UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

LOSSLESS MODEL COMPRESSION

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MARTÍN EDUARDO BRAVO DÍAZ

PROFESOR GUÍA:
ANDRÉS ABELIUK
PROFESOR GUÍA 2:
GONZALO NAVARRO

MIEMBROS DE LA COMISIÓN:
CAMILO GÓMEZ
ARTURO MERINO

Este trabajo ha sido parcialmente financiado por el Centro Nacional de Inteligencia Artificial (CENIA), financiado por la Agencia Nacional de Investigación y Desarrollo (ANID), proyecto FB210017.

SANTIAGO DE CHILE
2026

Resumen

Los modelos fundacionales, tales como los grandes modelos de lenguaje, los modelos generativos de imágenes y los clasificadores de visión, han aumentado considerablemente tanto en capacidad como en requerimientos computacionales. Debido a ello, reducir el costo de almacenar y transferir sus pesos se ha vuelto un problema importante. Los esquemas con pérdida, como la cuantización y la destilación, pueden producir representaciones mucho más pequeñas, pero modifican el modelo y pueden afectar su calidad. La compresión sin pérdida aborda un objetivo distinto: reducir el espacio utilizado y reconstruir exactamente los bits originales de cada tensor.

Un compresor relevante en este contexto es ZipNN, que aprovecha la estructura de las representaciones de punto flotante. En particular, observa que el campo de los exponentes suele concentrarse en pocos valores y, por lo tanto, puede codificarse eficientemente mediante códigos de Huffman. Sin embargo, ignora posibles relaciones que pueda haber entre los tensores.

En este trabajo presentamos NeuralZip, un compresor para pesos de modelos fundacionales inspirado desde una perspectiva de Estructuras de Datos Compactas. El método aprovecha dos propiedades observadas en los pesos: *heterogeneidad por bloques* y *redundancia local de los exponentes*. NeuralZip agrupa tensores con distribuciones de exponentes similares, reutiliza códigos de Huffman entre ellos y selecciona representaciones de exponentes empaquetados cuando su costo estimado justifica el tiempo adicional. Todos los modos pueden almacenar los datos sin codificar cuando la compresión no es conveniente, y la reconstrucción es exacta bit a bit.

La evaluación compara NeuralZip con ZipNN en modelos de lenguaje, visión, difusión y *mixture-of-experts*, además de escenarios con versiones intermedias de entrenamiento y comunicación federada. En almacenamiento directo, NeuralZip codifica más rápido en 26 de 34 artefactos de modelo, desde casi la paridad hasta 21.51×; en los ocho restantes es 1.01×–1.58× más lento. El ratio del campo de exponentes varía entre una reducción de 0.984 puntos porcentuales y un aumento de 0.224 puntos, lo que confirma que el beneficio depende de la arquitectura. Todos los archivos de punto flotante reconstruyen exactamente los bits originales.

Abstract

Foundation models, including large language models, image-generation models, and vision classifiers, have grown substantially in both capability and computational requirements. Reducing the cost of storing and transferring their weights has therefore become an important problem. Lossy techniques such as quantization and distillation can produce much smaller representations, but they modify the model and may affect its quality. Lossless compression addresses a different objective: reducing storage while reconstructing the original bits of every tensor exactly.

A relevant compressor in this setting is ZipNN, which exploits the structure of floating-point representations. In particular, values in the exponent field are often concentrated in a small set and can therefore be encoded efficiently with Huffman codes. However, it ignores possible correlations that may exist between the tensors.

In this thesis, we present NeuralZip, a compressor for foundation-model weights inspired by the perspective of Compact Data Structures. The method exploits two properties observed in model weights: *blockwise heterogeneity* and the *local exponent redundancy*. NeuralZip clusters tensors with similar exponent distributions, reuses Huffman codes within each cluster, and selects packed-exponent representations when their compression benefit justifies their measured encoding cost. Shared and packed modes fall back to local Huffman and then raw storage, preserving bitwise-exact reconstruction.

The evaluation compares NeuralZip with ZipNN across language, vision, diffusion, and mixture-of-experts models, as well as training-state and federated-communication settings. In direct model storage, NeuralZip encodes faster in 26 of 34 model artifacts, ranging from near parity to 21.51×, and is 1.01×–1.58× slower in the other eight. The exponent-field ratio ranges from a 0.984-percentage-point reduction to a 0.224-point penalty, confirming that the benefit depends on the architecture. Every floating-point archive reconstructs the original bits exactly.

A mis padres, por su apoyo incondicional y
por enseñarme el valor del esfuerzo y la perseverancia.

Agradecimientos

Me siento muy agradecido de toda la gente que me ha acompañado durante este largo, pero también bonito, camino que ha sido el paso por la universidad.

Primero, a mi familia, que es mi gran apoyo. A mis papás, Alisson y Luis, que siempre han estado para mí. A mis hermanos, Lucas y Santiago, que siempre me han acompañado. A mi primo Joaco, por siempre estar ahí y animarme a hacer deporte. A mis abuelas, tíos y primos, que siempre me han acogido y querido mucho.

Agradezco también a los amigos que estaban desde antes: Eduardo, Martín y Tomás. También a Antonia, que me acompañó durante esta gran etapa.

A mis amigos de industrial, Sebastián y Maximiliano, quienes me han acompañado desde plan común. También agradezco a Edgar, Felipe, Franco, Jean Paul y Roberto; gracias a ellos, mi paso por el Departamento de Computación se volvió una experiencia grata y divertida.

Agradezco a los amigos que hice durante mi tiempo en el extranjero, Jonas y Mateus, con quienes, incluso a miles de kilómetros de distancia, no pierdo el contacto.

Un agradecimiento especial a Iván Vidal, a quien conozco desde antes de la universidad. Durante estos últimos cinco años, siempre fue el mejor compañero de lecturas, debates y crecimiento.

Finalmente agradezco a mis profesores de la universidad, en especial a Andrés Abeliuk, Gonzalo Navarro y Ricardo Baeza-Yates, de quienes aprendí muchísimo estos últimos años. A mis profesores de plan común, Marcel Clerc y Carlos Flores, quienes me inspiraron a trabajar en las ciencias de la computación.

Muchas gracias a todos.

Declaración de uso de IA

En la elaboración de esta tesis se utilizaron herramientas de inteligencia artificial generativa, principalmente Codex CLI para el desarrollo de el informe y el código; y ocasionalmente ChatGPT para dudas puntuales sobre la literatura. En ambos se ocupó GPT-5, simplemente porque era el que se tenía al alcance.

Para el desarrollo de código, se diseñó un *Jupyter Notebook* inicial con el funcionamiento y lógica que debía contener NeuralZip, el cual se le hizo un *refactor* usando Codex. Programar desde cero o sin revisar con las herramientas permitía que tomaran decisiones sin consultar que llevaba a errores graves, e.g. crear datasets sintéticos en vez de usar datos reales, simular modelos con pesos aleatorios en vez de usar checkpoints entrenados, comprobar peso final comprimido usando estimaciones en vez de medirlo correctamente, etc.

Para el desarrollo del informe, las herramientas eran útiles para tareas repetitivas e.g. corregir errores gramaticales en inglés, crear tablas de resultados, fórmulas matemáticas, errores de compilación, o modificar la plantilla de LaTeX. Los modelos no fueron útiles a la hora de la redacción principal e interpretación de resultados, dado que alucinaban en la mayoría de las ocasiones, escribían información irrelevante para la memoria, o cometían errores que requerían supervisión constante.

Donde fueron mas útiles fue en dudas puntuales sobre la literatura, ChatGPT era capaz de buscar los trabajos relacionados a la tesis como ZipNN, resumirlos y responder dudas que luego calzaban con las implementaciones. También eran muy útiles a la hora de formalizar desarrollos matemáticos de el Lema 3.1, el Lema 3.2 y el Teorema 3.3.

Todo el contenido fue revisado y validado por el autor, quien asume plena responsabilidad sobre el trabajo presentado

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Problem

Modern society increasingly relies on intelligent systems, which has motivated the training of models at very large scale. These models solve problems in areas such as medicine [63], education [28], communication [76], and software development [5]. Foundation models, in particular, have undergone a rapid transformation in recent years, with substantial improvements in performance, precision, and practical usefulness [3].

A foundation model can be understood as a large neural network whose behavior is determined by a set of learned parameters. These models are commonly built from architectures such as transformers [67], convolutional networks [21], or related neural modules, and their execution consists largely of tensor operations, especially matrix multiplications, nonlinearities, and normalization steps [16, 1], one example of well-established foundation model is the GPT architecture shown in Figure 1.1. The learned parameters are stored as model weights, typically in floating-point tensor files. Depending on the training objective and architecture, the same general principle supports different tasks: large language models use these weights to model token sequences [4], diffusion models use them to generate data through denoising processes [23], and vision models use them to classify images or extract visual representations [21].

The exponential growth in the size of foundation models has introduced significant challenges in storage, computational efficiency, and energy consumption. These costs make deployment difficult on devices with limited resources, such as mobile phones, embedded systems, local agents, and edge environments. Given the trend toward local and edge execution [2], several model-reduction techniques have been developed [6]. *Quantization* [15, 20] reduces numerical precision to decrease model size and accelerate inference. *Pruning* [32, 77] removes low-relevance connections or parameters. *Knowledge Distillation* [19] trains a smaller model to imitate a larger one. *Low-Rank Factorization* [59, 64] and parameter sharing reduce internal redundancy in matrices and weights. Finally, *sparse or selectively activated models* [39] use only a fraction of their parameters during inference. Figure 1.2 summarizes the main families in model compression.

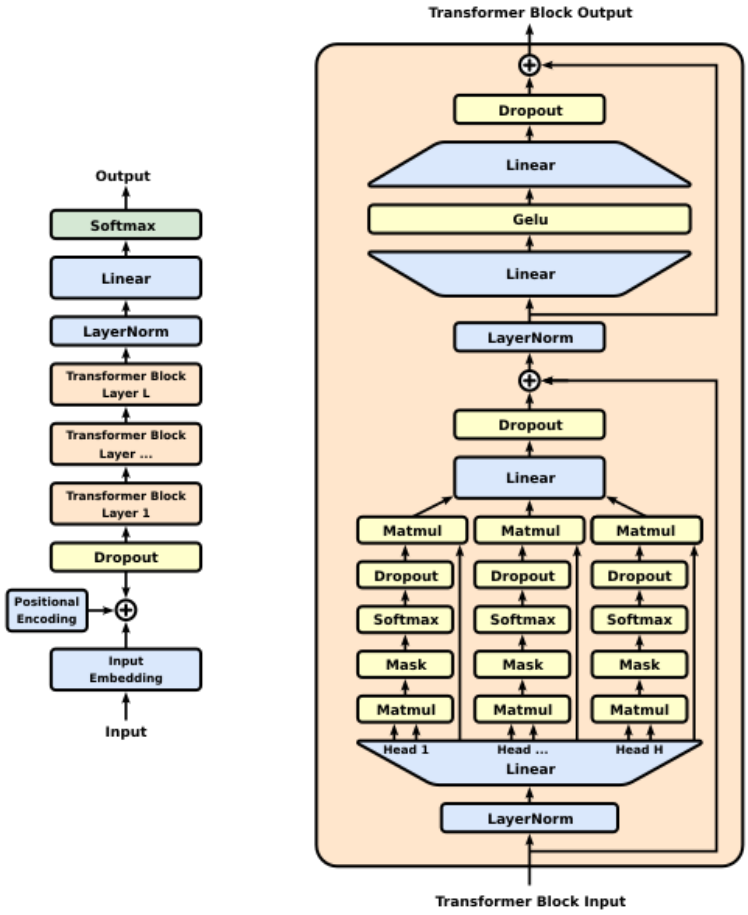
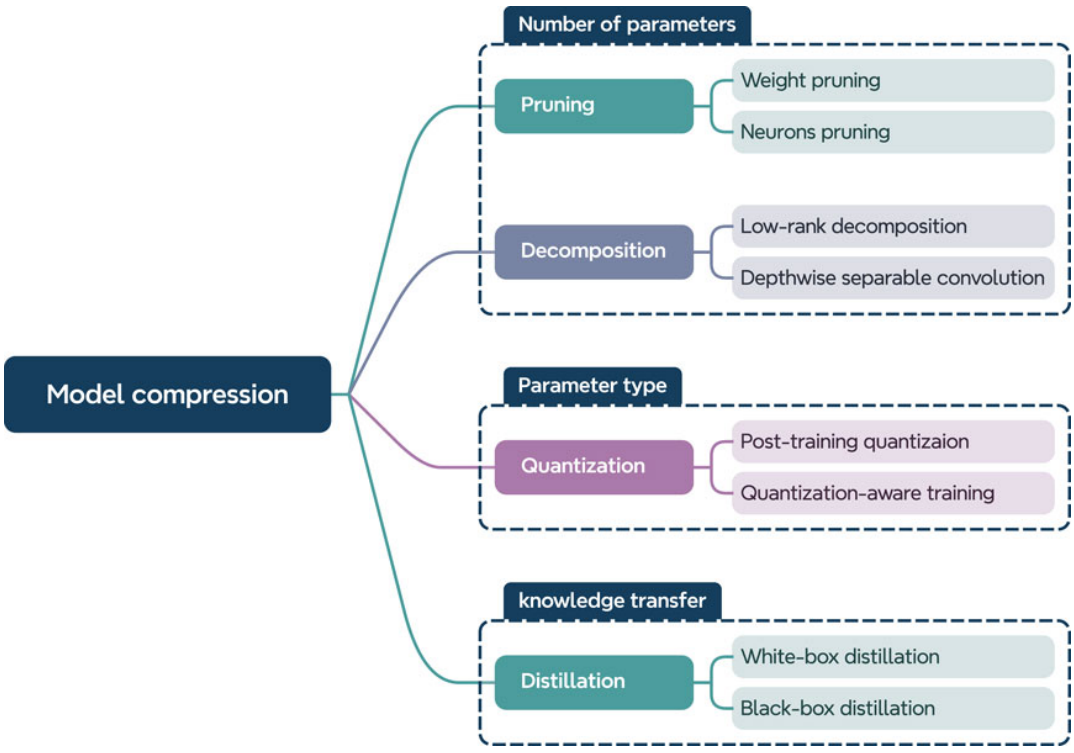


Figure 1.1: GPT architecture.

Although these techniques can make models smaller or faster, they introduce three important limitations. First, most of them do not preserve the original model: after pruning, quantization, distillation, or architectural changes, the exact original weights cannot be recovered unless an uncompressed copy is stored separately. Second, the reduction is therefore inherently lossy, so it can change the model's predictions and degrade accuracy or generation quality. Third, many of these methods require an additional post-training or fine-tuning stage, which adds computational cost after the original model has already been trained. These limitations motivate studying compression methods that reduce storage while keeping the original parameters exactly recoverable.

This is where *lossless* compression approaches address the gap. A lossless compressor encodes data into a shorter representation from which the original file can be reconstructed exactly. General-purpose compressors such as zlib, gzip, and Zstandard exploit repeated byte patterns and statistical redundancy in arbitrary files [10, 7]. More specialized methods from data compression and Compact Data Structures build on ideas such as entropy coding, variable-length codes, and indexed representations [60, 46]. Classical examples include Huffman coding, which assigns shorter codewords to more frequent symbols [24], and arithmetic coding, which can encode symbol sequences close to their entropy limit [56]. For model weights, the key question is how to adapt these principles to floating-point tensor data instead of treating the serialized model artifact only as an opaque byte sequence.



Source: Liu et al. [37].

Figure 1.2: Model-compression taxonomy.

The usefulness of lossless compression is not limited to storing the weights of a single model. Many current workflows require multiple model types, versions, or updates, so storage and network costs accumulate across the whole model lifecycle. Some motivating use cases are the following:

1. *Model hubs.* Large model repositories and hubs, such as the Hugging Face Hub, store collections of model artifacts and serve repeated upload and download requests [68]. In this setting, exact compression is useful in three ways: it reduces the amount of data stored by the hub, reduces the amount of data transferred over the network, and can reduce the time needed to upload or download large models.

2. *Sparse and mixture-of-experts models.* Sparse architectures such as mixture-of-experts models increase capacity by adding many expert parameters, but only activate a small subset for each input [61, 12]. This creates a gap between the number of parameters that must be stored and the number used in a given forward pass, making exact compression attractive for expert weights that remain available but inactive most of the time.

3. *Distributed and decentralized training.* During distributed training, nodes may exchange model weights, gradients, optimizer states, or partially trained models. In federated and decentralized learning, clients or contributors also transmit model updates repeatedly [40]. Since communication is often a limiting resource in these settings, lossless compression can reduce message sizes without changing the numerical update being communicated.

4. *Training checkpoints and model versions.* Training commonly produces intermediate

checkpoints for crash recovery, evaluation, hyperparameter search, and model selection. Development also produces multiple released model versions. Saving these states and artifacts increases storage and network pressure, and lossless compression reduces that overhead while keeping each saved state exactly recoverable.

5. *Lossy-plus-lossless pipelines.* Lossless compression can be applied after lossy reductions such as quantization. In that case, the compressed artifact no longer represents the original full-precision model, but it preserves the quantized model weights exactly and avoids introducing additional approximation.

6. *Compressed inference.* If a model artifact is decompressed before loading, lossless compression mainly improves storage and distribution. If the compressed representation remains active through the memory hierarchy, GPU kernels, or accelerator datapaths, it can also reduce memory footprint and bandwidth during serving; approaches such as DFloat11 and Huff-LLM demonstrate this direction [74, 73].

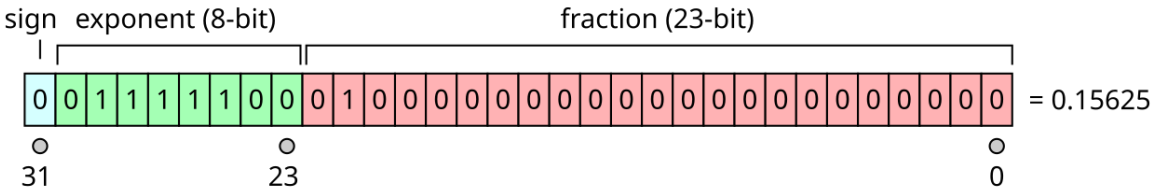


Figure 1.3: IEEE 754 single-precision format.

A notable example of a *lossless* compressor is ZipNN, which exploits the skewed distribution of floating-point exponents (represented in Figure 1.3) and compresses them with Huffman coding [22], as illustrated in Figure 1.4. ZipNN reports strong compression results across vision and language models while preserving the original parameters exactly. Another relevant approach is DFloat11 [74], which introduces a dynamic-length floating-point representation for efficient GPU inference while preserving the model parameters exactly. Huff-LLM also targets exact model preservation, but emphasizes end-to-end inference rather than only model-weight storage: it keeps LLM weights compressed across disk, main memory, and on-chip buffers, then decodes them only when they are consumed by accelerator compute units [73].

These methods show that lossless compression of model weights is both feasible and useful, but they also leave open a more structural question: whether current approaches exploit all the statistical regularity available in floating-point weights. In particular, a coding model fitted to a coarse group of values may miss differences among tensors and parameter types, while a separate Huffman code for every small group repeats construction and metadata costs. This suggests studying model compression from the perspective of Compact Data Structures, where the representation is designed around the internal organization of the data, the entropy of each component, and the metadata needed for practical decoding.

NeuralZip is proposed in this thesis as a lossless compressor for floating-point model weights. It retains the field-aware view introduced by ZipNN, but also leverages two phenomena observed across the evaluated architectures. First, exponent distributions vary among

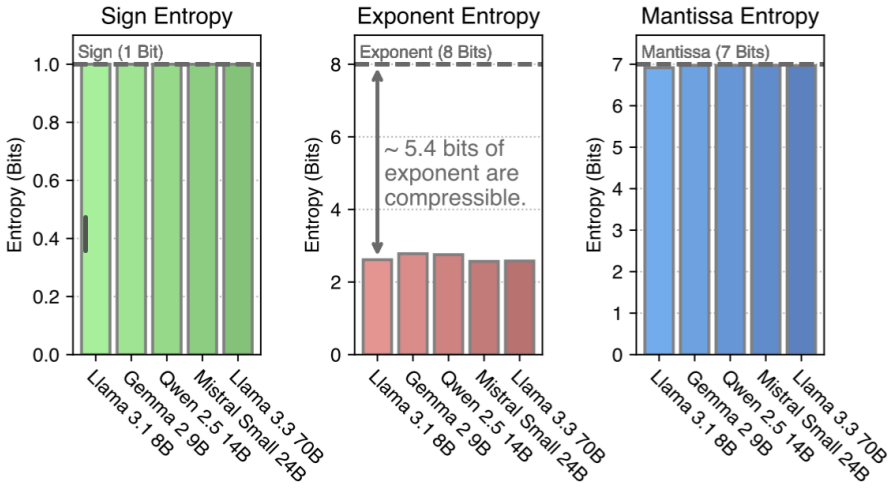


Figure 1.4: Representative exponent distribution.

parameter blocks while remaining similar within subsets of blocks, a property studied here as *blockwise heterogeneity*. Second, grouping adjacent exponent values into new symbols can reveal local regularities, a representation we call *local exponent redundancy*. NeuralZip combines both ideas in one algorithm: Jensen–Shannon clustering enables Huffman-code reuse, while a selector activates packed exponents only when sampled experiments predict a useful size reduction.

1.2 Hypothesis

Floating-point model weights contain statistical structure beyond overall exponent skew; exploiting blockwise heterogeneity together with local exponent redundancy can improve the compression-ratio–time trade-off of a lossless model compressor.

1.3 Research Questions

- How much of the lossless compression opportunity in floating-point model weights is explained by Exponent Skew, and how much additional structure appears through local dependence?
- Are exponent distributions heterogeneous across parameter blocks, and can groups derived from Jensen–Shannon distance reuse coding models effectively?
- What compression-ratio and compression-time trade-off does NeuralZip achieve relative to ZipNN across different model architectures?
- In which storage, training-state, and federated-communication settings is Huffman-code reuse useful, and what limits the resulting gains?

1.4 Objectives

1.4.1 General Objective

To design and evaluate a bitwise-lossless compressor for floating-point foundation-model weights that exploits blockwise heterogeneity and local exponent redundancy, and to characterize its compression-ratio–time trade-off relative to ZipNN across architectures and reuse scenarios.

1.4.2 Specific Objectives

1. To characterize statistical structure in floating-point exponent fields through entropy, Jensen–Shannon distance, and packed symbols, including exponent skew, blockwise heterogeneity, and local dependence.
2. To design and implement NeuralZip using histogram-based clustering, shared Huffman codes, packed exponents, time-aware selection, and exact fallbacks.
3. To compare NeuralZip with the authors' ZipNN source implementation under a consistent encode-versus-encode protocol, reporting complete exponent-stream size and verifying bitwise-exact round trips.

4. To evaluate structural reuse across model families, training checkpoints, and federated communication, together with its interaction with distilled and quantized artifacts.

1.5 Contributions

The contributions of this thesis can be summarized as follows:

- We identify and measure blockwise heterogeneity and local exponent redundancy in the exponent field of model weights.
- We present NeuralZip, a lossless model compressor that combines Jensen–Shannon clustering, shared Huffman codes, and packed exponent vocabularies.
- We provide ideal entropy bounds that motivate finer exponent partitions and packed symbols.
- We evaluate the resulting representation across several architectures and real-world scenarios, reporting both the benefits and the limits of exponent-focused lossless compression.
- The software developed for this thesis is publicly available in our GitHub repository [1].

1.6 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 introduces the related work required to understand the problem.

Chapter 3 presents ZipNN and the proposed NeuralZip compressor.

Chapter 4 defines the experimental setup and evaluates ZipNN and NeuralZip.

Chapter 5 summarizes the findings, limitations, and directions for lossless model compression.

The appendices provide supporting material, including detailed mathematical proofs, deferred techniques, ablations, and additional results.

[1] `https://github.com/MartinEBravo/bachelor-thesis-code`

Chapter 2

Related Work

2.1 Lossy Model Compression

Most work described as model compression reduces the computational or storage cost of a neural network by changing the model itself. Pruning removes weights, connections, neurons, or structured blocks judged to be less important, from early saliency-based methods to modern sparsification schedules [32, 77, 39]. Quantization replaces high-precision weights or activations with lower-bit representations [20, 15, 33]. Knowledge distillation trains a smaller student model to imitate a larger teacher, transferring behavior rather than preserving the original parameter values [19, 71]. Low-rank methods and related matrix factorizations replace dense operators with smaller algebraic decompositions [59, 64].

These approaches are essential for efficient deployment, but they are usually irreversible. After pruning, quantization, distillation, or factorization, the original tensor bits cannot be recovered unless a separate copy is retained. Consequently, lossy model compression and lossless model compression answer different questions. Lossless compressors only work for representations whose decoder reconstructs the original tensor shape, dtype, and bits exactly.

2.2 Lossless Model Compression

General-purpose compressors such as zlib, gzip, and Zstd combine entropy coding with dictionary or match-finding stages [10, 7]. They provide useful byte-oriented references but do not know that a model artifact consists of typed tensors.

ZipNN is the main field-aware lossless compressor [22]. It applies reversible bit and byte reordering, separates equal byte positions, and compresses sufficiently skewed groups with Huffman coding. In conventionally trained BF16 and FP32 models, the exponent field is particularly compressible. In rounded or "clean" models, mantissa groups containing repeated zeros may also be compressed. The decoder reverses the transformations and recovers the original model bits.

The Language Model Compressor (LMC) was developed for large training snapshots and is also based on byte grouping and Huffman coding [69]. Its main objective is high-throughput checkpointing: reducing the amount of data that must cross the network and reach persistent storage before the next training interval. LMC also evaluates incremental delta compression between snapshots. DFloat11 demonstrates that exact compression can be integrated with GPU serving rather than used only for archival storage [74]. It keeps BF16 signs and mantissas verbatim, entropy-codes the exponent field, and uses GPU-oriented lookup and scheduling structures to decode weights when they are consumed. Huff-LLM follows a related systems direction for FP16 language models: it partitions each value into hardware-friendly fields, keeps weights compressed across storage and memory levels, and places Huffman decoders near the accelerator datapath [73].

MM-RePair follows a different but related direction: it represents a matrix through a RePair grammar and supports matrix–vector multiplication over the grammar-compressed representation [13]. Rather than modeling floating-point fields, it exploits repeated adjacent patterns in a matrix-derived integer sequence and reuses the corresponding grammar nonterminals during computation. Its primary goal is therefore algebra over compressed matrices, not checkpoint storage. Since neural-network weights are organized as matrices, the method provides a useful structural comparison; the original implementation is evaluated on dense exponent matrices in Appendix C.1.

Git-Theta addresses another part of the model lifecycle: collaborative version control for model artifacts and tensor-level updates [27]..

Compressor	Coding mechanism	Primary objective
ZipNN	Local Huffman codes with raw fallback	Model storage and transfer
LMC	Parallel Huffman coding	High-throughput checkpointing
DFloat11	Huffman coding with GPU lookup structures	GPU inference
Huff-LLM	Huffman decoders placed near compute units	Accelerator inference
MM-RePair	RePair grammar over a matrix representation	Matrix–vector multiplication over compressed matrices

Table 2.1: Lossless compressors.

Chapter 3

Methods

This chapter first presents ZipNN and then introduces NeuralZip. If the reader is not familiar with certain Compact Data Structures or Machine Learning topics, it may find useful to read Appendix A.

3.1 ZipNN

ZipNN is a field-aware, chunk-based lossless compressor model weights [22]. For each supported dtype it applies a reversible bit reordering followed by byte grouping. In BF16 and FP32, the bit reordering consolidates the exponent field into one byte position and moves the sign bit into a group that otherwise contains mantissa bits. The byte-group transform then collects the equal byte position from every value. Each resulting group is divided into chunks and compressed independently when its empirical distribution is sufficiently skewed; otherwise it is stored verbatim.

3.1.1 Exponent Skew

Exponent values in trained weights are concentrated on a small part of their nominal alphabet. A uniform eight-bit exponent field would have entropy close to eight bits per value. Instead, a few exponents account for most observations, so Huffman coding can assign short codewords to them and longer codewords to the tail.

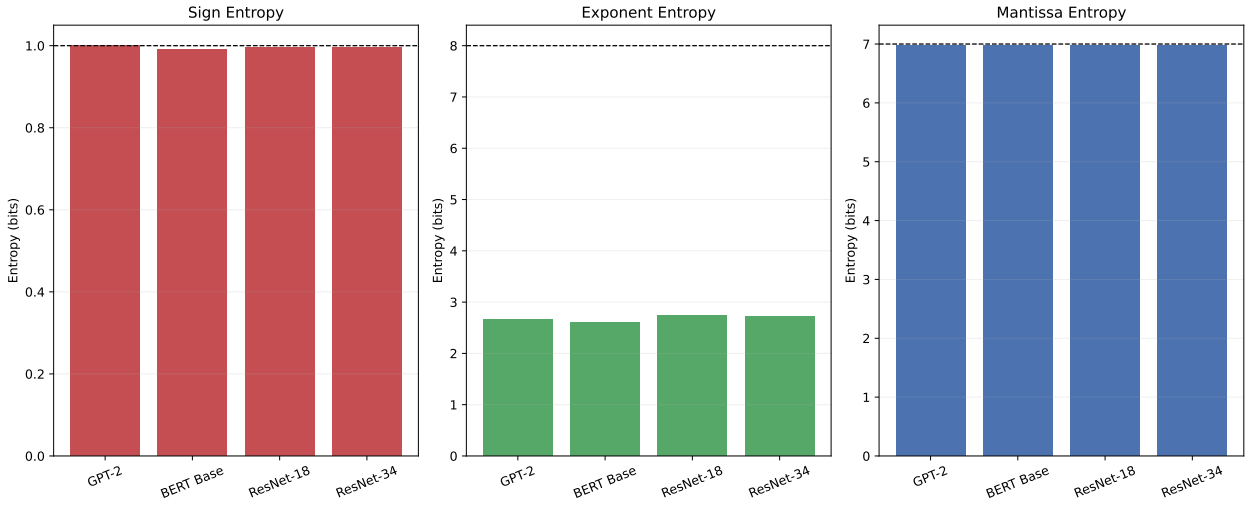


Figure 3.1: Entropy by floating-point field.

3.1.2 Algorithm

Algorithm 3.1 gives the relevant compression method. Huffman codes and chunk headers are part of the compressed representation, and the inverse bit-reordering and byte-grouping operations reconstruct the original tensor bits.

```
for each named tensor (u, B) in model Θ do
    R ← ReorderFloatingPointBits(B)
    G ← SplitIntoByteGroups(R)
    for each byte group g_b ∈ G do
        for each chunk x ∈ Chunk(g_b) do
            (z, H_x) ← HuffmanEncode(x, init = ∅)
            Store the smallest of (z, H_x) and raw x
        end for
    end for
    Store tensor metadata
end for
return compressed model
```

Algorithm 3.1: Conceptual field-aware ZipNN compression

The optional argument init is a precomputed Huffman code. Setting it to $\varnothing$ makes the encoder construct and return a local code H_x from the input chunk. Thus, the pseudocode emphasizes the defining baseline behavior: each chunk builds and stores its own Huffman code. Compression thresholds, chunk headers, and reversible archive metadata are implementation details described in the text.

The main result tables use the authors' ZipNN source package. Because that implementation mutates supported floating-point inputs during its reversible reordering, its reported encoding time includes the defensive input copy needed for a non-destructive API. The controlled component and deferred-technique ablations instead require instrumentation that the

compiled source extension does not expose, so they use our compatible local implementation. In those ablations, NeuralZip and the local baseline share the same native Huff0 primitives for common operations. Appendix D.3 quantifies the difference between the two implementations.

3.2 Proposed Method: NeuralZip

NeuralZip is motivated by two empirical phenomena in floating-point model weights: blockwise heterogeneity and local exponent redundancy. Their use within NeuralZip is described later in Section 3.2.3.

3.2.1 Blockwise heterogeneity

Our block perspective is inspired by Zhang et al. [75], who characterize variation among the Hessian spectra of parameter blocks using Jensen–Shannon distance. We apply this perspective to the empirical distributions of the exponent values.

Similar exponent histograms arise among structurally coupled projections with comparable tensor geometry and within common depth regimes. This produces numerical subfamilies such as parallel feed-forward projections, query/key projections, or value/output projections, while changes in scale across the model can split the same projection type into several depth-dependent groups.

Let the model weights contain parameter blocks $\mathcal{B}_1, \ldots, \mathcal{B}_L$. If block ℓ contains N_ℓ values, its normalized exponent histogram is

$$h^{(\ell)}(a) = \frac{1}{N_\ell} \left| \{ i : e_i^{(\ell)} = a \} \right|, \qquad a \in \Sigma_E.$$

We measure the dissimilarity between two histograms with Jensen–Shannon distance [34]. For distributions p and q, let

$$m = \frac{p+q}{2}, \qquad d_{\mathrm{JS}}(p,q) = \sqrt{\frac{1}{2} D_{\mathrm{KL}}(p \| m) + \frac{1}{2} D_{\mathrm{KL}}(q \| m)}.$$

Figure 3.2 shows that distances are not constant away from the diagonal. Among the 48 largest blocks per architecture, off-diagonal distances span the full observed range from 0 to 1. Only 2.4–41.5% of pairs lie below the default threshold, and the selected blocks form 6–32 groups. The simultaneous presence of near-identical and widely separated distributions is direct evidence of blockwise heterogeneity rather than one global exponent law. Repeated functions can share activation and gradient scales—for example, normalization blocks with normalization blocks or feed-forward projections with related projections—while changes in depth, stage, or tensor geometry alter those scales. The clustering ablation in Appendix D.2 examines how these groups align with coupled projection families, comparable tensor geometries, and depth-dependent numerical regimes.

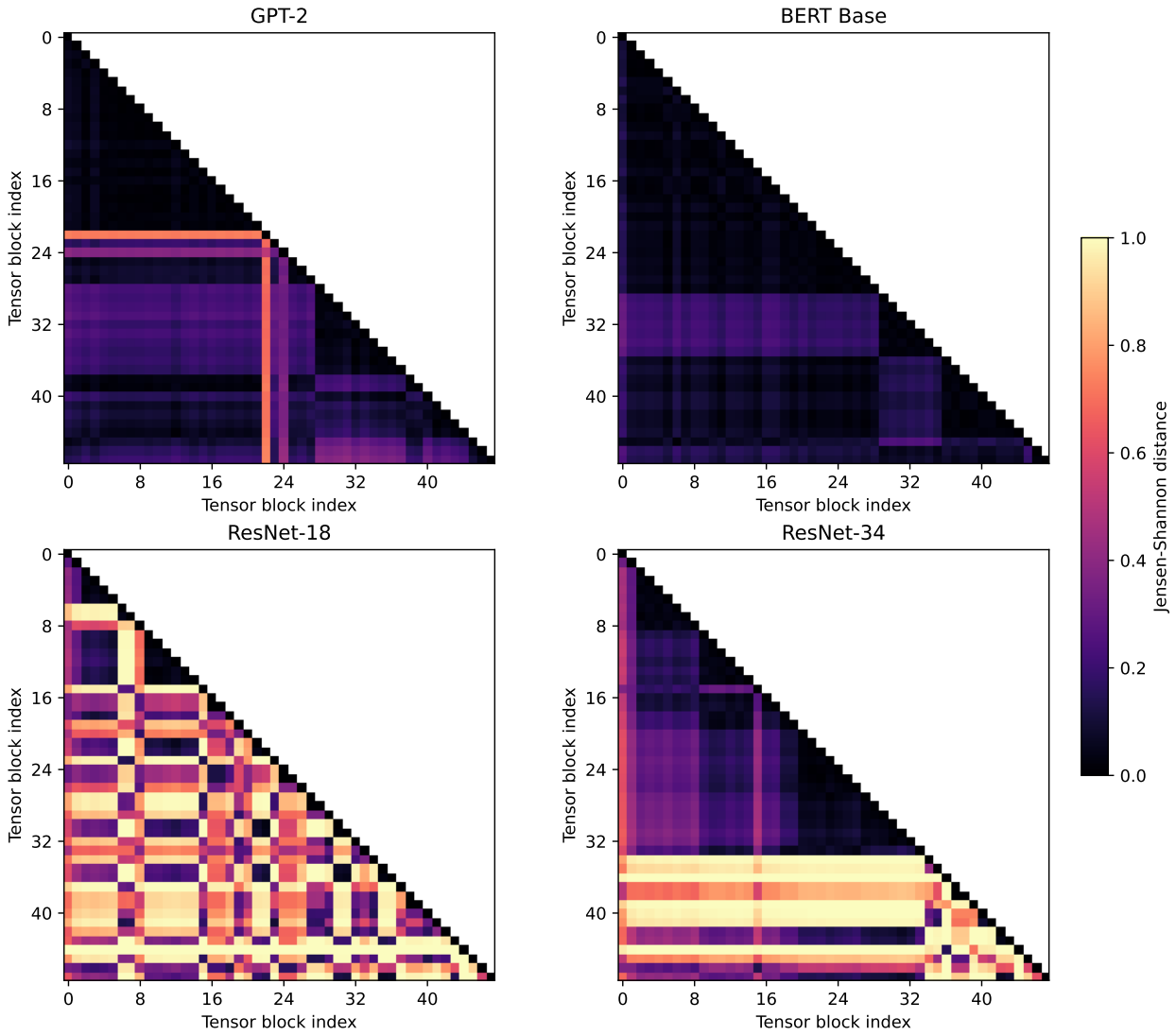


Figure 3.2: Blockwise Jensen–Shannon distance.

Ideal entropy under finer partitions Consider an ideal zero-order coder that partitions an exponent sequence into subsequences and fits one probability distribution to each subsequence. For a partition $\mathcal{Q}$, define the ideal payload cost

$$C_0(\mathcal{Q}) = \sum_{S \in \mathcal{Q}} |S| H_0(S).$$

Lemma 3.1 (Entropy monotonicity under tensor refinements) *If $\mathcal{P}_T$ refines a partition $\mathcal{P}$ by splitting its exponent sequences along parameter-block boundaries, then*

$$C_0(\mathcal{P}_T) \leq C_0(\mathcal{P}).$$

Proof sketch. Take $S \in \mathcal{P}$ and split it into block-induced subsequences $S_1, \ldots, S_t$. The empirical distribution of S is the length-weighted average of their distributions. By the concavity of Shannon entropy stated in Theorem A.10, $|S| H_0(S) \geq \sum_i |S_i| H_0(S_i)$. Summing over $\mathcal{P}$ proves the result. Appendix B.1 gives the detailed derivation. □

3.2.2 Local exponent redundancy

Local exponent redundancy refers to statistical dependence among adjacent exponent values. To expose this dependence, we treat n adjacent values as one composite symbol. For an exponent sequence

$$S = (e_1, e_2, \ldots, e_N), \qquad e_i \in \Sigma_E,$$

the k-th non-overlapping packed value is

$$y_k = \sum_{j=0}^{n-1} e_{kn+j+1} 2^{8j}, \qquad 0 \le k < \left\lfloor \frac{N}{n} \right\rfloor.$$

The ideal analysis considers only complete non-overlapping tuples and excludes any suffix shorter than n.

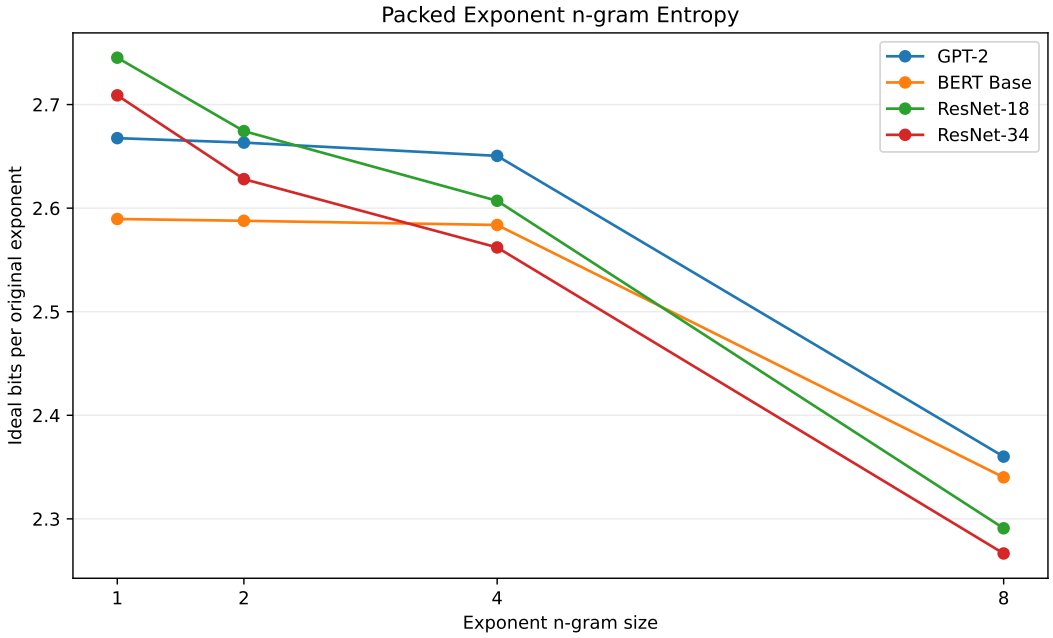


Figure 3.3: Packed-exponent entropy.

Figure 3.3 shows that normalized entropy can decrease when adjacent exponent values are considered jointly. Packing two and four adjacent values reduces the sampled empirical cost by 0.002–0.081 and 0.006–0.147 bits per original exponent, respectively. The $n = 8$ points show a larger exploratory decrease of 0.25–0.45 bits, but each estimate is formed from only about one million tuples over 0.57–0.66 million observed symbols. We therefore use $n = 2$–4 as the principal evidence and treat $n = 8$ as sample-limited. The consistent but non-uniform reduction indicates that local exponent redundancy exists while remaining architecture-dependent. A plausible explanation is that tensor serialization preserves neighborhoods such as rows, channels, and convolutional kernels. Adjacent weights in these neighborhoods participate in the same local operator and are trained under related activation and gradient scales, so their magnitudes, and therefore their exponents, need not be independent.

Ideal entropy under packed exponents For the ideal analysis, let $S^{[n]}$ be the longest prefix of S whose length is divisible by n, and let $G_n(S)$ be the sequence of packed symbols from that prefix. Define

$$H_0^{(n)}(S) = H_0(G_n(S)), \qquad \overline{H}_0^{(n)}(S) = \frac{1}{n} H_0^{(n)}(S).$$

For a partition $\mathcal{P}$, ignoring raw tails, define

$$C_0^{(n)}(\mathcal{P}) = \sum_{S \in \mathcal{P}} \left\lfloor \frac{|S|}{n} \right\rfloor H_0^{(n)}(S).$$

Lemma 3.2 (Entropy monotonicity under packed exponents) *Let $\mathcal{P}^{[n]}$ contain the non-empty complete prefixes of sequences in $\mathcal{P}$. For every $n \geq 1$,*

$$C_0^{(n)}(\mathcal{P}) \leq C_0(\mathcal{P}^{[n]}).$$

Equivalently, $\overline{H}_0^{(n)}(S) \leq H_0(S^{[n]})$ for each non-empty complete prefix.

Proof sketch. View the positions of a uniformly sampled packed symbol as random variables $X_1, \ldots, X_n$. By Theorem A.11, their joint entropy is at most the sum of their marginal entropies. The empirical distribution of $S^{[n]}$ is the average of those positional marginals, whose entropy is at least their average entropy by Theorem A.10. Thus $H_0^{(n)}(S) \leq nH_0(S^{[n]})$. Multiplying by the number of packed symbols and summing proves the result. The detailed proof is in Appendix B.2. □

3.2.3 Algorithms

Precomputation The implementation converts blockwise heterogeneity into a cluster map. It processes named tensors in deterministic order and compares each block with the current cluster centroids. A new cluster is created when the nearest centroid is farther than threshold τ; otherwise the block joins that cluster and its centroid is updated using an average weighted by the number of values. A user-provided cluster map, such as one derived from model metadata (e.g., Hugging Face model parameters), can replace this greedy procedure. All experiments described as JS clustering use the default rule.

To exploit local exponent redundancy without materializing an alphabet of size 256^n, the implementation evaluates $n \in \{2, 3, 4\}$ and retains the $K = 254$ most frequent packed values from a bounded sample. These values receive byte-valued tokens, while token 255 is reserved for escape. An unseen tuple is stored as an escape token followed by its original bytes, and an incomplete suffix is retained as a raw tail. The representation is therefore exactly reversible. Order-1 Huffman codes are an alternative to our packed-exponent technique; the difficulties of this alternative are detailed in Appendix C.3.

Shared Huffman codes are indexed by

$$(\text{cluster id}, \text{dtype}, \text{byte-group id}).$$

For every such key, NeuralZip collects at most M bytes from tensors assigned to the cluster. With the default configuration, $M = 8$ MiB. Tensor boundaries are preserved while sampling. The implementation uses the native Huff0 backend from Zstandard; a CTable is its encoder-side Huffman table. The sample is used to evaluate raw storage, local Huff0, and shared Huff0; for an exponent group, it also evaluates the packed candidates described above. Every estimated size includes the concrete Huff0 tables, vocabulary, escape, tail, and chunk metadata produced by the archive format.

Mode selection considers both compressed size and mode-dependent direct work. For candidate m, let b_m and t_m denote its estimated bytes and measured mode time on the same bounded logical-lane sample, and let $b^* = \min_m b_m$. Among the raw, local, and shared modes satisfying $b_m \le (1 + \varepsilon)b^*$, the selector chooses the smallest t_m. The default $\varepsilon = 0.03$ therefore permits a faster alternative to replace local Huff0 only within a controlled relative size tolerance.

A packed candidate becomes an *opportunity* only if it saves bytes relative to the selected base mode. Opportunities are ranked by estimated bytes saved per additional direct-encoding second and admitted under the global budget $\gamma \sum t_{\text{base}}$, with default $\gamma = 1.0$. The surviving opportunities are then reranked by bytes saved per source byte and admitted until they cover at most a fraction $\beta = 0.30$ of the exponent source. This second cap bounds packed-source coverage, target tokenization work, and transient memory. For candidate timing, the sampled logical lane is placed in stream zero of a synthetic tensor without bit reordering; the remaining lanes use raw mode. The production mixed native primitive then performs stream extraction, the candidate mode, and its exact local/raw fallback semantics. After one warm-up, the selector retains the best of three complete sample calls to reduce scheduler noise. This is a relative mode-selection signal rather than a complete-model encode measurement; reported encoding time is measured independently over the complete model.

After mode selection, the retained state is structural only: the cluster map, the selected per-stream modes, packed vocabularies, and shared Huff0 codebooks and CTables. It contains no reordered tensor streams, target packed tokens, escapes, tails, or target-local tables. Family transfer freezes and reuses this state. Direct mapping uses (name, dtype, byte-group) without reading target values, so it computes no target exponent histogram, Jensen–Shannon match, or alternative-mode probe. Reordering, byte-lane extraction, packed tokenization, any required local-code construction or fallback, and payload assembly remain part of every measured target encode.

$\kappa \leftarrow \text{ClusterByExponentDistribution}(\Theta)$
for each cluster c **and floating-point dtype** d **do**
  for each byte-group position b **do**
    $S \leftarrow \text{CollectByteGroup}(\Theta, \kappa, c, d, b)$
    $\mathcal{H}[c, d, b] \leftarrow \text{BuildSharedHuffmanCode}(S)$
    $\mu_c[c, d, b] \leftarrow \text{ChooseTimeAwareBase}(\text{Raw}, \text{Local}, \mathcal{H}[c, d, b])$
  end for
  $E \leftarrow \text{CollectExponentField}(\Theta, \kappa, c, d)$
  $\mathcal{P}[c, d] \leftarrow \text{BuildPackedExponentModel}(E)$
  $\mu_c[c, d, b_E] \leftarrow \text{ConsiderPackedMode}(\mu_c[c, d, b_E], \mathcal{P}[c, d])$
end for

```
for each tensor u do
    μ[u] ← μ_c[κ(u), dtype(u)]
end for
return (κ, H, P, μ)
```

Algorithm 3.2: Precomputation

Algorithm 3.2 deliberately leaves bounded sampling, candidate packed sizes, raw-mode tests, and both global budgets outside the conceptual flow. They determine the implementations of BuildPackedExponentModel, ChooseTimeAwareBase, and ConsiderPackedMode, as specified above and in Table 3.1.

Parameter	Default
Source chunk size	256 KiB
Compression acceptance threshold ρ	0.95
Minimum absolute saving	0 bytes
JS clustering threshold τ	0.05
Maximum sample per key M	8 MiB
Huffman pseudocount	1 per byte value
Packed sizes	$n \in \{2, 3, 4\}$
Packed vocabulary size K	254
Escape token	255
Ratio tolerance ε	0.03
Packed-time budget γ	1.0
Packed-source budget β	0.30
Exact-target cache	Disabled
Direct native batch threshold	128 MiB
Exponent layout	Original tensor order

Table 3.1: Default NeuralZip configuration.

Compression Once modes and tables are available, the selected mode is looked up by cluster, dtype, and byte-group position. The direct encoder reads the original tensor, performs reversible bit reordering and byte-lane extraction, and immediately applies the selected mode. Raw chunks become archive bytes. Local chunks build their Huff0 table and encode in the same call. Shared chunks use the retained cluster CTable. Packed chunks are tokenized directly from the interleaved, bit-reordered tensor through a hash lookup built for that batch, then encoded with their shared table while retaining escapes and the raw tail. A contiguous reordered packed stream is constructed only if packed coding is rejected and the chunk must try the local/raw fallback.

The native backend uses four-stream Huff0 frames for newly produced local, shared, and packed payloads. If a shared or packed payload does not meet ρ, the encoder constructs and tries the local Huff0 fallback; raw storage is used only when that frame is also not beneficial. Tensor jobs are grouped into windows whose default input-size threshold is 128 MiB; an individual tensor larger than the threshold forms its own window. The jobs within each

window share one native queue and thread pool, and all reordered lanes, tokens, and fallback workspace are released after that window. Only the final archive payloads remain; no exact-target cache is retained.

```
given precomputed (κ, H, P, μ)
for each input-size-windowed tensor batch Q in model Θ do
  in one native queue, parallel for each (u, B) in Q do
    c ← κ(u)
    d ← dtype(B)
    for each byte-group position b do
        g_b ← ExtractReorderedByteGroup(B, b)
        if g_b is the exponent group and μ[c, d, b] = Packed then
            x ← PackExponents(g_b, P[c, d])
            z ← HuffmanEncode(x, init = P[c, d].H)
        else if μ[c, d, b] = Raw then
            z ← g_b
        else if μ[c, d, b] = Local then
            z ← HuffmanEncode(g_b, init = ∅)
        else
            z ← HuffmanEncode(g_b, init = H[c, d, b])
        end if
        if μ[c, d, b] ∈ {Shared, Packed} and z fails ρ then
            z ← HuffmanEncode(g_b, init = ∅)
        end if
        if z does not satisfy ρ then z ← g_b
        Store z and its local code or shared-code reference
    end for
    Store tensor metadata
  end parallel for
end for
return compressed model
```

Algorithm 3.3: NeuralZip compression

The concrete encoder performs this flow chunk by chunk. It records local codes or shared-code references, escape data, tails, tensor boundaries, shapes, and dtypes so that decoding remains exact. We study what happens if we isolate shared Huffman codes and packed exponents in Appendix D.1.

Ideal Bound The two ideal entropy arguments can be combined, provided their assumptions are kept explicit.

Theorem 3.3 (Ideal bound) *Let $\mathcal{P}$ be a partition of exponent sequences, and let $\mathcal{P}_C$ be a refinement obtained by splitting its sequences along parameter-block groups. Fix $n \geq 1$ and assume every refined sequence has length divisible by n. Then*

$$C_0^{(n)}(\mathcal{P}_C) \leq C_0(\mathcal{P}).$$

Proof sketch. Lemma 3.2 gives $C_0^{(n)}(\mathcal{P}_C) \leq C_0(\mathcal{P}_C)$, and Lemma 3.1 gives $C_0(\mathcal{P}_C) \leq C_0(\mathcal{P})$. Chaining the inequalities proves the claim. Appendix B.3 gives the detailed proof. □

Theorem 3.3 excludes tails, escapes, integer code lengths, metadata, local-Huffman and raw fallbacks, clustering cost, and the fact that NeuralZip shares one Huffman code among several blocks. To determine whether NeuralZip improves over its baselines on a concrete model we need to measure size and compression time.

Chapter 4

Results

4.1 Setup

The model-storage and ablation experiments were executed on a CPU server with 1 TiB of RAM and two AMD EPYC 7343 16-core processors, for a total of 32 physical cores and 64 hardware threads across two NUMA nodes. Family-transfer, training, and federated experiments used a separate machine with one AMD EPYC 7282 16-core processor, 128 GiB of RAM, and three NVIDIA RTX A5000 GPUs. Compression timings use 12 pinned CPU threads unless stated otherwise.

Unless a table explicitly says "local," ZipNN denotes the authors' source package. Its measured encode includes the defensive copy required to preserve the caller's tensors. Both compressors use one verified warm-up and one measured repetition for direct encoding. Table values are shown to three decimal places; boldface and aggregate comparisons use the unrounded measurements.

Because the exponent field is the consistently compressible component in the evaluated weights, we report compression ratios only for this field. Let B_E be its uncompressed size in bytes. The reported ratio is

$$R_E = 100\frac{B_{\text{symbols}} + B_{\text{raw}} + B_{\text{metadata}}}{B_E},$$

Here, B_{symbols} is the encoded-symbol payload, and B_{raw} is data stored without coding. The term B_{metadata} contains the information required for exact decoding. Thus, the reported values include the complete exponent representation.

4.2 Precomputation

NeuralZip separates family-level structural precomputation from direct encoding. The reusable state contains only the cluster map, selected modes, packed vocabularies, and Huff0

codebooks and CTables. It contains no reordered target stream, packed target token, escape data, or target-local table. Thus, the measured direct encode includes target extraction and bit reordering, byte-group formation, packed tokenization, any required local-code construction or fallback, entropy coding, and payload assembly. Tensor jobs are submitted in input-size windows; all jobs within a window share one native parallel queue, and their transient workspace is released after that window. The default window threshold is 128 MiB, while a larger individual tensor forms its own window.

Candidate timing is part of structural mode selection, not the reported encode measurement. It places the sampled logical lane in stream zero of a synthetic tensor, assigns raw mode to the remaining lanes, and invokes the production mixed native primitive with its exact fallback semantics. It uses one warm-up and the best of three complete sample calls to reduce scheduler noise. In contrast, the complete-model results below use one warm-up followed by exactly one measured encoding repetition on 12 CPU threads. No exact-target cache is built in either measurement.

We evaluate target-specific direct encoding on a heterogeneous BF16 suite comprising transformer and convolutional architectures. For each checkpoint, NeuralZip builds its structural plan once and then encodes the original tensors without retaining an exact-target cache. The structural setup takes 1.32–5.82 s across the suite. This cost is stated separately; Table 4.1 reports only direct encoding.

	Encoding time (s)		Ratio (%)	
Model	ZipNN source	NeuralZip	ZipNN source	NeuralZip
GPT-2 [55]	0.334	**0.058**	31.278	**30.570**
BERT Base [11]	0.318	**0.051**	**32.598**	32.611
ResNet-18 [21]	0.081	**0.009**	33.372	**32.388**
ResNet-34 [21]	0.154	**0.014**	33.291	**32.650**

Table 4.1: Direct encoding.

Table 4.1 shows that, in this single measured repetition, NeuralZip encodes 5.74×–10.84× faster than ZipNN source across the suite. The exponent-field ratio is lower in three of the four cases, by 0.641–0.984 percentage points, while the remaining change is a 0.013-point penalty. Every model round-trips bitwise. Because each timing uses one repetition, the variation within this range is descriptive and does not establish a statistical ordering among models.

The ratio change follows the architecture rather than model size. Across the convolutional layouts, it decreases by 0.641–0.984 points and the measured first-order entropy reduction reaches 5.12–6.01%. Across the transformer layouts, the archive change ranges from a 0.013-point penalty to a 0.707-point improvement even though first-order entropy is only 0.20–0.31% below zero-order entropy. Local entropy alone therefore does not determine the final archive: shared codebooks, raw fallbacks, packed vocabularies, and metadata also contribute. The time gain has a separate mechanism: precomputed structural decisions and native batched execution avoid the repeated work performed by the source baseline.

We also evaluate structural transfer within language-model and convolutional families.

The source model first builds its clusters, Huffman codes, packed-exponent vocabularies, and selected modes. The target then uses this structural state through metadata mapping. No exact tensor-data cache from the source is transferred, and no such cache is built for the target. The target does not compute exponent histograms, run Jensen–Shannon matching, or probe alternative modes. Tensor names are shared within each evaluated family, so they provide the mapping to the source clusters. Target extraction, reordering, tokenization, and any local fallback remain part of structural-transfer encoding. The setup column reports the one-time source precomputation, and mapped tensors reports the exact source-to-target tensor mapping.

Model pair	Setup (s)	Encoding (s)	Mapped tensors	Ratio (%)
Qwen 3.5 2B → 0.8B [53, 51]	52.252	0.386	320/320	33.249
Gemma 3 4B → 1B [14]	123.723	1.121	340/340	32.981
ResNet-34 → ResNet-18 [21]	3.771	0.021	102/102	32.503

Table 4.2: Precomputation reuse.

Table 4.2 reports the one-time source setup and the transferred target encode. Every floating-point target tensor maps to the source plan, and all transferred encodings recover the target weights bitwise. The incremental mapping setup for a new family member is only 0.0005–0.0018 s, a 99.974–99.996% reduction relative to building a target-specific plan. Against ZipNN source, transferred encoding is 1.49×–1.59× faster in two cases and 1.15× slower in the remaining case. Two ratio differences are below 0.001 percentage points, while the remaining target improves by 0.909 points. The source setup can therefore be amortized across compatible members without assuming a universal direct-time advantage.

4.3 Model Storage

We study model storage across language, vision, and diffusion architectures. The suite contains both isolated checkpoints and multiple members of open model families, enabling a separate codebook-reuse analysis. All comparisons use the official ZipNN source implementation.

Table 4.3 shows that the post-setup trade-off depends on the architecture family. Across the vision families, NeuralZip is 8.91×–21.51× faster and obtains a lower ratio in seven of twelve cases; the ratio difference ranges from a 0.224 percentage-point penalty to a 0.984 percentage-point improvement. Across the remaining architectures, NeuralZip is faster in fourteen of twenty-two cases, ranging from near parity to 4.89×, and is 1.01×–1.58× slower in the other eight. The exponent ratio remains within 0.008 percentage points in sixteen of twenty-two cases; across the group, the difference ranges from a 0.140 percentage-point improvement to a 0.064-point penalty. Thus, precomputation provides substantial encoding benefits when the stored plan matches the target structure, but does not guarantee a uniform gain for every architecture. Structural setup spans 1.32–686.27 s across this suite and is reported separately from direct encoding.

Model	Encoding time (s)		Ratio (%)	
	ZipNN source	NeuralZip	ZipNN source	NeuralZip
ResNet				
ResNet-18 [21]	0.081	**0.009**	33.372	**32.388**
ResNet-34	0.154	**0.014**	33.291	**32.650**
ResNet-50	0.280	**0.018**	34.061	**34.013**
ResNet-101	0.390	**0.032**	34.193	**34.107**
ResNet-152	0.561	**0.042**	34.227	**34.211**
ConvNeXt				
ConvNeXt-Tiny [38]	0.158	**0.015**	33.199	**33.060**
ConvNeXt-Small	0.328	**0.035**	**33.242**	33.261
ConvNeXt-Base	0.432	**0.041**	**33.045**	33.056
ConvNeXt-Large	0.718	**0.081**	**32.978**	32.983
EfficientNet				
EfficientNet-B0 [65]	0.193	**0.009**	**32.961**	33.185
EfficientNet-B3	0.309	**0.022**	**33.321**	33.481
EfficientNet-B7	0.779	**0.051**	36.243	**36.149**
Qwen 2.5				
Qwen 2.5 1.5B [72]	1.812	**0.585**	**32.967**	32.975
Qwen 2.5 3B [72]	2.715	**1.381**	33.860	**33.720**
Qwen 2.5 7B [72]	**4.703**	6.717	**33.347**	33.411
Qwen 2.5 14B [72]	**8.979**	11.332	33.595	**33.517**
Qwen 3.5				
Qwen 3.5 0.8B [51]	1.683	**0.344**	**33.259**	33.267
Qwen 3.5 2B [53]	2.733	**0.976**	**33.160**	33.161
Qwen 3.5 27B [52]	**15.744**	18.389	**32.697**	32.704
OLMo-2				
OLMo-2-1B [66]	1.398	**0.531**	**32.957**	32.961
OLMo-2-7B [66]	6.343	**6.230**	**33.619**	33.630
OLMo-2-13B [66]	**9.872**	10.807	**33.630**	33.640
Llama				
Llama 3.2 1B [41]	1.082	**0.469**	**33.041**	33.049
Llama 3.2 3B [42]	2.618	**2.016**	**32.907**	32.914
Llama 3.1 Nemotron Nano 8B [47]	4.875	**4.869**	**32.665**	32.672
Mistral				
Mistral-7B [25]	4.393	**3.447**	**32.750**	32.757
Mistral Nemo 12B [44]	**7.280**	8.044	**32.770**	32.777
Phi				
Phi-4 14B [43]	**8.884**	10.770	**32.618**	32.625
Gemma				
Gemma 3 1B [14]	1.631	**0.395**	**32.972**	32.980
Gemma 3 4B [14]	4.533	**3.147**	**32.953**	32.958
Gemma 3 12B [14]	**9.008**	9.073	**32.846**	32.852
Gemma 4 31B [18]	**18.519**	29.337	**32.804**	32.811
LATAM-GPT				
LATAM-GPT 70B [31]	168.323	**80.107**	**32.382**	32.389
Stable Diffusion				
Stable Diffusion v1.5 UNet [57]	1.447	**0.354**	33.292	**33.291**

Table 4.3: Model storage.

4.4 Mixture-of-experts

We study sparse models where the total number of stored parameters is much larger than the number of active parameters used in a single forward pass. This motivates using NeuralZip, as repeated expert structures can benefit from Huffman-code sharing [61, 12]. The suite contains open sparse models with different expert organizations.

	Encoding time (s)		Ratio (%)	
Model	ZipNN source	NeuralZip	ZipNN source	NeuralZip
ZAYA1-8B [70]	14.599	**4.547**	33.192	**33.154**
LFM2.5-8B-A1B [35, 36]	15.816	**3.485**	32.638	**32.632**
JetMoE-8B [62]	**5.206**	6.126	**32.555**	32.562
OLMoE 1B-7B [45]	13.799	**4.788**	**32.664**	32.668
Qwen1.5-MoE-A2.7B [49]	24.056	**10.843**	**32.601**	32.609
Qwen3-30B-A3B [50]	63.497	**23.327**	32.685	**32.685**
Qwen3.5-35B-A3B [54]	**23.892**	33.199	**32.711**	32.719
DeepSeekMoE-16B [9]	27.262	**12.232**	**32.609**	32.617
Mixtral 8x7B [26]	**24.985**	43.330	**32.648**	32.655
Gemma 4 26B-A4B [17]	**16.688**	19.710	**32.843**	32.850

Table 4.4: Mixture-of-experts models.

Across the ten models in Table 4.4, the exponent ratios remain effectively aligned, with changes between a 0.008-point penalty and a 0.039-point improvement. Direct encoding is 2.22×–4.54× faster in six cases, whereas it is 1.18×–1.73× slower in the other four. Repeated expert structure therefore supports codebook reuse, but does not by itself guarantee a time advantage: tensor granularity and batching overhead still determine whether that reuse amortizes during encoding. The one-time structural setup spans 54.47–382.87 s and is reported separately from direct encoding.

4.5 Federated Learning

We evaluate online compression during federated training with two convolutional architectures on the real CIFAR-10 dataset [30]. Each FP32 trajectory runs for 200 communication rounds with 10 clients, one local minibatch of 64 examples per client and round, and FedAvg aggregation. In every round, the server encodes the global model for the downlink; each client decodes it, trains locally, and encodes its updated model for the uplink. The decoded states from the two compressors are checked for bitwise equality before the shared training trajectory advances.

The reported critical encoding time sums the server encode and the slowest client encode in each round, modeling parallel clients. Training, decoding, and network transfer are excluded. NeuralZip builds its structural plan from the initial state and distributes it once to each client. Setup takes 2.36–3.38 s and remains separate from direct encoding time; its 7.76–9.14 kB distributed payload is, however, charged once in the reported exponent ratio.

$P \leftarrow \emptyset$
if $\mathcal{A} =$ NeuralZip **then**
    $P \leftarrow \mathrm{BuildStructuralPlan}(\Theta^{(0)})$
    Send P **to every client once**
end if
for each communication round r **do**
    $Z_s \leftarrow \mathrm{Encode}_{\mathcal{A}}(\Theta^{(r)}, P)$
    for each client i **in parallel do**
        $\Theta_i \leftarrow \mathrm{Decode}_{\mathcal{A}}(Z_s, P)$
        $\Theta_i^{+} \leftarrow \mathrm{LocalTrain}(\Theta_i, D_i)$
        $Z_i \leftarrow \mathrm{Encode}_{\mathcal{A}}(\Theta_i^{+}, P)$
        Send Z_i **to the server**
        $\widehat{\Theta}_i \leftarrow \mathrm{Decode}_{\mathcal{A}}(Z_i, P)$
    end for
    Verify bitwise-equal decoded states
    $\Theta^{(r+1)} \leftarrow \mathrm{FedAvg}(\widehat{\Theta}_1, \ldots, \widehat{\Theta}_K)$
end for

Algorithm 4.1: Federated compression protocol

	Encoding time (s)		Ratio (%)	
Model	ZipNN source	NeuralZip	ZipNN source	NeuralZip
ResNet-18 [21]	57.380	**16.470**	**32.464**	32.521
ResNet-34	110.125	**26.342**	**32.434**	32.502

Table 4.5: Federated compression.

Across both trajectories, structural reuse reduces the direct critical encoding path by 3.48×–4.18×. Charging the plan once per client increases the exponent ratio by only 0.057–0.068 percentage points. Thus, the same frozen structure remains useful over 200 evolving

rounds, with a small and explicit communication-size trade-off rather than an unconditional gain.

4.6 Training Checkpoints

The checkpoint experiment studies how exponent compressibility changes during ordinary training across four convolutional vision architectures [21, 38, 65]. Each model is initialized randomly and trained from scratch for 50 epochs on real CIFAR-100 [30], using one A5000 per trajectory. We save the FP32 initialization and the weights after every epoch, obtaining 51 checkpoints per trajectory. The compressed artifact is used only to measure the exponent-field ratio and direct encoding time; it is not fed back into training, and there are no clients, communication rounds, or model aggregation.

For NeuralZip, the structural plan is built once from epoch zero and remains frozen for all subsequent checkpoints. Its 1.37–3.61 s setup and tensor mapping are excluded from direct encoding. For ZipNN source, the defensive copy required because the package mutates its input is included in encoding. Both compressors use one warm-up followed by one measured repetition on 12 CPU threads, and every archive passes a bitwise-exact round trip.

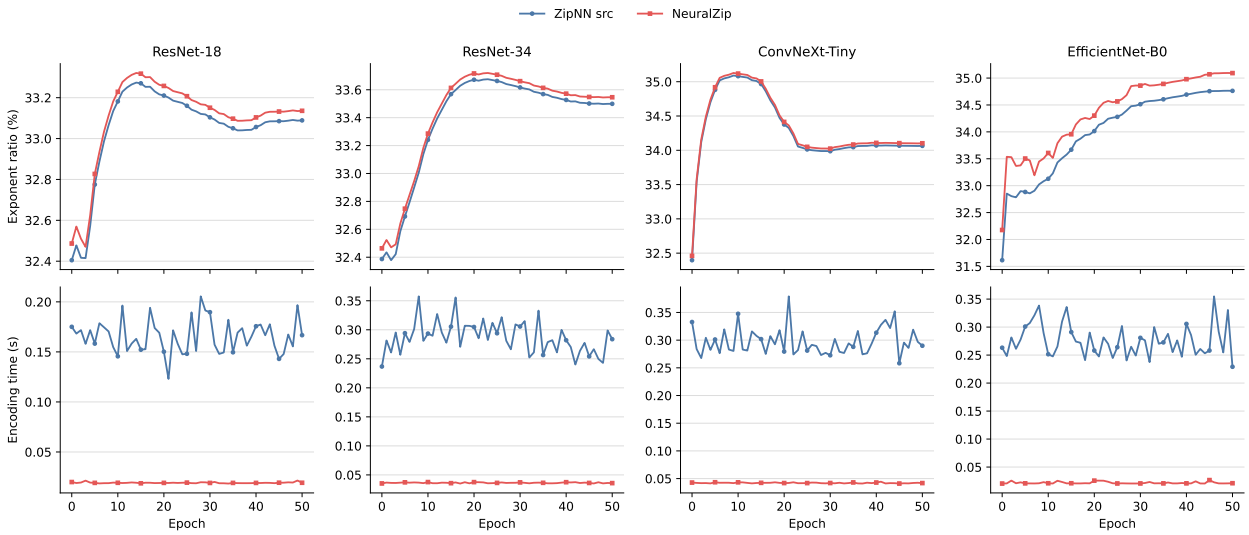


Figure 4.1: Checkpoint compression during training.

Across the trajectories, the exponent ratio spans 0.85–3.15 percentage points, and the largest adjacent-epoch change occurs within the first five epochs for both compressors. Their ratio curves remain closely aligned ($r = 0.994$–1.000), but the plan frozen at initialization increases the exponent-field ratio by 0.038–0.362 percentage points on average per trajectory. In exchange, NeuralZip is faster for every saved checkpoint: its encoding time is 0.018–0.044 s, compared with 0.123–0.379 s for ZipNN, yielding median per-trajectory speedups of 6.9×–12.8×. This is the expected time–size trade-off of freezing the plan: NeuralZip avoids rebuilding local Huffman codes, while ZipNN adapts its codes to each evolving checkpoint. Since every timing uses one measured repetition, these ranges are descriptive.

4.7 Lossy-Lossless Pipelines

We consider 4-bit quantized artifacts and distilled models. These cases require different interpretations. Distilled BF16 tensors retain the floating-point exponent field and can be analyzed with the components of NeuralZip. Packed integers do not contain that field, so applying the main compressor would be conceptually incorrect. Table 4.6 therefore reports a separate nibble-stream probe, not a comparison of the two main compressors. It reads the real tensor-data regions of pinned artifacts and samples at most 64 MiB from each authoritative weight shard. Coverage is 0.158–1.477% of the available tensor payload. These bytes represent the selected artifacts' tensor-data regions, which can mix quantized payloads with format-specific auxiliary tensors.

Every sampled byte is split into two four-bit symbols. The local strategy fits one Huffman code per chunk, whereas the shared strategy clusters nibble histograms with Jensen–Shannon distance and references shared codes, retaining a raw fallback. Sample I/O is outside the timer. Both variants use 12 threads, one verified warm-up, one measured repetition, and exact byte-for-byte round-trip checks. Shared-code setup takes 0.20–1.07 s and is excluded from encoding time. The probe ratio includes symbols, raw fallbacks, referenced codebooks, and per-block metadata; it is not a serialized full-checkpoint size.

	Encoding time (s)		Nibble-stream ratio (%)	
Model	Local Huffman	JS-shared Huffman	Local Huffman	JS-shared Huffman
Gemma 4 31B 4-bit [18]	0.889	**0.550**	**93.087**	96.923
Phi-4 14B 4-bit [43]	0.261	**0.232**	**88.592**	93.414
Qwen 3.5 27B 4-bit [52]	**0.115**	0.134	**88.582**	93.331
LATAM-GPT 70B 4-bit [31]	**0.112**	0.116	100.026	100.026

Table 4.6: Quantized nibble probe.

Across the probe, shared coding is 1.12×–1.62× faster in two cases and 1.04×–1.17× slower in the other two. For the three bitsandbytes samples, its ratio is 3.836–4.821 percentage points higher; for the GGUF sample, the difference is below 0.001 points and both strategies are dominated by raw fallback. Packed integers provide neither exponent histograms nor adjacent-exponent sequences, so the blockwise heterogeneity and local exponent redundancy exploited by NeuralZip are absent. The experiment therefore shows that transferring the coding strategy to nibbles does not transfer its statistical advantage. Because the probe covers only a bounded sample of each artifact, it does not establish full-checkpoint ratios.

	Encoding time (s)		Ratio (%)	
Model	ZipNN source	NeuralZip	ZipNN source	NeuralZip
Gemma 2 9B Distilled [29]	6.057	**3.875**	**32.813**	32.820
Qwen 2.5 7B Instruct Distilled [58]	4.892	**3.687**	**33.344**	33.408

Table 4.7: Distilled models.

Table 4.7 reproduces the main model-storage time pattern: direct encoding is 1.33×–1.56× faster after setup. The exponent ratios remain close to the source baseline, differing by only 0.007–0.064 percentage points, although both changes favor ZipNN in this run. Distillation therefore preserves the floating-point structure on which both compressors operate, but does not create an additional ratio gain here. Unlike packed quantized weights, these BF16 artifacts retain the exponent field and can still use the same blockwise and local analyses as the main model-storage setting.

Chapter 5

Conclusions

This thesis studied lossless model compression as a compact data structures problem. The starting point was *exponent skew*, the observation exploited by ZipNN that trained weights use a highly non-uniform subset of the exponents alphabet. Separating that field already captures a large fraction of the redundancy available to a lossless byte-level compressor.

Our analysis identified two additional properties. First, exponent distributions vary among parameter blocks but remain similar within subsets of them. We call this *blockwise heterogeneity*. It creates an opportunity to share a Huffman code among compatible blocks instead of rebuilding one for every small group. Second, using *local exponent redundancy*, packed exponents expose that dependence by representing frequent tuples as new symbols. We conclude that both effects depend on the architecture and are visible to different degrees in convolutional, transformer, diffusion, and sparse models.

NeuralZip combines these observations in one compressor. It groups tensors using Jensen–Shannon distance and builds Huffman codes indexed by cluster. Packed candidates are selected with a ratio-aware and time-aware policy, while every concrete chunk retains local-Huffman and raw fallbacks. The theoretical results are motivations for lower-bound comparisons, not a guarantee that NeuralZip must outperform ZipNN on every model; metadata remains a factor in the compression-ratio–time trade-off. The experimental evidence supports the same conclusion. Sharing Huffman codes can remove repeated construction work, and packed exponents can reduce the coded size when local patterns recur. Neither effect is uniform: some architectures exhibit stronger blockwise heterogeneity, while others present more local exponent redundancy.

In the final model-storage measurement, NeuralZip encoded faster in 26 of 34 model artifacts, ranging from near parity to 21.51×, and was 1.01×–1.58× slower in the remaining eight. The exponent-field ratio ranged from a 0.984-percentage-point reduction to a 0.224-point penalty. Training-checkpoint trajectories retained median direct speedups of 6.9×–12.8×, while the federated critical path was 3.48×–4.18× faster with a 0.057–0.068-point ratio penalty after charging the distributed plan once per client. These timings use one measured repetition after warm-up, so the variation within each range is descriptive rather than a statistical ordering. Every floating-point input in the comparison round-tripped bitwise, and no direct path retained an exact-target cache.

5.1 Limitations

Ratio gains over a strong field-aware baseline are often modest, and timing gains are not universal. In BF16, an exponent-oriented technique acts primarily on one of the two bytes per value. When the sign and mantissa byte is close to maximal entropy, even a substantial relative improvement in exponent coding produces a much smaller improvement in complete-model size. This places a natural ceiling on methods that leave those bits unchanged. Structural precomputation also adds a one-time family cost. Compatible checkpoints or family members with a complete exact name, dtype, and byte-group mapping can reuse the cluster map, selected modes, vocabularies, and codebooks, but every direct encode must still perform target-dependent extraction, reordering, tokenization, and any required local-code construction or fallback. Input-size-windowed native batching limits the lifetime of transient transformed workspace without retaining an exact-target cache; the final archive itself is necessarily kept as output. A one-shot archive without an existing compatible family plan must therefore pay the full structural-construction cost together with its complete direct-encoding cost; only a collection of compatible members can amortize that setup.

The current implementation is a CPU compressor for storage and transfer. It restores ordinary tensors before model execution and does not provide GPU kernels that consume the compressed representation directly. These limitations prevent the current results from establishing lower inference latency or reduced GPU memory use. Finally, blockwise heterogeneity and local exponent redundancy are empirical properties, not invariants of every trained model or numerical format. Quantized integers do not expose the same floating-point exponent field, and training procedures, dtypes, tensor layouts, and model families can change the observed distributions. The contribution is therefore evidence that this previously underused structure exists and can be exploited, together with an implementation that measures when the trade-off is favorable.

5.2 Future Work

We propose three directions for future work. First, a GPU implementation could retain NeuralZip archives in device memory and decode references to Huffman codes close to the matrix-multiplication kernels, following the systems direction of DFloat11 and Huff-LLM [74, 73]. This would require parallel decode tables, bounded synchronization, and a layout compatible with accelerator memory transactions. The second direction is *exponent-friendly* training or fine-tuning. An optimizer or regularizer could encourage a smaller effective exponent alphabet or stronger local regularity without changing the external floating-point format. The central question is whether such a constraint can improve lossless compressibility without functioning as an implicit quantizer or degrading the training objective. Finally, one could use the observed structure beyond compression. Blockwise exponent distributions provide compact signatures of how modules use numerical scale, while changes in packed-exponent statistics may reveal local training dynamics. These signals could be studied for model comparison, interpretability, anomaly detection, or uncertainty quantification.

Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai. *arXiv preprint arXiv:2506.02153*, 2025.

[3] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

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

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Tejalal Choudhary, Vipul Kumar Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53(7):5113–5155, 2020.

[7] Yann Collet and Murray S. Kucherawy. Zstandard compression and the application/zstd media type. RFC 8478, RFC Editor, 2018. Obsoleted by RFC 8878.

[8] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2 edition, 2006.

[9] Damai Dai, Chengqi Deng, Chenggang Zhao, et al. DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*, 2024.

[10] Peter Deutsch and Jean-Loup Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, RFC Editor, 1996.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[12] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[13] Paolo Ferragina, Giovanni Manzini, Travis Gagie, Dominik Köppl, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. Improving matrix-vector multiplication via lossless grammar-compressed matrices. *Proceedings of the VLDB Endowment*, 15(10):2175–2187, 2022.

[14] Gemma Team. Gemma 3. `https://goo.gle/Gemma3Report`, 2025.

[15] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[17] Google DeepMind. Gemma 4 26B-A4B model card. `https://huggingface.co/google/gemma-4-26B-A4B`, 2026. Accessed 2026-07-18.

[18] Google DeepMind. Gemma 4 31B model card. `https://huggingface.co/google/gemma-4-31B`, 2026. Accessed 2026-06-09.

[19] Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

[20] Robert M. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Or Ozeri, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, et al. Zipnn: Lossless compression for ai models. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pages 186–198. IEEE, 2025.

[23] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[24] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[25] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[26] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[27] Nikhil Kandpal, Brian Lester, Mohammed Muqeeth, Anisha Mascarenhas, Monty Evans, Vishal Baskaran, Tenghao Huang, Haokun Liu, and Colin Raffel. Git-theta: A git extension for collaborative development of machine learning models. *arXiv preprint arXiv:2306.04529*, 2023.

[28] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. ChatGPT for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103:102274, 2023.

[29] kazuyamaa. gemma-2-9b-opd-Distill-v001 model card. `https://huggingface.co/kazuyamaa/gemma-2-9b-opd-Distill-v001`, 2026. Hugging Face model card; accessed 2026-07-17.

[30] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[31] LatamGPT. Llama 3.1 70B LatamGPT SFT 1.0 model card. `https://huggingface.co/latam-gpt/Llama-3.1-70B-LatamGPT-SFT-1.0`, 2026. Accessed 2026-06-17.

[32] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605, 1989.

[33] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[34] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[35] Liquid AI. LFM2 technical report. *arXiv preprint arXiv:2511.23404*, 2025.

[36] Liquid AI. LFM2.5-8B-A1B: Personal assistant on your laptop. `https://huggingface.co/LiquidAI/LFM2.5-8B-A1B`, 2026. Model card; accessed 2026-07-17.

[37] Defu Liu, Yixiao Zhu, Zhe Liu, Yi Liu, Changlin Han, Jinkai Tian, Ruihao Li, and Wei Yi. A survey of model compression techniques: past, present, and future. *Frontiers in Robotics and AI*, 12:1518965, 2025.

[38] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022.

[39] Xiaolong Ma, Minghai Qin, Fei Sun, Zejiang Hou, Kun Yuan, Yi Xu, Yanzhi Wang, Yen-Kuang Chen, Rong Jin, and Yuan Xie. Effective model sparsification by scheduled grow-and-prune methods. In *International Conference on Learning Representations*, 2022.

[40] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

[41] Meta AI. Llama 3.2 1B model card. `https://huggingface.co/meta-llama/Llama-3.2-1B`, 2024. Accessed 2026-07-18.

[42] Meta AI. Llama 3.2 3B model card. `https://huggingface.co/meta-llama/Llama-3.2-3B`, 2024. Accessed 2026-07-18.

[43] Microsoft. Phi-4 model card. `https://huggingface.co/microsoft/phi-4`, 2024. Accessed 2026-06-09.

[44] Mistral AI. Mistral-Nemo-Base-2407 model card. `https://huggingface.co/mistralai/Mistral-Nemo-Base-2407`, 2024. Accessed 2026-07-18.

[45] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, et al. OLMoE: Open mixture-of-experts language models. *arXiv preprint arXiv:2409.02060*, 2024.

[46] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[47] NVIDIA. Llama-3.1-Nemotron-Nano-8B-v1 model card. `https://huggingface.co/nvidia/Llama-3.1-Nemotron-Nano-8B-v1`, 2025. Post-trained derivative of Meta Llama 3.1 8B Instruct; accessed 2026-07-18.

[48] powturbo. Turbo-Range-Coder: TurboRC. `https://github.com/powturbo/Turbo-Range-Coder`, 2026. Accessed 2026-07-10.

[49] Qwen Team. Qwen1.5-MoE-A2.7B model card. `https://huggingface.co/Qwen/Qwen1.5-MoE-A2.7B`, 2024. Accessed 2026-07-18.

[50] Qwen Team. Qwen3-30B-A3B model card. `https://huggingface.co/Qwen/Qwen3-30B-A3B`, 2025. Accessed 2026-07-18.

[51] Qwen Team. Qwen3.5-0.8B-Base model card. `https://huggingface.co/Qwen/Qwen3.5-0.8B-Base`, 2026. Accessed 2026-07-16.

[52] Qwen Team. Qwen3.5-27B model card. `https://huggingface.co/Qwen/Qwen3.5-27B`, 2026. Accessed 2026-06-09.

[53] Qwen Team. Qwen3.5-2B-Base model card. `https://huggingface.co/Qwen/Qwen3.5-2B-Base`, 2026. Accessed 2026-07-16.

[54] Qwen Team. Qwen3.5-35B-A3B model card. `https://huggingface.co/Qwen/Qwen3.5-35B-A3B`, 2026. Accessed 2026-07-18.

[55] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[56] Jorma J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[57] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Bjorn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[58] RZ412. Qwen2.5-7B-Instruct-S1K-DeepSeek-Distilled model card. `https://huggingface.co/RZ412/Qwen2.5-7B-Instruct-S1K-DeepSeek-Distilled`, 2025. Hugging Face model card; accessed 2026-07-17.

[59] Rajarshi Saha, Naomi Sagan, Varun Srivastava, Andrea Goldsmith, and Mert Pilanci. Compressing large language models using low rank and low precision decomposition. *arXiv preprint arXiv:2405.18886*, 2024.

[60] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 5 edition, 2017.

[61] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017.

[62] Yikang Shen, Zhen Guo, Tianle Cai, and Zengyi Qin. JetMoE: Reaching Llama2 performance with 0.1m dollars. *arXiv preprint arXiv:2404.07413*, 2024.

[63] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. Large language models encode clinical knowledge. *Nature*, 620:172–180, 2023.

[64] Sidhant Sundrani, Francesco Tudisco, and Pasquale Minervini. Low-rank compression of language models via differentiable rank selection. *arXiv preprint*, 2024. Submitted to ICLR 2025.

[65] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[66] Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin

Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Allyson Ettinger, Michal Guerquin, David Heineman, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Jake Poznanski, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 furious. *arXiv preprint arXiv:2501.00656*, 2024.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[68] Leandro von Werra, Lewis Tunstall, Abhishek Thakur, Alexandra Sasha Luccioni, Tristan Thrush, Aleksandra Piktus, Felix Marty, Nazneen Rajani, Victor Mustar, Helen Ngo, et al. Evaluate & evaluation on the hub: Better best practices for data and model measurements. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 128–136, 2022.

[69] Daniel G. Waddington and Cornel Constantinescu. Lossless compression for llm tensor incremental snapshots. *arXiv preprint arXiv:2505.09810*, 2025.

[70] Robert Washbourne, Rishi Iyer, Tomas Figliolia, Henry Zheng, Ryan Lorig-Roach, Sungyeon Yang, Pritish Yuvraj, Quentin Anthony, Yury Tokpanov, Xiao Yang, Ganesh Nanduru, Stephen Ebert, Praneeth Medepalli, Skyler Szot, Srivatsan Rajagopal, Alex Ong, Bhavana Mehta, and Beren Millidge. ZAYA1-8B technical report. *arXiv preprint arXiv:2605.05365*, 2026.

[71] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. A survey on knowledge distillation of large language models. *arXiv preprint arXiv:2402.13116*, 2024.

[72] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[73] Patrick Yubeaton, Tareq Mahmoud, Shehab Naga, Pooria Taheri, Tianhua Xia, Arun George, Yasmein Khalil, Sai Qian Zhang, Siddharth Joshi, Chinmay Hegde, et al. Huff-llm: End-to-end lossless compression for efficient llm inference. *arXiv preprint arXiv:2502.00922*, 2025.

[74] Tianyi Zhang, Mohsen Hariri, Shaochen Henry Zhong, Vipin Chaudhary, Yang Sui, Xia Hu, and Anshumali Shrivastava. 70% size, 100% accuracy: Lossless llm compression for efficient gpu inference via dynamic-length float (dfloat11). In *Advances in Neural Information Processing Systems*, 2025.

[75] Yushun Zhang, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. Why Transformers Need Adam: A Hessian Perspective. In *Advances in Neural Information Processing Systems*, volume 37, pages 131786–131823, 2024.

[76] Li Zhou, Jianfeng Gao, Di Li, and Heung-Yeung Shum. The design and implementation of XiaoIce, an empathetic social chatbot. *Computational Linguistics*, 46(1):53–93, 2020.

[77] Michael H. Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

Appendix A

Background

Notation: We use log as the logarithm in base 2.

A.1 Foundation Models

A foundation model is a large neural model trained on broad data and later adapted or prompted for many downstream tasks [3]. This section introduces the tensor operations and architectural blocks used by foundation models.

Definition A.1 (Tensor) *Let $\mathcal{A}$ be a set of scalar values. A tensor of order k and shape $(d_1, \ldots, d_k)$ over $\mathcal{A}$ is an element*

$$X \in \mathcal{A}^{d_1 \times \cdots \times d_k}.$$

Order-1 tensors are vectors, order-2 tensors are matrices, and higher-order tensors represent structured arrays such as images, convolutional kernels, or batches of activations.

Definition A.2 (Tensor operation) *A tensor operation is a map*

$$\Phi : \mathcal{A}^{d_1 \times \cdots \times d_k} \times \mathcal{A}^{r_1 \times \cdots \times r_m} \longrightarrow \mathcal{A}^{q_1 \times \cdots \times q_s}.$$

Neural computations combine entrywise arithmetic, tensor contractions, reshapes, concatenations, permutations, reductions, and nonlinearities. Learned layers are tensor operations with one or more parameter tensors as arguments.

A.1.1 Feed-Forward Networks

A feed-forward network applies tensor operations in a fixed order from input to output. For a batch $X \in \mathcal{A}^{B \times d_{\text{in}}}$, a dense layer has the form

$$Y = XW^\top + \mathbf{1}b^\top,$$

where $W \in \mathcal{A}^{d_{\text{out}} \times d_{\text{in}}}$, $b \in \mathcal{A}^{d_{\text{out}}}$, and $\mathbf{1}$ is the length-B all-ones vector. Multi-layer perceptrons alternate such affine maps with activation functions:

$$\text{MLP}(x) = W_2\sigma(W_1x + b_1) + b_2.$$

This simple block is used directly in classical neural networks and also appears inside transformer architectures.

A.1.2 Activation Functions

Activation functions introduce nonlinearity between affine tensor operations. If $g : \mathcal{A} \to \mathcal{A}$ is a scalar function, its elementwise extension to a tensor X is

$$\sigma(X)_{i_1,\ldots,i_k} = g(X_{i_1,\ldots,i_k}).$$

Three common examples are

$$\text{ReLU}(x) = \max(0, x), \qquad \text{sigmoid}(x) = (1 + e^{-x})^{-1}, \qquad \text{GELU}(x) = x\Phi(x),$$

where Φ is the cumulative distribution function of the standard normal distribution. Other activation-like operations, such as softmax, normalize a vector or a tensor slice rather than acting independently on each entry. These operations usually have no learned parameter tensors, but they are part of the composition that defines f_θ.

A.1.3 Convolutional Neural Networks

Convolutional neural networks use learned filters that are applied repeatedly across spatial positions [21]. For an input image tensor $X \in \mathcal{A}^{C_{\text{in}} \times H \times W}$ and a kernel tensor $K \in \mathcal{A}^{C_{\text{out}} \times C_{\text{in}} \times R \times S}$, a two-dimensional convolution can be written as

$$Y_{c,u,v} = b_c + \sum_{c'=1}^{C_{\text{in}}} \sum_{i=1}^{R} \sum_{j=1}^{S} K_{c,c',i,j} X_{c',u+i,v+j}.$$

The same kernel entries are reused at many locations.

A.1.4 Transformers

Transformers are the dominant architecture behind modern language models [67]. Let $X \in \mathcal{A}^{T \times d}$ be a sequence of T token representations. A single self-attention head forms

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V,$$

and computes

$$\text{Attn}(X) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

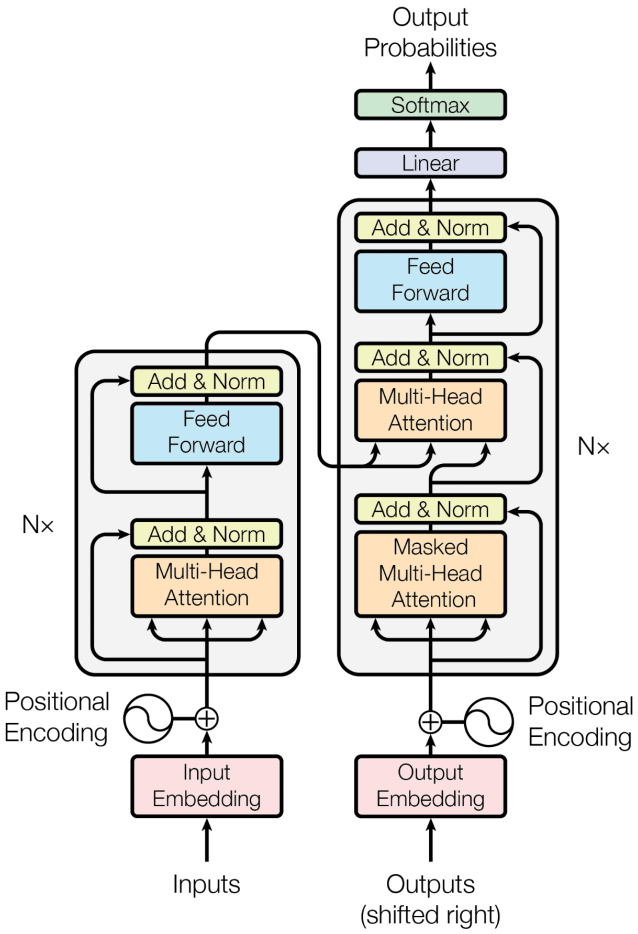


Figure A.1: Transformer encoder–decoder architecture.

Multi-head attention repeats this operation with several parameter sets, concatenates the resulting tensors, and applies an output projection. A transformer block combines attention, residual connections, normalization, and a feed-forward block:

$$U = \mathrm{LayerNorm}(X + \mathrm{MHA}(X)), \qquad Y = \mathrm{LayerNorm}(U + \mathrm{MLP}(U)).$$

Thus a transformer is again a composition of tensor operations with learned parameter tensors in the attention projections, feed-forward layers, embeddings, and normalization parameters. Figure A.1 shows the high-level transformer block structure [67].

The tensor-composition view is deliberately independent of the application domain. Language, vision, audio, and multimodal systems differ in their input representation and training objective, but their learned computation is built from the same kinds of tensor operations introduced above.

A.1.5 Types of Foundation Models

While large language models (LLMs) are the best-known examples, foundation models encompass a broader range of AI systems. The following categories differ mainly in their input

and output modalities and in how they combine the feed-forward, convolutional, attention, and embedding blocks described above.

Large Language Models are trained to model token sequences. Given a prefix $x_1, \ldots, x_t$, an autoregressive LLM estimates

$$p_\theta(x_{t+1} \mid x_1, \ldots, x_t),$$

and uses that distribution to generate, rank, or complete text. They are typically built from token embeddings, repeated transformer blocks, and an output layer that maps the final hidden state to a distribution over the vocabulary.

Vision Models process images or video frames to produce labels, dense predictions, detections, or visual representations. Convolutional vision models build hierarchical features by applying local filters across spatial positions. Vision transformers instead divide an image into patches, map each patch to an embedding, and process the resulting sequence with transformer blocks.

Multimodal Models combine inputs from several modalities, such as text, images, audio, or video. A common design uses one encoder per modality, projection layers that place the representations in a shared latent space, and attention-based fusion or decoder modules that produce the desired output. For example, a visual question-answering model may encode an image, encode a text question, use cross-attention to combine both representations, and decode a textual answer.

Audio and Speech Models are trained on waveform samples, spectrograms, or discrete audio tokens. They support tasks such as speech transcription, speaker identification, speech synthesis, and music generation. Their architectures often use convolutional front ends over time or frequency, transformer blocks over frames or audio tokens, and decoders that map latent representations back to text or audio.

A.2 Model Compression

Model compression is a broad term for techniques that reduce one or more costs of a trained model, including storage, memory traffic, arithmetic operations, latency, and energy consumption [6, 37]. These objectives are related but not equivalent. A representation can occupy fewer bytes without accelerating inference, and a model can execute fewer operations while retaining a large parameter file. It is therefore useful to distinguish the transformation applied to the model, whether additional training is required, and whether the original parameters remain exactly recoverable.

Quantization Quantization replaces high-precision numerical values with elements of a smaller discrete set [20, 15]. In a simple affine quantizer, a floating-point weight w_i is represented by an integer

$$q_i = \mathrm{clip}\left(\mathrm{round}\left(\frac{w_i}{s}\right) + z, q_{\min}, q_{\max}\right), \qquad \widehat{w}_i = s(q_i - z),$$

where s is a scale and z is a zero point. The integers may use eight, four, or fewer bits, with scales and zero points stored per tensor, channel, or group. Post-training quantization estimates these parameters after training, whereas quantization-aware training lets the model adapt to simulated low-precision error. Methods may quantize weights alone or weights and activations; practical speedups require compatible hardware kernels. Since multiple values map to the same integer, dequantization recovers $\widehat{w}_i$, not the original w_i. Activation-aware methods refine this mapping using observed activation statistics [33].

Pruning and sparsification Pruning removes parameters or computational structures that are estimated to contribute little to the model output [32, 77]. Unstructured pruning applies a binary mask to individual weights, selected by magnitude, saliency, gradients, or schedules that alternate pruning and regrowth [39]. It can reach high sparsity, but indices or masks must be stored and irregular zeros require specialized sparse kernels. Structured pruning instead removes complete channels, attention heads, neurons, filters, or blocks. It maps more naturally to dense hardware, although it offers less freedom in choosing individual weights. Both forms commonly use fine-tuning to recover quality, and neither can reconstruct removed values unless they are stored separately.

Knowledge distillation Knowledge distillation trains a smaller *student* model to reproduce the behavior of a larger *teacher* [19, 71]. The training objective combines the ordinary task loss with imitation of softened teacher outputs, intermediate features, attention maps, or relations among examples. The student may have fewer layers, smaller hidden dimensions, or a different architecture. Distillation therefore transfers behavior rather than encoding teacher parameters: it produces a newly trained model and cannot be decoded back into the original teacher.

Low-rank factorization, parameter sharing, and compact architectures Matrix and tensor decompositions reduce redundancy by replacing a dense operator with smaller

factors. For $W \in \mathbb{R}^{m \times n}$, a rank-$r$ approximation has the form

$$W \approx UV, \qquad U \in \mathbb{R}^{m \times r}, \quad V \in \mathbb{R}^{r \times n},$$

and stores $r(m+n)$ values instead of mn when $r \ll \min(m, n)$ [59, 64]. The approximation can be computed from a trained matrix and fine-tuned, or used as the training parameterization. Parameter sharing instead ties weights across layers or repeated operations, while compact architectures use narrower layers, efficient operators, or architecture search from the outset. These approaches can reduce storage and computation, but alter the parameterization and generally require training or adaptation. Unless a decomposition is exact and full-rank, the original dense tensors are not recoverable.

Sparse and selectively activated models Conditional computation reduces the number of parameters used for a particular input. Mixture-of-experts models, for example, route each token to only a small subset of experts [61, 12]. This can make active computation sparse while total capacity grows. It does not necessarily reduce storage: every expert remains part of the model unless compressed or removed separately. Selective activation primarily targets computation and memory movement, and may increase the complete parameter artifact.

Hybrid methods and exact recovery These techniques are often combined: a student can be pruned and quantized, a low-rank representation can use low-precision factors, and any resulting artifact can be entropy-coded. Hybrid pipelines must report size, quality, and runtime because gains are not necessarily additive. They must also identify the reference representation. Lossless compression after quantization reconstructs the quantized integers exactly, but not the full-precision model that preceded quantization.

A.3 Floating-point representations

The learned parameters of a neural model are stored as finite tensors. For a floating-point format with one sign bit, E exponent bits, and M mantissa bits, a normalized value has the form

$$x = (-1)^s \left(1 + \frac{m}{2^M}\right) 2^{e-\beta},$$

where $s \in \{0, 1\}$ is the sign, $e \in \{0, \ldots, 2^E - 1\}$ is the stored exponent, $m \in \{0, \ldots, 2^M - 1\}$ is the mantissa, and β is the exponent bias. Special exponent values represent zeros, subnormal values, infinities, and NaNs [16]. The most popular formats include FP32 $(1, 8, 23)$, BF16 $(1, 8, 7)$, and FP16 $(1, 5, 10)$, where each tuple gives the number of sign, exponent, and mantissa bits.

Example (FP32 representation) Consider $x = -13.25$. Its magnitude is

$$13.25 = (1101.01)_2 = (1.10101)_2\, 2^3.$$

FP32 uses bias $\beta = 127$, so $e = 3 + 127 = 130 = (10000010)_2$. The negative sign gives $s = 1$, and the 23-bit mantissa is $m = (10101000000000000000000)_2 = 5{,}505{,}024$. Thus,

$$\underbrace{\texttt{1}}_{s}\ \underbrace{\texttt{10000010}}_{e}\ \underbrace{\texttt{10101000000000000000000}}_{m} = \texttt{0xC1540000},$$

and

$$(-1)^1 \left(1 + \frac{5{,}505{,}024}{2^{23}}\right) 2^{130-127} = -13.25.$$

Example (BF16 representation) Consider $x = 0.15625$. In binary,

$$0.15625 = (0.00101)_2 = (1.01)_2\, 2^{-3}.$$

BF16 uses the same eight-bit exponent and bias $\beta = 127$ as FP32, but only seven mantissa bits. Hence $s = 0$, $e = -3 + 127 = 124 = (01111100)_2$, and $m = (0100000)_2 = 32$. The stored word is

$$\underbrace{\texttt{0}}_{s}\ \underbrace{\texttt{01111100}}_{e}\ \underbrace{\texttt{0100000}}_{m} = \texttt{0x3E20}.$$

The represented value is therefore

$$(-1)^0 \left(1 + \frac{32}{2^7}\right) 2^{124-127} = 0.15625.$$

A.4 Compact Data Structures

Lossless compression relies on assigning short binary descriptions to common events and longer descriptions to rare events. Entropy formalizes the best possible average description length under an assumed distribution.

Definition A.3 (Worst-case entropy) *Let U be a finite set. If every element of U must be represented by a fixed-length binary code, the minimum code length is governed by*

$$H_{wc}(U) = \log |U|.$$

Thus any fixed-length binary representation of U needs at least $\lceil \log |U| \rceil$ bits per element.

Example Let $U = \{A, B, C, D, E\}$. Then $H_{wc}(U) = \log 5 \approx 2.322$, so any fixed-length binary code requires at least three bits per symbol. For example, the code $A = 000$, $B = 001$, $C = 010$, $D = 011$, and $E = 100$ is valid, but it cannot use only two bits because two bits represent at most four symbols.

A.4.1 Shannon Entropy

Definition A.4 (Shannon entropy) *Let X be a discrete random variable over alphabet Σ with probability mass function $p(a) = \Pr[X = a]$. Its Shannon entropy is [8]*

$$H(X) = -\sum_{a \in \Sigma} p(a) \log p(a).$$

Terms with $p(a) = 0$ are treated as zero. Entropy is measured in bits.

If X is uniform over Σ, then $H(X) = \log |\Sigma|$, matching the worst-case fixed-length view. If the distribution is skewed, then $H(X)$ can be much smaller, which means that variable-length coding can reduce the average number of bits per symbol.

Example If X is uniform over $\{A, B, C, D\}$, then $H(X) = 2$ bits. If instead

$$p(A) = \frac{1}{2}, \qquad p(B) = \frac{1}{4}, \qquad p(C) = p(D) = \frac{1}{8},$$

then

$$H(X) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75$$

bits. The same alphabet has smaller entropy because the distribution is no longer uniform.

A.4.2 Entropy of a tuple

Definition A.5 (Joint entropy) *For discrete random variables $X_1, \ldots, X_n$, the entropy of the tuple $(X_1, \ldots, X_n)$ is the Shannon entropy of the joint random variable. For two variables,*

$$H(X, Y) = -\sum_{x,y} \Pr[X = x, Y = y] \log \Pr[X = x, Y = y].$$

Joint entropy satisfies the subadditivity inequality

$$H(X, Y) \leq H(X) + H(Y),$$

with equality when X and Y are independent. More generally, $H(X_1, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i)$.

Example Let X be a fair bit. If $Y = X$, then the pair (X, Y) only takes the values $(0, 0)$ and $(1, 1)$, each with probability $1/2$, so $H(X, Y) = 1$ bit. If Y is an independent fair bit, then the four pairs are equally likely and $H(X, Y) = 2$ bits. This distinction is important for packed exponents: neighboring symbols may be statistically dependent, so coding them jointly can expose structure that is invisible to independent symbol counts.

A.4.3 Empirical Entropy

Definition A.6 (Zero-order empirical entropy) *Let $S[1..n]$ be a finite sequence over alphabet Σ. For symbol $a \in \Sigma$, let n_a and $\widehat{p}_a = n_a/n$ be its count and empirical probability in S. The zero-order empirical entropy of S is*

$$H_0(S) = -\sum_{a \in \Sigma} \widehat{p}_a \log \widehat{p}_a.$$

Symbols with $n_a = 0$ contribute zero.

In model compression, the true distribution of weight fields is unknown. We therefore use the empirical distribution of the model weights. The corresponding ideal zero-order cost is $nH_0(S)$ bits. It is called zero-order because it ignores symbol order and only measures the frequency of each symbol.

Example Consider the sequence $S = (4, 4, 4, 5)$. Its empirical probabilities are $\widehat{p}_4 = 3/4$ and $\widehat{p}_5 = 1/4$, so

$$H_0(S) = -\frac{3}{4} \log \frac{3}{4} - \frac{1}{4} \log \frac{1}{4} \approx 0.811$$

bits per symbol. The ideal zero-order cost is therefore $4H_0(S) \approx 3.245$ bits, which is below the four bits required when each of the four sequence positions is represented by one fixed bit.

A.4.4 Coding

Definition A.7 (Binary code and prefix-free code) *A binary code for an alphabet Σ is a map $C : \Sigma \to \{0, 1\}^*$. Its code length for symbol a is $\ell(a) = |C(a)|$. The code is prefix-free if no codeword is the prefix of another codeword.*

Prefix-free codes are useful because a concatenation of codewords can be decoded unambiguously from left to right [60]. If a symbol a has code length $\ell(a)$ and empirical probability $\widehat{p}_a$, the average code length is

$$L = \sum_{a \in \Sigma} \widehat{p}_a \ell(a).$$

For any prefix-free code, the expected length is lower-bounded by entropy. Practical compressors also store metadata, such as Huffman codes, offsets, and block headers, so the measured compressed size is the entropy-coded payload plus implementation overhead.

Example For $\Sigma = \{A, B, C\}$, the code $C(A) = 0$, $C(B) = 10$, and $C(C) = 11$ is prefix-free. The encoded string 01011 decodes uniquely as A, B, C. If the probabilities are $p(A) = 1/2$, $p(B) = p(C) = 1/4$, the average length is

$$L = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{4} \cdot 2 = 1.5$$

bits per symbol.

A.4.5 Huffman Codes

Definition A.8 (Huffman code) *A Huffman code is a prefix-free code obtained by repeatedly merging the two least frequent symbols or subtrees until one binary tree remains. More frequent symbols are placed closer to the root and therefore receive shorter codewords.*

Huffman coding is a classical method for constructing an optimal prefix-free code for a known discrete distribution [24, 60, 46]. Given empirical counts n_a, Huffman coding assigns shorter codewords to more frequent symbols. If $\ell(a)$ is the Huffman length assigned to a, then

$$H_0(S) \leq \sum_{a \in \Sigma} \widehat{p}_a \ell(a) < H_0(S) + 1.$$

Thus Huffman coding is close to the zero-order empirical entropy, but it can still lose up to one bit per coded symbol because code lengths must be integers.

For an exponent sequence S, the Huffman payload length is

$$L_{\mathrm{Huff}}(S) = \sum_{a \in \Sigma_E} n_a \ell(a),$$

before metadata.

Algorithm A.1 gives the standard construction. It starts with one leaf per symbol and repeatedly merges the two minimum-weight trees. The resulting root-to-leaf paths define the codewords: following a left edge appends 0, and following a right edge appends 1. This merge process is the Huffman construction described in Navarro's treatment of entropy coding [46].

```
Q ← ∅
for each symbol a ∈ Σ do
    Insert Leaf(a, n_a) into Q
end for
while |Q| > 1 do
    u ← ExtractMin(Q)
    v ← ExtractMin(Q)
    z ← Node(weight(u) + weight(v), u, v)
    Insert z into Q
end while
T ← ExtractMin(Q)
for each leaf a in T do
    ℓ(a) ← Depth_T(a)
    C(a) ← RootToLeafBits_T(a)
end for
return T, C, ℓ
```

Algorithm A.1: Huffman tree construction

Definition A.9 (Canonical Huffman code) *Fix an order on the symbols of Σ. A canonical Huffman code for a set of Huffman lengths $\ell(a)$ is the prefix-free code obtained by sorting symbols by increasing $(\ell(a), a)$ and then assigning consecutive binary values, left-shifting whenever the next symbol has a longer length. It preserves every length $\ell(a)$ and therefore preserves the payload cost $\sum_{a\in\Sigma} n_a \ell(a)$.*

The concrete shape of a Huffman tree is not unique: ties can be broken in different ways and the left and right children of internal nodes can be swapped. A canonical Huffman code removes this ambiguity while preserving the same code lengths. The compressed payload length is unchanged because every symbol keeps its Huffman length, but the decoder can store a compact representation instead of a pointer-based tree. For model-weight compression, this matters because the metadata can record the observed symbols and their code lengths, or equivalent length tables, instead of serializing all tree edges or all explicit codewords. Following Navarro's canonical representation [46], Algorithm A.2 builds the canonical codes and the arrays L, F, and First used to identify the first code of each length.

```
T, C, ℓ ← BuildHuffmanTree(Σ, {n_a}_{a∈Σ})
A ← SortBy(Σ, (ℓ(a), a))
next ← 0
prev ← ℓ(A[1])
for each symbol a in A do
    next ← next ≪ (ℓ(a) − prev)
    C_can(a) ← Binary(next, ℓ(a))
    next ← next + 1
    prev ← ℓ(a)
end for
L ← A
h ← max_{a∈Σ} ℓ(a)
for d ← 1 to h do
    F[d] ← 0
    First[d] ← 0
end for
for i ← |Σ| downto 1 do
    d ← ℓ(L[i])
    F[d] ← i
    First[d] ← C_can(L[i])
end for
for d ← h − 1 downto 1 do
    if F[d] = 0 then
        F[d] ← F[d + 1]
        First[d] ← ⌊First[d + 1]/2⌋
    end if
end for
return C_can, L, F, First
```

Algorithm A.2: Canonical Huffman construction

A.4.6 Entropy Inequalities

Theorem A.10 (Concavity of entropy) *Let $p_1, \ldots, p_m$ be probability distributions over the same finite alphabet Σ, and let $\lambda_i \geq 0$ with $\sum_{i=1}^{m} \lambda_i = 1$. If*

$$p = \sum_{i=1}^{m} \lambda_i p_i,$$

then

$$H(p) \geq \sum_{i=1}^{m} \lambda_i H(p_i).$$

Proof. Define $f(0) = 0$ and $f(x) = -x \log x$ for $x > 0$. Since $f''(x) < 0$ for $x > 0$, the function is concave on $[0, 1]$, using its continuous extension at zero. For each $a \in \Sigma$, Jensen's

inequality gives

$$f\left(\sum_{i=1}^{m} \lambda_i p_i(a)\right) \geq \sum_{i=1}^{m} \lambda_i f(p_i(a)).$$

Summing over the alphabet yields

$$H(p) = \sum_{a \in \Sigma} f(p(a)) \geq \sum_{i=1}^{m} \lambda_i \sum_{a \in \Sigma} f(p_i(a)) = \sum_{i=1}^{m} \lambda_i H(p_i).$$

□

Theorem A.11 (Subadditivity of joint entropy) *Let $X_1, \ldots, X_n$ be discrete random variables with joint probability mass function P and marginal probability mass functions $P_1, \ldots, P_n$. Then*

$$H(X_1, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i).$$

Proof. For $\mathbf{x} = (x_1, \ldots, x_n)$, define the product distribution

$$Q(\mathbf{x}) = \prod_{i=1}^{n} P_i(x_i).$$

By the non-negativity of the Kullback–Leibler divergence,

$$\begin{aligned} D_{\mathrm{KL}}(P \parallel Q) &= \sum_{\mathbf{x}} P(\mathbf{x}) \log \frac{P(\mathbf{x})}{Q(\mathbf{x})} \\ &= -H(X_1, \ldots, X_n) - \sum_{i=1}^{n} \sum_{x_i} P_i(x_i) \log P_i(x_i) \\ &= -H(X_1, \ldots, X_n) + \sum_{i=1}^{n} H(X_i) \geq 0. \end{aligned}$$

Rearranging gives the claimed inequality. □

Appendix B

Mathematical Proofs

B.1 Proof of Lemma 3.1

Detailed proof. Let

$$e = (e_1, e_2, \ldots, e_N), \qquad e_i \in \Sigma_E,$$

be the sequence of exponent symbols obtained from floating-point model weights, where $\Sigma_E = \{0, \ldots, 2^E - 1\}$ is the exponent alphabet. For any ordered exponent sequence S, let $|S|$ denote its length and let

$$n_S(a) = |\{x \in S : x = a\}|$$

be the number of occurrences of exponent symbol $a \in \Sigma_E$ in S. Its empirical distribution is

$$\widehat{p}_S(a) = \frac{n_S(a)}{|S|},$$

and its zero-order empirical entropy is

$$H_0(S) = -\sum_{a \in \Sigma_E} \widehat{p}_S(a) \log_2 \widehat{p}_S(a).$$

For a probability vector p over Σ_E, write

$$H(p) = -\sum_{a \in \Sigma_E} p(a) \log_2 p(a).$$

Let

$$\mathcal{P} = \{S_1, S_2, \ldots, S_r\}$$

be a partition of the exponent sequence into ordered subsequences. A tensor-induced refinement $\mathcal{P}_T$ is obtained by splitting each sequence S_j into subsequences

$$S_{j,1}, S_{j,2}, \ldots, S_{j,t_j},$$

where each subsequence contains only symbols belonging to the same parameter block and preserves the original order. Empty subsequences are ignored. Thus,

$$|S_j| = \sum_{q=1}^{t_j} |S_{j,q}|.$$

For a fixed sequence S_j, define

$$\lambda_{j,q} = \frac{|S_{j,q}|}{|S_j|}.$$

The empirical distribution of S_j is the weighted average of the empirical distributions of its subsequences:

$$\widehat{p}_{S_j}(a) = \sum_{q=1}^{t_j} \lambda_{j,q} \widehat{p}_{S_{j,q}}(a), \qquad a \in \Sigma_E.$$

By Theorem A.10, Shannon entropy is concave, and hence

$$H_0(S_j) = H\left(\sum_{q=1}^{t_j} \lambda_{j,q} \widehat{p}_{S_{j,q}}\right) \geq \sum_{q=1}^{t_j} \lambda_{j,q} H_0(S_{j,q}).$$

Multiplying by $|S_j|$ gives

$$|S_j| H_0(S_j) \geq \sum_{q=1}^{t_j} |S_{j,q}| H_0(S_{j,q}).$$

Summing over every sequence in $\mathcal{P}$,

$$C_0(\mathcal{P}) = \sum_{j=1}^{r} |S_j| H_0(S_j) \geq \sum_{j=1}^{r} \sum_{q=1}^{t_j} |S_{j,q}| H_0(S_{j,q}) = C_0(\mathcal{P}_T).$$

Therefore $C_0(\mathcal{P}_T) \leq C_0(\mathcal{P})$. □

B.2 Proof of Lemma 3.2

Detailed proof. Let

$$S = (x_1, x_2, \ldots, x_t), \qquad x_i \in \Sigma_E,$$

be an ordered exponent sequence. For a fixed integer $n \geq 1$, define

$$q = \left\lfloor \frac{t}{n} \right\rfloor,$$

and let the complete prefix be

$$S^{[n]} = (x_1, x_2, \ldots, x_{qn}).$$

The remaining symbols

$$T_n(S) = (x_{qn+1}, \ldots, x_t)$$

form a raw tail of length $t - qn < n$ and are not included in the ideal entropy comparison.

The non-overlapping transform into packed exponents is

$$G_n(S) = (g_1, g_2, \ldots, g_q),$$

where

$$g_j = (x_{(j-1)n+1}, x_{(j-1)n+2}, \ldots, x_{jn}) \in \Sigma_E^n.$$

Equivalently, each packed exponent can be represented as a packed integer

$$\phi_n(g_j) = \sum_{b=0}^{n-1} x_{(j-1)n+b+1}\, 2^{Eb},$$

so the effective coding alphabet is $\Sigma_E^{(n)} = \{0, \ldots, 2^{En} - 1\}$. For byte-valued exponent symbols, this becomes

$$\phi_n(g_j) = \sum_{b=0}^{n-1} x_{(j-1)n+b+1}\, 2^{8b}.$$

For a packed exponent symbol $g \in \Sigma_E^n$, define

$$n_{S,n}(g) = |\{j : g_j = g\}|, \qquad \widehat{p}_{S,n}(g) = \frac{n_{S,n}(g)}{|G_n(S)|}.$$

The zero-order empirical entropy of the sequence of packed exponent symbols is

$$H_0^{(n)}(S) = -\sum_{g \in \Sigma_E^n} \widehat{p}_{S,n}(g) \log_2 \widehat{p}_{S,n}(g),$$

measured in bits per packed exponent. The corresponding entropy per original exponent symbol is

$$\overline{H}_0^{(n)}(S) = \frac{1}{n} H_0^{(n)}(S).$$

If $q = 0$, then there are no complete packed exponents and their ideal cost is zero, so the lemma is immediate. Assume $q > 0$. Draw uniformly one complete packed exponent from $G_n(S)$ and denote its b-th component by X_b, for $b = 1, \ldots, n$. The empirical distribution of X_b is

$$p_b(a) = \frac{1}{q} |\{j : x_{(j-1)n+b} = a\}|, \qquad a \in \Sigma_E.$$

The joint empirical distribution of $X_1, \ldots, X_n$ is $\widehat{p}_{S,n}$. Therefore,

$$H_0^{(n)}(S) = H(X_1, \ldots, X_n).$$

By Theorem A.11,

$$H(X_1, \ldots, X_n) \le \sum_{b=1}^{n} H(X_b) = \sum_{b=1}^{n} H(p_b).$$

The empirical distribution of the complete prefix $S^{[n]}$ is the average of the positional distributions:

$$\widehat{p}_{S^{[n]}}(a) = \frac{1}{n}\sum_{b=1}^{n} p_b(a), \qquad a \in \Sigma_E.$$

By Theorem A.10,

$$H_0(S^{[n]}) = H\left(\frac{1}{n}\sum_{b=1}^{n} p_b\right) \geq \frac{1}{n}\sum_{b=1}^{n} H(p_b).$$

Combining the two inequalities gives

$$H_0^{(n)}(S) \leq \sum_{b=1}^{n} H(p_b) \leq nH_0(S^{[n]}),$$

or equivalently

$$\overline{H}_0^{(n)}(S) \leq H_0(S^{[n]}).$$

Multiplying by $q = |G_n(S)|$ yields

$$|G_n(S)|H_0^{(n)}(S) \leq |S^{[n]}|H_0(S^{[n]}).$$

Now let $\mathcal{P} = \{S_1, S_2, \ldots, S_r\}$ be a partition of exponent sequences, and let $\mathcal{P}^{[n]} = \{S^{[n]} : S \in \mathcal{P},\ |S^{[n]}| > 0\}$ be the corresponding complete-prefix partition. Summing the previous inequality over each sequence in $\mathcal{P}$ gives

$$C_0^{(n)}(\mathcal{P}) = \sum_{S\in\mathcal{P}} \left\lfloor \frac{|S|}{n} \right\rfloor H_0^{(n)}(S) \leq \sum_{S\in\mathcal{P}} |S^{[n]}|H_0(S^{[n]}) = C_0(\mathcal{P}^{[n]}).$$

This proves the ideal entropy monotonicity claimed in Lemma 3.2. □

B.3 Proof of Theorem 3.3

Detailed proof. Let

$$\mathcal{P} = \{S_1, S_2, \ldots, S_r\}$$

be the baseline partition of exponent sequences used by a sequence-partitioned zero-order compressor. Let $\mathcal{P}_C$ be a cluster-aware refinement of $\mathcal{P}$. Thus, each baseline sequence S_j is split into cluster-induced subsequences

$$S_{j,1}, S_{j,2}, \ldots, S_{j,t_j},$$

where each subsequence contains symbols assigned to one cluster of parameter blocks and the original order inside each subsequence is preserved. Empty subsequences are ignored, and

$$\mathcal{P}_C = \{S_{j,q} : 1 \leq j \leq r,\ 1 \leq q \leq t_j\}.$$

The unpacked ideal zero-order cost is

$$C_0(\mathcal{P}) = \sum_{S \in \mathcal{P}} |S| H_0(S),$$

while the ideal clustered cost after transforming the sequences into packed exponents is

$$C_0^{(n)}(\mathcal{P}_C) = \sum_{S \in \mathcal{P}_C} \left\lfloor \frac{|S|}{n} \right\rfloor H_0^{(n)}(S).$$

Under the tail-free idealization, every sequence $S \in \mathcal{P}_C$ has length divisible by n. Therefore $S^{[n]} = S$, and Lemma 3.2 gives, for each clustered sequence,

$$\left\lfloor \frac{|S|}{n} \right\rfloor H_0^{(n)}(S) \leq |S| H_0(S).$$

Summing over all clustered sequences yields

$$C_0^{(n)}(\mathcal{P}_C) \leq C_0(\mathcal{P}_C).$$

Since $\mathcal{P}_C$ is a refinement of $\mathcal{P}$, Lemma 3.1 gives

$$C_0(\mathcal{P}_C) \leq C_0(\mathcal{P}).$$

Combining both inequalities,

$$C_0^{(n)}(\mathcal{P}_C) \leq C_0(\mathcal{P}_C) \leq C_0(\mathcal{P}),$$

and therefore

$$C_0^{(n)}(\mathcal{P}_C) \leq C_0(\mathcal{P}).$$

□

Appendix C

Deferred Techniques

This chapter examines five alternatives considered during the development of NeuralZip. They test whether adjacent-symbol substitution, exponent differences, conditional Huffman codes, or arithmetic coding expose enough structure to justify their cost. Table C.1 presents the complete comparison first, and the following subsections explain how each technique works and why the measured trade-off kept it outside the final method.

	Encoding time (s)						Ratio (%)					
Model	ZipNN local	NeuralZip*	MM-RePair†	Delta	Order-1	Arithmetic	ZipNN local	NeuralZip*	MM-RePair†	Delta	Order-1	Arithmetic
GPT-2 [55]	0.100	**0.052**	2.087	0.367	2.369	1.781	33.186	33.179	41.465	40.994	33.374	**32.525**
BERT Base [11]	0.102	**0.047**	1.758	0.280	1.843	1.612	32.610	32.609	39.638	40.685	32.909	**32.405**
ResNet-18 [21]	0.015	**0.007**	1.711	0.042	0.251	0.169	33.398	32.373	39.245	38.089	32.756	**31.746**
ResNet-34 [21]	0.026	**0.013**	1.642	0.048	0.435	0.314	33.318	32.636	38.009	37.421	32.179	**31.345**

Table C.1: Deferred compressors.

*Production direct path after structural setup. †Largest repeated-shape matrix pair only; its time is not directly comparable with the complete-stream measurements.

C.1 MM-RePair

MM-RePair is a grammar compressor designed to preserve matrix operations over a compact representation [13]. We evaluate the authors' original `matrepair` implementation. [1] Given a matrix E, the converter first writes a dictionary V of distinct values and a row-major integer sequence. Its standard CSRV representation combines each value identifier with its column position. This is useful for sparse matrices but redundant for a dense exponent matrix. We therefore also evaluate the repository's dense-row-value (DRV) path, which retains every entry but omits column identifiers, and report its smaller result.

The resulting integer sequence is compressed by RePair. At each iteration, its most frequent adjacent pair is replaced non-overlappingly by a fresh nonterminal. For example,

$$\underbrace{127\ 127}_{A}\ \underbrace{127\ 127}_{A}\ 124\ \underbrace{127\ 127}_{A} \quad\longrightarrow\quad A\ A\ 124\ A, \qquad A \to (127, 127).$$

```
(V, S) ← DenseRowValues(E)
R ← ∅
while a repeated adjacent pair exists in S do
    (a, b) ← MostFrequentPair(S)
    A ← FreshSymbol()
    S ← ReplaceNonOverlapping(S, (a, b), A)
    R[A] ← (a, b)
end while
return V, Pack(R), and ANSFold(S)
```

Algorithm C.1: MM-RePair

The official ReANS format stores the value dictionary in `.ivald`, the packed rules in `.dv.R.iv`, and the compressed start sequence in `.dv.C.ansf.1`. Its measured size is therefore

$$L_{\mathrm{MMR}} = |V_{\mathrm{ivald}}| + |R_{\mathrm{iv}}| + |C_{\mathrm{ansf}}|.$$

Across the evaluated matrix pairs, the denser DRV representation reaches ratios between 38.01% and 41.46%. All are larger than the corresponding complete-stream, field-aware baseline ratio. Because this technique operates on one repeated-shape pair rather than the full exponent stream, its encoding time is illustrative and is not directly comparable with the other columns. The grammar captures repeated pairs, but its rules and start sequence do not provide a stable size advantage over local byte Huffman codes for these low-alphabet sequences, so it is not included in NeuralZip.

[1] Reference implementation: `https://gitlab.com/manzai/mm-repair`

C.2 Delta exponent coding

The delta experiment encodes consecutive exponent differences instead of raw exponent values. For an exponent sequence

$$S = (e_1, e_2, \ldots, e_t),$$

we encode

$$d_1 = e_1, \qquad d_i = (e_i - e_{i-1}) \bmod 256.$$

If long runs of equal exponents were common, this would create many zeros and make the sequence easier to compress with Huffman coding. The transform is fully reversible through

$$e_1 = d_1, \qquad e_i = (e_{i-1} + d_i) \bmod 256.$$

For instance, $(127, 127, 124, 127)$ becomes $(127, 0, 253, 3)$: one repeated value becomes zero, but each change creates another delta symbol.

This trade-off is unfavorable across the suite. Delta coding increases the ratio by 4.10–8.08 percentage points relative to the ZipNN local baseline, while taking 1.82×–3.66× as long to encode. Equal adjacent exponents are therefore not frequent enough to offset the broader delta distribution, so the extra reversible pass has no role in the final compressor.

C.3 Order-1 Huffman Coding

We measured whether the previous exponent provides useful information about the next one. Let $S = (e_1, \ldots, e_t)$ be an exponent sequence with $t > 1$. For each context $c \in \Sigma_E$, let

$$S_c = (e_i : 2 \le i \le t,\ e_{i-1} = c) \qquad \text{and} \qquad n_c = |S_c|$$

where S_c is the sequence, in its original order, of exponents observed after context c, and n_c is its length. The first-order empirical entropy is the weighted conditional entropy

$$H_1(S) = \sum_{c \in \Sigma_E} \frac{n_c}{t-1} H_0(S_c).$$

Thus, H_1 is the average zero-order entropy of the successor distributions, weighted by how often each previous exponent occurs [46]. If $H_1(S) < H_0(S)$, the previous exponent contains predictive information that a zero-order code does not capture; the first exponent is handled separately.

We aggregate symbol and transition counts over all parameter blocks while excluding artificial transitions across tensor boundaries. As shown in Figure C.1, first-order conditioning reduces entropy by 0.005–0.163 bits per exponent, or approximately 0.20–6.01%, across the evaluated models. The reduction is consistent but modest, providing a reason to test conditional coding while also predicting limited practical gains after metadata is included.

To test whether this entropy reduction translates into a smaller representation, the Order-1 Huffman compressor uses the previous exponent as a context. Instead of one Huffman tree

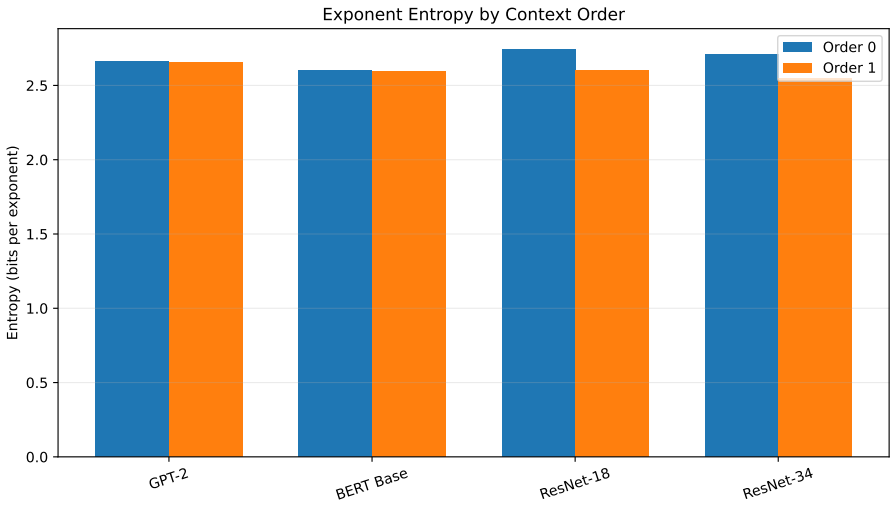


Figure C.1: Exponent entropy by order.

per exponent sequence, each cluster can store up to 256 trees, one for each previous exponent value. This is a natural alternative to packed exponents: both exploit sequential structure, but Order-1 coding uses conditional distributions rather than new symbols.

The conditional model changes the ratio by between a 1.14-point improvement and a 0.30-point penalty relative to the ZipNN local baseline, while taking 16.3×–23.7× as long to encode. A separate tree per observed context therefore adds substantial state without a stable gain across architectures; packed exponents capture short-range dependence with a much smaller coding interface.

C.4 Arithmetic coding

Huffman coding assigns an integral number of bits to every symbol. Arithmetic coding instead represents a complete sequence as a subinterval and can approach fractional information costs [56]. Let $[L, U)$ be the current interval and let $F_c(a)$ be the cumulative conditional probability of symbol a after context c. Encoding a updates the interval as

$$L' = L + (U - L)F_c(a), \qquad U' = L + (U - L)F_c(a + 1).$$

We evaluated the byte-wise order-1 `rccs` coder from Turbo-Range-Coder [48]. Each 256 KiB exponent chunk is encoded independently and stored raw whenever the range-coded payload is not smaller. The tensor layout is unchanged, so the comparison isolates the entropy coder rather than combining it with a transpose.

Arithmetic coding improves the ratio by 0.21–1.97 percentage points across the evaluated models, but requires 11.0×–17.8× the encoding time of the ZipNN local baseline. It is therefore excluded from the final method not because it fails to compress, but because its improvement consistently costs more compression time.

C.5 Direct access

The initial proposal considered an auxiliary sampled-offset index over the variable-length Huffman representation. Such an index would store the bit position of every K-th exponent, allowing a decoder to begin at the nearest preceding entry instead of scanning the compressed representation from its beginning. This would trade additional metadata for bounded local decoding.

We ultimately found that a separate index was unnecessary for the workloads in this thesis. The implemented representation already organizes compressed data as tensor records, byte groups, and independently decodable chunks. Once the archive metadata has been loaded, the references in these records act as pointers to the corresponding payloads and shared Huffman codes. Model storage, checkpoint compression, and distributed updates all consume complete tensors or complete model artifacts; none of the evaluated workloads requires retrieving a specific scalar value from inside a compressed tensor. An additional sampled index would therefore duplicate existing navigation metadata while increasing archive size and implementation complexity without benefiting the operations we measure.

Appendix D

Ablation Study

D.1 Component isolation

	Encoding time (s)				Ratio (%)			
Model	ZipNN local	Sharing Only	Packing Only	NeuralZip	ZipNN local	Sharing Only	Packing Only	NeuralZip
GPT-2 [55]	0.098	0.055	0.082	**0.048**	33.186	33.186	**33.177**	33.179
BERT Base [11]	0.099	0.057	**0.044**	0.066	32.610	32.610	32.610	**32.609**
ResNet-18 [21]	0.015	**0.006**	0.008	0.007	33.398	33.398	**32.366**	32.373
ResNet-34 [21]	0.025	**0.011**	0.024	0.012	33.318	33.316	**32.450**	32.636

Table D.1: Component isolation.

All four variants use the same byte grouping, chunk size, production Huff0 backend, acceptance threshold, and archive-size accounting. Sharing Only keeps JS clustering and shared codes but disables packed exponents. Packing Only assigns each tensor to its own cluster, thereby removing cross-tensor sharing while retaining packed exponents.

Across the suite, the complete method lowers the exponent-field ratio by 0.001–1.025 percentage points and encodes 1.49×–2.05× faster than the ZipNN local baseline after setup. Sharing Only changes the ratio by at most 0.002 percentage points while encoding 1.74×–2.47× faster, which isolates the effect of the precomputed structural plan and batched production path. Packing Only lowers the ratio by up to 1.032 percentage points and is the smallest representation in three of the four cases. The complete method remains within 0.186 percentage points of Packing Only while retaining a direct-encoding speedup in every case. These results support treating blockwise heterogeneity and packed exponents as complementary rather than universally interchangeable sources of redundancy.

D.2 Clustering strategy

We inspect the clusters after assignment to identify where their similarity comes from. Coupled projections with comparable tensor geometry tend to operate at similar numerical

scales, while changes across depth create separate regimes. We apply the default greedy rule ($\tau = 0.05$) to the floating-point tensors of the four BF16 models in the suite. Clustering uses only the normalized exponent histogram; architectural roles and layer indices are added afterwards for this analysis.

For each architecture, we inspect one representative pair of coupled projections. The same-cluster rate measures how often both tensors receive the same assignment within a block. Adjusted mutual information (AMI) compares the two sequences of cluster assignments across depth, treating each layer as one observation, and discounts agreement expected by chance. An AMI of 1 means that both projections follow the same depth-wise partition, 0 indicates chance-level agreement, and negative values indicate less agreement than expected by chance.

Model	Coupled pair	Same cluster (%)	AMI
GPT-2 [55]	FFN expansion–reduction	8.3	-0.11
BERT Base [11]	FFN expansion–reduction	100.0	1.00
ResNet-18 [21]	Adjacent block convolutions	100.0	1.00
ResNet-34 [21]	Adjacent block convolutions	58.3	0.40

Table D.2: Clustering agreement.

Table D.2 separates direct cluster sharing from agreement in how the pair changes across depth. At the default threshold, 42.7–48.2% of tensors belong to non-singleton clusters. The same-cluster rate for the inspected coupled roles ranges from 8.3% to 100%, while AMI ranges from -0.11 to 1.00. Inspection of the complete memberships also recovers pure groups of repeated normalization weights and recurring feed-forward or convolutional projections, together with mixed-role groups whose exponent distributions happen to be compatible. Comparable tensor geometry provides a second source of similarity.

Depth and stage provide the remaining structure. Although neither is used by the algorithm, a repeated role can move between clusters as its scale changes through the network, while coupled tensors can remain together within one depth or stage regime. The range of agreement therefore shows that JS clusteritsng is not a semantic layer classifier: semantic identity is neither required nor guaranteed. It identifies numerical regimes that are stable enough to share a coding model, which is the property required by NeuralZip.

D.3 ZipNN implementation

We compare the local compatible implementation against the authors' source package on the same pretrained checkpoints. Each run was verified through a bitwise-exact round trip.

	Encoding time (s)		Ratio (%)	
Model	ZipNN local	ZipNN source	ZipNN local	ZipNN source
GPT-2 [55]	**0.101**	0.183	33.186	**33.174**
BERT Base [11]	**0.094**	0.250	32.610	**32.596**
ResNet-18 [21]	**0.014**	0.039	33.398	**33.371**
ResNet-34 [21]	**0.026**	0.069	33.318	**33.290**

Table D.3: ZipNN implementations.

The exponent-field ratios are nearly identical across the suite: the source implementation is smaller by only 0.011–0.028 percentage points. The local implementation is 1.81×–2.86× faster. The small size differences arise from the distinct Huffman backends and their respective container metadata.

The local implementation is faster because it uses the same production native Huff0 primitives as NeuralZip and batches chunk encoding over 12 worker threads. The source package uses its official archive format and invokes its tensor path repeatedly across the state dictionary. Its required defensive copy represents only 3.4–6.8% of the measured source time; excluding that copy, the source codec remains 1.70×–2.77× slower. The comparison therefore verifies that the local baseline preserves nearly the same ratio while providing the shared, instrumentable backend required by the component ablations.