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

Compresión de Modelos de Lenguaje

PROPUESTA DE TEMA DE MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

Martín Eduardo Bravo Díaz

MODALIDAD:
Memoria

PROFESOR GUÍA:
Andrés Abeliuk
PROFESOR GUÍA 2:
Gonzalo Navarro

SANTIAGO DE CHILE
30 DE NOVIEMBRE DE 2025

1. Introducción

1.1. Qué es un modelo de lenguaje grande

Nuestra sociedad necesitaba la presencia de sistemas más inteligentes, por lo que empezamos a entrenar modelos a gran escala. Estos modelos resuelven muchos problemas en áreas como la medicina, educación y comunicación. En particular, los modelos de lenguaje grandes han tenido una revolución en los últimos años, donde su rendimiento y precisión han mejorado significativamente.

En esencia, un modelo de lenguaje grande es un tipo de modelo de aprendizaje automático que puede comprender y generar lenguaje humano mediante redes neuronales profundas.

La tarea principal de un modelo de lenguaje es calcular la probabilidad de que una palabra siga a una entrada dada en una oración; por ejemplo, "El cielo es _____", donde la respuesta más probable sería "azul". El modelo es capaz de predecir la siguiente palabra en una oración tras haber sido entrenado con un gran conjunto de textos (o corpus). Básicamente, aprende a reconocer diferentes patrones en las palabras. De este proceso resulta un modelo de lenguaje previamente entrenado.

1.2. Historia de los modelos de lenguaje grandes

La idea de los modelos de lenguaje grandes surgió por primera vez con la creación de Eliza en la década de 1960 [21]: fue el primer chatbot del mundo, diseñado por el investigador del MIT Joseph Weizenbaum. Eliza marcó el inicio de la investigación en procesamiento de lenguaje natural (NLP), sentando las bases para futuros modelos de lenguaje grandes más complejos.

Casi 30 años después, en 1997, aparecieron las redes de Long Short-Term Memory (LSTM) [9]. Su llegada permitió desarrollar redes neuronales más profundas y complejas, capaces de manejar mayores volúmenes de datos. La suite CoreNLP de Stanford [22], introducida en 2010, fue la siguiente etapa de crecimiento, al permitir a los desarrolladores realizar análisis de sentimiento y reconocimiento de entidades nombradas.

Posteriormente, en 2011, surgió una versión reducida de Google Brain con características avanzadas como las *word embeddings* [15], que permitieron a los sistemas de NLP obtener una comprensión más clara del contexto. Este fue un punto de inflexión importante, que desembocó en la irrupción de los modelos Transformer en 2017 [24]. Ejemplos destacados son GPT (Generative Pre-trained Transformer) [17], capaz de generar o "decodificar" nuevo texto, y BERT (Bidirectional Encoder Representations from Transformers) [4], que puede predecir o clasificar texto de entrada a partir de componentes encoder.

A partir de 2018, los investigadores comenzaron a construir modelos cada vez más grandes. En 2019, un equipo de Google presentó BERT, un modelo bidireccional con 340 millones de parámetros (el tercero más grande de su tipo en ese momento), capaz de determinar el contexto y adaptarse a múltiples tareas. Gracias a su preentrenamiento sobre una amplia variedad de datos no estructurados mediante aprendizaje autosupervisado, el modelo logró captar las relaciones entre palabras. En muy poco tiempo, BERT se convirtió en la herra-

mienta de referencia para tareas de NLP. De hecho, estuvo detrás de todas las consultas en inglés realizadas a través de Google Search.

A medida que BERT se refinaba, GPT-2 [18] de OpenAI, con 1,5 mil millones de parámetros, logró producir prosa convincente. Luego, en 2020, lanzaron GPT-3 con 175 mil millones de parámetros [3], que se convirtió en el estándar de los modelos de lenguaje grandes y sentó las bases de ChatGPT. Fue con el lanzamiento de ChatGPT en noviembre de 2022 que el público general comenzó realmente a notar el impacto de los modelos de lenguaje grandes. Incluso los usuarios no técnicos podían dar instrucciones al modelo, recibir una respuesta rápida y mantener una conversación, generando tanto entusiasmo como preocupación.

Más recientemente, OpenAI introdujo GPT-4 [1], estimado en alrededor de un billón de parámetros cinco veces el tamaño de GPT-3 y aproximadamente 3.000 veces el tamaño de BERT en su lanzamiento inicial.

1.3. El gran problema

El crecimiento exponencial en el tamaño de los modelos de lenguaje ha planteado desafíos significativos en términos de almacenamiento, eficiencia computacional y consumo energético, dificultando su despliegue en dispositivos con recursos limitados como teléfonos móviles o sistemas embebidos.

Ante esta tendencia hacia la ejecución local y en dispositivos periféricos [2], se han desarrollado diversas técnicas de alivianamiento de modelos, tales como la *quantization*, que reduce la precisión numérica de los parámetros para disminuir el tamaño y acelerar la inferencia; el *pruning*, que elimina conexiones o parámetros de baja relevancia; la *knowledge distillation*, donde un modelo más pequeño aprende a imitar el comportamiento de uno grande; las técnicas de *low-rank factorization* y *parameter sharing*, que reducen la redundancia interna de las matrices y pesos; además de los modelos *sparse* o de activación selectiva, que sólo utilizan una fracción de los parámetros por inferencia; y métodos de compresión *lossless* que preservan la integridad del modelo mientras optimizan su almacenamiento.

En particular, los métodos recientes de compresión *lossless* han cobrado gran relevancia. Un ejemplo destacado es ZipNN [8], que alcanzó una tasa de compresión de hasta 18.7× en modelos de visión como ResNet18 [7], superando significativamente a algoritmos tradicionales de compresión como GZip, Zstd y Brotli, así como a métodos especializados previos como DEEPSZ [10], Weightless [26] y ZIPLM [12]. Además, ZipNN logró reducir el tamaño de modelos de lenguaje (BERT, GPT-2, OPT) en un rango de 3–6×, manteniendo la exactitud original.

En esta memoria[1]., intentaremos encontrar métodos más eficientes para representar pesos que ZipNN, o analizar si es posible obtener mayores tasas de compresión permitiendo ligeras modificaciones en los parámetros del modelo sin afectar de manera significativa su rendimiento. El objetivo es avanzar en la frontera entre técnicas *lossless* y aproximaciones tolerantes a error, contribuyendo con evidencia empírica y propuestas concretas al desafío.

[1]Nuestra implementación puede encontrarse en https://colab.research.google.com/drive/1-rj5zl8UGVDuP7aGkE9bA4IMVQL5ohoo?usp=sharing

2. Estado del Arte

2.1. Compresión sin pérdida

El crecimiento de los modelos de lenguaje ha motivado la búsqueda de técnicas de *compresión sin pérdida* que reduzcan el tamaño del modelo manteniendo intactos sus parámetros. El método ZipNN identifica que el byte del exponente en los números de punto flotante presenta una distribución muy sesgada; al separarlo de la mantisa y comprimirlo con códigos de Huffman logra reducir los modelos BF16 a cerca de dos tercios de su tamaño original [8]. Para modelos *limpios* con pesos redondeados, se agrupan los bytes de la fracción y se comprimen sólo aquellos que contienen ceros, alcanzando reducciones de hasta 42 % en modelos FP32 como XLM-RoBERTa [8]. ZipNN también incorpora compresión de *deltas* entre checkpoints mediante un mecanismo que elige entre Huffman y Zstd según el porcentaje de ceros en cada fragmento [8].

2.1.1. Representación de los pesos en punto flotante

Sea N el número total de parámetros del modelo y $\{w_i\}_{i=1}^{N}$ los pesos en formato IEEE-754 `float32`. Cada peso w_i se descompone en tres campos:

$$w_i = (-1)^{s_i} \cdot 2^{e_i - 127} \cdot \left(1 + \frac{m_i}{2^{23}}\right),$$

donde $s_i \in \{0, 1\}, \quad e_i \in \{0, \ldots, 255\}, \quad m_i \in \{0, \ldots, 2^{23} - 1\}$. En términos de bits: s_i ocupa 1 bit (signo), e_i ocupa 8 bits (exponente), m_i ocupa 23 bits (mantisa). El modelo completo, sin comprimir, requiere entonces: $L_{\text{orig}} = 32N$ bits. Denotaremos por $\mathbf{s} = (s_1, \ldots, s_N)$, $\mathbf{e} = (e_1, \ldots, e_N)$ y $\mathbf{m} = (m_1, \ldots, m_N)$ las secuencias de signo, exponentes y mantisas, respectivamente.

La *tasa de compresión* de un esquema que utiliza L bits totales se define como

$$R = \frac{L}{L_{\text{orig}}} = \frac{L}{32N},$$

que en los experimentos reportamos como porcentaje $100 \cdot R\,\%$ del tamaño original.

2.1.2. ZipNN: compresión global del exponente con Huffman

La idea central de ZipNN es observar que la distribución de los exponentes $\mathbf{e}$ está altamente sesgada. Sea $\Sigma = \{0, \ldots, 255\}$ el alfabeto de posibles exponentes y para cada símbolo $a \in \Sigma$ definimos

$$n_a = |\{i : e_i = a\}|, \qquad p_a = \frac{n_a}{N}.$$

Un código de Huffman [16] para la distribución $\mathbf{p} = (p_a)_{a \in \Sigma}$ asigna a cada símbolo a una palabra de código binaria de longitud $\ell(a) \in \mathbb{N}$, con la cota clásica[2]

$$H(\mathbf{p}) \leq \mathbb{E}[\ell(e)] = \sum_{a \in \Sigma} p_a\, \ell(a) < H(\mathbf{p}) + 1, \quad H(\mathbf{p}) = -\sum_{a \in \Sigma} p_a \log_2 p_a$$

[2]Una introducción más amplia a técnicas clásicas de compresión puede encontrarse en *Compact Data Structures* [16]

con $H(\mathbf{p})$ la entropía de orden 0 de los exponentes. En nuestra implementación, el costo total en bits de codificar todos los exponentes con Huffman global se aproxima como

$$L_{\text{exp}}^{\text{Huff}} = \sum_{a \in \Sigma} n_a \, \ell(a).$$

Como primera aproximación, asumimos que los signos y mantisas se almacenan sin comprimir, utilizando exactamente N bits para $\mathbf{s}$ y $23N$ bits para $\mathbf{m}$. Así, el tamaño total del modelo bajo ZipNN es $L_{\text{ZipNN}} = L_{\text{exp}}^{\text{Huff}} + N + 23N$. En los experimentos con `GPT-2`, esto se traduce en una reducción hasta aproximadamente un 85,3 % del tamaño original.

2.2. Compresión amigable con la multiplicación matriz–vector

Más allá de reducir espacio, una línea de trabajo reciente busca formatos de compresión *lossless* que también aceleren directamente operaciones algebraicas. Ferragina et al. proponen representar matrices reales mediante una gramática que comprime una codificación tipo CSR y permite realizar productos matriz–vector recorriendo directamente la gramática, con coste proporcional al tamaño comprimido en lugar del tamaño original de la matriz [5]. De forma análoga, Francisco et al. estudian compresión de grafos para acelerar la multiplicación por la matriz de adyacencia en tareas como PageRank, explotando regularidades estructurales del grafo para reutilizar resultados parciales en cada fila [6]. Estos trabajos ilustran que es posible diseñar representaciones comprimidas que respetan la estructura algebraica subyacente, una idea que motiva parte de este trabajo al considerar formatos de compresión de modelos que no sólo ahorren memoria, sino que también puedan interactuar favorablemente con la multiplicación de matrices en la inferencia de LLMs.

2.3. Compresión con pérdida y codificación aproximada

Algunos trabajos permiten pequeñas modificaciones en los pesos a cambio de fuertes reducciones de tamaño. DeepSZ combina poda y compresión con error acotado (SZ) para seleccionar errores por capa que maximicen la relación de compresión. Puede comprimir redes como AlexNet y VGG16 con ratios de 46× y 116× respectivamente, manteniendo la pérdida de precisión bajo el 0,3 % [10]. Weightless utiliza filtros de Bloomier para codificar los pesos de forma probabilística; aunque introduce errores, el modelo se reajusta mediante reentrenamiento, logrando compresiones de hasta 496× con mejoras de 1,51× sobre el estado del arte [19].

2.4. Poda estructurada y destilación

La *poda estructurada* elimina bloques de parámetros completos para mejorar la velocidad de inferencia. ZipLM evalúa la relación entre pérdida de precisión y velocidad y elimina iterativamente los componentes con peor relación. El método produce variantes de BERT que igualan o superan métodos de destilación como CoFi y TinyBERT, e incluso genera versiones de GPT2 un 60 % más pequeñas y 30 % más rápidas que DistilGPT2 [12].

Por su parte, la destilación de conocimiento transfiere la información de un modelo grande a uno más pequeño. Un estudio reciente sintetiza la destilación de modelos de lenguaje

en tres pilares algoritmo, habilidad y verticalización y destaca que la generación de datos sintéticos puede mejorar la transferencia de habilidades y el alineamiento ético entre profesor y alumno [25].

2.5. Cuantización y descomposición de bajo rango

Los métodos de *cuantización* reducen la precisión numérica de los pesos. La técnica AWQ (*ActivationAware Weight Quantization*) demuestra que proteger sólo el 1 % de los canales más importantes permite cuantizar redes con 4 bits sin apenas degradar el rendimiento. Para identificar dichos canales se utilizan estadísticas de activación y se escala su amplitud antes de cuantizar; la propuesta consigue acelerar modelos LLaMA de 70 mil millones de parámetros hasta 3× en dispositivos móviles [13].

Las técnicas de *bajo rango* aproximan las matrices de pesos mediante descomposición. CALDERA descompone cada matriz como $\mathbf{W} \approx \mathbf{Q} + \mathbf{LR}$, donde $\mathbf{L}$ y $\mathbf{R}$ son factores de bajo rango y $\mathbf{Q}$, $\mathbf{L}$ y $\mathbf{R}$ se cuantizan. Formulando el problema como una regresión con restricciones de rango y precisión, logra mejores relaciones de compresión que métodos previos al aplicar menos de 2,5 bits por parámetro en modelos LLaMA2 y LLaMA3 [20].

En la misma línea, LLRC (*Learning to LowRank Compress*) utiliza un algoritmo diferenciable que aprende las máscaras óptimas para seleccionar valores singulares en cada capa. Este enfoque, libre de reentrenamiento, supera a métodos heurísticos como STRS y ARS, así como a LLMPruner, en tareas de razonamiento y preguntas de dominio abierto al comprimir modelos como LLaMA2 y LLaMA3 [23].

2.6. Herramientas de versionado y compresión de checkpoints

Además de la compresión de modelos, es relevante gestionar las versiones y diferencias de pesos. GitTheta ofrece un sistema de control de versiones que explota la estructura interna de los checkpoints para realizar actualizaciones eficientes y fusionar automáticamente cambios de distintos colaboradores [11]. Estas ideas se complementan con la compresión delta implementada en ZipNN, donde sólo se almacenan las diferencias entre modelos sucesivos, reduciendo el almacenamiento y la transferencia [8].

3. Trabajo Adelantado

En esta sección describimos las variantes de compresión implementadas sobre el modelo `GPT-2` en precisión `float32`, siguiendo y extendiendo la propuesta de ZipNN [8]:

Método	**Tamaño (Mbits)**	**% del original**
Original	3982.07	100.00 %
ZipNN [8]	3395.82	85.28 %
ZipNN+ (Sección 3.1)	3225.80	81.01 %
ZipNN++ (Sección 3.2)	3095.78	77.74 %
Límite teórico (sin exponentes)	2986.55	75.00 %

Tabla 1: Comparación de la tasa de compresión entre ZipNN y nuestras variantes ZipNN+ y ZipNN++.

3.1. ZipNN+: Compresión por capa

Sea el conjunto de parámetros del modelo particionado en L tensores (capas) $\{\mathbf{w}^{(\ell)}\}_{\ell=1}^{L}$, donde el tensor ℓ contiene N_ℓ pesos y $\sum_{\ell=1}^{L} N_\ell = N$.

Para cada capa ℓ descomponemos los pesos en

$$\mathbf{w}^{(\ell)} \longrightarrow \left(\mathbf{s}^{(\ell)}, \mathbf{e}^{(\ell)}, \mathbf{m}^{(\ell)}\right),$$

y calculamos las distribuciones empíricas de exponentes:

$$p_a^{(\ell)} = \frac{1}{N_\ell} \left|\{i : e_i^{(\ell)} = a\}\right|, \qquad q_b^{(\ell)} = \frac{1}{N_\ell} \left|\{i : m_i^{(\ell)} = b\}\right|.$$

Aplicamos Huffman *por capa* a exponentes. El costo total aproximado es

$$L_{\text{ZipNN+}}^{\text{normal}} = \sum_{\ell=1}^{L} \left(\sum_{a} n_a^{(\ell)}\, \ell_{\text{exp}}^{(\ell)}(a) + N_\ell \cdot 23 + N_\ell \right),$$

donde $n_a^{(\ell)}$ y $n_b^{(\ell)}$ son las frecuencias de a y b en la capa ℓ, y $\ell_{\text{exp}}^{(\ell)}(\cdot)$, $\ell_{\text{mant}}^{(\ell)}(\cdot)$ son las longitudes de código de Huffman para exponentes y mantisas de esa capa. El término N_ℓ corresponde a los bits de signo sin comprimir.

Esta variante reduce el tamaño del modelo a alrededor de un 81 % del original.

3.2. ZipNN++: Clustering + n-gramas + Codificación aritmética

3.2.1. n-gramas de exponentes

Los exponentes consecutivos muestran correlación; para explotarla, consideramos empaquetar n exponentes consecutivos en un único símbolo. Sea $\mathbf{e} = (e_1, e_2, \ldots, e_N)$, $\quad e_i \in \Sigma$, y fijemos un entero $n \geq 1$. Definimos los *n-gramas de exponentes* como

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

Cada y_k es un entero que codifica n exponentes en un símbolo de hasta $8n$ bits. Sobre la secuencia $\mathbf{y} = (y_k)$ construimos una distribución empírica

$$\pi_u = \frac{1}{\lfloor N/n \rfloor} \left| \{k : y_k = u\} \right|,$$

y un código de Huffman asociado, con longitudes $\ell_{\text{ngram}}(u)$. El costo en bits de los exponentes bajo este esquema es

$$L_{\text{exp}}^{(n)} = \sum_u \#\{k : y_k = u\}\, \ell_{\text{ngram}}(u).$$

El tamaño total del modelo se calcula análogamente a ZipNN++, sustituyendo $L_{\text{exp}}^{\text{Huff}}$ por $L_{\text{exp}}^{(n)}$. En la práctica, usamos $n = 2$ y $n = 3$, observando reducciones adicionales pequeñas (del orden de 0,3–0,7 puntos porcentuales) respecto a Huffman de orden 0.

3.2.2. Codificación aritmética adaptativa de orden 1

Para capturar dependencias de primer orden entre exponentes consecutivos, utilizamos un modelo de Markov de orden 1 junto con codificación aritmética [16]. Sea $X_1, \ldots, X_N$ la secuencia de exponentes, $X_t \in \Sigma$. Un modelo de orden 1 define probabilidades condicionales $\Pr(X_t = x \mid X_{t-1} = y) = p_{y,x}, \quad y, x \in \Sigma$. En una codificación aritmética ideal, el número de bits necesarios para codificar la secuencia sería $L_{\text{arith}} = -\log_2 \Pr(X_1, \ldots, X_N) = -\log_2 \Pr(X_1) - \sum_{t=2}^{N} \log_2 \Pr(X_t \mid X_{t-1})$. En nuestra implementación, estimamos $\Pr(X_t \mid X_{t-1})$ de forma adaptativa. Inicialmente se asigna un conteo suavizado $c_{y,x}^{(0)} = \varepsilon > 0, \qquad C_y^{(0)} = \sum_x c_{y,x}^{(0)} = 256\,\varepsilon$, y al procesar secuencialmente la secuencia (X_t) se actualizan

$$p_{y,x}^{(t)} = \frac{c_{y,x}^{(t)}}{C_y^{(t)}}, \quad c_{y,x}^{(t+1)} = c_{y,x}^{(t)} + \mathbf{1}_{\{(X_t, X_{t+1}) = (y,x)\}}, \quad C_y^{(t+1)} = C_y^{(t)} + \mathbf{1}_{\{X_t = y\}}.$$

El costo estimado en bits es

$$L_{\text{exp}}^{\text{arith}} \approx -\sum_{t=2}^{N} \log_2 p_{X_{t-1}, X_t}^{(t-1)}.$$

Clásicamente, un código de Huffman de orden 0 garantiza que el número esperado de bits por símbolo está acotado por $H(\mathbf{p}) \le \mathbb{E}[\ell(X)] < H(\mathbf{p}) + 1$, es decir, hasta 1 bit de overhead por símbolo con respecto a la entropía $H(\mathbf{p})$ [16]. Si se aplica Huffman sobre n-gramas, el overhead máximo pasa a ser de 1 bit por cada n símbolos, esto es, $1/n$ bit por símbolo. Por otro lado, la codificación aritmética permite aproximar la longitud ideal L_{arith} con un overhead global acotado por una constante: en implementaciones estándar, el número total de bits utilizados para codificar la secuencia completa difiere de L_{arith} en a lo más 2 bits, independientemente de la longitud de la secuencia [16].

3.2.3. Clustering de capas por distribución de exponentes

Empíricamente, la distribución de exponentes varía entre capas. Para aprovechar similitudes entre capas, calculamos, para cada capa ℓ, el histograma normalizado de exponentes:

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

Para medir la similitud entre capas utilizamos la distancia de Jensen–Shannon [14]. Dados dos histogramas p y q en el simplex,

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

donde D_{KL} es la divergencia de Kullback–Leibler.

Agrupamos capas en clusters $\mathcal{C}_1, \ldots, \mathcal{C}_K$ tales que, dentro de cada cluster, las distribuciones de exponentes tienen $\mathrm{JS}(h^{(\ell)}, h^{(\ell')})$ por debajo de un umbral τ. Para cada cluster $\mathcal{C}_k$ concatenamos los exponentes de todas sus capas y entrenamos un único modelo de compresión (Huffman, aritmético o σ-Huffman) sobre ese conjunto.

Si denotamos por E_k el conjunto de exponentes en el cluster $\mathcal{C}_k$ y por $L_{\mathrm{exp}}(E_k)$ el costo de comprimirlos con el método elegido, el costo total de exponentes es

$$L_{\mathrm{exp}}^{\mathrm{cluster}} = \sum_{k=1}^{K} L_{\mathrm{exp}}(E_k).$$

Para la mantisa y el signo se estima su tamaño como $(23{+}1)\cdot|E_k|$ cuando no se comprimen explícitamente.

La combinación de clustering, n-gramas y codificación aritmética alcanza en la práctica tasas cercanas a 77,7 % del tamaño original.

3.3. Técnicas Extras

Las siguientes técnicas también lograron mejoras en la compresión, pero no se incluyeron en la versión final dado que al acoplarlas con ZipNN el beneficio adicional era leve o más lento el proceso de compresión/descompresión.

3.3.1. Acceso directo

Consideramos el problema de *acceso directo* al flujo comprimido de exponentes. Sea $X_1, \ldots, X_N$ la secuencia de exponentes y $\ell(X_i)$ la longitud del código (por ejemplo, bajo Huffman). El flujo comprimido puede verse como una cadena de bits

$$\underbrace{c(X_1)}_{\ell(X_1)\ \text{bits}}\ \underbrace{c(X_2)}_{\ell(X_2)\ \text{bits}}\ \cdots\ \underbrace{c(X_N)}_{\ell(X_N)\ \text{bits}},$$

donde $c(\cdot)$ es la palabra de código. Definimos la posición de inicio en bits del símbolo i como

$$B_i = \sum_{j=1}^{i-1} \ell(X_j), \quad B_1 = 0.$$

Acceder directamente al símbolo X_i requeriría, en principio, conocer B_i, lo que es caro almacenar para todos los i.

Como aproximación, introducimos un *índice por bloques* de tamaño K (por ejemplo $K = 4096$). Para cada bloque $b = 0, 1, \ldots, \left\lfloor \frac{N}{K} \right\rfloor$, almacenamos sólo el desplazamiento

$$O_b = B_{bK+1} = \sum_{j=1}^{bK} \ell(X_j).$$

De este modo, para acceder al símbolo X_i:

$$b = \left\lfloor \frac{i-1}{K} \right\rfloor, \quad \text{se comienza a decodificar desde el bit } O_b \text{ hasta llegar a } X_i.$$

El costo extra del índice es aproximadamente

$$L_{\text{index}} \approx \left\lceil \frac{N}{K} \right\rceil \cdot \lceil \log_2 L_{\text{exp}} \rceil$$

bits, donde L_{exp} es la longitud total del flujo comprimido de exponentes. En nuestros experimentos, con $K = 4096$ y trigramas de exponentes, el tamaño total con índice se sitúa alrededor de un 83,3 % del original, sacrificando algo de compresión frente a la mejor variante pero habilitando acceso directo por bloques para futuras operaciones eficientes sobre el modelo comprimido.

3.3.2. σ-Huffman: Huffman con contexto

En el esquema σ-Huffman, en vez de un único código de Huffman global, se construye un código *distinto* para cada contexto definido por el exponente previo.

Para cada símbolo previo $\sigma \in \Sigma$ definimos la distribución condicional empírica

$$p_{x|\sigma} = \frac{\#\{t : X_{t-1} = \sigma,\, X_t = x\}}{\#\{t : X_{t-1} = \sigma\}}, \quad x \in \Sigma,$$

siempre que el denominador sea no nulo. Sobre cada distribución $(p_{x|\sigma})_{x\in\Sigma}$ se construye un código de Huffman propio, con longitudes $\ell_\sigma(x)$.

El número total de bits requerido para codificar la secuencia (ignorando el primer símbolo) es:

$$L_{\text{exp}}^{\sigma\text{-Huff}} = \sum_{\sigma\in\Sigma} \sum_{x\in\Sigma} n_{\sigma,x}\, \ell_\sigma(x),$$

donde $n_{\sigma,x} = \#\{t : X_{t-1} = \sigma,\, X_t = x\}$.

Este esquema captura dependencias de primer orden, similar a la codificación aritmética, pero usando códigos de Huffman condicionados al contexto.

4. Objetivos

Objetivo General

Explorar los límites prácticos de la compresión *lossless* en modelos de lenguaje grandes, partiendo de ZipNN pero yendo más allá de su enfoque original mediante modelos de contexto, agrupamiento por capas y estructuras de acceso directo sobre los pesos. El objetivo es diseñar y analizar esquemas de compresión que se acerquen al límite teórico dado por la entropía de los pesos (signo, exponente y mantisa), manteniendo exactitud bit a bit y costos razonables de descompresión en escenarios reales.

Objetivos Específicos

1. Sintetizar el estado del arte en compresión de modelos (quantization, pruning, distillation, bajo rango, parameter sharing y métodos lossless) identificando sus límites empíricos y teóricos.
2. Reproducir ZipNN y nuestras variantes preliminares de compresión de exponentes sobre modelos más grandes tipo GPT-2 (e.g. LLaMA, DeepSeek, GPT-OSS, etc.), comparando tasas de compresión y costos con compresores generales.
3. Analizar la estadística conjunta de signo, exponente y mantisa (histogramas, n-gramas, deltas, correlaciones por capa y por tipo de tensor) para estimar sus entropías y diseñar modelos de contexto *lossless* que se acerquen a dichos límites teóricos.
4. Combinar técnicas con pérdida (pruning, quantization, bajo rango) con ZipNN++ para medir si la estructura inducida mejora la compresión global sin degradar el modelo.
5. Diseñar índices de acceso directo (bloques, jerarquías, sampling de offsets) que equilibren compresión, memoria extra y tiempos de acceso parcial.
6. Evaluar si las representaciones comprimidas permiten acelerar productos matriciales identificando patrones repetidos o exponentes dominantes.
7. Integrar los hallazgos en un esquema *lossless* nuevo (NeuralZip[3]) que combine modelado de contexto, compresión de exponente/mantisa y acceso eficiente, acercándose al límite de entropía con tiempos prácticos.

Evaluación

El cumplimiento de los objetivos se evaluará mediante experimentos empíricos sobre modelos de lenguaje representativos (e.g. GPT-2) y sus pesos en el formato plano de referencia (típicamente FP32). Para cada método se medirán: (i) la tasa de compresión lograda, (ii) la distancia respecto del límite teórico dado por la entropía estimada de signo, exponente y mantisa, y (iii) el coste computacional en tiempo de compresión, descompresión y acceso aleatorio a bloques concretos del modelo.

El trabajo se considerará exitoso si las variantes propuestas de ZipNN alcanzan tasas de compresión *lossless* significativamente mejores que la implementación base de ZipNN, acercándose al límite de entropía, y si las estructuras de acceso directo permiten mantener tiempos de acceso razonables sin degradar la tasa de compresión.

[3]Este nombre es temporal y podría ser cambiado a uno acorde a la arquitectura final.

5. Solución Propuesta: NeuralZip

La propuesta de esta memoria es avanzar desde ZipNN hacia un esquema de compresión *lossless* más general, que llamaremos tentativamente NeuralZip. La idea central es tratar los pesos de un modelo de lenguaje grande no como una secuencia plana de números, sino como una estructura rica en redundancias: por tipo de campo (signo, exponente, mantisa), por capa, por posición y por patrón de uso en las multiplicaciones matriz–vector.

En un primer nivel, NeuralZip se construye sobre los resultados preliminares de ZipNN+ y ZipNN++ (Secciones 3.1 y 3.2), que ya mejoran la compresión del exponente mediante: (i) modelado por capa (Sección 3.1), (ii) uso de n-gramas de exponentes (Sección 3.2.1), (iii) codificación aritmética adaptativa de orden 1 (Sección 3.2.2) y (iv) clustering de capas con distribuciones similares (Sección 3.2.3). A partir de esta base, la solución propuesta se organiza en cuatro componentes principales:

5.1. Núcleo de compresión de exponentes (*ZipNN++*)

El primer componente consiste en consolidar un núcleo de compresión de exponentes que extienda ZipNN. Usaremos el pipeline ya implementado para:

- separar los campos $(\mathbf{s}, \mathbf{e}, \mathbf{m})$ de los pesos FP32;
- aplicar modelos de contexto sobre los exponentes $\mathbf{e}$: Huffman global, Huffman por capa, n-gramas y variantes de orden 1 (codificación aritmética, Sección 3.2.2, y σ-Huffman, Sección 3.3.2);
- agrupar capas mediante clustering basado en distancia de Jensen–Shannon para compartir modelos de probabilidad entre capas similares.

Este núcleo buscará acercarse lo más posible a la entropía observada de los exponentes, estudiando el compromiso entre complejidad del modelo (por ejemplo, σ-Huffman con 256 árboles distintos) y ganancia real en bits. La implementación se realizará en Python sobre `PyTorch` y `HuggingFace Transformers`, reutilizando y extendiendo el código experimental desarrollado para `GPT-2`.

5.2. Modelado de la mantisa y del signo

Mientras que ZipNN supone los campos de signo y mantisa como no comprimibles (o apenas explotados en modelos BF16), NeuralZip intentará incorporar explícitamente su estructura. Para la mantisa $\mathbf{m}$ se explorarán:

- histogramas y entropía marginal por capa y por tipo de tensor (matrices de atención, capas feed-forward, embeddings);
- modelos sencillos de contexto (por ejemplo, diferencias o deltas entre mantisas consecutivas, agrupación de bytes de mantisa, o tratamiento separado de los bits más significativos y menos significativos);
- la posibilidad de explotar patrones repetidos o bloques constantes que conecten con ideas de compresión amigable con la multiplicación matriz–vector [5, 6].

El objetivo no es necesariamente comprimir agresivamente la mantisa, sino entender cuánto margen *lossless* existe y qué variantes son compatibles con un futuro soporte eficiente de operaciones algebraicas sobre el modelo comprimido.

5.3. Esquemas híbridos con técnicas con pérdida

Un tercer componente consiste en estudiar cómo técnicas con pérdida (quantization, pruning, bajo rango) pueden facilitar la compresión *lossless*: por ejemplo, una cuantización previa en 8 o 12 bits induce pesos más estructurados, potencialmente con exponentes y mantisas más predecibles. En esta etapa se considerarán pipelines del tipo:

1. aplicar cuantización o poda moderada (con degradación de calidad controlada);
2. reexpresar los pesos cuantizados en FP32 o en un formato adecuado;
3. aplicar ZipNN++ sobre la nueva distribución de exponentes y mantisas.

La hipótesis a explorar es que, incluso admitiendo una pequeña pérdida de calidad por cuantización o poda, el *plus* de compresión *lossless* sobre esa representación puede producir esquemas combinados más atractivos en memoria total y en facilidad de implementación.

5.4. Estructuras de acceso directo y compatibilidad con operaciones de inferencia

Finalmente, NeuralZip incluirá estructuras de acceso directo sobre el flujo comprimido de pesos. Partiendo del índice por bloques ya implementado para los exponentes (Sección 3.3.1), se estudiarán:

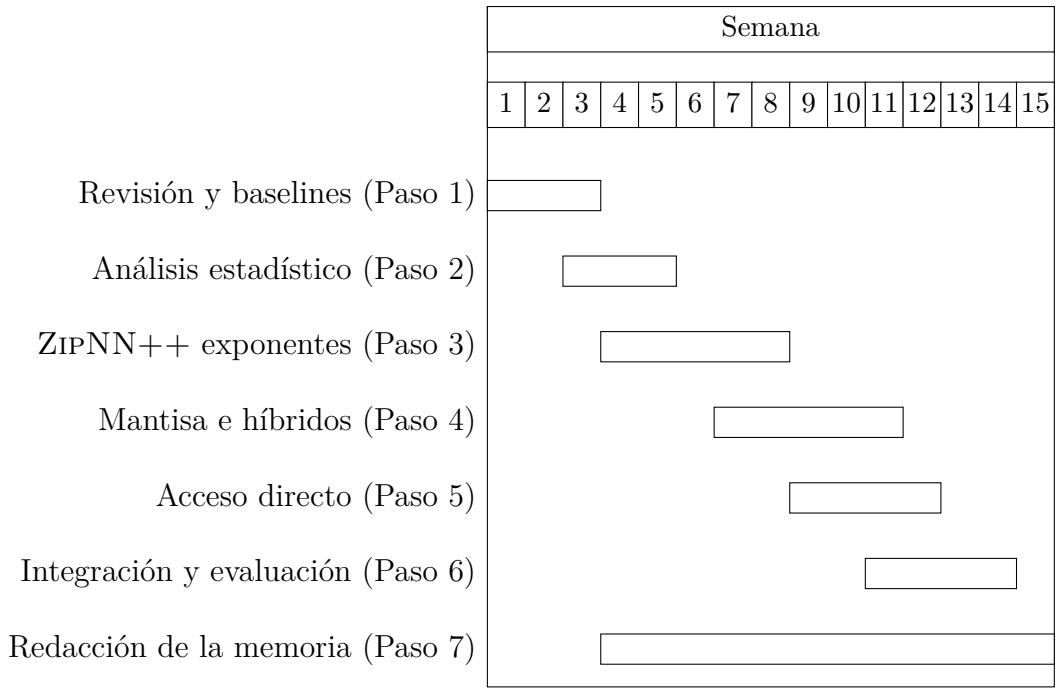
- distintos tamaños de bloque y jerarquías de índices (por ejemplo, índices a dos niveles: bloques grandes y sub-bloques dentro de ellos);
- el costo extra en bits de estos índices y su efecto en la tasa de compresión global;
- escenarios de uso donde el acceso es parcial (por capas o por subconjuntos de parámetros) y no necesariamente se descomprime todo el modelo a la vez.

A largo plazo, la meta es que la representación comprimida sea compatible con ideas de multiplicación matriz–vector sobre datos comprimidos, como las propuestas por Ferragina et al. y Francisco et al. [5, 6], aunque en esta memoria el foco principal estará en la compresión y el acceso eficiente, dejando la aceleración directa de la inferencia como línea de trabajo futura.

En conjunto, NeuralZip se concibe como una familia de esquemas *lossless* parametrizables (tipo de modelo de contexto, granularidad por capa, combinación con técnicas con pérdida e índice de acceso) que permiten trazar la frontera entre el límite de entropía de los pesos y las restricciones prácticas de tiempo y memoria en modelos de lenguaje grandes.

6. Plan de Trabajo

El desarrollo de NeuralZip se organizará en varias etapas secuenciales pero parcialmente solapadas, alineadas con los objetivos específicos de la Sección 4. Cada etapa produce artefactos concretos (código, *scripts* de experimentación, mediciones, figuras) que alimentan a la siguiente. A nivel temporal, un posible cronograma para un semestre de 15 semanas podría ser:



Este plan es preliminar y puede ajustarse en función de los resultados intermedios; sin embargo, proporciona una estructura clara para avanzar de un núcleo de compresión basado en exponentes hacia un esquema NeuralZip más completo, capaz de acercarse al límite de entropía de los pesos manteniendo la exactitud bit a bit y tiempos de acceso razonables.

1. **Revisión y consolidación de baselines.** Profundizar en la literatura relevante sobre compresión *lossless* y con pérdida en modelos de lenguaje, poniendo especial énfasis en ZipNN, compresores generales (GZip, Zstd) y trabajos de compresión amigable con la multiplicación matriz–vector [5, 6]. En paralelo, estabilizar la infraestructura experimental en Python: carga de modelos (GPT-2 y eventualmente otros LMs), extracción de pesos FP32, y ejecución reproducible de los baselines de compresión.
2. **Instrumentación y análisis estadístico de los pesos.** Implementar herramientas para medir, por modelo y por capa, la distribución de signo, exponente y mantisa: histogramas, entropía marginal, entropía condicional y correlaciones simples (por ejemplo, n-gramas de exponentes y mantisas). El objetivo de esta etapa es obtener una estimación clara del límite teórico de compresión y de cuánto espacio explican ya los métodos basados sólo en exponentes.
3. **Revisión de ZipNN++ sobre exponentes.** Extender y sistematizar las variantes de compresión de exponentes: Huffman por capa, n-gramas (pares, trigramas), codificación aritmética adaptativa y σ-Huffman (Sección 3.3.2), junto con clustering de capas según

distancia de Jensen–Shannon (Sección 3.2.3). Se explorará el espacio de parámetros (tamaño de n, umbral de clustering, etc.) y se compararán de forma sistemática las tasas de compresión y tiempos de ejecución frente a ZipNN y a los compresores generales.

4. **Modelado de la mantisa y esquemas híbridos con técnicas con pérdida.** A partir del análisis estadístico de la mantisa, se diseñarán modelos sencillos de compresión *lossless* para sus bits (Huffman de orden 0 u orden 1, codificación por bytes significativos, deltas entre valores) y se medirá cuánto aportan realmente a la compresión total. En paralelo, se implementarán pequeños experimentos de cuantización y poda moderada sobre GPT-2 para observar si la estructura inducida facilita la aplicación de ZipNN++ y cuánta memoria extra se puede ahorrar al combinar ambos enfoques.
5. **Diseño de estructuras de acceso directo.** Basado en el índice por bloques ya implementado para los exponentes, se estudiarán variantes más sofisticadas: cambios en el tamaño de bloque, índices jerárquicos (dos niveles), y posibles extensiones a la mantisa. Para cada variante se medirá el costo extra en bits y el tiempo promedio para acceder a una posición aleatoria o a una capa completa, cuantificando el compromiso entre compresión y capacidad de acceso parcial.
6. **Integración y evaluación de NeuralZip.** Seleccionar las combinaciones más prometedoras de las etapas anteriores (modelo de contexto para exponentes, tratamiento de la mantisa, presencia o no de cuantización/pruning, tipo de índice de acceso) y definir uno o dos esquemas concretos bajo el paraguas de NeuralZip. Estos esquemas se evaluarán en profundidad sobre uno o más modelos de lenguaje, midiendo: (i) tasa de compresión total, (ii) distancia al límite de entropía estimado, (iii) tiempos de compresión y descompresión y (iv) tiempos de acceso a bloques o capas.
7. **Redacción de la memoria y análisis crítico.** Sistematizar los resultados en el documento final de memoria, discutiendo no sólo las mejores tasas de compresión alcanzadas, sino también las limitaciones prácticas encontradas (costos computacionales, complejidad de implementación, sensibilidad a la arquitectura del modelo). Esta etapa incluirá la preparación de tablas comparativas, figuras y una discusión explícita de líneas futuras, como la posibilidad de ejecutar productos matriz–vector directamente sobre la representación comprimida.

Referencias

[1] Achiam, Josh, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat *y cols.*: *Gpt-4 technical report.* arXiv preprint arXiv:2303.08774, 2023.

[2] Belcak, Peter, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin y Pavlo Molchanov: *Small Language Models are the Future of Agentic AI.* arXiv preprint arXiv:2506.02153, 2025.

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

[4] Devlin, Jacob, Ming Wei Chang, Kenton Lee y Kristina Toutanova: *Bert: Pre-training of deep bidirectional transformers for language understanding.* En *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)*, páginas 4171–4186, 2019.

[5] Ferragina, Paolo, Travis Gagie, Dominik Köppl, Giovanni Manzini, Gonzalo Navarro, Manuel Striani y Francesco Tosoni: *Improving matrix-vector multiplication via lossless grammar-compressed matrices.* arXiv preprint arXiv:2203.14540, 2022.

[6] Francisco, Alexandre P, Travis Gagie, Dominik Köppl, Susana Ladra y Gonzalo Navarro: *Graph compression for adjacency-matrix multiplication.* SN Computer Science, 3(3):193, 2022.

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

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

[9] Hochreiter, Sepp y Jürgen Schmidhuber: *Long short-term memory.* Neural computation, 9(8):1735–1780, 1997.

[10] Jin, Sian, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao y Franck Cappello: *DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression.* En *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*, páginas 159–170, 2019.

[11] Kandpal, Nikhil, Brian Lester, Mohammed Muqeeth, Anisha Mascarenhas, Monty Evans, Vishal Baskaran, Tenghao Huang, Haokun Liu y Colin Raffel: *Git-Theta: A Git Extension for Collaborative Development of Machine Learning Models.* arXiv preprint

arXiv:2306.04529, 2023.

[12] Kurtić, Eldar, Elias Frantar y Dan Alistarh: *Ziplm: Inference-aware structured pruning of language models*. Advances in Neural Information Processing Systems, 36:65597–65617, 2023.

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

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

[15] Mikolov, Tomas, Kai Chen, Greg Corrado y Jeffrey Dean: *Efficient estimation of word representations in vector space*. arXiv preprint arXiv:1301.3781, 2013.

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

[17] Radford, Alec, Karthik Narasimhan, Tim Salimans, Ilya Sutskever *y cols.*: *Improving language understanding by generative pre-training*. 2018.

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

[19] Reagen, Brandon, Udit Gupta, Robert Adolf, Michael Mitzenmacher, Alexander Rush, Gu Yeon Wei y David Brooks: *Weightless: Lossy weight encoding for deep neural network compression*. En *International Conference on Learning Representations*, 2018.

[20] Saha, Rajarshi, Naomi Sagan, Varun Srivastava, Andrea Goldsmith y Mert Pilanci: *Compressing Large Language Models using Low Rank and Low Precision Decomposition*. arXiv preprint arXiv:2405.18886, 2024.

[21] Shum, Heung Yeung, Xiao dong He y Di Li: *From Eliza to XiaoIce: challenges and opportunities with social chatbots*. Frontiers of Information Technology & Electronic Engineering, 19(1):10–26, 2018.

[22] Song, Min y Tamy Chambers: *Text mining with the Stanford CoreNLP*. En *Measuring scholarly impact: Methods and practice*, páginas 215–234. Springer, 2014.

[23] Sundrani, Sidhant, Francesco Tudisco y Pasquale Minervini: *Low-Rank Compression of Language Models via Differentiable Rank Selection*. arXiv preprint, 2024. Submitted to ICLR 2025.

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

[25] Xu, Xiaohan, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao y Tianyi Zhou: *A Survey on Knowledge Distillation of Large Language Models*. arXiv preprint arXiv:2402.13116, 2024.

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