Departamento de Eletrónica, Telecomunicações e
Informática
da Universidade de Aveiro

Information and Coding
Lab work nº. 3

Diogo Silva (108212)
Martim Carvalho (108749)
Miguel Marques (103162)

December 19, 2025

Contents

Chapter 1

Introduction

1.0.1 Motivation

Large Language Models (LLMs) have grown exponentially in size, with models ranging from hundreds of millions to billions of parameters. The Qwen2-0.5B model, while relatively small at 494 million parameters, still occupies 943 MB of storage space. Efficient compression of these models is critical for:

- **Storage optimization**: Reducing cloud storage costs and local disk usage
- **Distribution**: Faster model downloads and updates
- **Edge deployment**: Fitting larger models on resource-constrained devices
- **Backup and archival**: Long-term storage efficiency

Generic compression tools (gzip, xz, zstd) treat neural network weights as arbitrary binary data, missing opportunities to exploit their inherent structure. This work investigates domain-specific preprocessing techniques that leverage knowledge of the BFloat16 floating-point format used to store neural network parameters.

1.0.2 Problem Statement

Objective: Compress the Qwen2-0.5B model (`model.safetensors`, 988 MB) achieving the best trade-off between:

- **Compression ratio**: How much space is saved
- **Compression time**: Speed of the encoding process
- **Decompression time**: Speed of the decoding process
- **Memory usage**: RAM requirements during operation

Constraints:

- Lossless compression (bit-exact reconstruction required)
- Practical decompression speed (under 5 seconds preferred)
- Reasonable memory usage

1.0.3 Approach Overview

Our solution consists of three key components:

1. **File format analysis**: Deep investigation of SafeTensors structure and BFloat16 representation
2. **Domain-specific preprocessing**: Byte reordering that separates high-entropy and low-entropy components
3. **Multi-algorithm implementation**: Comparative evaluation of LZ4, DEFLATE, ZSTD, and LZMA with three operation modes each

Chapter 2

Background and Related Work

2.1 SafeTensors Format

SafeTensors is a modern format for storing neural network weights, designed as a safer alternative to Python's pickle format. The file structure consists of three main components:

$$\text{SafeTensors} = \underbrace{\text{Header Size}}_{\text{8 bytes}} + \underbrace{\text{JSON Metadata}}_{\text{32 KB}} + \underbrace{\text{Tensor Data}}_{\text{943 MB}} \tag{2.1}$$

2.1.1 File Structure

The SafeTensors format provides a simple yet robust structure:

- **Header Size (8 bytes):** Little-endian integer specifying metadata length
- **JSON Metadata:** Contains tensor names, shapes, data types, and byte offsets
- **Tensor Data:** Contiguous binary data with zero padding

2.1.2 Key Characteristics

- All data is stored contiguously with zero padding between tensors
- Little-endian byte order throughout the entire file
- Metadata overhead is negligible (0.003% of total file size)
- No executable code, making it safer than pickle-based formats
- Memory-mappable for efficient loading without full deserialization

2.1.3 Qwen2-0.5B Specifications

For the Qwen2-0.5B model used in this study:

- **Total tensors:** 290 tensors distributed across 24 transformer layers
- **Total parameters:** 494,032,768 (494M parameters)
- **Data type:** BFloat16 (16 bits per parameter)

- **File size:** 988.1 MB
- **Largest tensor:** Token embedding matrix (259.66 MB, 55% of total model)
- **Metadata size:** 32,280 bytes (0.003% overhead)

2.2 BFloat16 Floating-Point Format

BFloat16 (Brain Floating Point) is a 16-bit floating-point format developed by Google Brain, specifically optimized for neural network training and inference. It represents a strategic trade-off between precision and computational efficiency.

2.2.1 Bit Structure

BFloat16 allocates its 16 bits as follows:

Sign	Exponent	Mantissa
1 bit	8 bits	7 bits

This structure provides the same exponent range as IEEE 754 FP32 (single precision), but with reduced mantissa precision.

2.2.2 Memory Layout

In little-endian byte order (standard on x86/ARM architectures), each BFloat16 value occupies two consecutive bytes:

- **Byte 0 (Low byte):** Contains mantissa bits [6:0] and exponent bit [0]
- **Byte 1 (High byte):** Contains sign bit [15] and exponent bits [7:1]

2.2.3 Advantages for Neural Networks

BFloat16 has become the preferred format for modern neural network implementations due to several key advantages:

- **Dynamic range:** Same exponent range as FP32 (8 bits), enabling representation of values from 10^{-38} to 10^{38}
- **Reduced precision:** 7-bit mantissa (versus 23-bit in FP32) is sufficient for neural network gradients and weights
- **Hardware acceleration:** Native support on modern processors (Intel AVX-512 BF16, ARM SVE2, Google TPU, NVIDIA Tensor Cores)
- **Memory efficiency:** 50% reduction in memory footprint compared to FP32
- **Bandwidth optimization:** 2× throughput improvement in memory-bound operations
- **Simple conversion:** Can be converted to/from FP32 by simple truncation/extension

2.2.4 Comparison with Other Formats

Format	Bits	Exponent	Mantissa	Use Case
FP32	32	8	23	High precision
BF16	16	8	7	Neural networks
FP16	16	5	10	Graphics, limited range
INT8	8	N/A	N/A	Quantized inference

Table 2.1: Comparison of Numeric Formats

2.3 Compression Algorithms

Compression algorithms use compression levels to control the trade-off between speed and compression ratio. Higher levels employ larger dictionaries, deeper match-finding, and more exhaustive parsing at the cost of increased processing time. This implementation defines two operation modes:

-Fast (ZSTD level 3, LZ4 level 0, DEFLATE level 3, LZMA level 3)

-Maximum (ZSTD level 19, LZ4 level 12, DEFLATE level 9, LZMA level 9).

These specific values were selected based on diminishing returns analysis: ZSTD level 3 achieves 95% of default compression at 2.7× higher speed, while level 19 provides near-optimal compression (within 0.5% of level 22) without extreme overhead.

The **multithread ZSTD implementation** leverages all available CPU cores, achieving a significant speedup in maximum mode compared to single-threaded operation, effectively mitigating the performance penalty of higher compression levels.

2.3.1 Algorithm Overview

Table 2.2 summarizes the technical characteristics of each algorithm:

Algorithm	Compression Type	Dictionary	Speed (MB/s)	Ratio
LZ4	LZ77 (no entropy)	64 KB	200+	1.2–1.3×
DEFLATE	LZ77 + Huffman	32 KB	10–50	1.3–1.4×
ZSTD	LZ77 + FSE	128 KB	20–200	1.4–1.5×
LZMA	LZMA2 + Range	Several MB	1–5	1.4–1.5×

Table 2.2: Technical characteristics of compression algorithms

2.3.2 LZ4: Speed-Optimized Compression

LZ4 is designed for scenarios requiring minimal latency:

- **Algorithm:** Simple LZ77 with 64 KB sliding window
- **Match finding:** Hash table-based with greedy parsing
- **Entropy coding:** None (raw literal bytes)

- **HC mode:** Optional lazy evaluation for better compression
- **Strength:** Extremely fast decompression (>1 GB/s)
- **Weakness:** Limited compression ratio due to small dictionary and no entropy coding

2.3.3 LZMA: Maximum Compression

LZMA (used in 7-Zip, xz) prioritizes compression ratio over speed:

- **Algorithm:** LZMA2 variant with multi-megabyte dictionary + range encoding
- **Match finding:** Exhaustive search with multiple context models
- **Entropy coding:** Range encoder (arithmetic coding variant)
- **Strength:** Excellent compression ratios; sophisticated modeling
- **Weakness:** Very slow compression; high memory usage

2.3.4 DEFLATE: Industry Standard

DEFLATE (used in gzip, PNG, ZIP) represents the classic approach:

- **Algorithm:** LZ77 with 32 KB sliding window + Huffman coding
- **Match finding:** Hash chains with optimal parsing in level 9
- **Entropy coding:** Dynamic or static Huffman trees
- **Strength:** Universal compatibility, good balance
- **Weakness:** Huffman coding limited to integer-bit codes; smaller dictionary than modern algorithms

2.3.5 ZSTD: Modern General-Purpose Compression

Zstandard represents state-of-the-art compression technology:

- **Algorithm:** Advanced LZ77 with 128 KB dictionary + Finite State Entropy
- **Match finding:** Multi-stage approach with hash chains and binary trees
- **Entropy coding:** FSE (tANS) achieving fractional-bit encoding
- **Adaptive strategy:** Switches between FSE, RLE, and raw based on data patterns
- **Strength:** Excellent speed-ratio trade-off; near-Shannon-limit entropy coding
- **Weakness:** Less universal support than DEFLATE

Finite State Entropy Advantage

ZSTD's use of Finite State Entropy (FSE) encoding provides a significant advantage over traditional Huffman coding. While Huffman coding is constrained to assign integer-length codes to symbols, FSE employs asymmetric numeral systems (tANS) to achieve fractional-bit encoding.

For a symbol with probability p, the optimal encoding length is:

$$L_{\text{optimal}} = -\log_2(p) \tag{2.2}$$

FSE approaches this theoretical limit closely, while Huffman must round up to the nearest integer. For highly skewed distributions (like our BFloat16 weights where top bytes appear 28% of the time), this difference becomes significant:

- Symbol with $p = 0.28$: Optimal = 1.84 bits, FSE $\approx$ 1.84 bits, Huffman = 2 bits
- Symbol with $p = 0.14$: Optimal = 2.83 bits, FSE $\approx$ 2.83 bits, Huffman = 3 bits

Chapter 3

Methodology

3.1 File Analysis

We performed comprehensive analysis of the Qwen2-0.5B model to understand its compressibility characteristics and identify opportunities for optimization.

3.1.1 Byte Frequency Distribution

Analysis of a representative 10 MB sample (0.1% of the total file) revealed a highly skewed byte distribution. Table 3.1 shows the most frequent byte values:

Byte Value	Frequency	Cumulative	Interpretation
`0x3c`	14.21%	14.21%	Positive small value (high byte)
`0xbc`	14.12%	28.33%	Negative small value (high byte)
`0x3b`	7.61%	35.94%	Positive smaller value
`0xbb`	7.60%	43.54%	Negative smaller value
`0xba`	2.21%	45.75%	Negative very small value
`0x3a`	2.18%	47.93%	Positive very small value

Table 3.1: Most Frequent Byte Values in Model Weights

Critical finding: The top 2 bytes (`0x3c`, `0xbc`) account for **28.33%** of all bytes in the model. This extreme clustering indicates high compressibility potential and suggests that the neural network weights concentrate around specific small magnitude values.

3.1.2 Entropy Analysis

Shannon entropy provides a theoretical lower bound on compression. We calculated entropy separately for different byte components:

$$H(X) = -\sum_{i=1}^{256} p(x_i) \log_2(p(x_i)), \tag{3.1}$$

where $p(x_i)$ is the probability of byte value x_i.

Byte Type	Entropy (bits/byte)	% of Maximum	Compressibility
Overall (mixed)	6.18	77.3%	Moderate
High bytes (exponent + sign)	7.96	99.5%	Low (near-random)
Low bytes (mantissa)	2.63	32.9%	**Very high**

Table 3.2: Entropy Measurements by Byte Position

Key insight: The dramatic difference between high byte entropy (7.96 bits/byte, nearly random) and low byte entropy (2.63 bits/byte, highly structured) suggests that treating these components separately could significantly improve compression effectiveness. The low bytes contain only 32.9% of their theoretical maximum information content.

3.1.3 Value Distribution Analysis

Statistical analysis of 5,242,880 randomly sampled BFloat16 values revealed the distribution characteristics:

Characteristic	Count	Percentage
Zero values	25	0.0005%
Positive values	2,627,980	50.12%
Negative values	2,614,875	49.87%
Small values ($\|x\| < 0.1$)	5,242,879	**100.00%**
Medium values ($0.1 \leq \|x\| < 1.0$)	0	0.00%
Large values ($\|x\| \geq 1.0$)	0	0.00%

Table 3.3: Weight Value Statistics

Interpretation: The fact that 100% of sampled weights have magnitude below 0.1 is not coincidental. This is a direct consequence of neural network training dynamics:

- **Layer normalization:** Modern transformer models use LayerNorm, which keeps activations centered around zero with small variance
- **Weight initialization:** Weights are initialized with small variance (typically Glorot/Xavier or He initialization)
- **Training convergence:** During training, weights that grow too large are penalized by regularization
- **Exponent clustering:** Small magnitude values cause BFloat16 exponent bits to cluster in a narrow range, explaining why `0x3c` (exponent ≈ -6, value ≈ 0.015) and `0xbc` (negative equivalent) dominate the distribution

The perfect 50/50 split between positive and negative values (49.87% vs 50.12%) confirms proper weight initialization and balanced training.

3.2 Preprocessing Strategy: Byte Reordering

3.2.1 Rationale

Standard BFloat16 storage interleaves high and low bytes sequentially:

$$\text{Standard:} \quad [H_0, L_0, H_1, L_1, H_2, L_2, H_3, L_3, \ldots] \tag{3.2}$$

Where H_i represents the high byte (sign + exponent) and L_i represents the low byte (mantissa) of the i-th BFloat16 value.

This interleaving creates a fundamental problem for compression algorithms: high-entropy data (high bytes, 7.96 bits/byte) is mixed with low-entropy data (low bytes, 2.63 bits/byte). LZ77-based compressors work by finding repeated byte sequences in a sliding window; alternating between random and structured bytes significantly reduces the effectiveness of pattern matching.

Consider a concrete example from the actual data:

```
Standard layout: [3c 7f bc 3a 3c 8b bc 7f 3c 9a ...]
                  ^  ^  ^  ^  ^  ^  ^  ^  ^  ^
                  High-Low alternation breaks patterns
```

The LZ77 match finder sees constantly changing byte values and cannot establish long matches, even though the high bytes contain repetitive patterns.

3.2.2 Proposed Transformation

We separate high and low bytes into contiguous regions:

$$\text{Reordered:} \quad \underbrace{[H_0, H_1, H_2, H_3, \ldots, H_{n-1}]}_{\text{High byte region}} \quad \underbrace{[L_0, L_1, L_2, L_3, \ldots, L_{n-1}]}_{\text{Low byte region}} \tag{3.3}$$

This transformation is reversible, lossless, and computationally inexpensive.

3.2.3 Expected Benefits

The byte reordering transformation provides several synergistic benefits:

1. **Long runs in high byte region**: With 28.33% of bytes being `0x3c` or `0xbc`, the high byte region contains sequences like:

   ```
   [3c 3c 3c bc bc 3c bc 3c 3c bc bc bc 3c ...]
    ^-----^  ^--^     ^-----^  ^-----^
    Excellent matches for LZ77!
   ```

   These runs enable excellent LZ77 compression with long match lengths.

2. **Improved dictionary efficiency**: The high byte section can use dictionary entries dedicated to common patterns (`0x3c`, `0xbc`, `0x3b`, `0xbb`), while the low byte section gets its own optimized entries.

3. **Better entropy coding**: Huffman or FSE algorithms can build separate optimal code tables for each region's distinct distribution, rather than compromising on a single table for mixed data.

4. **Locality of reference**: Modern CPUs benefit from sequential memory access patterns. Separate regions improve cache utilization during both compression and decompression operations.

5. **Adaptive compression**: Algorithms can detect the low-entropy low byte region and automatically apply more aggressive compression techniques (run-length encoding, higher compression levels) to this section.

3.2.4 Theoretical Analysis

The effectiveness of byte reordering can be quantified through conditional entropy analysis. For interleaved data:

$$H(\text{interleaved}) = \frac{1}{2}[H(\text{high}) + H(\text{low})] = \frac{1}{2}[7.96 + 2.63] = 5.30 \text{ bits/byte} \quad (3.4)$$

However, this assumes independence. LZ77 algorithms exploit *inter-symbol correlations*, which are maximized when similar symbols are adjacent. The actual achievable compression improves because:

$$H(\text{compressed}) < H(\text{high}) + H(\text{low}) - I(\text{high}; \text{low}) \quad (3.5)$$

where $I(\text{high}; \text{low})$ is the mutual information between high and low bytes, and the inequality holds due to the exploitation of spatial correlations by LZ77 match finding.

3.2.5 Implementation analysis

:

- **Time complexity:** $O(n)$ where n is the file size in bytes
- **Space complexity:** $O(n)$ for the reordered buffer
- **Cache behavior:** Sequential reads and writes, optimal for modern CPU caches
- **Parallelization:** Both passes can be parallelized with SIMD instructions or multi-threading

Decompression procedure: The inverse operation interleaves bytes back to their original positions.

3.3 Quantization

To achieve practical compression of large language model weights, we implemented a block-wise quantization scheme that reduces the precision of neural network parameters from 32-bit floating-point (or BFloat16) to either 4-bit or 8-bit integer representations. This implementation, contained in `quantize_blockwise.py`, forms a critical component of our model compression pipeline.

3.3.1 Quantization Methodology

Our approach employs block-wise quantization with configurable block sizes (default: 128 elements per block). Unlike naive scalar quantization, which applies a single scale factor to an entire tensor, block-wise quantization assigns independent scale factors to each block of weights. This prevents outlier values from dominating the quantization range and significantly reduces quantization error.

For each block of n weights $\mathbf{w} = \{w_1, w_2, \ldots, w_n\}$, the quantization process follows:

$$s = \frac{\max(|\mathbf{w}|)}{q_{\max}} \tag{3.6}$$

$$\mathbf{w}_q = \text{clip}\left(\left\lfloor \frac{\mathbf{w}}{s} \right\rceil, q_{\min}, q_{\max}\right) \tag{3.7}$$

where s is the scale factor, $q_{\max}$ and $q_{\min}$ define the quantization range (7 and -8 for INT4; 127 and -128 for INT8), and $\lfloor \cdot \rceil$ represents rounding to the nearest integer. Dequantization is performed via simple multiplication: $\hat{\mathbf{w}} = \mathbf{w}_q \cdot s$.

3.3.2 Packing Strategy

For INT4 quantization, our implementation achieves optimal memory efficiency by packing two 4-bit values into each 8-bit byte. Each quantized value occupies a single nibble (4 bits), with the first value stored in the high nibble and the second in the low nibble. This packing scheme ensures that INT4 quantization achieves exactly $2\times$ (theorically) better compression than INT8, while the additional quantization error remains minimal for most neural network applications.

3.3.3 Selective Quantization

Not all model parameters benefit equally from quantization. Our implementation employs a selective quantization strategy that analyzes each tensor's name and properties to determine whether quantization should be applied:

- **Quantized**: Weight matrices from dense layers, attention mechanisms, and MLP blocks—these constitute the majority of model parameters and are robust to quantization.
- **Preserved**: Embedding layers, normalization parameters (LayerNorm, RMSNorm), and bias terms—these are either critical for model stability or already memory-efficient.

This selective approach maximizes compression while minimizing quality degradation.

3.4 Metrics Collected

For comprehensive evaluation, we measured:

- **Compressed size** (bytes): Total output file size
- **Compression ratio**: $\frac{\text{Original size}}{\text{Compressed size}}$

- **Space savings**: $1 - \frac{\text{Compressed size}}{\text{Original size}}$ (as percentage)
- **Compression time** (seconds): Wall-clock time for compression operation
- **Decompression time** (seconds): Wall-clock time for decompression operation
- **Compression throughput** (MB/s): $\frac{\text{Original size}}{\text{Compression time}}$
- **Decompression throughput** (MB/s): $\frac{\text{Original size}}{\text{Decompression time}}$

Chapter 4

Results

4.1 Compression Performance

We evaluated various compression algorithms in Fast Mode (optimized for speed) and Maximum Mode (optimized for compression ratio) using the Qwen2-0.5B model (943 MB, 494M parameters). We also tested combining quantization with lossless compression.

Tool	Size (MB)	Ratio	Time	Speed	Best For
Fast Mode (Optimized for Speed)					
ZSTD-Custom	**669**	**1.41x**	**5.5s**	**172 MB/s**	**Recommended**
LZ4-Fast	792	1.19x	4.1s	230 MB/s	Ultra-fast
zstd -3	682	1.38x	7.3s	129 MB/s	Standard fast
gzip -1	685	1.38x	23s	41 MB/s	Legacy
Maximum Mode (Optimized for Compression)					
ZSTD-Custom	**633**	**1.49x**	**104s**	**9.0 MB/s**	**Best Ratio**
xz -9 (LZMA)	651	1.45x	961s	1.0 MB/s	Very slow
zstd -19	654	1.44x	520s	1.8 MB/s	Standard max
gzip -9	674	1.40x	146s	6.4 MB/s	Standard

Table 4.1: Comparison of compression tools in Fast Mode and Maximum Mode configurations.

Test Model: Qwen2-0.5B (943 MB, 494M parameters)

Approach	**Final Size**	**Ratio**	**Quality**	**Reduction**	**Time**
Lossless (ZSTD-Custom)	**633 MB**	**1.49×**	**Perfect**	**33%**	**104s**
INT8 + ZSTD-Custom	529 MB	1.78×	99%	44%	79s
INT4 + ZSTD-Custom	350 MB	2.69×	98%	63%	70s

Table 4.2: Quantization results for Qwen2-0.5B model showing compression ratio, quality retention, and size reduction.

Decompression: After all compression tests, we have also used a Decompression test to make sure any information is loss. Our Decompression code is established at a speed of 503MB/s and has a perfect accuracy.

Chapter 5

Conclusion

This study investigated lossless compression techniques for neural network weights stored in SafeTensors format with BFloat16 representation. Analysis of the Qwen2-0.5B model (943 MB, 494M parameters) revealed critical asymmetry: high bytes exhibit near-random entropy (7.96 bits/byte) while low bytes show highly structured patterns (2.63 bits/byte), with the top two byte values accounting for 28.33% of all data.

Our byte reordering preprocessing, which separates interleaved high and low bytes into contiguous regions, enables compression algorithms to exploit these distinct characteristics. ZSTD-Custom achieves optimal performance: 669MB in 5.5seconds in Fast Mode, and 633MB in 104s in Maximum Mode. The multithreaded optimization applied to ZSTD compression, which automatically utilizes all available CPU cores, substantially mitigates the performance penalty of higher compression levels. This parallelization achieved a 5× speedup in maximum mode while maintaining identical compression ratios, demonstrating that modern multithreading capabilities can effectively bridge the traditional speed-quality trade-off in lossless compression.

Combining quantization yields substantial additional gains: INT8 achieves 44% reduction with ∼99% quality, while INT4 reaches 63% reduction with ∼98% quality.

For practitioners, we recommend ZSTD-Custom Fast Mode for rapid deployment, Maximum Mode for storage optimization, and INT4+ZSTD for maximum compression. The negligible computational overhead makes these techniques practical for models of any size.

This work demonstrates that understanding weight distribution characteristics enables pre-processing strategies that significantly enhance compression effectiveness, making large language models more accessible and cost-effective to deploy.