ to the global codebook $\mathbf{C}$, and assign new indices for each centroid in $\mathbf{C}$
    for each vector $g_{i,j}$ in subspace V_s **do**
        $X_{j,i}^q \leftarrow \arg\min_{\mathbf{c}_j} \|g_{i,j} - c_j\|$ ▷ Quantize to nearest centroid within group
        Replace the centroid index in $X_{j,i}^q$ with the new index in the global codebook $\mathbf{C}$
return Quantized matrix X^q, Codebook $\mathbf{C}$

As shown in Algorithm 1, the matrix quantization process begins by partitioning the input matrix $X \in \mathbb{R}^{n \times d}$ into m column groups, each containing $s = \frac{d}{m}$ columns. For each dimension k within these groups, subspace intervals are generated based on the data distribution's quantiles, and these intervals are stored in $\mathbf{I}_k$. Each data point within the matrix is then assigned a subspace ID by determining the interval indices across all dimensions for the corresponding group.

Once the subspace IDs have been computed, the quantization process begins. For each unique subspace identifier, the corresponding vectors are collected, and a clustering algorithm (e.g., k-means) is applied to these vectors to determine centroids. The number of centroids k_s for each

subspace is determined based on the distribution of the vectors within that subspace. The centroids are stored in subspace-specific codebooks $\mathbf{C}_s$, which are then merged into a global codebook $\mathbf{C}$. Each data point is quantized by replacing it with the nearest centroid, and the index of the centroid from the global codebook is stored in the quantized matrix X^q.

Finally, the quantized matrix X^q, containing indices that point to centroids in the global codebook $\mathbf{C}$, and the codebook $\mathbf{C}$ itself are returned.

The HyperChr algorithm optimizes the quantization process through a structured approach, as illustrated in Figure 2. The matrix partition phase pre-partitions the columns of the matrix to simplify computational requirements. Subspace generation then creates high-dimensional subspaces by combining intervals from the grouped columns based on their data distribution. Subspace IDs are generated by mapping each vector to these subspaces based on its elements' interval locations. Finally, matrix quantization is performed within each subspace, enhancing quantization effectiveness and reducing overall memory usage.

3.2.1 *HyperChr* DEQUANTIZATION

As shown in Algorithm 2, the matrix dequantization process reconstructs the original matrix X from the quantized matrix X^q and the codebook $\mathbf{C}$. For each element $X^q_{i,j}$ in the quantized matrix, the algorithm retrieves the corresponding centroid c_k from the codebook $\mathbf{C}$ using the index stored in $X^q_{i,j}$. The retrieved centroid c_k is then placed into the corresponding position in the reconstructed matrix X, such that $X_{i,j} \leftarrow c_k$. This process is repeated for all elements in X^q, resulting in the full reconstruction of the original matrix X.

Algorithm 2 Matrix Dequantization Algorithm

1: **Input**: Quantized matrix X^q, Codebook $\mathbf{C}$
2: **Output**: Reconstructed matrix X
3: **for** each element $X^q_{i,j}$ in the quantized matrix X^q **do**
4: Retrieve the corresponding centroid c_k from the codebook $\mathbf{C}$ using the index $X^q_{i,j}$
5: Assign $X_{i,j} \leftarrow c_k$ ▷ Place the centroid value into the reconstructed matrix X
6: **return** Reconstructed matrix X

3.3 THEORETICAL ANALYSIS OF *HyperChr*

We first present a comparison of *HyperChr* and PQ in terms of MSE, followed by a comparison of the time complexity between PQ and *HyperChr*.

Theorem 1. (Theoretical analysis of MSE) The Mean Squared Error (MSE) of the *HyperChr* algorithm is lower than that of the PQ algorithm:

$$\text{MSE}_{\text{HyperChr}} = \Gamma \cdot \text{MSE}_{\text{PQ}}, \tag{1}$$

where $0 < \Gamma < 1$ is the effective variance reduction factor achieved by *HyperChr*, and the proof can be found in Appendix A.1.

According to Theorem 1, since $\Gamma < 1$, it follows that:

$$\text{MSE}_{\text{HyperChr}} = \Gamma \cdot \text{MSE}_{\text{PQ}} < \text{MSE}_{\text{PQ}}. \tag{2}$$

Thus, the MSE of the *HyperChr* algorithm is lower than that of the PQ algorithm.

Theorem 2. (Time Complexity for Quantization) The quantization time complexity of *HyperChr* method is:

$$QT_{\text{HyperChr}} = \frac{1}{m \cdot l^{\frac{d}{m}}} \cdot QT_{\text{PQ}}, \tag{3}$$

where QT_{HyperChr} and QT_{PQ} are the time complexities of our method and the PQ algorithm, respectively. The proof can be found in Appendix A.2.

Since $\frac{1}{m \cdot l \frac{d}{m}} < 1$, the time complexity of *HyperChr* is lower than that of PQ.

Theorem 3. (Time Complexity for Dequantization) The time complexity of the dequantization process in the Matrix Dequantization Algorithm is:

$$DQT_{\text{HyperChr}} = \frac{1}{m} \cdot DQT_{\text{PQ}}, \tag{4}$$

where DQT_{HyperChr} and DQT_{PQ} represent the time complexities of the Matrix Dequantization Algorithm and the PQ algorithm, respectively. The proof can be found in Appendix A.3.

The Matrix Dequantization process of *HyperChr* has a lower time complexity than that of PQ, as $\frac{1}{m} < 1$.

Thus, we draw three conclusions:

- The *HyperChr* algorithm achieves a lower Mean Squared Error (MSE) compared to the PQ algorithm due to its effective subspace partitioning strategy.
- The quantization time complexity of the *HyperChr* method is lower than that of the PQ algorithm.
- The dequantization process of the *HyperChr* algorithm also has a lower time complexity compared to the PQ algorithm.

4 Experiments

4.1 Experiment setup

Platform and Implementation: We performed our algorithm evaluations on a high-performance server configured with an Intel Core i9-10980XE processor, which boasts 18 cores and 36 threads, operating at a base frequency of 3.00 GHz. The server is equipped with 128GB of 3200MHz DDR4 memory and a 24.8MB L3 cache, ensuring robust computational capabilities. All algorithms were implemented in Python, specifically version 3.8.10. In each experimental scenario, we repeated the evaluation 100 times.

Dataset: To evaluate the effectiveness of our proposed algorithm, we conducted experiments using one synthetic dataset and three real-world datasets: a synthetic normal distribution dataset, LLM weight dataset, GIST dataset, and SIFT dataset. Below, we provide a detailed description of each dataset. Each element in the matrix is represented with 32 bits.

(1) Synthetic normal distribution dataset: The dataset was constructed by sampling each entry of the matrix from a truncated normal distribution, characterized by a mean of 0.5 and a standard deviation of 0.16. Furthermore, approximately one element out of every ten thousand was substituted with an outlier, randomly selected from the interval [-100, 100]. The matrix dimensions are set to 1,024 $\times$ 128.

(2) LLM weight dataset: This dataset includes weight matrices derived from the large language model (LLM) LLaMA2 (Touvron et al. (2023)). The dimensions of the LLM weight matrices are 11,008 $\times$ 4096.

(3) GIST dataset: The GIST dataset consists of feature vectors used for image recognition and retrieval tasks. It contains the first 100,000 features extracted from the Tiny Image dataset (Torralba et al. (2008)). Moreover, this dataset has been employed in Jegou et al. (2010) and Oliva & Torralba (2001). The dimensions of the GIST dataset are 100,000 $\times$ 960.

(4) SIFT dataset: The SIFT descriptors were obtained by extracting the learning set from Flickr images, while the database and query descriptors were sourced from the INRIA Holidays image collection Jegou et al. (2008). The matrix size for the SIFT dataset is 100,000 $\times$ 128.

Metrics: We primarily assess the accuracy and time efficiency of the algorithm. Accuracy is evaluated using Mean Absolute Error (MAE), Mean Relative Error (MRE), and Mean Squared Error (MSE). For time efficiency, we measure the Quantization Time (QT) and DeQuantization Time (DQT).

Let $x_{(i,j)}$ represent the elements of the original matrix to be quantized, while $x'_{(i,j)}$ denotes the elements of the dequantized matrix.

$$\text{MAE} = \frac{1}{n \cdot d} \sum_{i,j} \left| x_{(i,j)} - x'_{(i,j)} \right|, \quad \text{MSE} = \frac{1}{n \cdot d} \sum_{i,j} \left(x_{(i,j)} - x'_{(i,j)} \right)^2$$

Comparative Algorithms: We compare our approach with PQ (Product Quantization)(Jegou et al. (2010)), OPQ (Optimized Product Quantization)(Ge et al. (2013)), and LOPQ (Locally Optimized Product Quantization)(Kalantidis & Avrithis (2014)). For the comparative algorithms, under the same space constraints, we use the same common parameters as our algorithm, while other parameters are set according to the recommended configurations from their respective papers.

Parameter Selection for *HyperChr*: For an $n \times d$ matrix with a bits per element and a compression ratio θ, the number of subgroups is m. Therefore, after the matrix partition operation, the dimension of each vector becomes $\frac{d}{m}$. We first determine the total number of cluster centroids, k_s, by solving the following inequality using binary search to find the maximum value of k_s:

$$k_s \times \frac{d}{m} \times a + n \times m \times \log_2(k) \leq \frac{n \times d \times a}{\theta}$$

We use quantiles to divide each dimension into l intervals. Therefore, there are a total of $l^{\frac{d}{m}}$ high-dimensional subspaces. The number of centroids in each subspace is allocated based on the number of subvectors in the subspace.

$$k_{s,j} = k_s \times \frac{n_j}{n \times m} \quad (j = 1, 2, 3, \ldots, l^{\frac{d}{m}})$$

where n_j is the number of vectors in the j-th subspace. Typically, we adopted the parameters $\frac{d}{m} = 4$ and $l = 3$.

4.2 Matrix Quantization

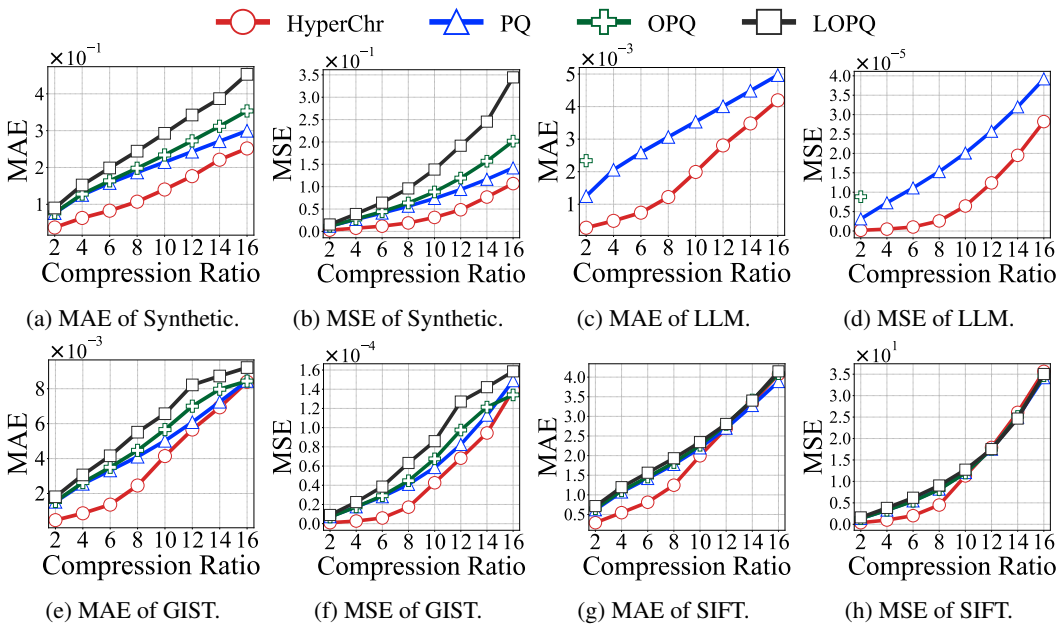
This section primarily presents the accuracy (MAE and MSE) and computational efficiency (QT and DQT) of the *HyperChr* algorithm.

MAE and MSE: The experimental results for MAE and MSE are presented in Figure 3. The results clearly demonstrate that *HyperChr* consistently outperforms PQ, OPQ, and LOPQ in most scenarios across various datasets and compression ratios, in both MAE and MSE metrics. For lower compression ratios ($\theta = 2 - 8$), the *HyperChr* algorithm shows a significant improvement, reducing MAE by an average of 55.3% and MSE by 75.3% compared to PQ. However, for higher compression ratios ($\theta = 10 - 16$), the improvements are more moderate, with an average reduction of 14.9% in MAE and 25.9% in MSE compared to PQ.

For the Synthetic dataset, *HyperChr* exhibits the lowest MAE and MSE across all compression ratios, significantly outperforming the other algorithms. At lower compression ratios, compared to the PQ algorithm, *HyperChr* reduces MSE by 71.2% and MAE by 47.8%. At higher compression ratios, it still achieves reductions of 41.2% in MSE and 24.1% in MAE.

In the LLM dataset, *HyperChr* exhibits a similar trend in maintaining lower MAE and MSE values. At lower compression ratios, compared to the PQ algorithm, *HyperChr* reduces MSE by 90.3% and MAE by 71.4%. At higher compression ratios, it still achieves reductions of 46.7% in MSE and 27.9% in MAE. It is worth noting that due to the additional space requirements of LOPQ, it was unable to handle the compression of the LLM dataset.

The results on the GIST dataset further reinforce the strength of *HyperChr*, particularly in maintaining lower errors across all compression ratios. While PQ and OPQ occasionally show comparable performance, *HyperChr* consistently achieves better results. At lower compression ratios, compared to the PQ algorithm, *HyperChr* reduces MSE by 76.8% and MAE by 58.5%. At higher compression ratios, it still reduces MSE by 16.8% and MAE by 7.3%.

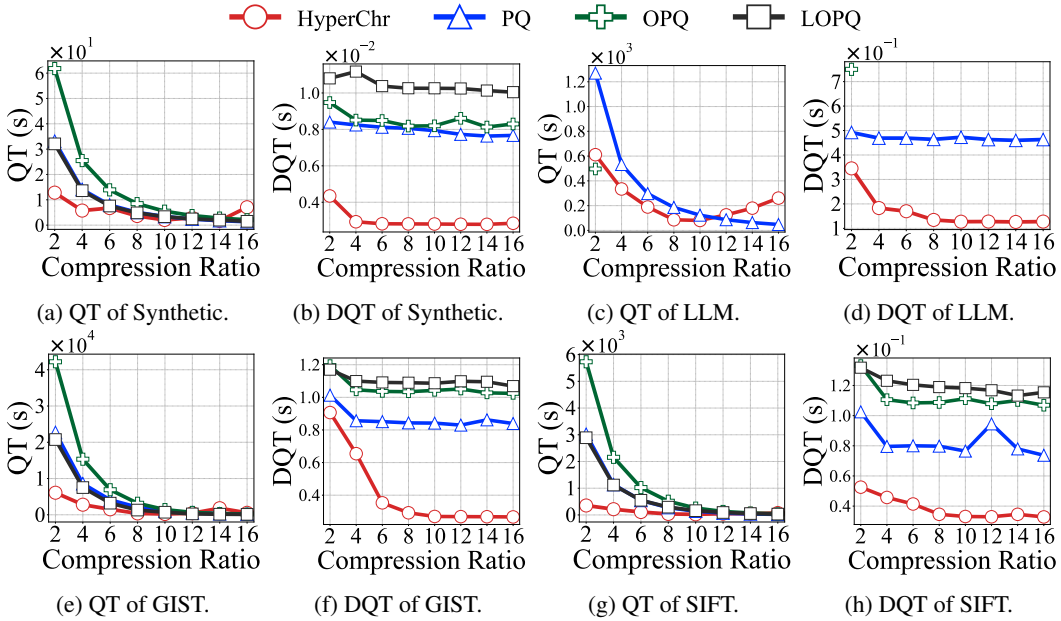


(a) MAE of Synthetic. (b) MSE of Synthetic. (c) MAE of LLM. (d) MSE of LLM.

(e) MAE of GIST. (f) MSE of GIST. (g) MAE of SIFT. (h) MSE of SIFT.

Figure 3: MAE and MSE of different datasets.

Lastly, the analysis of the SIFT dataset reveals that *HyperChr* performs well at lower compression ratios, but its performance is comparable to the PQ algorithm at higher compression ratios. At lower compression ratios, compared to the PQ algorithm, *HyperChr* reduces MSE by 62.7% and MAE by 43.6%. However, at higher compression ratios, it shows a slight increase of 1.2% in MSE and a reduction of 0.5% in MAE.



(a) QT of Synthetic. (b) DQT of Synthetic. (c) QT of LLM. (d) DQT of LLM.

(e) QT of GIST. (f) DQT of GIST. (g) QT of SIFT. (h) DQT of SIFT.

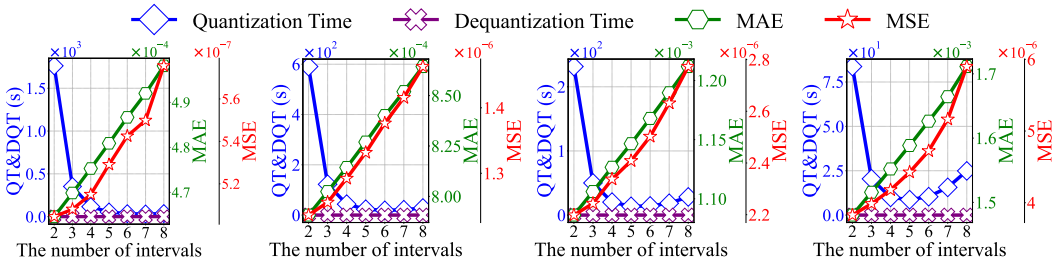
Figure 4: QT (s) and DQT (s) of different datasets.

QT and DQT: The experimental results for Quantization Time (QT) and DeQuantization Time (DQT) are presented in Figure 4. The results clearly demonstrate that *HyperChr* consistently outperforms PQ, OPQ, and LOPQ in most scenarios across various datasets and compression ratios, in both QT and DQT metrics. The impact of compression ratio on QT is significant, while its effect on

DQT is minimal. At low compression ratios, the QT of *HyperChr* is reduced by 60.2% compared to PQ. However, at higher compression ratios, where the QT is already small, the QT of *HyperChr* increases by 113.4% compared to PQ. On the other hand, DQT remains relatively stable; for lower compression ratios, the DQT of *HyperChr* is reduced by 60.2% compared to PQ, while at higher compression ratios, the DQT of *HyperChr* is reduced by 65.7%. Specifically, for the Synthetic dataset, LLM dataset, GIST dataset, and SIFT dataset, the DQT is reduced by 62.2%, 64.5%, 53.8%, and 53.6%, respectively. At low compression ratios, the QT is reduced by 40.9%, 44.8%, 71.4%, and 83.9%, respectively.

4.3 Trade-off between accuracy and computational efficiency

We explored the tradeoff between accuracy and computational efficiency of the *HyperChr* algorithm by varying the number of intervals l based on the LLM dataset. The experimental results are shown in Figure 5. Figures 5a, 5b, 5c, and 5d present the accuracy (MAE/MSE) and computational efficiency (QT/DQT) for compression ratios of 2, 4, 6, and 8, respectively.



(a) $\theta = 2$. (b) $\theta = 4$. (c) $\theta = 6$. (d) $\theta = 8$.

Figure 5: Trade-off between accuracy and computational efficiency.

For different compression ratios θ, the algorithm exhibits a similar trade-off characteristic. As the number of intervals increases, the DQT of the *HyperChr* algorithm decreases, while QT remains relatively stable, but both MAE and MSE increase. Upon closer analysis of the algorithm, we observe that when the number of intervals increases, fewer centroids exist for each high-dimensional subspace, resulting in faster computation. During the dequantization phase, *HyperChr* retrieves the corresponding vectors from the codebook (arrays of centroids) to reconstruct the matrix, which explains the stability in QT. Additionally, increasing the number of intervals provides finer partitioning of each dimension, thus reducing MAE and MSE.

5 Conclusion

Matrix quantization is essential for reducing memory usage while maintaining accuracy across various applications, especially with the increasing scale of models like GPT-3. Traditional quantization methods often overlook the heterogeneous distribution characteristics of real-world datasets, which can lead to suboptimal results.

In this paper, we introduced *HyperChr*, a matrix quantization algorithm tailored for heterogeneous data distributions. By partitioning high-dimensional subspaces based on column-specific distribution properties, *HyperChr* enhances both compression effectiveness and computational efficiency. Our experimental results demonstrate that *HyperChr* significantly reduces quantization errors and dequantization time, particularly at lower compression ratios, outperforming traditional methods.

These improvements highlight the potential of *HyperChr* in optimizing large-scale matrix quantization for real-world applications, especially in large language models and other machine learning tasks.

REFERENCES

How many users does facebook have? URL https://www.oberlo.com/statistics/how-many-users-does-facebook-have. Accessed: 2024-09-09.

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127, 2024.

Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *International symposium on string processing and information retrieval*, pp. 18–30. Springer, 2009.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Francisco Claude and Susana Ladra. Practical representations for web and social graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 1185–1190, 2011.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

Haojie Duanmu, Zhihang Yuan, Xiuhong Li, Jiangfei Duan, Xingcheng Zhang, and Dahua Lin. Skvq: Sliding-window key and value cache quantization for large language models. *arXiv preprint arXiv:2405.06219*, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):744–755, 2013.

Robert M. Gray and David L. Neuhoff. Quantization. *IEEE transactions on information theory*, 44 (6):2325–2383, 1998.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Computer Vision–ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part I 10*, pp. 304–317. Springer, 2008.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

Yannis Kalantidis and Yannis Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2321–2328, 2014.

Zayd Muhammad Kawakibi Zuhri, Muhammad Farid Adilazuarda, Ayu Purwarianti, and Alham Fikri Aji. Mlkv: Multi-layer key-value heads for memory efficient transformer decoding. *arXiv e-prints*, pp. arXiv–2406, 2024.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6: 87–100, 2024.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024.

Qingqun Ning, Jianke Zhu, Zhiyuan Zhong, Steven CH Hoi, and Chun Chen. Scalable image retrieval by sparse product quantization. *IEEE Transactions on Multimedia*, 19(3):586–597, 2016.

Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42:145–175, 2001.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2023.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*, 2019.

Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958–1970, 2008.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pp. 38087–38099. PMLR, 2023.

Donna Xu, Ivor W Tsang, and Ying Zhang. Online product quantization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2185–2198, 2018.

Tan Yu, Junsong Yuan, Chen Fang, and Hailin Jin. Product quantization network for fast image retrieval. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 186–201, 2018.

Yuxuan Yue, Zhihang Yuan, Haojie Duanmu, Sifan Zhou, Jianlong Wu, and Liqiang Nie. Wkvquant: Quantizing weight and key/value cache for large language models gains more. *arXiv preprint arXiv:2402.12065*, 2024.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

A APPENDIX

A.1 THE PROOF OF THEOREM 1

Lemma 1. Suppose each element in the matrix is independently sampled with a mean μ and variance σ^2. After quantization by the PQ algorithm, the Mean Squared Error (MSE) is given by:

$$\text{MSE}_{\text{PQ}} = \sigma^2 \left(1 - \frac{1}{n_k}\right).$$

Proof. Let $c_{(i,j)}$ denote the cluster centroid matrix, and let n_k be the number of vectors in the cluster. Define the residual matrix as $s_{(i,j)}$.

The expectation of the residual matrix is:

$$E[s_{(i,j)}] = E[x_{(i,j)} - c_{(i,j)}] = \mu - \mu = 0.$$

The Mean Squared Error (MSE) can be expressed as the variance of the residual matrix:

$$\text{MSE}_{\text{PQ}} = \text{Var}[s_{(i,j)}] = \text{Var}[x_{(i,j)}] + \text{Var}[c_{(i,j)}] - 2 \cdot \text{Cov}(x_{(i,j)}, c_{(i,j)}).$$

Given that $c_{(i,j)}$ is the centroid within the cluster:

$$\begin{aligned}\text{MSE}_{\text{PQ}} &= \sigma^2 + \frac{\sigma^2}{n_k} - 2E\left[(x_{(i,j)} - \mu)\left(\frac{1}{n_k}\sum_{i=1}^{n_k} x_{(i,j)} - \mu\right)\right] \\ &= \sigma^2 + \frac{\sigma^2}{n_k} - 2 \cdot \frac{1}{n_k}\left(E\left[(x_{(i,j)} - \mu)^2\right] + \sum_{i \neq i'} E\left[(x_{(i,j)} - \mu)(x_{i'j} - \mu)\right]\right).\end{aligned}$$

Since $x_{(i,j)}$ and $x_{i'j}$ (for $i \neq i'$) are independent:

$$\text{MSE}_{\text{PQ}} = \sigma^2 + \frac{\sigma^2}{n_k} - 2 \cdot \frac{\sigma^2}{n_k} = \left(1 - \frac{1}{n_k}\right)\sigma^2. \tag{5}$$

□

Theorem 1. (Theoretical analysis of MSE) The Mean Squared Error (MSE) of the *HyperChr* algorithm is lower than that of the PQ algorithm:

$$\text{MSE}_{\text{HyperChr}} = \Gamma \cdot \text{MSE}_{\text{PQ}}, \tag{6}$$

where $0 < \Gamma < 1$ is the effective variance reduction factor achieved by *HyperChr*.

Proof. We assume that each element $x_{i,j}$ in the matrix X is independently sampled from a distribution with mean μ and variance σ^2.

From Lemma 1, the MSE of the PQ algorithm is: $\text{MSE}_{\text{PQ}} = \sigma^2 \left(1 - \frac{1}{n_k}\right)$

The *HyperChr* algorithm reduces quantization error through two main techniques:

Subspace Partitioning: Partitioning the data into subspaces based on interval indices, effectively grouping similar data points together.

Localized Clustering: Clustering within each subspace allows for more precise centroids tailored to the local data distribution.

Within each subspace s, the variance of the data σ_s^2 is less than the global variance σ^2:

$$\sigma_s^2 = \gamma_s \sigma^2, \quad \text{with} \quad 0 < \gamma_s < 1. \tag{7}$$

This variance reduction occurs because the data within a subspace are more homogeneous due to the partitioning based on interval indices.

The MSE for *HyperChr* can be expressed as the weighted sum of the MSEs within each subspace:

$$\text{MSE}_{\text{HyperChr}} = \sum_s p_s \cdot \text{MSE}_s, \tag{8}$$

where p_s is the proportion of data points in subspace s and MSE_s is the MSE within subspace s.

Within each subspace s:

$$\text{MSE}_s = \sigma_s^2 \left(1 - \frac{1}{n_{k_s}}\right) = \gamma_s \sigma^2 \left(1 - \frac{1}{n_{k_s}}\right), \tag{9}$$

where n_{k_s} is the number of data points per centroid in subspace s.

Assuming that the average n_{k_s} across subspaces is approximately equal to n_k in PQ, we can compare the MSEs:

$$\begin{aligned} \text{MSE}_{\text{HyperChr}} &= \sum_s p_s \cdot \gamma_s \sigma^2 \left(1 - \frac{1}{n_{k_s}}\right) && (10) \\ &\approx \left(\sum_s p_s \gamma_s\right) \sigma^2 \left(1 - \frac{1}{n_k}\right) && (11) \\ &= \Gamma \sigma^2 \left(1 - \frac{1}{n_k}\right), && (12) \end{aligned}$$

where we define the effective variance reduction factor:+

$$\Gamma = \sum_s p_s \gamma_s, \quad \text{with} \quad 0 < \Gamma < 1. \tag{13}$$

□

A.2 THE PROOF OF THEOREM 2

Lemma 2. Assuming the same compression ratio, the number of centroids k_1 per subspace in the PQ algorithm is approximately equal to $\frac{k_2}{m}$:

$$k_1 \approx \frac{k_2}{m}. \tag{14}$$

Proof. The compressed memory size of the PQ algorithm is:

$$\text{Memory}_{\text{PQ}} = nm \log_2(k_1) + k_1 d, \tag{15}$$

where n is the number of data points, m is the number of subspaces, and d is the dimensionality of the data.

In our method, since the total number of centroids across all subspaces is k_2, the average number of centroids per subspace is $\frac{k_2}{m}$. Thus, the compressed memory size for our method is:

$$\text{Memory}_{\text{HyperChr}} = nm \log_2 \left(\frac{k_2}{m}\right) + k_2 \frac{d}{m}. \tag{16}$$

Assuming both methods achieve the same compression ratio, we set the compressed memory sizes equal:

$$nm \log_2(k_1) + k_1 d = nm \log_2 \left(\frac{k_2}{m}\right) + k_2 \frac{d}{m}. \tag{17}$$

Given that k_1 and $\frac{k_2}{m}$ represent the respective centroid counts per subspace in each method, we can deduce that for the equality to hold, $k_1 \approx \frac{k_2}{m}$. Therefore,

$$k_1 \approx \frac{k_2}{m}.$$

□

Theorem 2. (Time Complexity for Quantization) The quantization time complexity of *HyperChr* method is:

$$QT_{\text{HyperChr}} = \frac{1}{m \cdot l^{\frac{d}{m}}} \cdot QT_{\text{PQ}}, \tag{18}$$

where QT_{HyperChr} and QT_{PQ} are the time complexities of our method and the PQ algorithm, respectively. The proof can be found in Appendix A.2.

Proof. We will compare the time complexity of our method with that of the PQ algorithm by analyzing both grouping and clustering steps.

The time complexity of the PQ algorithm is:

$$QT_{\text{PQ}} = O(nk_1d), \tag{19}$$

where n is the number of data points, k_1 is the number of centroids per subspace, and d is the data dimensionality.

In our method, each data point is divided into $\frac{d}{m}$ dimensions, and each dimension is segmented into l intervals. Determining the appropriate group for each data point involves assigning it to one of $l^{\frac{d}{m}}$ groups. The time complexity per data point for this assignment is $O\left(\frac{d}{m}\log l\right)$, due to sorting or searching through the intervals.

Therefore, the total grouping time complexity for all data points is:

$$QT_{\text{group}} = O\left(n\frac{d}{m}\log l\right).$$

Since $l^{\frac{d}{m}}$ is assumed to be a constant and $k_2 \gg l$, the grouping time QT_{group} is negligible compared to the clustering time, which dominates the overall time complexity.

After grouping, there are $l^{\frac{d}{m}}$ groups in total. Each group contains approximately $\frac{n}{l^{\frac{d}{m}}}$ data points. Assuming the number of centroids in each group is proportional to the number of data points, each group will have about $\frac{k_2}{l^{\frac{d}{m}}}$ centroids.

The clustering time complexity per group is:

$$O\left(\frac{n}{l^{\frac{d}{m}}} \cdot \frac{k_2}{l^{\frac{d}{m}}} \cdot \frac{d}{m}\right) = O\left(\frac{nk_2d}{ml^{2\frac{d}{m}}}\right).$$

Multiplying by the total number of groups $l^{\frac{d}{m}}$, the total clustering time complexity becomes:

$$QT_{\text{cluster}} = l^{\frac{d}{m}} \cdot O\left(\frac{nk_2d}{ml^{2\frac{d}{m}}}\right) = O\left(\frac{nk_2d}{ml^{\frac{d}{m}}}\right).$$

Combining both grouping and clustering times, the total time complexity of our method is:

$$QT_{\text{HyperChr}} = QT_{\text{group}} + QT_{\text{cluster}} = O\left(n\frac{d}{m}\log l\right) + O\left(\frac{nk_2d}{ml^{\frac{d}{m}}}\right).$$

Since QT_{group} is negligible compared to QT_{cluster}, we can simplify:

$$QT_{\text{HyperChr}} = O\left(\frac{nk_2d}{ml^{\frac{d}{m}}}\right).$$

From Lemma 2, we have $k_1 \approx \frac{k_2}{m}$. Substituting $k_2 = mk_1$ into the expression for QT_{HyperChr}, we get:

$$QT_{\text{HyperChr}} = O\left(\frac{n(mk_1)d}{ml^{\frac{d}{m}}}\right) = O\left(\frac{nk_1d}{l^{\frac{d}{m}}}\right).$$

Comparing this with QT_{PQ}:

$$\frac{QT_{\text{HyperChr}}}{QT_{\text{PQ}}} = \frac{O\left(\frac{nk_1d}{l^{\frac{d}{m}}}\right)}{O(nk_1d)} = \frac{1}{l^{\frac{d}{m}}}.$$

However, since PQ processes m subspaces and our method effectively reduces computations by a factor of $m \cdot l^{\frac{d}{m}}$, the actual ratio is:

$$\frac{QT_{\text{HyperChr}}}{QT_{\text{PQ}}} = \frac{1}{m \cdot l^{\frac{d}{m}}}.$$

Thus, the total time complexity of our method is approximately $\frac{1}{m \cdot l^{\frac{d}{m}}}$ times that of the PQ algorithm.

This completes the proof, showing that our method is significantly more efficient compared to the PQ algorithm when m is large and $l^{\frac{d}{m}}$ is constant.

□

A.3 THE PROOF OF THEOREM 3

Theorem 3. (Time Complexity for Dequantization) The time complexity of the dequantization process in the Matrix Dequantization Algorithm is:

$$DQT_{\text{HyperChr}} = \frac{1}{m} \cdot DQT_{\text{PQ}}, \tag{20}$$

where DQT_{HyperChr} and DQT_{PQ} represent the time complexities of the Matrix Dequantization Algorithm and the PQ algorithm, respectively.

Proof. For the Matrix Dequantization Algorithm, the time complexity for retrieving the corresponding centroid is:

$$T_{\text{HyperChr}} = O(1), \tag{21}$$

as the centroid is directly accessed via an index lookup.

For the PQ algorithm, since it divides the vector into m sub-vectors and retrieves the corresponding sub-centroid from each sub-codebook, the time complexity is:

$$T_{\text{PQ}} = O(m). \tag{22}$$

Therefore, the time complexity of *HyperChr* relative to PQ can be expressed as:

$$T_{\text{HyperChr}} = \frac{1}{m} \cdot T_{\text{PQ}}. \tag{23}$$

□