Lossy Compression of LLM Weights in Safetensors Format

Aaron Pinto

Abstract

Large language models (LLMs) such as DeepSeek and Google's Gemma require hundreds of gigabytes of storage when shared via Hugging Face. These models are typically split into many safetensor files—secure, fast tensor storage formats (1)--each often 4-5 GB in size. Downloading or distributing such massive models is time-consuming. Common solutions like 8-bit or 4-bit quantization reduce size by lowering precision (6), but may still yield sizable files or require retraining. In contrast, error-bounded scientific compressors (for example, SZ) can achieve much higher compression ratios with controlled error. This work applies the Python SZ implementation (PySZ) to a DeepSeek-V3 safetensor shard. A compression ratio of approximately 13.33× was obtained (from ≈4.4 GB to ≈0.33 GB) while successfully reconstructing the tensor file for potential model use. Methodology, compression statistics, and implications for model fidelity versus quantization are described.

1. Background

Hugging Face's safetensors format is a zero-copy tensor storage format designed to safely replace pickle-based weight files (1). It is widely used for sharing LLM weights (for example, DeepSeek and Google's Gemma) because it is secure and fast (2), (3). However, modern LLMs can be extremely large. For example, DeepSeek-V3 is partitioned into many safetensor shards totaling hundreds of gigabytes; most shards are roughly 4.3–5.3 GB each (2). Similarly, Google's Gemma-3-27B-IT model uses multiple safetensor files with several shards near 4.9 GB each, for a large overall footprint (3). Downloading tens to hundreds of gigabytes to run or fine-tune these models locally poses a major bottleneck.

Model compression techniques offer solutions. Quantization (for example, 8-bit or 4-bit) is widely used and reduces storage by lowering numeric precision; tools such as bitsandbytes facilitate these approaches (6). These methods can achieve substantial size reduction without significant performance degradation in many cases, but extreme quantization can still leave sizable files and may require calibration or retraining.

Error-bounded lossy compression is common in scientific computing for floating-point data. Tools such as SZ and ZFP exploit spatial and numerical structure to compress floating-point arrays by large factors when a user-specified error tolerance is acceptable. Prior work shows such compressors can achieve orders-of-magnitude reductions on scientific datasets (5). In neural networks, SZ-based approaches applied to convolutional networks have yielded high

compression ratios with negligible accuracy loss in certain settings (5). Lossless compressors tailored to model files (for example, ZipNN) produce only modest reductions for floating-point checkpoints (7). The combination of these observations motivates exploration of lossy, error-bounded compressors for LLM weight shards as an alternative or complement to quantization.

2. Materials

The computational environment consists of a MacBook Air (M1, 2020), a Wi-Fi connection, and Python version 3.12.3. The core compression tool is PySZ, the Python bindings for the SZ3 scientific lossy compression library, installed by cloning the *SZ3/PySZ* repository and building it from source.

The following additional libraries, models, and datasets are employed:

1. safetensors (and optionally *Hugging Face transformers*) — for reading and writing safetensor model files.
2. transformers and accelerate — for loading large language models (LLMs) and running inference.
3. bitsandbytes — used to establish quantization baselines where applicable.
4. Standard Python stack: NumPy, pandas, tqdm, and matplotlib for data processing, logging, and visualization.
5. DeepSeek-V3-0324 safetensor shards — used as the initial test model.
6. One to two additional open safetensor-based LLMs (e.g., Gemma) — for cross-model generalization.
7. Evaluation datasets: a small custom prompt list, a subset of the GLUE benchmark, and held-out text segments for perplexity measurement.

3. Methodology

3.1 Model Selection and Baseline Verification

1. Select the DeepSeek-V3-0324 model as the initial test subject.
2. Download all model safetensor shards using the Hugging Face CLI or *transformers* utilities.
3. Store these shards in models/DeepSeek/original_shards/.
4. Load the uncompressed model and perform a short baseline text-generation test to establish reference outputs.
5. Save the generated baseline output and record the total original safetensor size (sum of all shard sizes).

6. Create an inventory file (models/DeepSeek/inventory.csv) listing shard filenames, tensor names, and sizes.

3.2 Environment Preparation and PySZ Validation

1. Clone the *PySZ/SZ3* repository to tools/pysz/.
2. Follow the build and installation instructions, resolving any dependency issues encountered.
3. Implement and execute a small validation script (tools/pysz/test_simple_array.py) that compresses and decompresses a random NumPy array.
4. Verify reconstruction error and record both compression ratio and runtime.
5. Confirm correct API usage of pysz.compress() and pysz.decompress() functions.
6. Create a reproducible Python environment file (environment.yml or requirements.txt).

3.3 Utility Scripts for Safetensor Operations

1. Tensor Extraction:
    - Implement scripts/extract_tensors.py to read each safetensor shard and iterate over all contained tensors.
    - Save each tensor as a .npy file in a structured directory: work/DeepSeek/tensors/<shard_name>/<tensor_name>.npy.
    - Log tensor shapes and data types.
2. Compression:
    - Implement scripts/compress_tensor.py to compress individual .npy tensors using PySZ with specified parameters.
    - Generate .pysz compressed artifacts and JSON metadata files containing compression settings and output sizes.
    - Include command-line flags for absolute and relative error bounds, block size, and other relevant PySZ options.
3. Batch Compression:
    - Develop scripts/batch_compress_model.py to iterate over all model tensors and invoke compress_tensor.py with a consistent configuration.
    - Track the original and compressed sizes, compression ratio, parameters used, and runtime per tensor.

3.4 Compression Runs

1. Begin with conservative PySZ configurations (small absolute error bounds).
2. Execute batch_compress_model.py to generate run_001.
3. Record total compressed size and compute the overall compression ratio (original_total ÷ compressed_total).

4. Repeat the procedure using progressively larger error bounds (e.g., 1e-5, 1e-4, 1e-3), generating run_002, run_003, and run_004 respectively.
5. Observe scaling behavior of compression ratios with respect to increasing tolerance.
6. Preserve all metadata and manifests for subsequent analysis.
7. For each run, compute:
    - Mean absolute error per tensor.
    - Maximum absolute error.
    - Percentage of tensors exceeding the target error bound.

3.5 Reconstruction and Safetensor Reassembly

1. Implement scripts/decompress_tensor.py to reconstruct .npy tensors from .pysz and metadata files using PySZ decompression.
2. Verify tensor shape and data type consistency with the original.
3. Implement scripts/reassemble_safetensor.py to reconstruct safetensor shard files identical in structure and filenames to the originals.
4. Use checksums or validation steps to ensure correct file and tensor name preservation.
5. Repeat for each compression run.

3.6 Functional Loading and Smoke Tests

1. Attempt to load each decompressed safetensor model using *transformers* or safetensors utilities.
2. If model loading fails, review mismatch logs and validate tensor shapes.
3. Correct assembly issues before proceeding.
4. Re-run the original baseline prompt against the decompressed model.
5. Save outputs as results/decompressed_run_00X_outputs.txt.
6. Compare the outputs qualitatively and record any visible degradation.

3.7 Benchmarking and Quantitative Evaluation

1. Define a reproducible benchmark suite comprising:
    - 50 curated prompts for generation-quality evaluation.
    - A small held-out text sample for perplexity computation.
    - A simple QA or classification subset, if supported.
2. Implement scripts/evaluate_models.py to perform inference for all model variants:
    - Original uncompressed model.
    - Decompressed versions from runs 001–004.
    - Quantized baselines (8-bit and 4-bit).
3. Compute the following metrics:
    - Generation quality (BLEU/ROUGE-style overlap).

- Perplexity on held-out text.
- Classification accuracy or F1 score, where applicable.
- Runtime and peak memory usage.

4. Save evaluation results as results/eval_summary_run_00X.csv.

3.8 Quantization Baselines and Comparisons

1. Generate 8-bit and 4-bit quantized models using Hugging Face quantization tools.
2. Store outputs in models/DeepSeek/quantized_8bit/ and related directories.
3. Document all quantization parameters and procedures.
4. Execute the same benchmark suite used for decompressed runs and record comparative results.

3.9 Parameter Sweep and Sensitivity Study

1. Automate a grid-based parameter sweep across multiple PySZ error bounds and, if desired, compression modes (absolute/relative) and block sizes.
2. Identify thresholds at which model quality metrics decline below acceptable limits.

3.10 Generalization Tests

Repeat the compression and evaluation pipeline on one or more additional open-source safetensor LLMs to test generality across architectures and shard configurations. Record and analyze differences in behavior by model type and tensor distribution.

4. Results

Applying the described pipeline to model-00034-of-000163.safetensors produced a compression ratio of 13.33×. Reported approximate size values were:

- Original Size: ≈ 4403 MB (≈ 4.3 GiB)
- Compressed Size: ≈ 330 MB
- Compression Ratio: 13.33× (compressed ≈ 7.5% of original)

All tensors compressed and decompressed successfully, and a reconstructed safetensor file named reconstructed_model.safetensors was written. No decompression exceptions were reported. The obtained compression ratio substantially exceeds that achievable by naive precision reduction alone; for comparison, uniform reduction from 32-bit to 8-bit would yield at most a 4× reduction in raw storage (6). Lossless safetensor-centered compression solutions have also reported modest shrinkage in practice, often substantially less than the observed 13.33× (7).

5. Discussion

Error-bounded lossy compression appears to be a promising direction for reducing LLM model size for distribution. Prior studies indicate that SZ and ZFP can compress scientific floating-point arrays by very large factors when appropriate error bounds are selected (5). Application of SZ to neural network weights has achieved large compression ratios in other model classes with acceptable accuracy trade-offs (5). The 13.33× reduction obtained in the present process suggests that similar benefits may extend to transformer-based LLM weights.

Comparisons with quantization reveal different trade-offs. Quantization reduces storage by lowering numeric precision uniformly; libraries such as bitsandbytes make such quantization for inference accessible without retraining (6). For storage/transmission-focused goals, error-bounded compressors can outperform quantization at the same size budget because compression adapts to numerical patterns in the data rather than enforcing uniform reduced precision. Empirical work on vision models has shown that error-bounded compression can retain accuracy better than uniform quantization at similar storage sizes (5). For LLMs, the hypothesis follows that SZ-compressed weights might preserve inference behavior more faithfully than heavily quantized weights of equal size, but this requires direct evaluation on downstream tasks.

Operational considerations include the following. The compressed format is lossy, so acceptance depends on allowable error for downstream tasks. The decompression step imposes CPU or I/O overhead at load time; however, this overhead can be offset by reduced network transfer time and disk I/O when retrieving models from remote storage. Integration into model distribution pipelines could allow serving compressed shards and decompressing on first load or on-demand, similar in spirit to existing model packaging plugins but for lossy formats. Additionally, compatibility issues (for example, dtype handling and conversion of bfloat16 to float32) need explicit handling in any production pipeline. The current pipeline converts bfloat16 to float32 before compression; this increases pre-compression size for those tensors but ensures compatibility with SZ. A more sophisticated approach could handle bfloat16 directly if the compressor and configuration support that dtype.

Limitations of the present report include absence of quantitative model-performance evaluation after decompression. Successful reconstruction and safetensor saving demonstrate structural integrity but do not measure inference fidelity. The next step should be to run the reconstructed model on representative tasks and compute metrics such as perplexity, accuracy on benchmark tasks, or output similarity measures (for example, cosine similarity of logits or classification metrics). Also, exploration of error-bound tuning per tensor group (for example, smaller error bounds for attention or layer-norm parameters) may produce better size–accuracy trade-offs. Hybrid schemes that combine lightweight quantization with per-tensor lossy compression might also be fruitful. Finally, comparison with state-of-the-art model-compression techniques that use

retraining (for example, advanced quantization-aware methods) could clarify application-specific choices.

6. Conclusion

The present study provides a proof-of-concept showing that SZ-based lossy compression (via PySZ) can reduce a DeepSeek safetensor shard by approximately 13.33× and reconstruct a valid safetensor checkpoint. Error-bounded lossy compression is a viable candidate for reducing transmission and storage costs for large model checkpoints. Future work must include systematic evaluation of model inference fidelity after decompression, tuning of per-tensor error bounds, and exploration of integration into model distribution ecosystems. If downstream accuracy is preserved within acceptable margins, lossy scientific compression could complement or in some cases outperform conventional quantization for storage-focused model sharing.

7. References

1. Hugging Face. *safetensors — Hugging Face Documentation*. Hugging Face, n.d. Web.
2. DeepSeek AI. *DeepSeek-V3-0324 — Model Files*. Hugging Face Model Hub, 2024. Web.
3. Google. *google/gemma-3-27b-it — Model Files*. Hugging Face Model Hub, 2024. Web.
4. PySZ. *pysz — Python Package Index (PyPI)*. Python Software Foundation, 2024. Web.
5. Lim, Seung Moo, and Seunghyeon W. Jin. “Neural Network Compression Using Error-Bounded Lossy Compression Techniques.” *Electronics*, vol. 11, no. 6, 2022, article 858. Web.
6. Hugging Face. *Quantization and bitsandbytes — Transformers Documentation*. Hugging Face, 2024. Web.
7. ZipNN. *ZipNN: Lossless Compression for AI Models (GitHub)*. ZipNN Project, 2024. Web.