A Detailed, Pedagogical, and Mathematical Explanation of GPTQ Quantization

Abderrahman Skiredj
College of Computing, Mohammed VI Polytechnic University, Benguerir, Morocco
abderrahman.skiredj@um6p.ma

February 2025

Abstract

We present a detailed, pedagogical, and mathematical explanation of GPTQ quantization, a post-training technique for large language models that reduces weight precision while preserving model accuracy. By leveraging second-order Hessian information to guide a column-wise quantization process with error compensation, GPTQ enables significant model compression with minimal performance loss. This paper provides clear derivations, intuitive examples, and a rigorous exploration of the underlying mathematics, making the complex concepts accessible to both practitioners and researchers.

1 Introduction to Quantization

1.1 What is Quantization?

Quantization is the process of mapping high-precision numerical values (e.g., 32-bit floating-point numbers) to a lower-precision representation (e.g., 4-bit integers). In neural networks, this typically applies to the model's weights, reducing memory usage and computational cost, which is critical for deploying large models on resource-constrained devices like smartphones or edge systems.

1.1.1 Understanding Number Encoding and Quantization

Before diving into quantization, it's important to understand how numbers are encoded in computers. High-precision numbers, such as 32-bit floating-point numbers, can represent a wide range of values with fine granularity. However, they require significant memory and computational resources. Quantization reduces this precision by mapping these high-precision numbers to a smaller set of discrete values, typically integers, which are easier to store and process.

For example, consider a 32-bit floating-point number like 3.14159. Storing this number requires 32 bits of memory. If we quantize it to a 4-bit integer, we might represent it as 3, which only requires 4 bits. While this introduces some approximation error, it drastically reduces memory usage and speeds up computations.

1.1.2 Simple Uniform Quantization: Intuition and Example

Uniform quantization is one of the simplest and most widely used quantization methods. The core idea is to divide the range of possible values into equal-sized intervals (or "bins") and map each high-precision value to the nearest bin. This process involves two key steps:

1. **Scaling**: High-precision values are scaled to fit the range of a lower-precision representation.
2. **Rounding**: Scaled values are rounded to the nearest integer.

Let's break this down with a concrete example.

Step 1: Scaling Consider a vector of floating-point numbers:

$$\mathbf{W} = [1.2, -0.5, 3.7, -2.1]$$

We aim to quantize this vector to 4-bit signed integers. A 4-bit signed integer can represent $2^4 = 16$ distinct values, typically ranging from $[-8, 7]$. To map the values in $\mathbf{W}$ to this range, we first identify the minimum and maximum values:

$$W_{\min} = -2.1, \quad W_{\max} = 3.7$$

The range of the vector is $[-2.1, 3.7]$. To fit this into the 4-bit signed integer range $[-8, 7]$, we calculate a scaling factor s:

$$s = \frac{W_{\max} - W_{\min}}{2^n - 1}$$

Here, $n = 4$ (for 4-bit quantization), so $2^n - 1 = 15$, and:

$$s = \frac{3.7 - (-2.1)}{15} = \frac{5.8}{15} \approx 0.3867$$

This scaling factor s represents the size of each quantization bin.

Step 2: Quantization With s determined, we quantize each value in $\mathbf{W}$ to an integer Q_i using:

$$Q_i = \text{round}\left(\frac{W_i - W_{\min}}{s}\right)$$

Since we're using signed 4-bit integers, the resulting Q_i values must be clipped to the range $[-8, 7]$. During inference, the quantized values are scaled back to approximate the original values:

$$\hat{W}_i = Q_i \times s + W_{\min}$$

where $\hat{W}_i$ is the dequantized approximation of W_i.

Example Let's quantize $\mathbf{W} = [1.2, -0.5, 3.7, -2.1]$ using $s \approx 0.3867$:

$$Q_1 = \text{round}\left(\frac{1.2 - (-2.1)}{0.3867}\right) = \text{round}\left(\frac{3.3}{0.3867}\right) = \text{round}(8.53) = 9$$
$$Q_2 = \text{round}\left(\frac{-0.5 - (-2.1)}{0.3867}\right) = \text{round}\left(\frac{1.6}{0.3867}\right) = \text{round}(4.14) = 4$$
$$Q_3 = \text{round}\left(\frac{3.7 - (-2.1)}{0.3867}\right) = \text{round}\left(\frac{5.8}{0.3867}\right) = \text{round}(15.0) = 15$$
$$Q_4 = \text{round}\left(\frac{-2.1 - (-2.1)}{0.3867}\right) = \text{round}\left(\frac{0.0}{0.3867}\right) = \text{round}(0.0) = 0$$

For $Q_1 = 9$ and $Q_3 = 15$, the values exceed the 4-bit signed integer range $[-8, 7]$. We clip them to the maximum representable value, 7:

$$\mathbf{Q} = [7, 4, 7, 0]$$

Now, dequantize $\mathbf{Q} = [7, 4, 7, 0]$ to approximate the original values:

$$\hat{W}_1 = 7 \times 0.3867 + (-2.1) \approx 2.7069 - 2.1 = 0.6069$$
$$\hat{W}_2 = 4 \times 0.3867 + (-2.1) \approx 1.5468 - 2.1 = -0.5532$$
$$\hat{W}_3 = 7 \times 0.3867 + (-2.1) \approx 2.7069 - 2.1 = 0.6069$$
$$\hat{W}_4 = 0 \times 0.3867 + (-2.1) = -2.1$$

The dequantized vector is:

$$\hat{\mathbf{W}} = [0.6069, -0.5532, 0.6069, -2.1]$$

These values approximate the originals, but notice that clipping $Q_3 = 15$ to 7 limits the maximum dequantized value ($\hat{W}_3 \approx 0.6069$), underestimating $W_3 = 3.7$. This illustrates a trade-off: quantization

reduces precision to save memory and speed up computations. Additionally, rounding $s \approx 0.3867$ introduces small errors—e.g., the exact maximum $W_3 = 3.7$ isn't preserved, a common artifact of finite precision.[1]

Why Quantization Speeds Up Operations Quantization reduces memory usage by representing values with fewer bits (e.g., 4-bit integers vs. 32-bit floats) and enables faster computations. Integer arithmetic is more efficient than floating-point operations, especially on hardware like GPUs and TPUs, which have optimized support for low-precision integer instructions. This makes quantization a cornerstone of efficient machine learning inference.

Intuition Behind the Scaling Method The scaling factor s ensures that the range of the original high-precision values is evenly distributed across the available quantization levels. By dividing the range into equal-sized bins, uniform quantization provides a simple and efficient way to approximate high-precision values. However, this method assumes that the values are uniformly distributed, which may not always be the case. In practice, more advanced quantization techniques (e.g., non-uniform quantization) are often used to minimize accuracy loss.

1.2 Overview of Quantization Methods

- **Uniform Quantization**: As above, applies a fixed step size across all weights.
- **Post-Training Quantization (PTQ)**: Quantizes a pre-trained model without retraining, using a calibration dataset to adjust the process.
- **Quantization-Aware Training (QAT)**: Incorporates quantization into training, adjusting weights to anticipate low-precision effects.

GPTQ is a PTQ method, leveraging second-order information to enhance accuracy preservation, distinguishing it from simpler PTQ techniques.

1.3 Contribution

This paper makes the following contributions:

- It offers a clear and intuitive introduction to quantization, establishing the fundamental concepts with concrete examples.
- It provides a step-by-step, mathematically rigorous derivation of the GPTQ algorithm, highlighting how second-order Hessian information guides the quantization process.
- It details practical implementation insights, including error compensation through column-wise weight updates and efficient computation via Cholesky decomposition.

2 What is GPTQ?

2.1 Definition and Objectives

GPTQ is a quantization technique tailored for large language models (LLMs), applied post-training to reduce weight precision (e.g., from 32-bit floats to 4-bit integers). Its primary goal is to minimize the output difference between the original and quantized models, preserving performance while achieving significant compression. Specifically, GPTQ minimizes layer-wise output differences as a heuristic for maintaining overall model performance, rather than directly optimizing the downstream task loss (e.g., perplexity for LLMs), which would require full model retraining.

For a linear layer with weight matrix W and input X, the original output is WX. After quantization to Q, the output becomes QX. GPTQ aims to minimize:

$$\|WX - QX\|_2^2$$

where:

[1] The scaling factor s is an approximation due to rounding, which can slightly shift dequantized values. For exact preservation of extremes, alternative methods like asymmetric quantization with a zero-point offset might be used, but that's beyond this simple example.

- W: Original weight matrix (e.g., $m \times n$, full precision).
- Q: Quantized weight matrix (same shape, lower precision).
- X: Input matrix (e.g., $n \times N$, where N is the number of samples). This represents the **activations** (or hidden states) fed into the linear layer, obtained by passing a calibration dataset through the model up to that layer.
- $\|\cdot\|_2^2$: Squared Euclidean norm, summing squared differences over outputs.

2.2 Understanding X: Input to the Linear Layer

- X is **not** the raw input (e.g., sentences from a dataset). Instead, it represents the **activations** (hidden states) that are fed into the linear layer being quantized.
- **Source of X:** X is derived from a **calibration dataset**, which consists of representative inputs (e.g., sentences from C4 [2]). These inputs are passed through the model to generate the activations for each layer.
- **Role in Quantization:** The activations X are used to guide the quantization process, ensuring that the quantized weights Q minimize the difference in outputs compared to the original weights W.
- **Shape and Structure:** For a single sample i, the input to the layer is denoted X_i, which is a vector of shape $(n,)$, where n is the number of input features of the layer. For a batch of N samples, X becomes a matrix of shape (n, N).
- **Layer-Specific Nature:** Even for deep layers in the model, X represents only the input to that specific layer, regardless of the transformations applied in earlier layers.

2.3 How GPTQ Differs

Unlike uniform quantization, GPTQ:

- Is **post-training**, requiring no retraining.
- Uses the **Hessian matrix** (second-order derivatives of the loss) to guide quantization, capturing weight sensitivity.
- Processes weights **column by column**, adjusting remaining weights to compensate for errors iteratively.

This Hessian-based, structured approach makes GPTQ both effective and efficient for billion-parameter models.

3 Mathematical Foundations

3.1 The Hessian Matrix

The Hessian matrix H of a function $f(W)$ with respect to a vectorized parameter W is the matrix of second-order partial derivatives:

$$H_{i,j} = \frac{\partial^2 f}{\partial W_i \partial W_j}$$

In neural networks, f is typically the loss function $L(W)$, and W represents the weights. For a linear layer $Y = WX$, where W is $m \times n$ and X is $n \times N$, the loss depends on Y, and the Hessian describes the curvature of L around W.

3.2 Second-Order Taylor Approximation

To understand quantization's impact, consider the loss change when weights shift from W to Q. The Taylor expansion of $L(Q)$ around W is:

$$L(Q) = L(W) + \nabla L(W)^T(Q - W) + \frac{1}{2}(Q - W)^T H(Q - W) + \text{higher-order terms}$$

Since W is from a trained model, the gradient $\nabla L(W) \approx 0$. Ignoring higher-order terms, the change in loss is:

$$\Delta L \approx \frac{1}{2}(Q - W)^T H(Q - W)$$

This quadratic form measures the loss perturbation due to quantization. GPTQ minimizes this, but computing the full Hessian for large W (e.g., $mn \times mn$ elements) is impractical, necessitating an approximation.

4 Approximating the Hessian

4.1 Calibration Dataset

GPTQ uses a small calibration dataset (e.g., 128–512 samples from a corpus like C4 [2]) to estimate the Hessian. For a transformer layer, raw inputs (e.g., token embeddings) are processed through preceding layers, yielding activations X as inputs to the layer being quantized. For N samples, X is $n \times N$, where each column $X_{:,i}$ is an input vector of length n.

4.2 Hessian Approximation

Consider a linear layer $Y = WX$ where $W \in \mathbb{R}^{m \times n}$, and assume a quadratic loss (e.g., MSE) for simplicity:

$$L = \frac{1}{2}\|Y - Y_{\text{target}}\|_2^2$$

Under this framework, the Hessian with respect to W can be approximated through the input activations X.

4.2.1 Quadratic Loss Simplification

For analytical tractability, assume $Y_{\text{target}} = 0$ (equivalent to modeling residual errors). This simplification models the curvature of the residual error, aligning with GPTQ's goal of minimizing output perturbation (i.e., $\|WX - QX\|_2^2$) by focusing on the local sensitivity of the output $Y = WX$ rather than zeroing it outright. Then:

$$L = \frac{1}{2}\|Y\|_2^2 = \frac{1}{2}\text{Tr}(Y^\top Y) = \frac{1}{2}\text{Tr}(X^\top W^\top W X)$$

For a single sample $X_{:,i}$, this reduces to $L_i = \frac{1}{2}\|WX_{:,i}\|_2^2$.

4.2.2 Hessian Derivation

1. **First Derivative:** For a single sample $X_{:,i}$, the loss is $L_i = \frac{1}{2}\|WX_{:,i}\|_2^2$. The gradient of L_i with respect to W is:

   $$\frac{\partial L_i}{\partial W} = WX_{:,i}X_{:,i}^\top \quad \in \mathbb{R}^{m \times n}$$

   This represents the sensitivity of the loss to changes in the weight matrix W.

2. **Second Derivative:** The Hessian H_i is the second derivative of L_i with respect to the vectorized form of W, denoted $\text{vec}(W)$. It is given by:

   $$H_i = \frac{\partial^2 L_i}{\partial \text{vec}(W) \partial \text{vec}(W)^\top} = X_{:,i}X_{:,i}^\top \otimes I_m \quad \in \mathbb{R}^{mn \times mn}$$

   Here:

   - $\otimes$ is the Kronecker product, which expands $X_{:,i}X_{:,i}^\top$ into a block matrix.

- I_m is the identity matrix of size $m \times m$, ensuring the Hessian accounts for interactions across all output dimensions.

3. **Diagonal Approximation:** The full Hessian H_i is computationally expensive due to its large size ($mn \times mn$). For quantization, we only need the **block-diagonal terms** of H_i, which correspond to interactions within the same column of W. This simplification is valid because quantization typically updates weights column-wise (per output neuron), and off-diagonal terms (cross-column interactions) are assumed negligible or computationally impractical to include. The block-diagonal approximation is:
$$H_i^{\text{diag}} = X_{:,i} X_{:,i}^\top \quad \in \mathbb{R}^{n \times n}$$
This matrix captures the curvature of the loss with respect to the input dimensions (rows of $X_{:,i}$) for a single column of W, which is sufficient for per-column weight updates.

4. **Sample Average:** To approximate the Hessian over the entire calibration dataset, we average the per-sample Hessian approximations H_i^{diag} across all N samples:
$$H \approx \frac{1}{N} \sum_{i=1}^{N} X_{:,i} X_{:,i}^\top = \frac{1}{N} X X^\top$$
This final approximation $H \in \mathbb{R}^{n \times n}$ is efficient to compute and provides the necessary curvature information for quantization.

This aligns with Fisher information approximations in optimization literature.

5 GPTQ Quantization Algorithm

5.1 Overview & Key Intuitions

The GPTQ algorithm quantizes a weight matrix $W \in \mathbb{R}^{m \times n}$ column-wise while minimizing output distortion $\|WX - QX\|_2^2$, where Q is the quantized matrix. It uses a sequential approach with error compensation through Hessian-guided updates.

- **Sequential Quantization**: Columns are quantized from left to right. Once quantized, $Q_{:,j}$ remains fixed.
- **Error Compensation**: Errors from quantizing column j are partially canceled by adjusting subsequent columns before their quantization.
- **Hessian Guidance**: Uses second-order information (Hessian H) to make optimal weight updates that minimize output distortion.

5.2 Algorithm Steps

Initialization

- Compute Hessian $H = \frac{1}{N} X X^T$ (size $n \times n$)
- Perform Cholesky decomposition $H = LL^T$
- Create working copy $\widetilde{W} \leftarrow W$

Column Processing (For each column $j = 1$ to n)

1. **Quantize Current Column**:
   - Quantize $\widetilde{W}_{:,j}$ to $Q_{:,j}$ using uniform quantization as described earlier
   - Compute quantization error: $\text{err}_k = Q_{k,j} - \widetilde{W}_{k,j}\ \forall k$
2. **Update Subsequent Columns**:
   - For each row k:
   $$\Delta \widetilde{W}_{k,j+1:} = -\text{err}_k \cdot \left(H^{-1}_{j+1:,j+1:} H_{j+1:,j}\right)^T$$
   - Update working copy:
   $$\widetilde{W}_{k,j+1:} \leftarrow \widetilde{W}_{k,j+1:} + \Delta \widetilde{W}_{k,j+1:}$$

5.3 Understanding Weight Updates

Why Update Subsequent Columns?

- Quantization introduces output error: $(Q_{:,j} - W_{:,j})X_j$
- Goal: Adjust unquantized weights $\widetilde{W}_{:,j+1:}$ to cancel this error
- Optimal update found via least squares using Hessian information

Working Copy Explanation

- $\widetilde{W}$ is a temporary full-precision copy
- Allows making reversible adjustments before quantizing subsequent columns
- Original W remains unchanged - only Q is the final quantized output

6 Detailed Mathematical Derivations

6.1 Error in Output

After quantizing columns 1 to j, the output is:

$$Q_{:,1:j}X_{1:j} + W_{:,j+1:}X_{j+1:}$$

Originally, it's:

$$W_{:,1:j}X_{1:j} + W_{:,j+1:}X_{j+1:}$$

The error from column j's quantization is:

$$(Q_{:,j} - W_{:,j})X_j$$

For row k, this is $\text{err}_k X_j$. We adjust $W_{k,j+1:}$ by $\Delta W_{k,j+1:}$ to produce:

$$\Delta W_{k,j+1:}X_{j+1:}$$

Ideally:

$$\Delta W_{k,j+1:}X_{j+1:} = -\text{err}_k X_j$$

6.2 Least-Squares Formulation

Since $X_{j+1:}$ is $(n-j) \times N$ and $N > n - j$, this is overdetermined. Minimize the squared residual:

$$\min_{\Delta W_{k,j+1:}} \|\Delta W_{k,j+1:}X_{j+1:} + \text{err}_k X_j\|_2^2$$

Define:

- $A = X_{j+1:}^T$ $(N \times (n-j))$,
- $x = \Delta W_{k,j+1:}^T$ $((n-j) \times 1)$,
- $b = -\text{err}_k X_j^T$ $(N \times 1)$.

Minimize:

$$\|Ax - b\|_2^2$$

Normal equations:

$$A^T Ax = A^T b$$

where:

- $A^T = X_{j+1:}$,
- $A^T A = X_{j+1:}X_{j+1:}^T$,
- $A^T b = X_{j+1:}(-\text{err}_k X_j^T) = -\text{err}_k X_{j+1:}X_j^T$.

Thus:

$$X_{j+1:}X_{j+1:}^T \Delta W_{k,j+1:}^T = -\text{err}_k X_{j+1:}X_j^T$$

6.3 Connecting to the Hessian

Since $H = \frac{1}{N} X X^T$:

- $H_{j+1:,j+1:} = \frac{1}{N} X_{j+1:} X_{j+1:}^T$,
- $X_{j+1:} X_{j+1:}^T = N H_{j+1:,j+1:}$,
- $H_{j+1:,j} = \frac{1}{N} X_{j+1:} X_j^T$,
- $X_{j+1:} X_j^T = N H_{j+1:,j}$.

Substitute:

$$N H_{j+1:,j+1:} \Delta W_{k,j+1:}^T = -\text{err}_k N H_{j+1:,j}$$

$$H_{j+1:,j+1:} \Delta W_{k,j+1:}^T = -\text{err}_k H_{j+1:,j}$$

Solve:

$$\Delta W_{k,j+1:}^T = -\text{err}_k H_{j+1:,j+1:}^{-1} H_{j+1:,j}$$

$$\Delta W_{k,j+1:} = -\text{err}_k (H_{j+1:,j+1:}^{-1} H_{j+1:,j})^T$$

Proof of Equivalence Verify by plugging back:

$$H_{j+1:,j+1:} (-\text{err}_k H_{j+1:,j+1:}^{-1} H_{j+1:,j}) = -\text{err}_k H_{j+1:,j}$$

The inverse cancels, confirming correctness.

7 Efficiency and Practical Considerations

7.1 Column-Wise Efficiency

Quantizing column by column batches the process (e.g., m weights at once), unlike per-weight methods (e.g., OBQ), reducing complexity from $O(mn)$ individual updates to $O(n)$ column updates.

7.2 Cholesky Decomposition

$H_{j+1:,j+1:}$ shrinks with j, but precomputing $H = LL^T$ allows updates via triangular solves. For $Hv = w$:

- Solve $Ly = w$ (forward substitution).
- Solve $L^T v = y$ (backward substitution).

Complexity drops from $O(n^3)$ for inversion to $O(n^2)$ per solve, amortized over rows.

8 Conclusion

GPTQ compresses LLMs by quantizing weights iteratively, using the Hessian to minimize output errors. Its column-wise approach, coupled with efficient linear algebra (Cholesky), makes it scalable and effective, reducing model size (e.g., 8x) with minimal accuracy loss. This detailed derivation ensures you can fully grasp its mathematical underpinnings without external references.

References

[1] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers," *arXiv preprint arXiv:2210.17323*, 2023. Available: `https://arxiv.org/abs/2210.17323`.

[2] J. Dodge, M. Sap, A. Marasović, W. Agnew, G. Ilharco, D. Groeneveld, M. Mitchell, and M. Gardner, "Documenting Large Webtext Corpora: A Case Study on the Colossal Clean Crawled Corpus," *arXiv preprint arXiv:2104.08758*, 2021. Available: `https://arxiv.org/abs/2104.08758`.