Everything You Always Wanted to Know About Storage Compressibility of Pre-Trained ML Models but Were Afraid to Ask

Zhaoyuan Su*
University of Virginia
acf7ea@virginia.edu

Ammar Ahmed
University of Minnesota
ahme0599@umn.edu

Zirui Wang
University of Virginia
eeb9sd@virginia.edu

Ali Anwar
University of Minnesota
aanwar@umn.edu

Yue Cheng
University of Virginia
mrz7dp@virginia.edu

ABSTRACT

As the number of pre-trained machine learning (ML) models is growing exponentially, data reduction tools are not catching up. Existing data reduction techniques are not specifically designed for pre-trained model (PTM) dataset files. This is largely due to a lack of understanding of the patterns and characteristics of these datasets, especially those relevant to data reduction and compressibility.

This paper presents the first, exhaustive analysis to date of PTM datasets on storage compressibility. Our analysis spans different types of data reduction and compression techniques, from hash-based data deduplication, data similarity detection, to dictionary-coding compression. Our analysis explores these techniques at three data granularity levels, from model layers, model chunks, to model parameters. We draw new observations that indicate that modern data reduction tools are not effective when handling PTM datasets. There is a pressing need for new compression methods that take into account PTMs' data characteristics for effective storage reduction.

Motivated by our findings, we design ELF, a simple yet effective, error-bounded, lossy floating-point compression method. ELF transforms floating-point parameters in such a way that the common exponent field of the transformed parameters can be completely eliminated to save storage space. We develop ELVES, a compression framework that integrates ELF along with several other data reduction methods. ELVES uses the most effective method to compress PTMs that exhibit different patterns. Evaluation shows that ELVES achieves an overall compression ratio of 1.52×, which is 1.31×, 1.32× and 1.29× higher than a general-purpose compressor (zstd), an error-bounded lossy compressor (SZ3), and the uniform model quantization, respectively, with negligible model accuracy loss.

PVLDB Reference Format:
Zhaoyuan Su, Ammar Ahmed, Zirui Wang, Ali Anwar, and Yue Cheng. Everything You Always Wanted to Know About Storage Compressibility of Pre-Trained ML Models but Were Afraid to Ask. PVLDB, 17(8): 2036 - 2049, 2024.
doi:10.14778/3659437.3659456

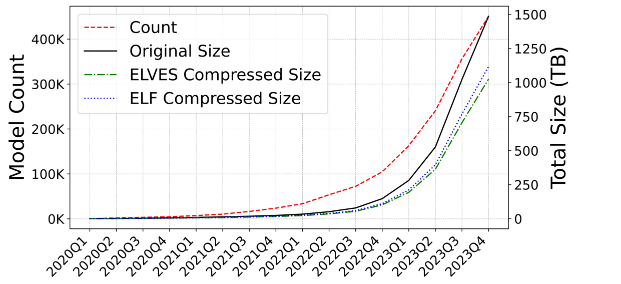

Figure 1: Increasing trend of model count (left Y) and aggregate storage size (right Y) of Hugging Face. *"ELVES Compressed Size" or "ELF Compressed Size" represents the storage trend after ELVES alone or ELF alone is applied to Hugging Face's PTM storage.*

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ds2-lab/ELF.

1 INTRODUCTION

As artificial intelligence (AI) and machine learning (ML) continue to evolve at a fast pace, a plethora of diverse models are being created and refined. These models reveal several emerging trends. First, the sheer number of pre-trained models (PTMs) is skyrocketing. These models are *pre-trained* to achieve a desirable accuracy for numerous tasks. PTMs are further reused to build task-specific models, which are fine-tuned with expertise [45, 51]. Typically, PTM datasets are persistently stored and managed in the format of *files* by model registry services such as Hugging Face [3] and TensorFlow Hub [7] to facilitate model sharing. For example, Hugging Face hosts over 450K PTMs (1,486.72 TB in size) as of December 31, 2023, and this number has been growing exponentially, as shown in Figure 1.

Second, the exponential growth of training datasets and the vast range of problem domains lead to more complex model architectures, enriched features, a significant rise in the number of parameters, and as a result, increasingly large model sizes [4, 13, 57]. As of the fourth quarter of 2023, stored PTM datasets in Hugging Face have already exceeded 1,400 TB (Figure 1) and the need for storage is projected to continue in the foreseeable future. These trends impose huge storage requirements for MLOps to store PTMs.

The extensive storage requirements associated with large pre-trained ML models could theoretically be alleviated through data reduction techniques. These techniques include general-purpose

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659456

lossless compression algorithms [1, 10], data deduplication methods for enterprise storage systems [23, 67], and floating-point compression algorithms for scientific datasets [8, 65], time series (TS) datasets [14, 33, 38, 47], and data lakes [32]. However, these strategies are not effective when dealing with PTM datasets becasue: (1) none of existing techniques are *aware of the data patterns of PTMs*; and (2) there is *a lack of understanding in those patterns, especially those relevant to data reduction and storage compressibility.*

Model pruning and quantization techniques [18, 27, 34], on the other hand, are typically used for reducing the memory and computational requirements during training or inference, posing many constraints and challenges in model integrity, usability, and accuracy when applied to PTM storage reduction.

To fill this gap, we present, to the best of our knowledge, the first comprehensive study of a large, real-world dataset of PTMs collected from Hugging Face on PTM storage compressibility. Our analysis seeks **comprehensiveness** in three dimensions.

- **Scale.** We collected a total of 8, 238 PTMs from Hugging Face, which include over 75K files and occupy around 13 TB storage capacity. We performed the analysis on file formats, file storage footprint, and model sizes under different model categories.
- **Data reduction techniques.** We sampled a representative set of 900 models from our large-scale dataset and performed an in-depth, what-if analysis of various widely used data reduction techniques. We first studied hash-based deduplication at the model layer level and chunk level. Then, we proceeded with data similarity detection to see whether model layers or model chunks are highly similar. Finally, we examine the dictionary coding compression technique at the parameter level.
- **Data granularity.** Our analysis explored the aforementioned techniques at three data granularity levels: model layers, model data chunks, and model parameters.

Our multifaceted analysis induces the following **observations**.

- **Real-world PTMs are large and deep.** Our analysis shows that 65.97% of the 8, 238 models fall within the size range between 100 MB and 1,000 MB. 45.83% of models have 150-250 layers.
- **Parameters of these models are highly concentrated.** The main contents of PTMs, model parameters, are predominantly floating-point values, and most of these floating-point parameters are `float32`. Across all 900 sampled models, 98.91% of all parameters fall within the range of $(-1, 1)$.
- **Most model layers and chunks are non-duplicate, nor are they similar.** Our analysis reveals that duplicate layers across all different model categories constitute only 5.72% of the overall storage footprint and our data similarity detection algorithm uncovers that only 7.98% of all 512-byte model data chunks bear any resemblance to each other.
- **A majority of models exhibit modest-to-high parameter redundancy.** About 48.94% of models have at least 50% of their parameters repeated at least once. However, widely used general-purpose dictionary coding compressors are generally not effective due to the limited length of floating-point parameters and the long distance between duplicate parameters. Nonetheless, dictionary coding is effective for 11.56% of models, where over 99% of their parameters are duplicated.

A high-level **ramification** of many of these observations is *a previously undisclosed insight: PTM storage compression is very challenging and existing techniques are generally ineffective due to the randomness of PTMs.* There is a need for new compression methods that *account for PTMs' data characteristics* in order to extract most of the compressibility from the datasets.

This paper makes the following **contributions**.

- We conduct the first, exhaustive study of the storage compressibility of real-world pre-trained ML model datasets and make key observations that motivate the design of new compression methods for PTM storage.
- We propose Elf (Exponent-Less Float-point encoding), a new, error-bounded, near-lossless floating-point compression method motivated by the observations from our analysis. The idea of Elf is simple yet effective: since most parameters in PTMs are within $(-1, 1)$, Elf maps all parameters $\in (-1, 1)$ to $[1, 2)$ so that the common exponent field `0b01111111` can be completely eliminated to save storage space. Elf is easily parallelizable and has fast compression and decompression speed.
- We develop Elves, an offline compression framework for efficient PTM storage. Elves incorporates Elf along with hash-based deduplication, length-distance dictionary coding, and a general-purpose lossless compressor. Our hybrid approach collectively compresses PTM datasets that exhibit different data patterns.
- We develop a validation framework that generates random inputs to validate the accuracy of Elves-decompressed models at scale.
- Experimental results show that: (1) Elves achieves the highest compression ratio (1.52) for our collected dataset compared to a wide range of 11 compression methods with near-zero model accuracy loss; and (2) in terms of compression and decompression speed, Elf outperforms all 13 selected baseline methods.

A CR of 1.52 might not appear remarkable if viewed in isolation. However, this result is significant compared with the state-of-the-art compression methods, a factor of 1.29× improvement compared to the best baseline compressor zfp. A 34% reduction in storage will translate to a cost reduction of hundreds of TBs of storage hardware: if we apply Elves (or Elf) to Hugging Face's PTM storage, it could have saved 509 TB (369 TB) of storage by end of 2023 Q4 as shown in Figure 1. The saved storage also means a potential improvement in datacenter TCOs, encompassing benefits such as reduced cooling, lower energy usage, and decreased carbon footprint [2].

2 RELATED WORK

Model Pruning and Quantization. There is a large body of research focusing on reducing the **memory and computational requirement** of ML models for *online* tasks such as model serving [19–21, 24–26, 28, 35, 62, 63].

- **Why pruning may not be ideal for PTM storage reduction:** Pruning removes insignificant connections or layers in the model, resulting in a smaller model representation [26]. Pruning can be generally categorized into structured pruning [12, 56] and unstructured pruning [24, 37, 54]. Structured pruning may remove entire channels, filters, or layers, leading to significant model size reduction. However, structured pruning impacts the integrity of PTMs: (1) from a model provider perspective (e.g., Hugging Face), such irreversible changes are often not acceptable to those who share PTM datasets via model registries; (2) these changes to model structures may affect subsequent MLOps operations. Unstructured pruning involves a trade-off: reducing the model's size requires a higher magnitude threshold, which can potentially compromise model performance.

- **Why quantization may not be ideal for PTM storage reduction:** Quantization [21, 25, 31, 62, 63] involves representing the parameters and activations of a model using fewer bits than the original. Quantization methods can be generally categorized into quantization-aware training [46], dynamic quantization [39], and post-training quantization [44, 62, 63]. The first two are typically not directly applicable to PTM storage. Post-training quantization typically reduces the precision of model parameters, such as converting from `float32` to formats like `float16`, `int8`, or 3-bit representation. However, this process introduces non-negligible errors that build up across the neural network, potentially leading to a significant deviation from the original model's performance.

Pruned and quantized models cannot use re-training or fine-tuning to "regain" the information loss for PTM storage, further affecting the models' accuracy, and diminishing their usability.

Elf is fundamentally different from pruning and quantization. Elf supports both compression and decompression and, thus, is capable of preserving model structures and recovering model parameters to their original data type, though with bounded loss. In contrast, pruning and quantization are one-way processes, meaning that once a model has undergone pruning and quantization, it cannot be fully recovered to its original state due to a lack of decompression. That is, quantization and pruning have irreversible effects on model information, therefore hindering subsequent operations on PTMs. Due to these drawbacks, these two techniques do not serve as ideal solutions for reducing the **storage requirement** of persistently-archived PTMs [3, 7].

Data Reduction and Compression. Large-scale enterprise and cloud storage systems often rely on data deduplication [23, 60, 67] and delta compression [11, 40, 64] to reduce storage costs, as these data—documents, source code, binary executables, webpage objects, and more—typically show high duplication rates or are highly similar. General-purpose lossless compressors can reduce file sizes by identifying redundant information and representing them in a more compact form [1, 6, 10, 68]. However, these data reduction techniques are not designed to handle floating-point-based datasets, which renders them largely ineffective for PTM datasets. Floating-point compression techniques for TS datasets [14, 16, 33, 38, 47] exploit the temporal data patterns and redundancies of TS data and use delta compression or XOR operations on successive values to eliminate redundant information or resulting XOR'ed zeros for space savings. Columnar storage formats [32, 55] are designed to compress large column datasets efficiently in data lakes by utilizing data-reduction and compression techniques, such as dictionary encoding, bit packing, and novel floating-point encoding. Lossy floating-point compression methods [17, 53] for scientific datasets (e.g., visualizations), such as SZ3 [65] and zfp [8], encode floating-point values by leveraging correlations among values. However, these floating-point compressors are not effective when it comes to PTM datasets since model parameters are cluttered, making it impossible to extract correlations or patterns.

Table 1: Distribution of model categories (full dataset). *NLP: natural language processing. CV: computer vision. RL: reinforcement learning. Uninformed: models with no category tag information.*

Category	Count (%)	Total Size in GB (%)
NLP	6,220 (75.5%)	7,661.22 (78.82%)
Audio	430 (5.22%)	466.79 (4.8%)
Multimodal	394 (4.78%)	358.6 (3.69%)
CV	195 (2.37%)	134.62 (1.39%)
RL	1 (0.01%)	0.0062 (0.0001%)
Uninformed	998 (12.12%)	1,098.5 (11.3%)
Overall	8,238 (100%)	9,719.73 (100%)

Table 2: Distribution of model categories (sampled dataset).

Category	Count (%)	Total Size in GB (%)
NLP	300 (33.33%)	170.85 (29.67%)
Audio	150 (16.67%)	154.30 (26.79%)
Multimodal	150 (16.67%)	97.81 (16.99%)
CV	150 (16.67%)	58.74 (10.20%)
Uninformed	150 (16.67%)	94.18 (16.35%)
Overall	900 (100%)	575.88 (100%)

3 DATASET OVERVIEW

Full Dataset. We have downloaded pre-trained ML models from a total of 8, 238 Hugging Face repositories as of October 20, 2022. These repositories include 75, 871 files, accounting for around 13.2 TB of storage space. `.json` files represent the largest proportion, comprising approximately 42.5% of all files. `.bin` files account for 16.5% of all files and primarily contain the binary data of models. Regarding the size distribution of different file formats, it is evident that the `.bin` files, which store PTMs, occupy the largest portion (71.9%) of the storage footprint. We obtained category tag information from each model repository. Based on this information, we categorized all collected models into six categories as shown in Table 1. Out of all the 8, 238 models, 75.5% are NLP models, consuming 78.82% of the storage size, highlighting the popularity of language models.

Sampled Dataset. We observe from the full dataset that `.bin` files that store the binary data of PTMs predominantly occupy the storage footprint, therefore, we focus on examining the characteristics and compressibility of these binary data throughout the rest of the paper. To do so, we use smaller samples of the full dataset that can fit within the storage capacity of typical storage server machines. The sampled dataset features a more balanced distribution of model categories to avoid bias (Table 2). This dataset includes 150 models each from the Audio, Multimodal, CV, and Uniformed categories. The NLP category contains 300 models, reflecting its prominence and prevalence in current applications. Unless stated otherwise, the rest of the paper will be focused on the 900-model, sampled dataset.

4 ANALYSIS: SIZES AND CONTENTS

This section presents our model size and content analysis that aims to answer the following research questions (RQs):

- **RQ1:** What are the sizes of PTMs in different categories?
- **RQ2:** What are the layer counts and sizes in these models?
- **RQ3:** What types of data (parameters) are stored in these models?

Model Sizes. We first analyze model sizes. Figure 2 shows that 64.78% of models fall within the size range between 100 MB and 1,024 MB, with an additional 25.22% surpassing the 1 GB threshold.

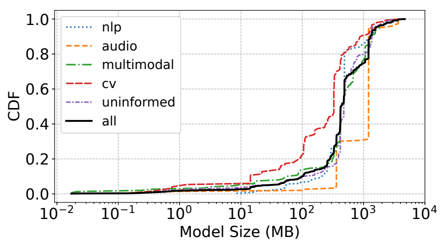

Figure 2: Model size distribution of the sampled dataset.

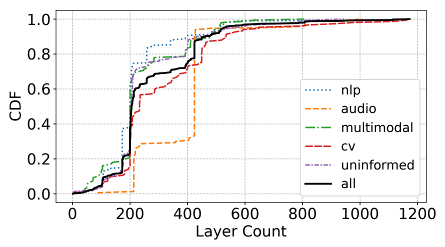

Figure 3: Model layer number CDF for different categories.

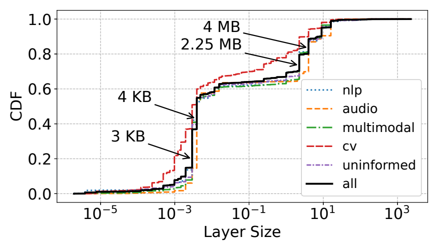

Figure 4: Model layer size CDF for different categories.

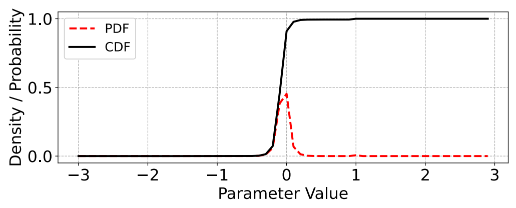

Figure 5: Parameter value distribution.

Model Layers. We extract the layer information of all the 900 models in the sampled dataset and report the details in Figure 3–4. The key observations are: (1) Around 75% of models have over 200 layers, likely because most models need a deep structure to capture complex features. (2) In the audio category, about 70% of models are very deep, with more than 400 layers. (3) Layer sizes exhibit a step-like distribution, with 58% of layer sizes concentrated around the scale of 3 KB, 4 KB, 2.25 MB, and 4 MB.

Table 3: Model layer data type distribution. *Others include* `float64`, `uint8`, *and* `int64`.

Layer Type	Count (%)	Total Sz in GB (%)	Avg Para #	Avg Sz in MB
float32	240,966 (96.95%)	557.84 (96.87%)	621,421	2.37
float16	4,018 (1.62%)	14.51 (2.52%)	1,939,421	3.70
others	3,561 (1.43%)	3.53 (0.61%)	595,181	1.02
Overall	248,545 (100%)	575.88 (100%)	642,352	2.37

Model Data Types. Next, we examine the data types of the sampled model dataset. We find that all parameters within an individual layer are of the same data type. Thus, we break down all model layers by data types and report the statistics of layer data types in Table 3. Most layers are of `float32` type with around 97% in both count and storage footprint. Layers of type `float16` and others make up 2.52%, and 0.61% of the total storage space, respectively.
Model Parameters. We then analyze the distribution of parameter values. As shown in Figure 5, 98.91% of all parameters fall within the range of $(-1, 1)$, with 50.50% in the interval $(-1, 0]$ and 48.41% in the interval $(0, 1)$.

Implications

• *Real-world, pre-trained ML models are considerably large in size. The rapidly increasing number of ML models poses significant challenges to MLOps for storing and managing the exponentially increasing volume of model datasets.*
• *These models typically contain a massive number of layers, potentially providing intriguing opportunities for data reduction using layer-based (or chunk-based) deduplication techniques and/or similarity-based delta compression techniques.*

• *Parameter values are concentrated within a narrow range of* $(-1, 1)$, *implying potential opportunities for the application of compression/encoding methods [1, 8, 16, 47, 65, 68]. These compression methods use a combination of techniques including prediction [36, 65], XOR-based zero encoding [16, 47], and error-encoding [66] to compress floating-point model datasets.*

5 ANALYSIS: COMPRESSIBILITY

In this section, we present our two-dimensional, what-if analysis of model data reduction and compressibility. Along the *data granularity dimension*, our analysis explores the potential data reduction yield at three different levels: model layers, data chunks, and individual parameters. Along the *data reduction and compression technique dimension*, we consider three widely used techniques: hash-based deduplication [23], similarity-based delta compression [64], and distance-encoding [68]. Specifically, our analysis aims to answer the following RQs:

- **RQ1:** Does duplication exist among model layers or model data chunks? If so, would data deduplication help in reducing the model data storage footprint?
- **RQ2:** Are there any model layers or chunks that are highly similar?
- **RQ3:** What is the repetition pattern of model parameters? If parameter-level repetition exists, how can this redundancy be eliminated or mitigated?

Table 4: Model layer duplication statistics based on data types. *The percentages in Columns 3 and 5 represent the proportion of the total count and total size under that row.*

Layer Type	Count	Dup %	Total Sz in GB	Dup Sz in GB (%)
float32	240,966	8.35%	557.84	30.14 (5.40%)
float16	4,018	3.61%	14.51	0.14 (0.96%)
float64	199	0%	0.81	0 (0%)
uint8	1,597	99.81%	1.75	1.74 (99.43%)
int64	1,765	96.77%	0.97	0.94 (96.91%)
Overall	248,545	9.48%	575.88	32.96 (5.72%)

5.1 Model Layer- and Chunk-level Duplication

Does Duplication Exist among Model Layers? Hash-based data deduplication method partitions the target dataset into fine-grained chunks, computes the hash values (i.e., fingerprints) of all data chunks, scans the partitioned dataset, and performs deduplication by removing duplicate chunks with identical fingerprints to save storage space. To understand if there is any duplication among model layers, we scanned the 900 models in the sampled dataset to

Table 5: Distribution of duplicate chunks based on data type. *Columns 3 and 4 display the size of duplicate chunks using fixed-size chunking (FSC) of 4 KB and 512 B, and Column 5 indicates the total size of duplicate chunks determined by content-defined chunking (CDC) with chunk sizes ranging from 128 B to 128 KB.*

Data Type	Total Sz (GB)	Size of Duplicates in GB (%)		
		4 KB (FSC)	512 B (FSC)	CDC
float32	557.84	40.35 (7.23%)	42.92 (7.69%)	44.50 (8.16%)
float16	14.51	0.14 (0.96%)	0.14 (0.96%)	0.15 (1.03%)
float64	0.81	0 (0%)	0 (0%)	0 (0%)
uint8	1.75	1.74 (99.43%)	1.74 (99.43%)	1.74 (99.43%)
int64	0.97	0.94 (96.91%)	0.96 (98.97%)	0.96 (98.97%)
Overall	575.88	43.17 (7.50%)	45.76 (7.95%)	47.35 (8.22%)

compute the layer fingerprints. Our results are discouraging. The analysis reveals that a mere 5.72% of the total storage footprint for model layers is accounted for by duplication (see Table 4). For `float32`-typed layers, duplicate layers only occupy 30.14 GB out of a total of 557.84 GB. Although layers of type `uint8` and `int64` are largely duplicate, their overall fraction is negligible.

Does Duplication Exist among Model Chunks? Next, we conduct chunk-based duplication analysis based on fixed-size chunking (FSC) and content-defined chunking (CDC) [49] approaches at chunk granularity. Table 5 shows the results. With FSC, chunks of 512 B exhibit a higher duplication ratio across the majority of data types compared to chunks of 4 KB. We utilized FastCDC [59], a widely used, state-of-the-art, gear-based CDC method, on our dataset. We find that the size of the duplicates detected by FastCDC is 47.35 GB, accounting for 8.22% of the total dataset size. This is 9.68% and 3.47% higher than FSC with a chunk size of 4 KB and 512 B, respectively. *Nevertheless, both the FSC and CDC duplication analysis shows similarly negative results, indicating that hash-based data deduplication might not effectively reduce the storage size of PTM datasets.*

5.2 Model Layer- and Chunk-level Similarity

We next study whether PTMs contain data that is similar but not exactly identical. In storage systems, delta compression often complements deduplication as a data reduction technique in order to eliminate redundancy among non-duplicate yet highly similar chunks [11, 40, 60]. For example, if chunk D_2 is similar to a base chunk D_1, a delta compressor will only store the differences, i.e., the *delta*, and the mapping between D_2 and D_1, by removing the redundant data for improved storage efficiency.

How to Detect Data Chunk Similarity? Widely used data similarity (resemblance) detection methods compute "super features" (SFs) [15, 48] based on the Rabin fingerprints [50][1] of data chunks and use the computed SFs to detect similar chunks. For example, Finesse [64], a state-of-the-art method, works as follows. (1) The base and target data chunks, D_1 and D_2, are partitioned into four sub-chunks each, and a group of hash values based on Rabin fingerprints are computed for all eight sub-chunks. (2) For both D_1 and D_2, three SFs are constructed. The first SF is constituted by using the largest hash values from each of its four sub-chunks, the second SF from the second largest hash values from the four sub-chunks, and the third SF from the third largest hash values from the four sub-chunks. (3) A hash value based on SFs for D_1 and D_2 is computed. If the hash value of D_1's SFs is the same as that of D_2, it indicates that D_1 and D_2 are highly similar.

[1]Rabin fingerprints compute a group of hash values using a sliding window that slides from the start to the end of the data chunk. The size of the sliding window is configurable and is set to 48 bytes as per [64].

Table 6: Similarity ratios for different granularities. *Similarity ratio is defined as the size of similar layers/chunks divided by the total size. (Note that the true similarity ratios are lower than reported in this table as the entire sampled dataset includes duplicate data.)*

Data Type	Total Sz (GB)	Size of Similar Data in GB (%)		
		Layer	4 KB	512 B
float32	557.84	30.17 (5.41%)	40.39 (7.24%)	43.12 (7.73%)
float16	14.51	0.14 (0.96%)	0.14 (0.96%)	0.14 (0.96%)
float64	0.81	0 (0%)	0 (0%)	0 (0%)
uint8	1.75	1.74 (99.43%)	1.74 (99.43%)	1.74 (99.43%)
int64	0.97	0.94 (96.91%)	0.95 (97.94%)	0.96 (98.97%)
Overall	575.88	32.99 (5.73%)	43.22 (7.51%)	45.96 (7.98%)

Are Model Layers or Chunks Highly Similar? Methods like Finesse introduce significant computational overhead. To explore the potential of delta compression, we designed and implemented a simple and efficient data similarity detection algorithm that approximates Finesse. Our approximation algorithm samples a parameter for every N parameters from each model layer or model chunk, with N setting to 32 in this method, computes the hashes of sampled parameters, and compares the hashes of two layers/chunks. Compared to Finesse which uses a sliding window to compute SF hashes, our algorithm reduces computational requirements by using sampling and a sliding window of 1. A side effect of the approximation is a potentially higher false positive rate, where two layers/chunks with sparse similarities might be inaccurately detected as highly similar. However, as shown in Table 6, the similarity ratios for model layers and 4 KB/512 B chunks remain remarkably low. For example, for the entire dataset, only 7.98% of 512 B blocks are identified as similar. *These negative results suggest that similarity-based delta compression will not be effective for reducing the storage footprint of PTMs.*

5.3 Model Parameter-level Duplication

What is the Repetition Pattern of Model Parameters? Recall we have shown in §4 that the values of model parameters are concentrated within a small range of $(-1, 1)$. Going one step deeper, we study the repetition pattern of individual parameters by counting the duplication ratio of parameters for each individual model from our sampled dataset. Here the parameter duplication ratio for a model is defined as the fraction of repetitive parameters. Figure 6 shows that 48.94% of models exhibit a parameter duplication ratio of over 50%, meaning that these models have at least half of their parameters duplicated at least once. Interestingly, 11.56% of models have over 99% duplicated parameters. The high duplication ratios imply potential opportunities for utilizing general-purpose compression methods to reduce parameter redundancies.

Will Off-the-Shelf Dictionary Coding Work? The high parameter duplication ratio motivates us to conduct a what-if analysis to study the feasibility of existing compression methods in reducing the parameter-level redundancy. We first explore a compression technique that is commonly used in today's general-purpose compressors, called dictionary coding [52]. Dictionary coding works

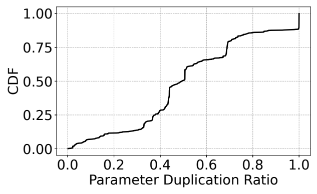

Figure 6: The parameter duplication ratio of all 900 models. Each data point in the CDF curve represents a model's duplication ratio.

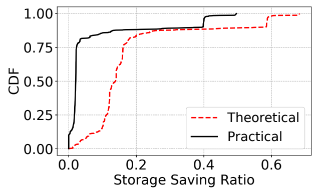

Figure 7: Storage saving ratios (% storage space saved) by practical DE (with overhead included) and theoretical DE (without overhead).

by using an external macro scheme, i.e., a separately stored dictionary data structure, to maintain a mapping between duplicated sequence patterns, e.g., text strings, and codes that locate the sequences. Given the high duplication ratio of PTM datasets, it seems that this might work. However, we found that, while most models have duplicate parameters, the absolute amount of unique duplicate parameters, i.e., the working set size of duplicate parameters, in a model can be enormous, which makes the space complexity of the dictionary extremely large. Worse, the average repetition frequency for all duplicate parameters is on the lower end. For example, in our dataset, 34.35% of models have over 60% of parameter duplicates; however, these duplicate parameters have an average repetition frequency of around 12, with a total unique parameter count exceeding 6.8 billion. This implies that at least 33 bits in the length code would be required to encode the whole dictionary. Given this, dictionary coding offers no data reduction benefit for PTM datasets.

We then consider a more efficient dictionary coding variant. Without external macros, this method encodes duplicate sequence patterns in pointers. A pointer is a length-distance (D, L) pair [52, 68], where the "distance" D tells the compressor (and the decompressor) how far back to look for the start of the repeated sequence, and the "length" L tells the compressor how many characters make up the repeated sequence. Real-world implementations such as LZ77 [68] typically use a sliding window to provide a dynamic dictionary of duplicate sequences that can be referred back to.

This general-purpose compression method can be more space-efficient to store a short pointer that refers to an earlier occurrence of a string than to store the whole string again, especially if the string itself is long and/or repeated frequently. This method typically works well for text-based datasets [1, 9]. Dealing with floating-point datasets such as PTM datasets, however, becomes challenging due to the following reasons. (1) Unlike string-based text datasets where duplicate sequences can be long, a duplicate model parameter is short, e.g., the most common data type in PTM datasets—`float32`-typed parameters (Table 3)—are only 4 bytes long. While the good news, in our case, is that the length value can be omitted if we target `float32`-typed parameters only, simply because `float32` parameters are of fixed length; the bad news is that the limited length of duplicate parameters puts a hard constraint on the potential gain in storage reduction. (2) Duplicate parameters are sparsely distributed within a model, requiring a large sliding window size and relatively long distance values. In a common implementation, the distance values might be represented as 16-bit integers, which can encode a distance of at most $2^{16} = 65,536$. Unfortunately, for most models in our dataset, the distance between two adjacent duplicate parameters is longer than that. Of course, this problem can be addressed by using longer distance values. However, the longer the distance value, the less data reduction gains the compressor would achieve. In theory, we need to control the distance value length to be shorter than 4 bytes (2^{32}) in order to receive gains.

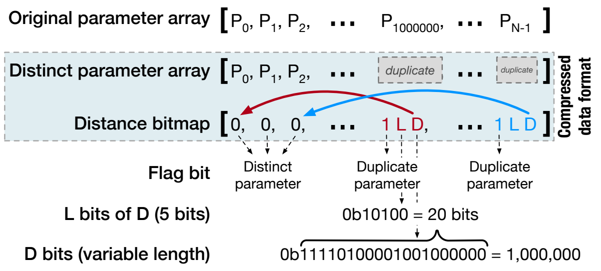
Original parameter array [P0, P1, P2, ... P1000000, ... PN-1]
Distinct parameter array [P0, P1, P2, ... duplicate ... duplicate]
Distance bitmap [0, 0, 0, ... 1 L D, ... 1 L D]
Compressed data format
Flag bit
Distinct parameter
Duplicate parameter
Duplicate parameter
L bits of D (5 bits)
0b10100 = 20 bits
D bits (variable length)
0b11110100001001000000 = 1,000,000

Figure 8: An example of distance-encoding compression. *L: length. D: distance. The distinct parameter array does not store duplicate parameters marked in dashed-line, shady boxes. In this example, $P_{1000000}$ duplicates with P_0. Therefore, DE toggles $P_{1000000}$'s flag bit as 1 and adds a length-distance pair (L, D). The 5-bit L field encodes 20, indicating that the bit length for D is 20 bits. D encodes a decimal value of* 1,000,000*, meaning that this duplicate refers back to a parameter that is* 1,000,000 `float32` *parameters ahead.*

How to Minimize Model Parameter Redundancy? To verify whether length-distance dictionary coding is effective, we designed and implemented an efficient length-distance-based compression method and data format, which we call distance-encoding (DE), targeting `float32`-typed and `float64`-typed model parameters. Our DE method stores a compressed model parameter file into two logically decoupled arrays: a distinct parameter array that is used to store duplicated and unique, and non-duplicate parameters and a distance bitmap that is used to store metadata to keep track of the pointers for duplicate parameters. Figure 8 gives an example of the data format. To compress, our DE *compressor* takes a linear pass of the model dataset. DE stores distinct parameters as is in the distinct parameter array and records a flag bit of 0 in the distance bitmap. DE skips a duplicate parameter in the distinct parameter array and records a flag bit of 1 followed by an (L, D) pair in the distance bitmap. L is a fixed-length, 5-bit field, which records the bit length of the next field, distance D, so that *the decompressor* knows how many bits to read in order to decode D. The 5 bits in L can encode a distance of at most 2^{31} if all five bits are used ($2^5 - 1 = 31$). As such, the D field can have variable length, ranging from 1 to 31 bits. Note that the fields of flag bit and L bits introduce storage overhead.

We evaluated DE on our 900-model dataset. Figure 7 plots the storage saving ratios achieved by: (1) a theoretical compressor without adding the flag bit and L-field overhead, representing a best-case baseline, and (2) a practical compressor that includes the extra overhead. DE, in theory, can achieve at least 10% storage savings for 83.67% of 900 models. However, this comes with the catastrophic consequence of being not decompressible due to the absence of metadata. Taking into account the extra metadata overhead, DE's efficacy decreases dramatically—it can only provide the same space savings for 13.22% of the models, which corresponds to the vertical curve at the top-right corner of Figure 6. But the good news is that the average storage saving ratio for these models is remarkable, at about 33%. This is because these models have over 99% of parameters duplicated, representing a best-case scenario for DE to be effective. *This mixed result suggests that general-purpose length-distance-based compression methods can achieve a reasonably high compression ratio only if model parameters are highly duplicated.*

Implications

- *The effects of hash-based data deduplication approaches are double-edged. First, duplicate layers in our sampled dataset make up only 5.72% of the storage size. Both the FSC-based and CDC-based deduplication see a very limited duplication ratio. We thus expect hash-based data deduplication approaches to be generally ineffective in reducing storage costs. Second, a much higher level of layer duplication exists in the integer-typed model layers, suggesting a potential avenue for future research. Overall, the effectiveness of data deduplication for PTM datasets appears to be minimal.*
- *Our approximation data similarity detection algorithm reveals that only 7.98% of all 512 B model chunks, including duplicate chunks, bear resemblance to each other. This result suggests that there is limited similarity within PTM datasets that could be leveraged by delta compression techniques.*
- *Model parameter-level duplication is virtually universal—with all 900 models in our dataset having duplicate parameters. However, this does not imply that widely used, general-purpose compression algorithms, such as length-distance-based dictionary coding, will be effective for PTMs. Two main reasons contribute to such lack of compressibility. First, the unit of the duplicate sequence—floating point numbers—is short and most duplicate parameters repeat infrequently. Second, the distance between duplicate parameters is typically long, requiring lengthy bits to encode the distance. On a positive note, though, 11.56% of models have over 99% duplicated parameters, and therefore, could benefit from high storage savings by using general-purpose compressors.*

6 ELF AND ELVES DESIGN

In this section, we first introduce a new, error-bound, lossy floating-point compression method ELF (Exponent-Less Float-point encoding), motivated by the observations from §4 and §5. We then present a compression framework named ELVES, which integrates two main compression methods, ELF (§6.1) and DE (§5.3), along with several other data reduction methods, to compress pre-trained ML model datasets. ELF compresses models that primarily consist of floating-point parameters that fall within the range of $(-1, 1)$, while DE targets models that have a significant proportion of out-of-range parameters. These two methods complement each other, thereby maximizing overall storage efficiency.

6.1 The ELF Compression Algorithm

Two key observations motivate the design of ELF. (1) Recall we have observed in §4 that around 99% parameters in our model dataset fall within the range of $(-1, 1)$. (2) Take the single-precision floating-point (`float32`) format as an example: In accordance with IEEE 754 Standard [30], a `float32` value p is stored with 32 binary bits, where 1 bit is for the sign s, 8 bits for the exponent $\vec{e} = \langle e_1, e_2, e_3, \dots, e_8 \rangle$, and 23 bits for the mantissa $\vec{m} = \langle m_1, m_2, m_3, \dots, m_{23} \rangle$ (there is a

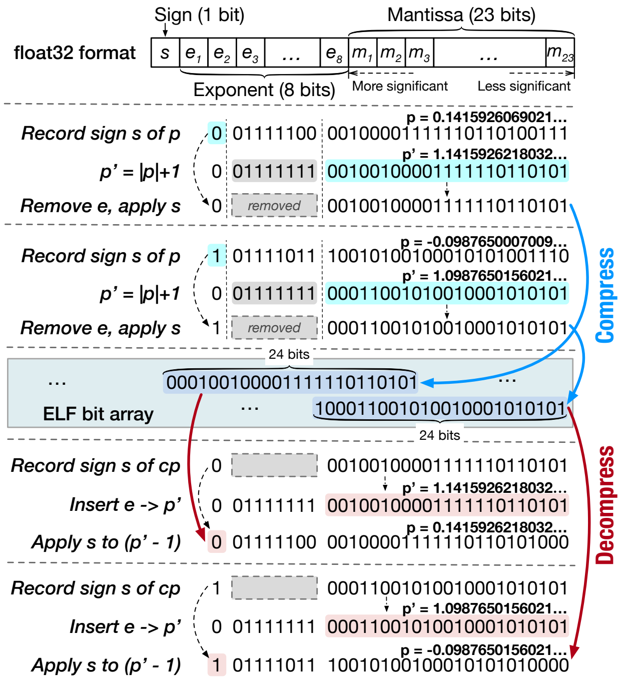

Figure 9: IEEE 754 Standard `float32` format and examples of the ELF compression process marked using blue arrows and the decompression process marked using red arrows. *Take parameter $p = 0.1415926069021\dots$ (binary machine representation) as an example, the compression process follows: transform p to the intermediate value $p' = 1.1415926218032\dots$; take the sign bit of p and the 23-bit mantissa of p' to construct the 24-bit compressed parameter cp, and finally, append it to ELF's binary bit array. The decompression process is the inverse of the compression process.*

default bit, `0b1`, which is hidden on the most significant side of the 23-bit mantissa), as shown in Figure 9 (top). p's value satisfies:

$$p = (-1)^s \times 2^{e-127} \times (1.m_1 m_2 \dots m_{23})_2$$
$$= (-1)^s \times 2^{e-127} \times (1 + \sum_{i=1}^{23} m_i \times 2^{-i}) \qquad (1)$$

where e is the decimal value of $\vec{e}^2$. Equation 1 decides that, for all p where $p \in [1, 2)$, the binary representation of p has the same exponent bits `0b01111111` with a decimal representation of 127. This is because, when s is 0 and e is `0b01111111` (which equals 127 in decimal), the value of the exponent field of $e - 127 = 127 - 127 = 0$. This means that this `float32` parameter is positive and its value is $(-1)^0 \times 1 \times (1.m_1 m_2 \dots m_{23})_2 \in [1, 2)$.

ELF is based on these two observations. The main idea of ELF is to map all floating-point parameters p where $p \in (-1, 1)$ to p' where $p' \in [1, 2)$, so that the common exponent field `0b01111111`, in case of `float32`, can be eliminated in order to save storage cost. Figure 9 illustrates ELF's compression and decompression using simple `float32` examples.

Compression. A sequential version of ELF scans model parameters linearly, and for each floating-point parameter p where $p \in (-1, 1)$, the ELF compression performs the following three steps:

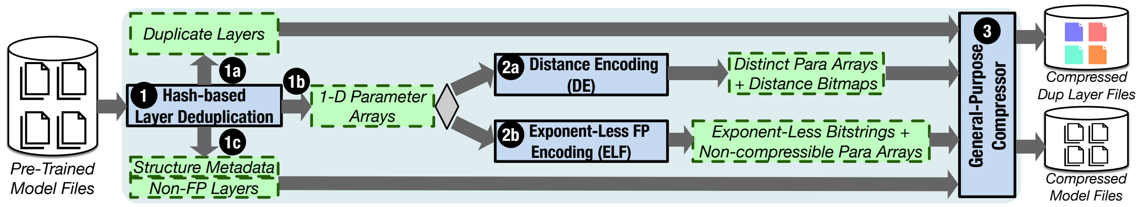
Duplicate Layers
1a
1 Hash-based Layer Deduplication
1b
1-D Parameter Arrays
1c
Structure Metadata
Non-FP Layers
Pre-Trained Model Files
2a Distance Encoding (DE)
Distinct Para Arrays + Distance Bitmaps
2b Exponent-Less FP Encoding (ELF)
Exponent-Less Bitstrings + Non-compressible Para Arrays
3 General-Purpose Compressor
Compressed Dup Layer Files
Compressed Model Files

Figure 10: The ELVES workflow. *Boxes with solid lines represent the stages of ELVES, and boxes with dashed lines denote intermediate data.*

(1) Record the sign bit s of p for later use in Step (3).
(2) Convert p where $p \in (-1, 1)$ to p' where $p' \in [1, 2)$.
(3) Remove the exponent bits $\vec{e}$ of p', concatenate the 23-bit $\vec{m}$ of p' after the recorded sign bit s, and append the 24-bit compressed parameter cp to the end of the bit array file.

Decompression. The decompression process of a sequential version of ELF reads the compressed bit array file and restores all 24-bit compressed parameters sequentially. To restore a 24-bit unit cp to p, the ELF decompression performs the following three steps:

(1) Record the first bit s (sign bit) of cp for later use in Step (3), and take the next 23 bits as the mantissa to restore an intermediate representation p'.
(2) Set the sign bit of p' with `0b0` and insert the exponent bits `0b01111111` between the sign bit and the 23-bit mantissa $\vec{m}$ of p' to restore p' so that $p' \in [1, 2)$.
(3) Construct a new intermediary p'' where $p'' = p' - 1 \in [0, 1)$, and apply the recorded sign bit s to p'' to restore the original parameter p where $p \in (-1, 1)$.

Generality and Storage Savings. ELF can be applied to all three types of floating-point values: `float32`, `float16`, and `float64`, although the storage efficiency varies depending on the data type. The storage savings of ELF come entirely from the removal of e from the binary representation of each floating-point model parameter. For `float32` data, ELF can yield a 25% ($\frac{8}{32}$) reduction in storage space. For `float16` and `float64`, the storage savings are 31.25% ($\frac{5}{16}$) and 17.19% ($\frac{11}{64}$), respectively. According to Table 3, over 99% of parameters of our PTM dataset are composed of `float32` or `float16`, making these datasets particularly well-suited for ELF's utility.

Parallelizability and Performance. ELF is easily parallelizable as it is embarrassingly parallel via *data parallelism*: a PTM dataset can be divided into chunks and each chunk can be compressed independently using a thread or a CPU core. Similarly, ELF's decompression process can be easily parallelized using data parallel as well. This property guarantees ELF's superior compression and decompression speed, which we evaluate in §7.3.

Compression Loss. Since ELF involves data transformation and encoding for all floating-point parameters that fall within $(-1, 1)$, this transforming process introduces bounded errors and the error bound varies depending on the data type. Before giving ELF's error bound, we briefly discuss the process of converting a `float32` $p \in (-1, 1)$ to $p' \in [1, 2)$ in ELF. There is no information loss when obtaining p's absolute value $|p|$. Then for $1 + |p|$, the exponent e of $|p|$ needs to be shifted right (by adding the difference of $127 - e$) so that the exponents of $|p|$ and 1 equal. This operation results in a right shift of the mantissa of $|p|$. After aligning and adding the mantissas of $|p|$ and 1, the result is normalized and rounded, and this process is where the error occurs. In other words, only the information from $|p| \in [0, 1)$ captured by the first 23 bits of its mantissa is retained in $p' \in [1, 2)$, after that the less significant bits are rounded and discarded. Therefore, the maximum error introduced by ELF for `float32` parameters is $5.96046448 \times 10^{-8}$, or 2^{-24}. Similarly, the error bound is 4.8828125×10^{-4} or 2^{-11} for `float16` and 1.110223×10^{-16} or 2^{-53} for `float64`, respectively.

Storage Overhead. ELF stores out-of-range, non-compressible parameters $p \notin (-1, 1)$ separately and uses an external table to keep track of the positions of these parameters. This extra storage overhead might outweigh the storage reduction obtained from ELF's transformation and encoding, especially if the percentage of these out-of-range parameters is considerable. We address this problem using a hybrid approach, ELVES, which will be described in §6.2.

6.2 The ELVES Compression Framework

We present the design of our offline compression framework ELVES. ELVES incorporates the insights of ELF and a series of data reduction methods that we have explored in §5. Potential use case of ELVES is to run ELVES as a background process to scan all the PTM datasets that have already been written to storage [41, 58] and select the most effective methods for data reduction.

Figure 10 depicts the stages of ELVES. In Stage 1, ELVES performs a scan over the entire PTM file dataset to compute the fingerprint of each model layer (§5.1). The fingerprint is computed based on the content of the layer by using a cryptographic hash function and is stored in a table that maps the layer ID to its fingerprint. When a new layer is encountered, its fingerprint is computed and compared with the fingerprints of existing layers in the table. If a matching fingerprint is found in the table, ELVES detects that this layer is a duplicate, and instead of storing the layer again, a reference to the existing layer (stored as a separate layer file) is recorded. If no match is found, the new layer is unique, and it is stored along with its fingerprint. By the end of this stage, ELVES stores all duplicate layers exactly once (intermediate output 1a in Figure 10) and continues to compress the rest of the non-duplicate layers (intermediate output 1b) in next stages.

Non-duplicate layers are flattened to 1-dimensional (1-D) arrays of parameters of different floating-point types, which are streamed to our DE (distance-encoding) compressor (§5.3) in Stage 2a and ELF for exponent-less floating-point encoding (§6.1) in Stage 2b for parameter-level compression. ELVES applies both DE and ELF to intermediate data 1b and chooses the compressor with better effect. The rationale is that, for models that have a substantial proportion of parameters $p \notin (-1, 1)$, ELF might not be beneficial as it needs to keep track of these non-compressible parameters, which introduces extra storage overhead. Thus, for these models, ELVES opts to use DE over ELF, or the other way around, depending on which is more effective. ELVES then deletes the intermediate files generated by the less effective compressor. In Stage ❸, the outputs of DE or ELF, together with the duplicate layers (the intermediate output 1a), model structure metadata files, and non-floating-point model layers (the intermediate output 1c) are further compressed by a general-purpose lossless compressor Zstandard (zstd) [10].

The decompression process for ELVES is the inverse of the compression process. (1) Compressed files are decompressed by zstd to restore model structure metadata, non-floating-point layer files, duplicate layers, and DE-compressed / ELF-compressed intermediate files. (2) DE-compressed / ELF-compressed intermediate files are decompressed by ELVES to obtain the 1-D parameter arrays. (3) Layers of the original models are recovered based on the model structure metadata. For each layer, there are three possibilities: (*i*) a duplicate layer will be retrieved from the corresponding decompressed duplicate layer file referenced by its fingerprint; (*ii*) a floating-point layer will be restored from the 1-D array, based on the model structure specified by the model structure metadata; (*iii*) a non-floating-point layer will be restored from the non-floating-point layer file. Upon completing these steps, the model dataset is decompressed.

7 EVALUATION

Setup and Dataset. We performed all of our tests on a server with 56 Intel(R) Xeon(R) Gold 6330 CPU cores and 256 GB memory running Ubuntu 20.04 with a kernel version of 5.4.0. Our evaluation is focused on our sampled dataset of 900 real-world pre-trained ML models (§3) collected from Hugging Face. The total size of the binary format of the models that we tested is 575.88 GB.

Baselines and Configurations. We selected a total of 11 representative compressors divided into four categories.

- General-purpose lossless compressors: Gzip [1] and zstd [10]. Both are based on LZ77 [68]. They operate at the binary byte level and look for repetitive patterns among the bytes. We tested zstd's compression levels from 3 (default) to 19 (highest compression ratio). We found that all configurations led to the same compression ratio, but level 19 had extremely slow compression speed. Thus, we chose to use a compression level of 3 for zstd.
- Time series and data lake compressors: Sprintz [14], Buff [38], Chimp [33] and Gorilla [47] for TS datasets, and BTRBLOCKS [32] for data lakes. Sprintz uses a lookup table to predict values based on preceding entries and encodes the delta between predicted and original values. Buff divides the sign, exponent, and mantissa, and tailors the storage scheme based on the specific bounds and precision requirements of the dataset. Chimp and Gorilla exploit the TS predictability, XOR successive values with previous ones, and compress away redundant zeros. BTRBLOCKS uses Pseudodecimal Encoding to convert `float64` into two integers, significant digits with the sign and the exponent, to save storage. We used the default settings that their GitHub repositories specified for Sprintz, Chimp, and Gorilla, and matched the delta precision of Buff with the error bound of ELF. For BTRBLOCKS we used the single-column configuration given by its examples.
- Error-bounded, lossy compressors for floating-point, scientific datasets: SZ3 [65] and zfp [8]. SZ3 is a modular, error-bounded lossy compression framework for scientific datasets. SZ3 uses Lorenzo predictor [29] and regression predictor [36] to predict next parameters. SZ3 relies on the quantizer [66] to enable error control for prediction. zfp is designed for multidimensional numerical datasets. zfp uses transform to reduce the dynamic range of the floating-point data and then quantizes the transformed data. Since integer-typed and boolean-typed layers in PTMs might be involved in mission-critical functionality, such as input layers, where no information loss is allowed, we tested SZ3 and zfp on floating-point layers of all models only. The error bound for SZ3 and zfp was set to match that of ELF.
- Model pruning and quantization methods: global magnitude pruning (Global MP) [24], and Gaussian and outliers uniform quantization (GOUQ) based on GOBO [62]. Global MP prunes parameters based on their magnitude while preserving the original model structure. Its parameter error can be controlled by a threshold. GOUQ is modified from GOBO, a cutting-edge quantization method for NLP inference. GOUQ categorizes parameters into a "Gaussian" (G) and an "Outliers" (O) group. Parameters in the O group retain their original format (e.g., `float32`), while parameters of the G group are quantized to a small set of representative values for space saving. GOUQ reduces the errors introduced by GOBO by increasing the number of representatives in the G group, which introduces a tradeoff in storage saving and model accuracy. We set the error bound of these two methods to align with the maximum error potentially introduced by ELF.

Goals. Our evaluation aims to answer the following questions:

- How does ELVES compare to other baseline compressors in terms of compression ratio (§7.1)?
- How does each stage of ELVES contribute to storage saving (§7.2)?
- What is the compression and decompression speed of ELF (§7.3)?
- What is the impact of the lossiness of error-bounded ELF on the model accuracy (§7.4)?

7.1 Comparison with Baselines

First, we compare the 11 baseline methods with ELVES. Figure 11 and 12 show the compression ratio (CR)[2] statistics.

General-Purpose Compressors. As Figure 11 shows, Gzip and zstd perform almost the same, with an overall CR of 1.16×. zstd outperforms Gzip slightly on maximum CR (see Figure 12). Both of them are able to reduce the model file sizes for almost all models, with around 72% of models having file sizes reduced by 10% (a CR of 1.1×). The modest data reduction is due to that general-purpose compressors like Gzip and zstd are not specifically optimized for compressing floating-point datasets. However, it is noteworthy that for about 11% of models, Gzip and zstd still achieve a good CR of 1.5×. This is because these models have nearly 100% parameter-level duplication, a pattern that can be exploited by dictionary coding.

State-of-the-Art, Encoding-based, Floating-Point Compressors. TS data compressors perform poorly on our PTM dataset. Specifically, Sprintz, Chimp, and Gorilla yield a CR of *less than one*—indicating an increase in file size after compression—for 98.54%, 74.77%, and 75.23% of models, respectively. Approximately 82% of models can save an average of 5% storage space with Buff, but poor CR is achieved for about 11% of the models with a minimum CR of

[2]CR is defined as the ratio between the size of the original, uncompressed dataset and the size of the compressed dataset. The higher, the better.

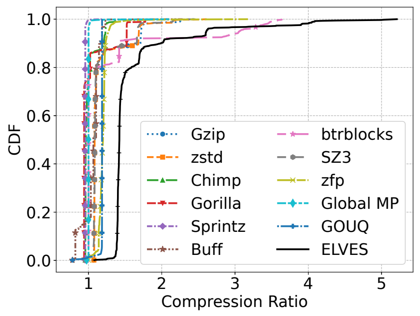

Figure 11: Compression ratio comparison of different compressors for the 600-model dataset. *Each data point in a curve is the CR of a model.*

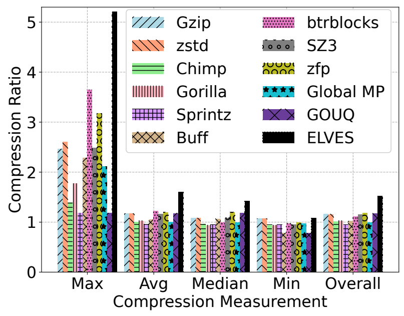

Figure 12: Compression ratio breakdown. *Overall CR: the aggregate size of the original, uncompressed dataset divided by that of the compressed dataset.*

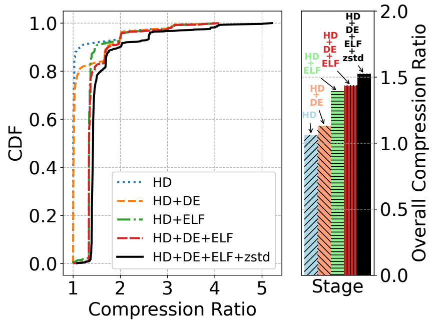

Figure 13: Compression ratio CDF (left) and the cumulative fraction of overall compression ratio (right) for different ELVES setups.

0.78×. The ineffectiveness of these TS compressors stems from significant differences in data characteristics. In TS data, two notable characteristics are often observed: (1) the precision of the data may be relatively low, and (2) there is typically a predictable trend or correlation among adjacent values. In PTM datasets, however, these features no longer exist. Floating-point parameters come with high precision and adjacent parameters are randomly different. In terms of the overall CR, Sprintz, Buff, Chimp, and Gorilla achieve 0.96×, 1.03×, 1.02×, and 1.03× for the entire dataset, respectively.

The columnar compressor, BTRBLOCKS, yields an overall CR of 1.12. Specifically, it exhibits a CR of precisely 1.0 for 74.67% of the models, indicating its ineffectiveness for PTM datasets. Furthermore, BTRBLOCKS attains a CR of 1.1 or less for 80.11% of the models; and a mere 18.56% of PTMs achieving a CR of 1.3 or higher, with the maximum CR at 3.65×. The reason is that the Pseudodecimal Encoding algorithm is more effective on the floating points with fixed, low precision—take 3.25 for example, BTRBLOCKS encodes 3.25 as (+325, 2), where 325×10^{-2}. However, most if not all parameters of PTMs exhibit a higher degree of precision (e.g., 1.0389173), rendering the encoding scheme of BTRBLOCKS ineffective.

State-of-the-Art, Error-Bound, Lossy Compressors. SZ3 attains a CR of 1.1× for 76.78% of models, which is slightly higher than that of Gzip and zstd, since SZ3 is a prediction-based compressor that is specifically designed and optimized for floating-point-based scientific datasets that exhibit relatively "smooth" data patterns. But for PTMs that contain mostly trend-cluttered parameters, SZ3 requires a larger overhead to store points that fall outside the range that can be encoded through quantization and predictions. zfp generally performs better than the other baselines, with a CR ranging from 1.1× to 1.3× for 93.78% of models. But it produces a CR greater than 1.3× on only 3% of the models. zfp achieves an overall CR of 1.18× for the entire dataset, which is marginally elevated than that of zstd due to a lack of parameter correlations in model datasets.

Model Pruning and Quantization Methods. The storage savings achieved by Global MP on the PTM dataset are negligible. A significant majority of PTMs, about 98.11%, exhibit a CR between 1.0× and 1.05×, with an overall CR of 1.02×. To ensure the model integrity, Global MP preserves the original model structure by exploiting a bit string consisting of 0 and 1 indicating whether a parameter is removed or retained, which introduces a storage overhead. Furthermore, to prevent significant accuracy loss from pruning, even without re-training or fine-tuning, the magnitude threshold must be kept relatively low, which means only a minor portion of the model parameters are eligible for pruning, leading to modest compression results. The quantization method, GOUQ, slightly outperforms Global MP, achieving a CR ranging from 1.15× to 1.19× for 97.51% of the models, with an overall CR of 1.18×. However, 1.22% of the models exhibit a CR below 1, attributed to a considerable proportion of parameters being classified as outliers; and the overhead caused by the bit table distinguishing parameters as either Gaussian or outliers, surpasses the storage savings achieved from parameter mapping to representative values. Similarly, to ensure the accuracy of quantized models, parameter errors introduced during the quantization are strictly controlled.

Comparing with ELVES. Figure 11 shows a big margin between ELVES and the baselines, because ELVES' hybrid approach is designed based on the data characteristics of PTM datasets. Specifically, ELVES achieves 28.81%, 47.57%, and 28.81% higher overall CR than zfp (the best in the error-bounded, lossy compressor set), Gorilla (the best in the TS data compressor set) and GOUQ (the best in the model quantization and pruning set), respectively, as shown in Figure 12.

7.2 Evaluating ELVES Stages

One way to understand the differences among hash-based layer deduplication (HD), distance-encoding (DE) compression, and ELF is to view the effectiveness of ELVES' individual stages in terms of CR. The CR distribution and cumulative improvement of overall CR are depicted in Figure 13.

HD (Hash-based Deduplication). In this test we only enabled Stage ❶ of ELVES (Figure 10). While for most models the storage savings from HD are quite limited due to a low duplication ratio and small sizes of duplicate layers, around 5.33% of models exhibit a substantial level (CR $\geq$ 2) of data reduction, with a CR of up to 5.21×. We anticipate that as the number of PTMs in real-world production environments continues to increase (Figure 1), the proportion of duplicate layers among models will also grow. This, in turn, will likely amplify the effectiveness of HD.

HD + DE (Distance-Encoding). In this test we enabled Stage ❶ and 2a of ELVES. Figure 13 (left) shows that DE results in a CR of 1.5× and more for about 11% models due to the fact that these models have close to 100% of their parameters duplicated. Enabling DE atop HD improves the overall CR for all models by 6.55%.

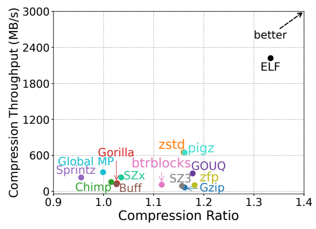

Figure 14: CR vs. compression throughput.

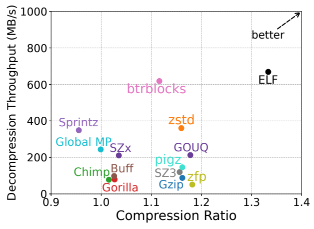

Figure 15: CR vs. decompression throughput.

HD + Elf (Exponent-Less Floating-Point Encoding). Then, we enabled Stage ❶ HD and 2b Elf. Elf is generally effective for floating-point models. Elf has a significant CR boost: *for over* 87% *of models where HD yields close-to-zero CRs, enabling Elf achieves an average CR of* 1.35×, *and improves the overall CR by* 31.32%.

HD + DE + Elf. Next, we combined Stage ❶, 2a and 2b together. The compression strategies used by DE and Elf capitalize on different data patterns, rendering them complementary for some PTMs. As shown in the left subfigure of Figure 13, Elf guarantees an effective CR for a broad range of PTMs, while DE achieves better CRs on a small set of selected models. Integrating the three approaches yields enhancements in compression efficiency, outperforming HD+DE and HD+Elf by 27.06% and 3.11%, respectively.

HD + DE + Elf + zstd (Zstandard). Finally, by enabling Stage ❸ zstd to compress all intermediate data generated by previous stages, we observe an additional CR improvement of 8.69%.

In summary, our ablation test demonstrates that: (1) Elf has the greatest impact on reducing dataset sizes compared to other techniques, accounting for 65.46% of the end-to-end, overall CR improvement enabled by all stages. (2) Elves' hybrid design is aware of the diverse data patterns of PTMs. Elves effectively tailors the best compression method when compressing models with different patterns, yielding an overall CR of 1.52× for the whole dataset. Moreover, 99% of models (891 of 900) see a CR of over 1.35×, and 24.44% of them achieve a CR ≥ 1.5× with the highest CR of 5.21×.

7.3 Evaluating Elf Performance

We have implemented the core compression and decompression algorithm of Elf using C++ and pthread. Next, we compare the compression and decompression speed of Elf with 13 baselines. To ensure the best throughput performance, we enabled multi-threading configuration for those baselines with data parallelism support: zstd, pigz (parallel Gzip) [5], BtrBlocks, and SZx [61].

Figure 14 and 15 show the results. The throughput was calculated using the ratio of the aggregate size of the whole dataset and the total compression (or decompression) time. Note that, since we target storage compression and decompression, our throughput metric includes the I/O time each method took for reading files from the disk and writing files to the disk. We used a 1.6TB Intel Optane DC P5800X SSD, which provides a sequential read (write) throughput of 4.2 GB/s (938 MB/s). *This is to ensure that the compression and decompression processes are not bottlenecked by the disk I/O.* From Figure 14 and 15 we can see that Elf significantly outperforms all other baselines in compression throughput, and also has the fastest decompression speed among all 14 compressors. By using all the available 56 CPUs, Elf achieves an average compression (decompression) throughput of 2,170.56 MB/s (653.58 MB/s), respectively. zstd with multi-thread setting using all 56 CPUs is still 3.6× (1.89×) slower in compression (decompression) speed, compared to Elf. While BtrBlocks exhibits a comparable decompression throughput with Elf, Elf achieves a compression throughput that is 35.29× greater than BtrBlocks's. With a single thread, Elf achieves a compression and decompression throughput of 121.68 MB/s and 135.02 MB/s, respectively, still competitive compared to baselines such as Gzip, Buff, zfp, Chimp, and Gorrila. The strong performance results indicate that Elf can serve as a practically useful tool for PTM storage compression.

Table 7: Accuracy degradation of different model task categories. *% in Column 2 represent the proportion of all tested models.*

Model Task (Category)	Count (%)	Accuracy Degradation
Image Classification (CV)	69 (23.00%)	0.87%
Text Generation (NLP)	68 (22.67%)	0%
Text Classification (NLP)	60 (20.00%)	0%
Token Classification (NLP)	30 (10.00%)	0%
Translation (NLP)	25 (8.33%)	0.4%
Question Answering (NLP)	24 (8.00%)	0%
Audio Classification (Audio)	9 (3.00%)	0%
Summarization (NLP)	9 (3.00%)	1.11%
Speech Recognition (Audio)	6 (2.00%)	0%
Overall	**300 (100%)**	**0.27%**

7.4 Quantifying Impact on Model Accuracy

This section evaluates the impact of Elf on model accuracy using two methods: fuzz-testing-inspired validation (§7.4.1) and benchmark validation (§7.4.2).

7.4.1 Fuzz-Testing-Inspired Validation. Elf introduces bounded errors for floating-point parameters during compression. We evaluated the impact of Elf on model accuracy using a dataset comprising 300 out of 900 models from 9 different tasks, ranging from image classification to text generation, across three model categories.

We quantify the impact of compression on model accuracy using a metric called *accuracy degradation* (AD) for a given task and dataset, which is defined as follows:

$$AD = \frac{\sum_{i=1}^{N} \Delta A_i}{N} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{M} compare(O_{i,j}, O'_{i,j})}{N \cdot M} \quad (2)$$

where ΔA_i represents the difference between the original model and the decompressed model for the same task and under the same dataset. N is the total number of models tested this time. O is the output of the original model and $O^{'}$ is the output of the model after decompression. Each model, original or decompressed, generates an output tensor that consists of M float numbers. We use M to define the size of the output and use a function *compare*() to compare the outputs of two models under a defined output precision. In the case of the fuzz-testing validation, we compare each bit of two floats. Here is how we make the comparison: for a tensor with an output length of 1,000, we compare the float numbers at the corresponding positions of the two tensors. If the floats are consistent within all digits, we consider the two model outputs to be the same, and *compare*() outputs 0; otherwise, it outputs 1. If 999 out of 1,000 floats are the same, then we obtain an AD of 0.1%.

Table 8: Accuracy degradation (AD) of all loss compression frameworks with the benchmark datasets. *Benchmark validation has been evaluated on a total of 46 models across 9 tasks in 4 domains. The error bound of* Elves *is denoted with e. All tested baselines and their abbreviations are as follows: SZ3, zfp, Global MP (mp), Global MP with 2× error bound (mp2e), GOUQ (gouq), GOUQ with 2× error bound (gouq2e), and half-precision quantization (half). The overall AD is averaged across all tasks for a particular compression method, while the overall CR is calculated by dividing the combined size of the original models by the total size of the compressed models.*

Domain	Task(# of tested model)	Dataset	Accuracy Degradation							
			Elves	SZ3	zfp	mp	mp2e	gouq	gouq2e	half
CV	image classification(4)	mini_imagenet	0.2%	0.3%	0.2%	0.1%	0.2%	0.4%	1.1%	65.0%
		cifar100	0.2%	0.3%	0.1%	0.2%	0.2%	0.4%	1.2%	48.4%
	object detection(4)	detection-datasets/coco	0.1%	0.2%	0.2%	0.1%	0.2%	0.2%	0.2%	1.6%
		cppe-5	0.2%	0.3%	0.2%	0.2%	0.3%	0.2%	0.3%	2.6%
	image segmentation(6)	scene_parse_150	0.2%	0.6%	0.4%	0.1%	0.2%	0.2%	0.8%	38.6%
		sidewalk-semantic	0.3%	1.4%	0.5%	0.2%	0.3%	0.2%	0.7%	35.1%
Multimodal	feature extraction(7)	Open-Orca/OpenOrca	0.1%	0.2%	0.1%	0.1%	0.1%	0.2%	0.3%	18.1%
		imdb-movie-reviews	0.1%	0.1%	0.1%	0.1%	0.1%	0.2%	0.5%	24.5%
	image-to-text(4)	conceptual_captions	0%	0%	0%	0%	0%	0%	0%	0%
		red_caps	0%	0%	0%	0%	0%	0%	0%	0%
Audio	speech recognition(5)	librispeech_asr_dummy	0%	0%	0%	0%	0%	0%	0%	0%
		lj_speech	0%	0%	0%	0%	0%	0%	0%	0%
NLP	sentiment classification(7)	glue-sst2	0%	0%	0%	0%	0%	0%	0%	0%
		imdb	0%	0%	0%	0%	0%	0%	0%	0%
	sentence similarity(5)	glue-stsb	0%	0%	0%	0%	0.1%	0.1%	0.2%	3.6%
		paws-x	0%	0%	0%	0%	0.1%	0.1%	0.2%	4.2%
	Fill-mask(4)	wikitext	0%	0%	0%	0%	0.1%	0.1%	0.1%	0.1%
		ptb_text_only	0%	0%	0%	0%	0.1%	0.1%	0.1%	0.1%
Overall AD (Overall CR)			**0.07% (1.52)**	**0.18% (1.16)**	**0.1% (1.18)**	**0.06% (1.00)**	**0.22% (1.01)**	**0.13% (1.18)**	**0.32% (1.20)**	**13.44% (1.99)**

Testing Methodology. Our methodology for evaluating the model accuracy degradation is inspired by *fuzz testing* [22, 42, 43]. In fuzz testing, random inputs are generated and fed to a program to verify the correctness. Here, we generate random inputs for the original and decompressed models and compare the outputs from them.

In the text generation task, the model behaves like a chatbot, producing outputs based on the training data for any given prompt. Due to the inherent randomness in the search algorithms, these models are intended to generate non-repeated text for the same inputs. The non-deterministic nature of the outputs makes it challenging to evaluate the accuracy degradation. Thus, we modified the $compare()$ to calculate the cosine similarity of the embeddings and attention weights generated by the model. If the cosine similarity of the embeddings is *exactly* 1, it produces 0 else 1. Cosine similarity for output $\{\mathbf{O_i}, \mathbf{O_j}\}$ from two models is defined as:

$$C_{sim}(O_i, O_j) = \frac{\mathbf{O_i} \cdot \mathbf{O_j}}{\|\mathbf{O_i}\| \|\mathbf{O_j}\|} \tag{3}$$

To fuzz test classification models, we generated images and text based on random noise and content, fed them into the networks, and compared the top k labels predicted by the original and decompressed models. We here define the $compare()$ function as let Y^k be the set of the top k labels predicted by model m, so that we compare the output of these models and produce 0 if and only if $Y^k_m - Y^k_{m'} = \emptyset$ where $Y^k_m, Y^k_{m'}$ are top k labels output by the original model m and decompressed model m'.

Validation Results. Even with the rigorous definition of model accuracy comparison, we were able to achieve an AD of 0% (0% indicates that every decompressed model for the given task generates the *exact same output* as its original version) for 6 out of the 9 tasks, and < 1.2% for remaining tasks, leading to an overall accuracy degradation of 0.27% (Table 7).

7.4.2 Benchmark Validation. While our large-scale fuzz-testing-based accuracy validation in §7.4.1 covers a broad set of 300 models, standard benchmark datasets offer a more comprehensive way of validating the model accuracy. We conducted benchmark tests on a total of 9 tasks across 4 domains. Table 8 shows the results. We see that Elves is the only method that achieves both an overall accuracy degradation close to zero (0.07%) and a high CR. This result demonstrates that Elves has negligible influence on the performance of models and all outputs generated using decompressed models are almost identical to those generated by the original ones.

8 CONCLUSION

This paper dissects the data characteristics of real-world pre-trained ML model datasets and studies their compressibility across various dimensions. Our analysis considers different representative data reduction and compression techniques and spans three data granularities: model layers, model chunks, and model parameters. Our thorough analysis reveals that PTM dataset compression is notably challenging, with existing data reduction and compression methods generally ineffective for reducing the storage size of PTM datasets. Based on the observations, we have proposed Elf, a simple and effective, near-lossless floating-point compression algorithm and developed Elves, a compression framework that integrates Elf and several other techniques. Elves achieves an overall compression ratio of 1.52×, which is up to 1.32× higher than state-of-the-art lossy floating-point compressors, while introducing near-zero model accuracy loss. We hope that our study will provide valuable insights into the design, implementation, and optimization of data reduction techniques and systems for efficient storage of PTM datasets.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable feedback and comments. This work was sponsored in part by NSF grants: CCF-2318628, CCF-1919113, CNS-2322860, OAC-2106446, and CMMI-2134689.

REFERENCES

[1] [n.d.]. gzip. https://www.gzip.org/.

[2] [n.d.]. How Much Energy Do Data Centers Really Use? . https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/.

[3] [n.d.]. Hugging Face: The AI community building the future. https://huggingface.co/.

[4] [n.d.]. Introducing LLaMA: A foundational, 65-billion-parameter large language model. https://ai.meta.com/blog/large-language-model-llama-meta-ai/.

[5] [n.d.]. pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines. https://zlib.net/pigz/.

[6] [n.d.]. Snappy, a fast compressor/decompressor. https://github.com/google/snappy.

[7] [n.d.]. TensorFlow Hub. https://www.tensorflow.org/hub.

[8] [n.d.]. zfp. https://computing.llnl.gov/projects/zfp.

[9] [n.d.]. zip. https://www.iana.org/assignments/media-types/application/zip.

[10] [n.d.]. Zstandard. https://facebook.github.io/zstd/.

[11] 2012. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 12)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/Shilane

[12] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.

[13] Guangji Bai, Zheng Chai, Chen Ling, Shiyu Wang, Jiaying Lu, Nan Zhang, Tingwei Shi, Ziyang Yu, Mengdan Zhu, Yifei Zhang, Carl Yang, Yue Cheng, and Liang Zhao. 2024. Beyond Efficiency: A Systematic Survey of Resource-Efficient Large Language Models. arXiv:2401.00625 [cs.LG]

[14] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[15] A.Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[16] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *2007 Data Compression Conference (DCC'07)*. 293–302. https://doi.org/10.1109/DCC.2007.44

[17] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

[18] Xuepeng Chang, Huihui Pan, Weiyang Lin, and Huijun Gao. 2021. A mixed-pruning based framework for embedded convolutional neural network acceleration. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 4 (2021), 1706–1715.

[19] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *International conference on machine learning*. PMLR, 2285–2294.

[20] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure within Convolutional Networks for Efficient Evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 1269–1277.

[21] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[22] Justin Forrester and Barton Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *4th USENIX Windows Systems Symposium (4th USENIX Windows Systems Symposium)*. USENIX Association, Seattle, WA.

[23] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. 2015. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 331–344. https://www.usenix.org/conference/fast15/technical-sessions/presentation/fu

[24] Manas Gupta, Efe Camci, Vishandi Rudy Keneta, Abhishek Vaidyanathan, Ritwik Kanodia, Chuan-Sheng Foo, Wu Min, and Lin Jie. 2022. Is complexity required for neural network pruning? a case study on global magnitude pruning. *arXiv preprint arXiv:2209.14624* (2022).

[25] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[26] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[27] Benjamin Hawks, Javier Duarte, Nicholas J Fraser, Alessandro Pappalardo, Nhan Tran, and Yaman Umuroglu. 2021. Ps and qs: Quantization-aware pruning for efficient low latency neural network inference. *Frontiers in Artificial Intelligence* 4 (2021), 676564.

[28] Tianxing He, Yuchen Fan, Yanmin Qian, Tian Tan, and Kai Yu. 2014. Reshaping deep neural network for fast decoding by node-pruning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 245–249.

[29] L Ibarria, P Lindstrom, J Rossignac, and A Szymczak. 2003. Out-of-core Compression and Decompression of Large n-dimensional Scalar Fields. 22, 3 (2 2003). https://doi.org/10.1111/1467-8659.00681

[30] William Kahan. 1996. IEEE Standard 754 for Binary Floating-Point Arithmetic. *Lecture Notes on the Status of IEEE 754* (1996).

[31] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv:1806.08342 [cs.LG]

[32] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[33] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3058–3070. https://doi.org/10.14778/3551793.3551852

[34] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[35] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[36] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. 438–447. https://doi.org/10.1109/BigData.2018.8622520

[37] Zhu Liao, Victor Quétu, Van-Tam Nguyen, and Enzo Tartaglione. 2023. Can Unstructured Pruning Reduce the Depth in Deep Neural Networks?. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1402–1406.

[38] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2586–2598.

[39] Zhenhua Liu, Yunhe Wang, Kai Han, Siwei Ma, and Wen Gao. 2022. Instance-aware dynamic neural network quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12434–12443.

[40] Dirk Meister and André Brinkmann. 2009. Multi-Level Comparison of Data Deduplication in a Backup Scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (Haifa, Israel) *(SYSTOR '09)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/1534530.1534541

[41] Dutch T. Meyer and William J. Bolosky. 2011. A Study of Practical Deduplication. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast11/study-practical-deduplication

[42] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[43] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[44] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. 2021. Loss aware post-training quantization. *Machine Learning* 110, 11-12 (2021), 3245–3262.

[45] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

[46] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 580–595.

[47] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[48] University of Massachusetts Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. 2004. Redundancy Elimination Within Large Collections of Files. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2004-usenix-annual-technical-conference/redundancy-elimination-within-large-collections

[49] Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival data storage. In *Conference on file and storage technologies (FAST 02)*.

[50] Michael O. Rabin. 1981. Fingerprinting by random polynomials. Note: Harvard Aiken Computational Laboratory TR-15-81.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[52] James A. Storer and Thomas G. Szymanski. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (oct 1982), 928–951. https://doi.org/10.1145/322344.322346

[53] Zhaoyuan Su, Sheng Di, Ali Murat Gok, Yue Cheng, and Franck Cappello. 2022. Understanding Impact of Lossy Compression on Derivative-related Metrics in Scientific Datasets. In *2022 IEEE/ACM 8th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD)*. 44–53. https://doi.org/10.1109/DRBSD56682.2022.00011

[54] Saeed Vahidian, Mahdi Morafah, and Bill Lin. 2021. Personalized federated learning by structured and unstructured pruning under data heterogeneity. In *2021 IEEE 41st international conference on distributed computing systems workshops (ICDCSW)*. IEEE, 27–34.

[55] Deepak Vohra and Deepak Vohra. 2016. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (2016), 325–335.

[56] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2019. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732* (2019).

[57] BigScience Workshop and Scao et al. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs.CL]

[58] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710. https://doi.org/10.1109/JPROC.2016.2571298

[59] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. {FastCDC}: A fast and efficient {Content-Defined} chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 101–114.

[60] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. 2017. Online Deduplication for Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1355–1368. https://doi.org/10.1145/3035918.3035938

[61] Xiaodong Yu, Sheng Di, Kai Zhao, Jiannan Tian, Dingwen Tao, Xin Liang, and Franck Cappello. 2022. Ultrafast Error-Bounded Lossy Compression for Scientific Datasets. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 159–171. https://doi.org/10.1145/3502181.3531473

[62] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 811–824.

[63] Shuyu Zhang, Donglei Wu, Haoyu Jin, Xiangyu Zou, Wen Xia, and Xiaojia Huang. 2021. QD-Compressor: a Quantization-based Delta Compression Framework for Deep Neural Networks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 542–550.

[64] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-Grained Feature Locality based Fast Resemblance Detection for Post-Deduplication Delta Compression. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 121–128. https://www.usenix.org/conference/fast19/presentation/zhang

[65] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[66] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2020. Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/3369583.3392688

[67] Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-08/avoiding-disk-bottleneck-data-domain-deduplication-file-system

[68] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714