Facultad de Ciencias

Análisis de flexibilidad de los modelos LLM cuantizados
(Flexibility analysis of quantized LLM models)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Pablo González San José

Directores: Pablo Abad Fidalgo y Pablo Prieto Torralbo

Febrero – 2025

RESUMEN

Los *Large Language Models* (LLM) utilizados en tareas de procesado de lenguaje, como GPT, Gemini o Llama, han representado un punto de inflexión en el potencial impacto de la Inteligencia Artificial (IA) en nuestra vida diaria. Los requierimientos computacionales derivados del tamaño de estos modelos han favorecido la utilización de mecanismos de optimización como la cuantización de los pesos y funciones de activación. En este contexto, el proyecto trabajará analizando si es posible la aplicación directa de técnicas de mejora sencillas, tanto desde un punto de vista *hardware* (reducción del coste computacional) como *software* (mejora de la precisión del modelo). Haciendo uso de los modelos derivados del LLM de *Meta* (de nombre Llama), analizaremos la viabilidad de dos técnicas: la compresión de modelo basado en la presencia de valores recurrentes y la refactorización de pesos en función de la distribución de valores de los mismos.

ABSTRACT

Large Language Models (LLM) used in language processing tasks, such as GPT, Gemini or Llama, have become a turning point in the potential impact of Artificial Intelligence (AI) on our daily lives. Computational requirements derived from the size of these models have favoured the use of optimization mechanisms, such as weight and activarion functions quantization. In this context, the project will work on analizing if the straightforward approach of simple improvement techniques is possible, both from a hardware point of view (reduced computational cost) and software (improved model precission). Making use of the models derived from Meta's LLM (named Llama), the viability of two techniques will be analized: the compression of the model based on the pressence of recurring values and the refactorization of the weights based on the distribution of their values.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Introducción y Motivación

En los últimos años, los avances en las técnicas de Inteligencia Artificial (IA) han dado lugar a modelos cada vez más sofisticados. Desde las primeras pruebas a finales de los 50, hasta los algoritmos de *deep learning* y la importancia del *Big Data*, en menos de 70 años se ha llegado muy lejos y, recientemente, con la aparición de la arquitectura *transformer*, la IA generativa y los Grandes Modelos de Lenguaje (LLM) se ha alcanzado un nuevo grado de sofisticación.

Los LLM son un tipo de modelo de IA capaz de reconocer y entender el lenguaje natural y utilizar el contexto para inferir significado, siendo capaces de generar respuestas relevantes y coherentes, entre otras tareas. Esto se logra mediante un mecanismo basado en predecir la siguiente palabra (*token*) de la frase que están construyendo, utilizando como entradas tanto la pregunta como el fragmento ya generado de su respuesta. Sin embargo, aunque sus capacidades se enfoquen en el procesamiento del lenguaje natural, los LLM tienen muchas aplicaciones en otros campos que abordan el problema de la transducción de secuencias.

Gracias a los avances técnicos que tuvieron lugar entre 2005 y 2017, el aumento de potencia de los computadores posibilitó la utilización de algoritmos de aprendizaje de IA más sofisticados, siendo el más importante el de *deep learning*. Aplicando estas técnicas, se lograba mejorar el rendimiento de los modelos mediante el incremento del tamaño del *dataset* en 2 o 3 órdenes de magnitud [24], en lugar de alterando el algoritmo de entrenamiento (como había estado ocurriendo hasta entonces). Al mismo tiempo, la digitalización de las infraestructuras de datos dio lugar a la posibilidad de generar mucha más información que, al ser tratada mediante técnicas de *Big Data*, facilitó la creación de *datasets* cada vez más grandes para el entrenamiento de IA [22]. La combinación de estos dos fenómenos es lo que se conoce como la tercera revolución de la IA.

El principal obstáculo de este acercamiento es que cuanto mayor sea el tamaño del modelo, menos eficiente se vuelve, y uno de los principales mecanismos para solucionarlo es paralelizarlo. Sin embargo, los modelos que operan con lenguaje natural tienen un grado de paralelización muy bajo debido a la naturaleza secuencial de las entradas. El problema radica en que, para entender la última parte de una frase, hace falta haber entendido primero todo lo anterior, y el mismo problema surge a la hora de formular la respuesta. Para solucionarlo, los modelos tradicionales aplicaban una mezcla de recursión y mecanismos de atención, mientras que la arquitectura *transformer* ha logrado mejorar el rendimiento de los modelos en paralelo prescindiendo de las primeras [16]. Este avance ha posibilitado el uso de IA, no solo para tareas de procesado de

lenguaje, sino para resolver problemas en multitud de campos de estudio, dando lugar a un crecimiento significativo de su uso.

No obstante, aunque la arquitectura *transformer* ha tenido un impacto positivo en la eficiencia de los LLM, factores como el tamaño de los modelos, el uso intensivo de la memoria o el apoyo en GPUs hacen más difícil aprovecharlos en casos más limitados, como teléfonos móviles, *edge computing*, etc... Así, el objetivo de este trabajo es comprobar la efectividad de mecanismos simples de cuantización y técnicas de *sparsity* para reducir el tamaño de un modelo y mejorar su rendimiento, con el objetivo de facilitar su utilización desde equipos con menor potencia.

La cuantización es una técnica de optimización que reduce el tamaño con el que el modelo se almacena aplicando operaciones de transformación a los tensores para pasar de formatos numéricos flotantes grandes a enteros más pequeños. La pérdida de precisión que conlleva este proceso se mitiga revirtiendo el cambio en el momento de operar, empleando el formato completo.

Las técnicas de *sparsity*, por su parte, reducen el tamaño del modelo y mejoran su rendimiento convirtiendo en 0 muchos valores con poco impacto en el resultado final. Al anularse, guardarlos y procesarlos resulta trivial, mientras que su reducido impacto minimiza la pérdida de precisión en los datos de salida.

Para llevar a cabo el trabajo, se ha empleado el modelo Llama versión 2, desarrollado por *Meta*. La naturaleza *open source* de Llama otorga mayor control sobre el funcionamiento del modelo, así como la capacidad de acceder y cambiar el código fuente y la arquitectura.

El objetivo de este trabajo es comprobar la efectividad de técnicas de cuantización sencillas que permitan aprovechar el *sparsity* para reducir el tamaño de un modelo y mejorar su rendimiento, para facilitar su utilización en equipos menos potentes. De este modo se pretende evaluar, por un lado, en qué medida la compresión de los datos ayuda a mejorar el aprovechamiento de la memoria y a reducir el coste computacional y, por otro, comprobar si es posible implementar un mecanismo alternativo de cuantización tal que la refactorización ofrezca mejor precisión al mismo coste.

El resto del documento se organiza de la siguiente forma:

En primer lugar, se incluye una explicación de la arquitectura de Llama 2 y cómo ésta diverge de la arquitectura *transformer* original, seguido de una descripción del *framework* que se ha utilizado durante el trabajo, llama.cpp. En el final de la sección, se detallan tanto aspectos teóricos sobre la cuantización, como una explicación de su implementación particular en el *framework*.

A continuación, se aborda el primer acercamiento a comprimir el modelo partiendo de la cuantización Q4_0; tanto el análisis inicial de las matrices de pesos, como de la primera implementación de la cuantización y el modelo multi-cero. De manera similar, la siguiente

sección detalla el segundo acercamiento a la compresión, esta vez partiendo de la cuantización Q8_0, mediante implementaciones basadas en la agrupación de valores en rangos de cuantización.

Finalmente, se explican las conclusiones obtenidas.

2. Llama, llama.cpp y cuantización

2.1. Modelos Llama

Este trabajo se centra en la cuantización del modelo Llama, desarrollado por *Meta*, en concreto en su versión Llama 2. Llama es un tipo de modelo *transformer* optimizado para tareas conversacionales, aunque en este caso existen diferencias respecto a la arquitectura *transformer* convencional.

La arquitectura *transformer* se aplica a Redes Neuronales donde el orden de la secuencia de elementos de entrada es relevante (como modelar lenguaje o traducir). El funcionamiento de este tipo de modelos se basa en la predicción y generación de *tokens* utilizando como contexto la entrada y los *tokens* generados anteriormente, hasta generar un *token* de fin. Aunque la naturaleza secuencial de este problema podría tener un impacto negativo en la escalabilidad del modelo, la aplicación de técnicas de atención, así como la arquitectura codificador-decodificador han logrado mitigar este problema en los *transformers* haciéndolo una arquitectura altamente paralelizable. Aplicando las técnicas de autoatención y *Multi-head attention* ha sido capaz de reducir la complejidad del número de operaciones a un valor constante [16], posibilitando el uso de modelos cada vez más grandes.

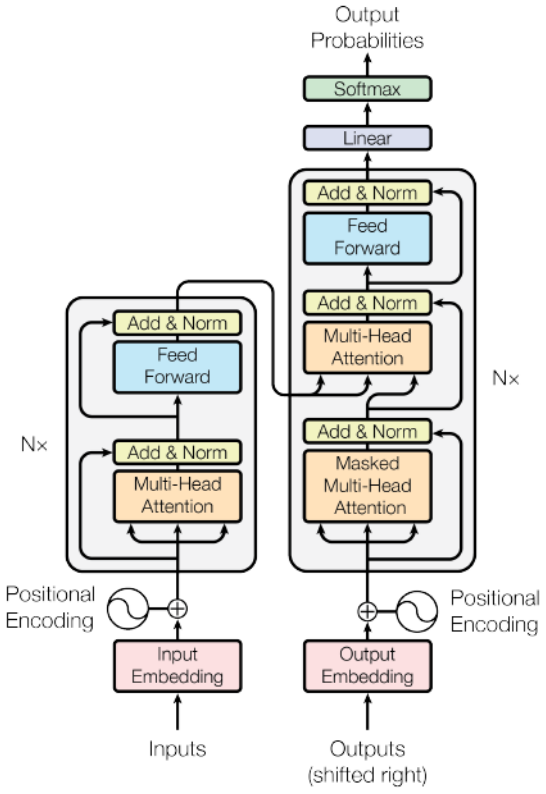

Figura 2.1.1: Esquema de la arquitectura *transformer* [16].

La arquitectura *transformer* sigue la estructura codificador-decodificador típica de los modelos de transducción de secuencias, donde el codificador mapea la secuencia de entrada (símbolo) a una secuencia de representación contínua, que es enviada al decodificador para generar la secuencia de salida (símbolo) *token* a *token*. Sin embargo, a diferencia de otras arquitecturas, en lugar de utilizar recurrencia para consumir los *tokens* generados anteriormente, los modelos *transformer* utilizan *scaled dot-product attention*.

El mecanismo de atención que se aplica en la arquitecrura *transformer* es una variación de la atención multiplicativa, que aplica el producto escalar para mapear vectores de consultas (Q) y parejas llave-valor (K-V) a salidas. Para obtener el resultado final, esta función de atención se aplica paralelamente a *h* proyecciones lineales de Q, K y V (cabezas). Esta técnica es *multi-head attention* (MHA).

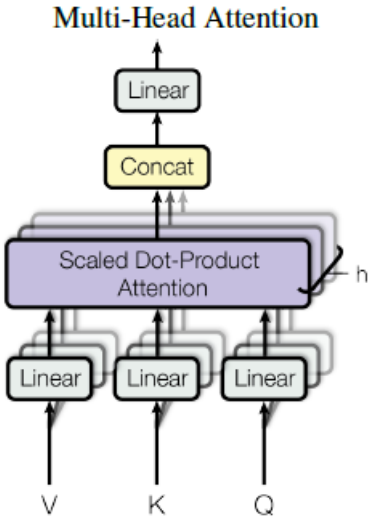

Figura 2.1.2: Esquema del funcionamiento de MHA [16].

De este modo, como durante la operación de atención se computan tanto el *token* actual como los anteriores, existe una cantidad significativa de operaciones repetidas en cada paso. Para abordar este problema, Llama aplica la técnica de *KV cache*, que consiste en guardar las parejas KV de los *tokens* anteriores, convirtiendo las operaciones repetidas en accesos a memoria.

Sin embargo, a medida que crece el tamaño del modelo, el uso intensivo de la memoria lo convierte en un cuello de botella, limitando el rendimiento y la escalabilidad. En el caso de Llama, se aplica la técnica de *grouped-query attention* (GQA). En lugar de mantener una Q, K y V únicas por cada *h*, un número *G* de Q en cabezas diferentes comparten el mismo KV. GQA con $G = h$, por ejemplo, es equivalente a MHA. Esta técnica ha demostrado mejorar significativamente en aprovechamiento de la memoria por parte de Llama, y con ello su escalabilidad, al mismo tiempo que mantiene una precisión muy similar a la de MHA [8].

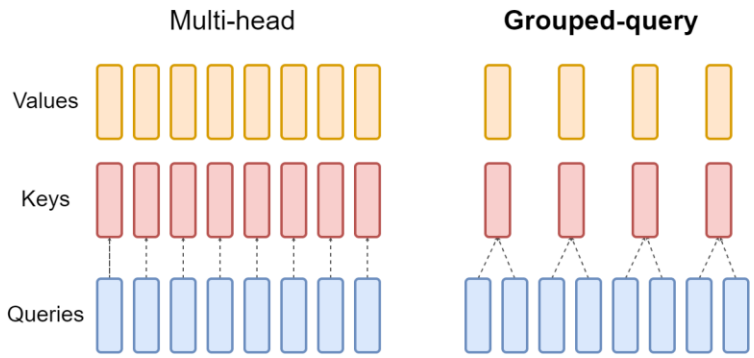

Figura 2.1.3: Comparación de MHA y GQA para *h* consultas [17].

	BoolQ	PIQA	SIQA	Hella-Swag	ARC-e	ARC-c	NQ	TQA	MMLU	GSM8K	Human-Eval
MHA	**71.0**	**79.3**	48.2	75.1	71.2	**43.0**	12.4	44.7	**28.0**	4.9	**7.9**
MQA	70.6	79.0	47.9	74.5	71.6	41.9	**14.5**	42.8	26.5	4.8	7.3
GQA	69.4	78.8	**48.6**	**75.4**	**72.1**	42.5	14.0	**46.2**	26.9	**5.3**	**7.9**

Tabla 2.1.1: Comparación de la precisión alcanzada por 3 modelos similares aplicando MHA, GQA y *multi-query attention* [8].

Otra diferencia significativa entre la arquitectura de Llama y la *transformer* es el cambio en la capa de normalización. En lugar de normalizar la salida de cada capa siguiendo la función LayerNorm, Llama normaliza la entrada siguiendo la función RMSNorm. Mientras que LayerNorm se caracteriza por la invarianza en el recentrado y reescalado de los valores para mantener al modelo estable frente al ruido y escalado de los tensores, respectivamente, RMSNorm propone que únicamente la última de estas propiedades tiene un efecto significativo [18]. También emplea la función de activación *SwiGLU*, en lugar de *ReLU*.

Finalmente, Llama cambia el método de codificación posicional en la entrada de uno estático a uno rotatorio (RoPE). Con este sistema, el modelo es capaz de rotar el vector de atención QK al recibir la entrada. Esta implementación reduce el ritmo de decadencia del modelo a largo plazo, al mismo tiempo que propone una implementación más directa y, teóricamente, más computacionalmente eficiente de la función de atención [19].

Meta ofrece diversos modelos para aprovechar las capacidades de Llama y para este trabajo se ha utilizado la versión 2, que era la más actual en ese momento. Esta versión abarca desde los 7B hasta los 70B parámetros y para este trabajo se ha utilizado el modelo de 7B, ya que, aunque pequeño, es razonablemente preciso y, al cuantizarse, llega a generar entre 13 y 20 tokens/s, siendo superior a la velocidad de lectura humana, que ronda los 7 a 10 tokens/s. Para el modelo de 13B, este valor ya se reduce a los 8 a 10 tokens/s [23].

2.2. Llama.cpp

Llama.cpp es una *framework* de código abierto que implementa la lógica interna de otros LLM, como Llama o mistral, utilizando C++. El objetivo del proyecto es optimizar los LLM para mejorar el rendimiento de su ejecución en la CPU, reduciendo los requerimientos computacionales para que investigadores o particulares puedan experimentar con los modelos aunque no tengan acceso a las potentes infraestructuras que se requieren para el desarrollo de LLM. Esto ha hecho que se convierta en uno de los *frameworks* más extendidos para trabajar con LLM.

Llama.cpp se desarrolla paralelamente a la librería de gestión de operaciones con tensores GGML, del mismo autor y también de código abierto, y se enfoca en ofrecer herramientas que permitan el uso de inferencia con LLM en equipos *hardware* menos potentes.

GGML ofrece multitud de mecanismos de cuantización que se organizan en 2 partes de implementación y logística. La parte de implementación consiste en 2 ficheros que tienen todo lo necesario para hacer funcionar todos los tipos de cuantización, desde las propias funciones, hasta las estructuras de datos de gestión de cada una. La parte de logística está formada por el árbol de funciones que se recorre para llevar el flujo de ejecución a la implementación de la cuantización escogida. Está repartido por varios ficheros y desempeña tareas tales como la gestión de memoria previa y posterior a la cuantización, la lectura y escritura del tensor en ficheros, la subdivisión del tensor en bloques, etc… así como la gestión de estructuras de datos especiales que identifican y configuran cada cuantización (tipos de datos que utiliza, punteros a la función de cuantización y decuantización correspondientes, punteros a la estructura de datos específica, etc…).

De este modo, para implementar una función de cuantización nueva será necesario abordar ambas partes.

2.3. Cuantización

Actualmente uno de los mayores problemas a la hora de desarrollar IA es el alto coste de la infraestructura que las soporta. Esto es especialmente importante en los *transformers*, con modelos que se miden en miles de millones de parámentros. Un problema radica en el uso de formatos numéricos de alta precisión en elementos como las matrices de pesos, que mejoran la exactitud y el rendimiento del modelo, pero tienen un impacto negativo tanto en el rendimiento computacional de las aplicaciones de *deep learning*, como en los requerimientos de almacenamiento físico. Una solución a este problema consiste en la utilización de formatos numéricos más pequeños, como enteros de 8b, siempre y cuando sea posible mantener la precisión del modelo en un rango razonable.

La cuantización es una técnica que puede mejorar el rendimiento computacional de las Redes Neuronales y reducir el tamaño de las mismas aprovechando las características de las operaciones con números de menor precisión. En primer lugar, la reducción del tamaño de las palabras tiene como consecuencia un mayor aprovechamiento de la memoria, lo cual repercute positivamente en otros aspectos como la eficiencia de la jerarquía de memoria. La utilización de palabras más pequeñas alivia la presión sobre el ancho de banda del sistema, aumentando el *thoughput* y mejorando el rendimiento en infraestructuras limitadas por el ancho de banda. Adicionalmente, muchos procesadores y unidades de aceleración hardware cuentan con unidades de procesamiento matemático más rápidas para formatos numéricos con menos bits, mejorando el rendimiento y el *throughput* al realizar operaciones matemáticamente intensas [12].

Tipo de dato Entrada	Tipo de dato Acumulación	Throughput Matemático	Reducción Ancho de Banda
FP32	**FP32**	**1x**	**1x**
FP16	**FP16**	**8x**	**2x**
INT8	**INT32**	**16x**	**4x**
INT4	**INT32**	**32x**	**8x**
INT1	**INT32**	**128x**	**32x**

Tabla 2.3.1: Beneficios de tipos de datos de precisión reducida para operaciones con tensores en GPUs de NVidia [12].

El objetivo del proceso de cuantización es la reducción del tamaño del modelo final. Para lograrlo, se lleva a cabo una reducción del número de bits de los parámetros de aprendizaje de todas las capas de la Red Neuronal, tratando de minimizar al mismo tiempo, el impacto en la exactitud del modelo. Para lograrlo, el proceso se puede dividir en 2 fases:

1. Determinar el rango de números reales a cuantizar y aproximar aquellos que queden fuera al más cercano.
2. Mapear los números reales a enteros representables en el número de bits escogido para el valor cuantizado.

El rango de valores reales a cuantizar se representa como $[\beta, \alpha]$ y el número de bits que se van a utilizar para guardar la transformación es b. De este modo, el rango de valores cuantizados es $[-2^{b-1}, 2^{b-1} - 1]$ y el objetivo es, dado $r \epsilon [\beta, \alpha]$, encontrar un valor equivalente perteneciente a este dominio de cuantización. Un aspecto relevante de la cuantización es la función que calcula esta equivalencia y existen varios acercamientos, principalmente la cuantización uniforme y la no uniforme. En la cuantización uniforme, el dominio de cuantización se divide en valores equidistantes a los que se asignan los números reales (Figura 2.3.1. izquierda), mientras que en la cuantización no uniforme, la distancia entre los valores cuantizados puede ser variable (Figura 2.3.1. derecha).

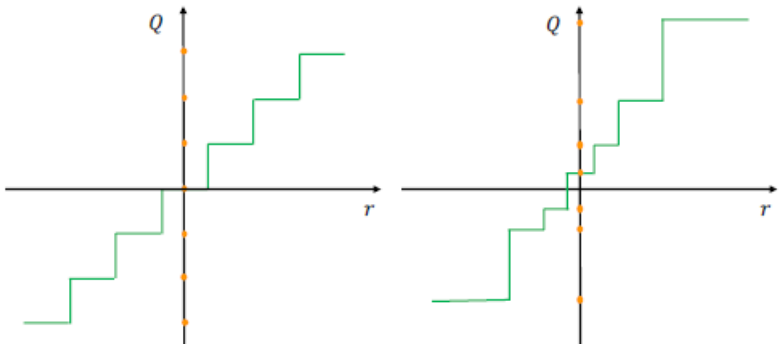

Figura 2.3.1: Comparación entre cuantización uniforme (izquierda) y no uniforme (derecha) [11]. Valores cuantizados en el eje vertical, valores reales en el eje horizontal.

Otros aspectos relevantes de la cuantización son las constantes *S* y *Z*. *S* es el factor de escalado, que se calcula a partir del rango del dominio de origen y el rango del dominio de cuantización y determina la proporción en la que los valores originales se transforman, teniendo un alto impacto en la pérdida de precisión de la cuantización. *Z*, por otro lado, es un desplazamiento que se añade para mover los valores cuantizados, mejorando el aprovechamiento del ancho de palabra.

Al mismo tiempo que se aplica una operación de cuantización para convertir los valores, para utilizar el modelo es necesario poder revertir la transformación mediante una función de decuantización. Sin embargo, debido a errores de redondeo, los valores recuperados de esta manera no tienen por qué coincidir con los valores originales, resultando en una pérdida de precisión del modelo.

La función que describe la cuantización uniforme es la siguiente:

$$Q(r) = Int(r \cdot S) + Z \quad (1)$$

Donde $r, S, Z \in \mathbb{R}$ y *r* es el valor real a cuantizar, *S* es el factor de escalado, que se encarga de dividir el rango de valores reales en particiones uniformes, y *Z* es el desplazamiento del punto medio del dominio de origen respecto al punto medio del dominio de cuantización. La función *Int* se encarga de redondear el número real a un entero en el dominio de cuantización y corregirlo si se escapa del rango.

De este modo, el comportamiento de la función de cuantización viene determinado por cómo se calcule el factor de escalado *S*.

$$S = \frac{2^b - 1}{\alpha - \beta} \quad (2)$$

A este tipo de cuantización uniforme se la denomina cuantización asimétrica, dado que el valor de β y α no se ve atado a ninguna restricción, ofreciendo mayor control sobre el rango de números reales. Esto permite establecer dominios más ajustados a las necesidades de cada problema, haciendo un uso más eficiente de los valores disponibles a costa de aumentar la complejidad lógica del sistema y empeorar el rendimiento en ciertos casos.

En este caso la función de decuantización sigue la siguiente fórmula:

$$D(q) = \frac{q - Z}{S} \quad (3)$$

Alternativamente, existe otro tipo de cuantización uniforme, denominada cuantización simétrica donde, tanto el rango de números reales y como el de cuantización son simétricos alrededor de 0. Esto simplifica la fórmula de cuantización haciendo que el mapeo de rangos durante la cuantización se realice como una transformación de escala. En este tipo de cuantización se cumple que:

$$-\beta = \alpha = \max(|r_{min}|, |r_{max}|) \quad (4)$$

Siendo r_{min} y r_{max} el valor real mínimo y máximo del dominio de origen, respectivamente.

Cabe destacar que este funcionamiento implica que para enteros de 8b, por ejemplo, es necesario utilizar el rango de cuantización restringido [-127, 127], desechando el valor 128. Mientras que la pérdida de un valor del rango de cuantización no tiene por qué repercutir de manera significativa en la precisión, en casos donde el ancho de palabra es menor, la utilización de la cuantización simétrica puede resultar perjudicial.

En cualquier caso, el uso de la cuantización simétrica simplifica la fórmula de cuantización al cumplirse que *Z = 0*.

$$Q(r) = Int(r \cdot S) \quad (5)$$

$$S = \frac{2^{b-1} - 1}{\alpha} \quad (6)$$

$$D(q) = \frac{q}{S} \quad (7)$$

La principal ventaja de utilizar cuantización simétrica es la mejora de rendimiento que supone la simplificación de los cálculos al hacer que $Z = 0$.

Una Red Neuronal está formada por un conjunto de capas que transforman las entradas a medida que éstas las atraviesan. Para conseguirlo, están formadas por un conjunto de matrices de pesos, denomiadas filtros, que aplican las modificaciones. Dichos filtros son los tensores que interesa cuantizar y las características entre ellos, principalmente el rango del dominio de origen, no tienen por qué ser homogéneas. Por este motivo, se opta por establecer una granularidad de cuantización, que determina cuántos elementos compartirán los parámetros de cuantización a lo largo del modelo. Una granularidad demasiado pequeña, donde muy pocos elementos comparten los parámetros de cuantización, supondría una mejora de espacio muy baja respecto al modelo original, mientras que una granularidad demasiado grande empeoraría significativamente la precisión. De este modo, el objetivo es encontrar un equilibrio entre ambos aspectos. Existen varias alternativas:

- **Granularidad por capa**: El rango del dominio de origen se determina a partir de todos los filtros de una capa y es compartido por todos ellos. Este acercamiento es el más sencillo, pero generaliza las particularidades de cada filtro, independientemente de cómo de significativas son las variaciones entre ellos.
- **Granularidad por grupo**: En este caso se agrupan varios filtros de la misma capa y el rango del dominio de origen se determina y es compartido por todos los elementos de cada grupo. Este acercamiento tiene la ventaja de reducir el tamaño del modelo de manera similar a la granularidad por capa, mejorando la precisión en casos donde la distribución de los parámetros varía mucho.
- **Granularidad por filtro**: El rango del dominio de origen se determina y es compartido por todos los elementos de un mismo filtro. Este acercamiento es la opción más popular ya que mejora la precisión del modelo respecto a las anteriores, a cambio de una reducción de tamaño aceptable.
- **Granularidad sub-tensor o sub-filtro**: En este caso se agrupan varios elementos dentro de un tensor y el rango del dominio de origen se determina y es compartido únicamente por ellos. Esta opción ofrece la mayor preservación de la precisión a costa de aumentar el espacio del modelo cuantizado significativamente y es la que se ha utilizado en las cuantizaciones observadas.

GGML ofrece múltiples mecanismos de cuantización y es posible escoger aquel que aproveche mejor las características de cada modelo, aunque hay aspectos de su funcionamiento que son comunes a todos ellos. En este estudio nos hemos centrado en entender las cuantizaciones Q4_0 y Q8_0.

En la cuantización Q4_0, primero se divide el tensor por filas y se pasan a la función de cuantización. Después, para cada fila, la función de cuantización las divide en bloques más pequeños de tamaño *QK4_0 = 32* elementos (este valor es parametrizable en la librería de *ggml*) y cuantiza cada uno de ellos de manera independiente, siendo la granularidad sub-tensor. Finalmente, se lleva a cabo la cuantización bloque por bloque, que a su vez tiene 3 fases:

1. Se recorre el bloque entero para determinar el valor real máximo α.
2. Se calcula *S* siguiendo una variación de la fórmula de la cuantización simétrica:

$$S = \frac{-2^{b-1}}{\alpha}$$

3. Se cuantizan los elementos del bloque.

De cara al proceso de decuantización, los resultados de cada bloque se guardan en una estructura *block_q4_0* que está compuesta de una lista de valores cuantizados agrupados en enteros de 8b *unsigned*, así como la variable *S*. Esta información es posteriormente escrita en un fichero de salida junto a una cabecera que indica, entre otras cosas, el tipo de cuantización utilizada. Gracias a esto, posteriormente, llama.cpp es capaz de identificar los procesos de decuantización correspondientes de manera automática. De este modo, para la decuantización Q4_0 se lleva a cabo la misma división en filas y luego en bloques *QK4_0 = 32* y se decuantizan los datos con la *S* correspondiente.

Atendiendo a la implementación concreta de la cuantización de los bloques, se sigue el mecanismo detallado en la Figura 2.3.2:

```
for (int j = 0; j < qk/2; ++j) {

    const float x0 = x[i*qk + 0    + j]*id;
    const float x1 = x[i*qk + qk/2 + j]*id;

    const uint8_t xi0 = MIN(15, (int8_t)(x0 + 8.5f));
    const uint8_t xi1 = MIN(15, (int8_t)(x1 + 8.5f));

    y[i].qs[j]  = xi0;
    y[i].qs[j] |= xi1 << 4;
}
```

Figura 2.3.2: Implementación del mecanismo de cuantización de elementos del bloque en la cuantización Q4_0 de la librería *ggml*.

Observando el código, se aprecia como la cuantización Q4_0 presenta algunas peculiaridades. En primer lugar, se observa como, al tratar valores en formato de 4b, se hace un uso más eficiente del ancho de banda y la memoria operando los valores *xi0* y *xi1* por parejas, y guardándolos en el mismo *Byte*. En segundo lugar, aunque está utilizando la cuantización uniforme simétrica, después de cuantizar se aplica un desplazamiento $Z = 8.5$. Este valor cumple 2 propósitos: por un lado el desplazamiento de 8 permite eliminar los números negativos y poder guardar los datos en enteros sin signo, haciendo un uso más eficiente del ancho de palabra, y por otro, el desplazamiento de 0.5 fuerza un redondeo hacia arriba cuando se lleve a cabo la conversión de flotante a entero.

En la cuantización Q8_0 la función recibe el tensor entero y después lo divide en bloques de tamaño *QK8_0 = 32* elementos que cuantiza de manera independiente, siendo la granularidad también sub-tensor. La cuantización se lleva a cabo en 3 fases:

1. Se recorre el bloque entero para determinar el valor real máximo absoluto α.
2. Se calcula S siguiendo la fórmula de la cuantización simétrica

$$S = \frac{2^{b-1} - 1}{\alpha}$$

3. Se cuantizan los elementos del bloque.

Al igual que en la cuantización Q4_0, los resultados de cada bloque se guardan en una estructura *block_q8_0* que está compuesta de una lista de valores cuantizados en enteros de 8b, así como la variable *S*.

Atendiendo a la implementación concreta de la cuantización de los bloques, se sigue el mecanismo detallado en la Figura 2.3.3:

```
for (int j = 0; j < QK8_0; ++j) {

    const float x0 = x[i*QK8_0 + j]*id;

    y[i].qs[j] = roundf(xi0);
}
```

Figura 2.3.3: Implementación del mecanismo de cuantización de elementos del bloque en la cuantización Q8_0 de la librería *ggml*.

A diferencia de la cuantización Q4_0, Q8_0 refleja una implementación más directa de la cuantización simétrica, destacando la ausencia de desplazamiento en la conversión ya que cada *byte* contiene un único valor cuantizado. Cabe recordar que Q8_0 aprovecha el rango negativo de valores de 8b [-127, 127], omitiendo -128 para mantener la simetría.

3. Compresión adicional de modelos

En esta sección aplicaremos cambios sencillos en los mecanismos de cuantización de llama.cpp, comprimiendo todavía más el modelo, y estudiaremos cómo afecta al rendimiento de la ejecución sobre la CPU.

En *hardware*, la presencia de valores cero habituales puede proporcionar ventajas. Por un lado, las operaciones de multiplicación por cero son triviales y podrían evitarse, beneficiando la velocidad de procesamiento, mientras que los datos 0 podrían comprimirse si están suficientemente agrupados, reduciendo el espacio que ocupan tanto en disco como en memoria. Una de las características principales de la cuantzación simétrica es que el dominio de números reales está centrado en cero, pero sin tener en cuenta la distribución de los valores a lo largo de dicho rango. Analizaremos la distribución de valores 0 en el modelo cuantizado para, a continuación, aprovechar la transformación que se lleva a cabo durante la cuantización para acumular los valores más frecuentes en 0 con la intención de comprimir el modelo.

3.1. Análisis de pesos del modelo

El primer paso para crear el mecanismo de cuantización de ceros consiste en determinar cuál es la distribución de valores en la cuantización original, concretamente cuáles son los valores más comunes en el dominio de cuantización, con el objetivo de anularlos. Para ello, se recogen los valores de los tensores ya cuantizados y se analizan.

Para tomar los datos, la librería propia construye un histograma del número de ocurrencias de cada valor del dominio de cuantización, obteniendo resultados para todo el modelo de Llama 2. En la Figura 3.1.1 se muestran los resultados normalizados para la cuantización Q4_0 que lleva a cabo llama.cpp (simple), con rango de cuantización [0, 15], para pasar de valores *fp32* a *int8*.

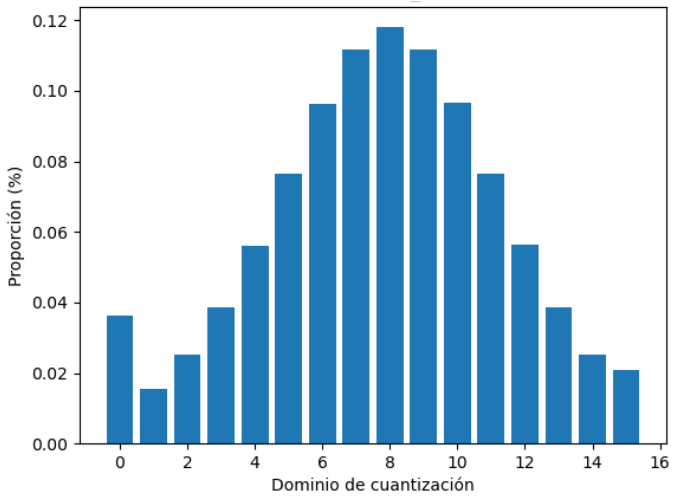

Figura 3.1.1: Histograma de la distribución de los valores cuantizados para todo el modelo de Llama 2 (llama.cpp) aplicando la cuantización Q4_0 simple.

Como se puede apreciar, la cuantización se centra alrededor de 8 y no hay demasiados 0, aunque, esto es algo que se puede cambiar usnado una codificación distinta de los valores, aplicando un desplazamiento.

3.2. Distribución de ceros

A la vista de lo observado en la Figura 3.1.1, vamos a tratar de aumentar el número de ceros en nuestro modelo cambiando la codificación de los valores con 4 bits. La implementación de la nueva función de cuantización *pablo_q4_0* tiene como objetivo replicar el comportamiento de la original de llama.cpp, centrando los valores en 0. Para ello, aprovechando que la distribución de valores original sigue la normal alrededor de 8, aplica un desplazamiento de -8 a todo el dominio de cuantización, pasando al rango [-8, 7]. El resto de valores como el tipo de dato de origen se mantienen con respecto a la implementación original.

Para realizar las pruebas e implementar los cambios en el modo de cuantización se han creado funciones propias y se han modificado aspectos del código original de llama.cpp. Todo el código desarrollado se encuentra en la librería propia *pablo.h* y el fichero *pablo.c*. Todas las llamadas que hagan uso del código modificado caen hasta este fichero.

Con el objetivo de introducir la menor cantidad de modificaciones posibles al código original de llama.cpp, se ha estudiado a fondo el flujo de ejecución de la cuantización y decuantización. Cada tipo de cuantización tiene asociadas una serie de entradas en un conjunto de estructuras de datos que la librería *ggml* utiliza para modificar el flujo de ejecución. Estas tablas es como llama.cpp relaciona las funciones de decuantización con sus homólogas, ajusta los tipos de dato para reducir el tamaño del modelo cuantizado, se adapta a los mecanismos de gestión de la memoria, etc…

En la primera implementación de la librería propia, se añadieron entradas nuevas para crear una cuantización particularizada e independiente sin restringirse a las peculiaridades de la implementación de cada cuantización. Sin embargo, debido a la complejidad del sistema, esta solución probó ser insuficiente a la hora de cambiar el tipo de cuantización a estudiar.

La segunda implementación no añade ninguna estructura nueva y en su lugar aprovecha las entrada ya existentes, alterando el funcionamiento de las cuantizaciones básicas. Para ello, cuando se hace una llamada a la función de cuantización, primero se ejecutan las funciones planficadoras de la librería *pablo.h*. Luego, en función de la información del fichero de configuración *pablo.conf*, se llama a las funciones de cuantización originales o a las modificadas. Esta aproximación es mucho más fácil de implementar para múltiples tipos de cuantización, ya que no requiere cambios en el entorno de llama.cpp y asegura que el estado del sistema durante la cuantización sea el mismo. A cambio, las modificaciones están sujetas a la granularidad, ancho de palabra de cuantización, división previa de los tensores, etc… por defecto de los tipos de cuantización que hay disponibles.

De este modo, se han obtenido los resultados siguientes:

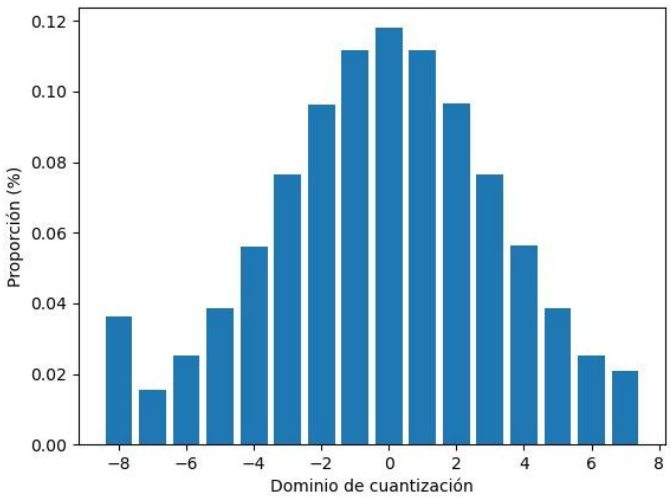

Figura 3.2.1: Histograma de la distribución de los valores cuantizados para todo el modelo de llama.cpp aplicando la cuantización Q4_0 propia.

Como se puede observar, la función centra los valores en 0 como se esperaba.

A continuación, para estudiar la eficacia del nuevo mecanismo de cuantización de cara a una posible compresión de los valores, es necesario comprobar el grado de agrupación de los 0 en los tensores. Idealmente, para sacar el máximo partido a la compresión, debería observarse una mayor presencia de grupos grandes de 0 seguidos. En el caso de que la distribución de los mismos fuera dispersa, la eficiencia del mecanismo de compresión se vería reducida.

Para averiguarlo, la librería propia utiliza un nuevo histograma con 16 entradas que mide el número de ocurrencias de *clusters* de ceros de tamaños 1 a 15, juntando aquellos grupos mayor de 15 en la última entrada.

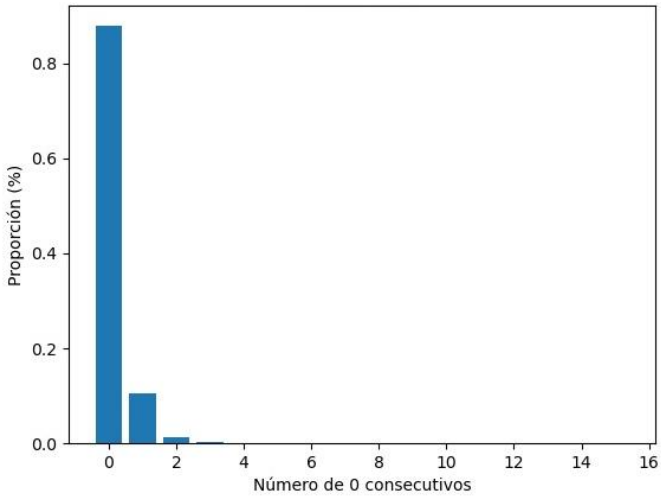

Figura 3.2.2: Histograma del número de ocurrencias de *clusters* de ceros consecutivos en todo el modelo de llama.cpp aplicando la cuantización Q4_0 propia.

Como se puede observar, la inmensa mayoría de los ceros cuantizados (87,83%) se encuentran aislados. Este resultado implica que la distribución de los ceros en el sistema es prácticamente aleatoria, por lo que un mecanismo de compresión basado únicamente en su adyacencia resulta ineficiente. Aunque todavía es posible mejorar el rendimiento evitando las operaciones con 0 en la CPU, el cuello de botella de estas aplicaciones está en el uso del ancho de banda a memoria, y reducir el tamaño del modelo tendría un mayor impacto en el rendimiento.

3.3. Modelo Multi-Cero

A continuación se propone un mecanismo que trata de aumentar el número de ceros adyacentes en el modelo cuantizado aumentando la cantidad de valores que se cuantizan a 0 en general. Como se comprobó en las primeras medidas, los valores en el modelo cuantizado de Q4_0 siguen una distribución normal, por lo que los valores adyacentes al centro son los segundos más frecuentes.

De este modo, una forma sencilla de aumentar el número de ceros del modelo resultante es aumentar el rango de valores alrededor del centro del dominio de cuantización que se convierten en cero. Este cambio tendría un impacto claro en la eficiencia del modelo, pero nos servirá para averiguar que ganancia es posible obtener. Para lograrlo, se han implementado los cambios en la función de cuantización propia, y se ha incluido un nuevo parámetro en el fichero de configuración *pablo.conf* llamado *Q4_0_RADIUS* que determina el valor de este radio. Así, es posible tomar medidas para diferentes radios de forma más rápida y sencilla.

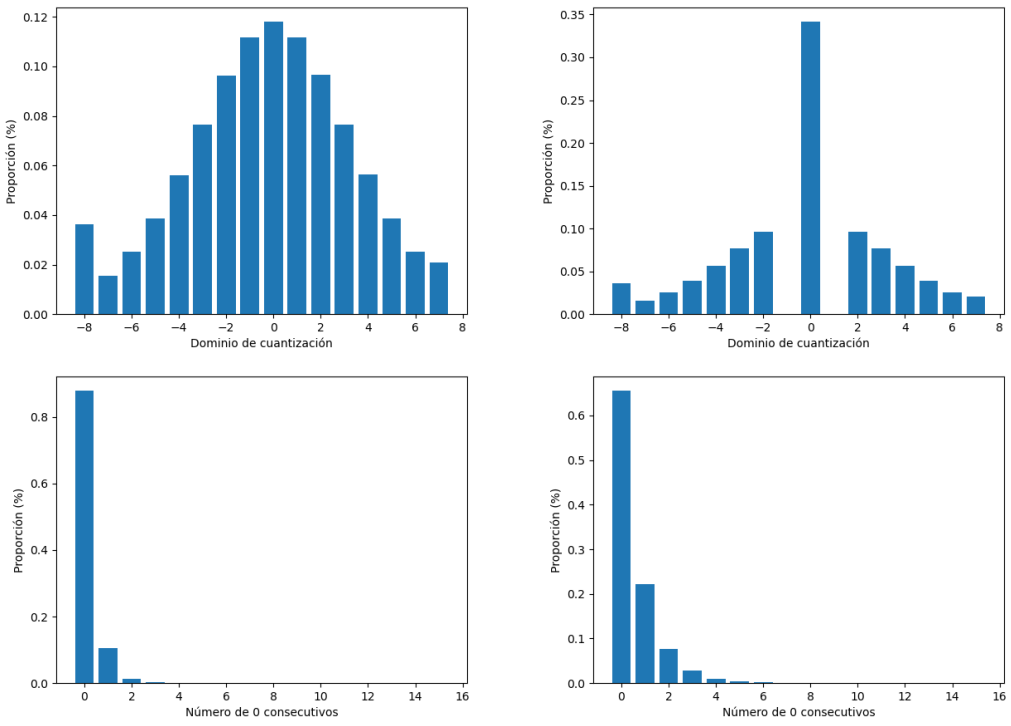

Figura 3.3.1: Comparación de los histogramas de distribución de los valores y grado de agrupación de la cuantización Q4_0 propia normal (izquierda) y haciendo 0 los valores en un radio de 1 alrededor del centro (derecha), para todo el modelo de llama.cpp.

Como se puede observar, el cambio tiene el efecto esperado en la cuantización y agrupación del modelo y, aunque la distribución sigue siendo aleatoria, la proporción de ceros aislados baja a un 65,54%. De hecho, es posible reducir este porcentaje en mayor medida aumentando el radio.

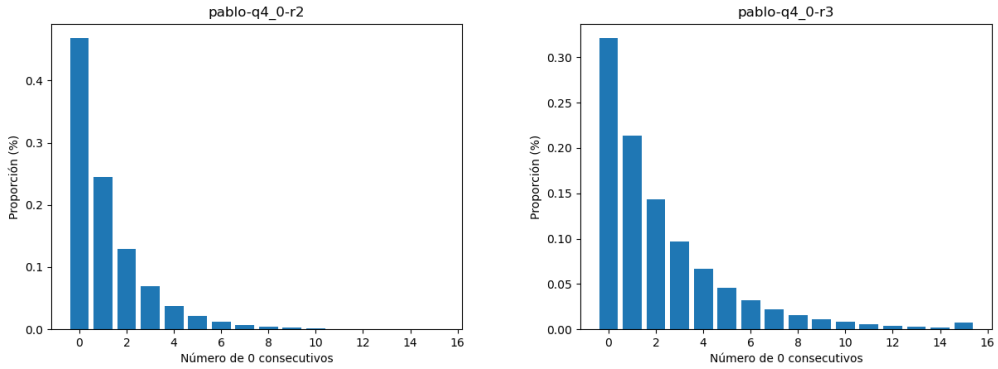

Figura 3.3.2: Comparación de los histogramas del grado de agrupación de la cuantización Q4_0 propia haciendo 0 valores en un radio de 2 (izquierda) y 3 (derecha) valores alrededor del centro, para todo el modelo de llama.cpp.

Sin embargo, al aumentar el radio del modelo multi-cero, también aumenta el número de valores reales que se mezclan y, al decuantizarse al mismo número del dominio de números reales, se pierden. Esto resulta en una reducción en la precisión del modelo final, por lo que es importante estudiar el impacto que tiene el mecanismo. Para ello se ha utilizado la métrica de *perplexity*, que cuantifica en qué medida una distribución de probabilidad es capaz de realizar una predicción a partir de una muestra dada. En el campo de la IA, es una métrica medida muy útil para determinar hasta qué punto el modelo ha sido condicionado por los datos de entrenamiento. Cuanto menor sea el valor de *perplexity* para un modelo, más precisas serán sus predicciones. Sin embargo, tras analizar los datos, la precisión de las cuantizaciones con ceros suponen un deterioro significativo en la capaciad del modelo de realizar predicciones correctas.

De cualquier manera, a la vista de los resultados obtenidos, se puede concluir que no ha sido posible generar un modelo cuantizado más comprimido que mantenga el mismo grado de precisión, empleando mecanismos de cuantización tan sencillos. Sin embargo, los resultados proporcionados por las pruebas realizadas han aportado información sobre el funcionamiento de la cuantización en llama.cpp que puede ser aplicado para probar mecanismos de cuantización alternativos.

4. Recuantizado de pesos

En el mecanismo de cuantización propio implementado para Q4_0 la optimización se consigue convirtiendo los valores cuantizados del dominio de cuantización original al modificado con una relación de uno a uno. El objetivo del nuevo mecanismo de cuantización es comprimir el modelo generado agrupando múltiples valores del dominio de cuantización original en rangos, que se convierten en un mismo valor. Para ello se ha modificado la función de cuantización Q8_0 para convertir los valores al dominio de cuantización de Q4, es decir, pasar de un modelo de cuantización de 8b a uno de 4b. Lo que se busca es obtener una cuantización de 4b con una precisión más cercana a 8b, aprovechando lo que conocemos sobre su distribución de valores.

El principal problema asociado a este mecanismo es la pérdida de precisión del modelo. Dado que todos los valores del mismo rango se reducen a uno solo, la información sobre la variabilidad del intervalo se pierde en el proceso. Además, como en este caso el rango del dominio de cuantización está fijado en 4b, todo el peso de la precisión del modelo final recae sobre cómo se haga este reparto. Una primera aproximación consistiría en utilizar rangos más pequeños para guardar aquellos valores que requieran mayor precisión.

4.1. Análisis inicial

El primer paso para implementar el mecanismo consiste en analizar la distribución de valores en Q8_0, con el objetivo de diseñar un mecanismo de cuantización a medida, tratando de minimizar la pérdida de información (teniendo en cuenta que ya se ha perdido parte en la cuantización Q8_0). De este modo, se ha modificado la función de cuantización para generar el histograma en formato *.json*.

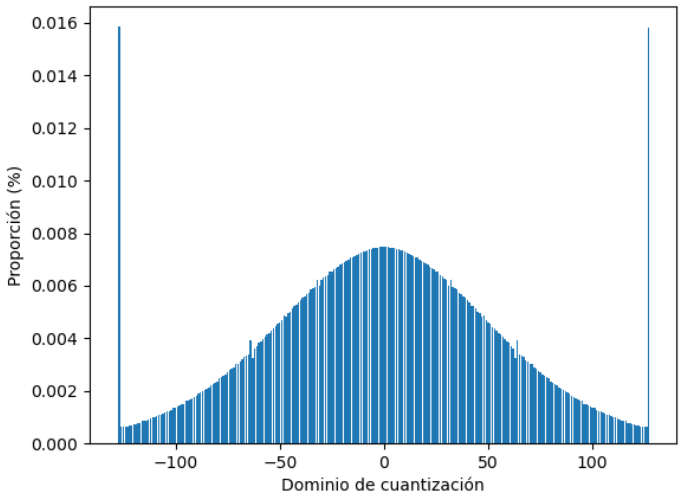

Figura 4.1.1: Histograma de la distribución de los valores cuantizados para todo el modelo de llama.cpp aplicando la cuantización Q8_0 simple.

Como se puede apreciar, al igual que en el mecanismo Q4_0, los valores cuantizados siguen una distribución normal alrededor de 0, destacando la alta concentración de ocurrencias en los extremos, valores denominados *outlayers*. Por este motivo, el mecanismo de cuantización implementado debe priorizar la precisión en el centro, al mismo tiempo que refleja la presencia de valores *outlayers*.

La implementación del mecanismo de cuantización comprimido está en la función *pablo_q8_0_quantize_row*, que primero invoca la función de cuantización normal de Q8_0 para luego analizar el tensor de salida y traducir sus valores siguendo una tabla de traducción. De este modo, como el proceso de traducción se basa en utilizar los valores de la tabla, se puede ajustar el funcionamiento de la cuantización con la mayor granularidad realizando la menor cantidad de modificaciones en el código. Del mismo modo, existe una tabla homóloga para aplicar el proceso inverso en la decuantización.

Siguiendo el flujo de ejecución de la nueva cuantización, el peso de la precisión del modelo recae sobre los contenidos de la tabla de traducción. Como se ha indicado al principio, esta cuantización debe priorizar mantener la precisión de los valores centrales, aumentando el tamaño

del rango a medida que se aproxima a los extremos. Para determinar cómo crecen, en una primera aproximación, se busca un mecanismo que se acerque a ese comportamiento normal de la distruibución, aumentando el tamaño de cada rango en un factor de 2 partiendo de los valores centrales $\in [-2,1]$ que se codifica con un único bit cada uno, obteniendo así las siguiente tablas de traducción:

Tabla de traducción

8 bits	4 bits
[-128, -64)	-8
[-64, -32)	-7
[-32, -16)	-6
[-16, -8)	-5
[-8, -4)	-4
[-4, -2)	-3
-2	-2
-1	-1

8 bits	4 bits
0	0
1	1
[2, 4)	2
[4, 8)	3
[8, 16)	4
[16, 32)	5
[32, 64)	6
[64, 128)	7

Tabla de traducción inversa

4 bits	8 bits
-8	-128
-7	-64
-6	-32
-5	-16
-4	-8
-3	-4
-2	-2
-1	-1

4 bits	8 bits
0	0
1	1
2	2
3	4
4	8
5	16
6	32
7	64

Tabla 4.1.1: Tablas de traducción empleadas durante la cuantización basada en tablas básicas.

Cabe destacar que los 4 valores del centro [-2, -1, 0, 1] se cuantizan con precisión 1:1. En el resto de casos, los valores son decuantizados a su valor más próximo al valor central (0), debido a la distribución normal de los valores, salvo en los extremos, donde se ha decidido que en la decuantización se vuelva con el valor más extremo, ya que representarán en su mayoría los valores *outlayers*.

A continuación, se comprueba la efectividad del mecanismo, tanto estudiando la distribución de los valores cuantizados con el mismo histograma que las pruebas de Q4_0, como analizando los resultados de *perplexity*.

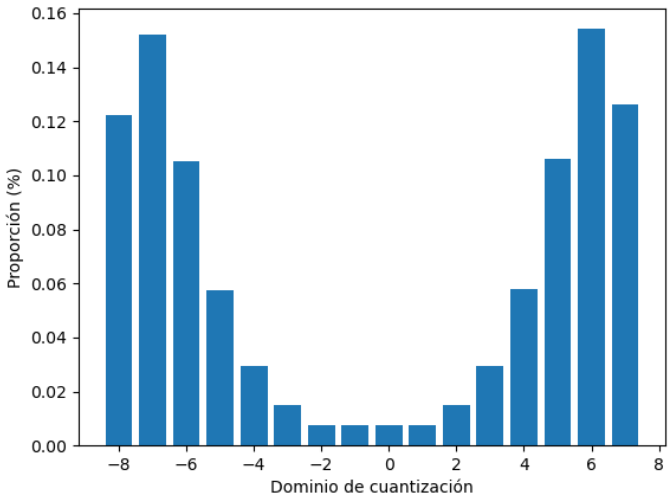

Figura 4.1.2: Histograma de la distribución de los valores cuantizados para todo el modelo de llama.cpp aplicando la cuantización Q8_0 basada en las tablas de traducción básicas.

Como se puede observar, el modelo mantiene una tendencia similar a su versión sin cuantizar, aunque los valores del centro para los que se ha mantenido la precisión parecen ser los menos significativos. Esto tiene sentido, ya que el resto de rangos agrupan cada vez más elementos a medida que se acercan a los extremos. También se aprecia el impacto de los valores *outlayers* en la versión cuantizada, aunque en menor medida que la original debido a la cantidad de valores en los penúltimos rangos.

A la vista de los resultados, cabe analizar el impacto que la reducción tiene en la precisión del modelo.

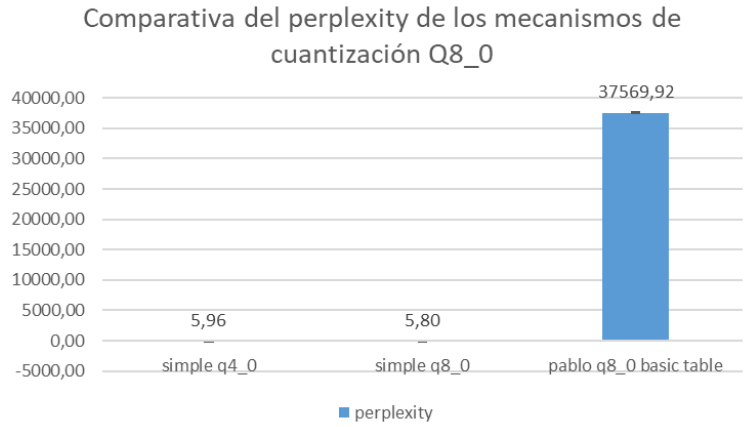

Figura 4.1.3: Comparación de *perplexity* de los mecanismos de cuantización Q4_0 simple, Q8_0 simple y Q8_0 basada en las tablas de traducción básicas.

A la vista de los resultados, se busca una solución que consiga una distribución de datos más homogéneos, donde cada valor de la cuantización tenga el mismo peso en la distribución. La idea principal es dividir el dominio de cuantización de Q8_0 en rangos de área similar, de modo que todos los números del dominio de cuantización de 4b contengan la misma cantidad de valores de 8b.

Para implementar el cambio se ha escrito una nueva tabla de traducción en la librería, así como su inversa para la decuantización. Al mismo tiempo, se ha añadido un nuevo parámetro al fichero de configuración *pablo.conf* llamado *TABLE_MODE*, que se utiliza para seleccionar la tabla.

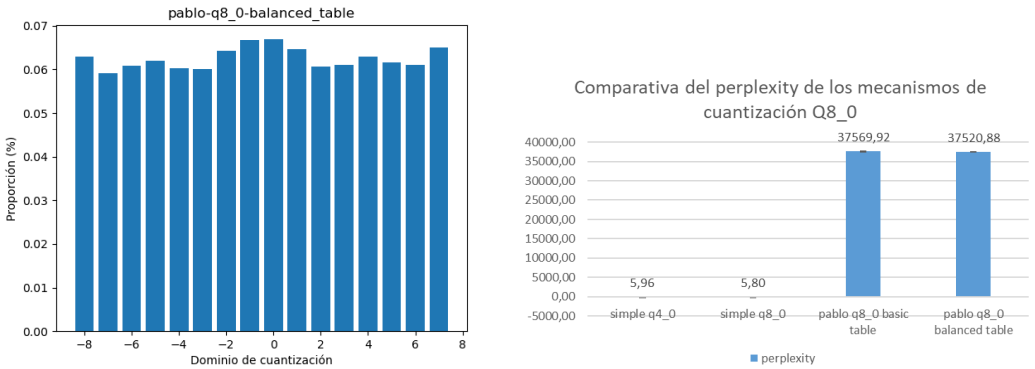

Figura 4.1.4: Histograma de la distribución de los valores cuantizados para todo el modelo de llama.cpp aplicando la cuantización Q8_0 basada en la tabla equilibrada (izquierda) y comparación de *perplexity* con Q4_0 simple, Q8_0 simple y Q8_0 basada en las tablas de traducción básicas y equilibradas (derecha).

Como se puede apreciar, los resultados de las pruebas reflejan un rendimiento muy similar al del modelo anterior. Modificar la cuantización de 4b mediante rangos de longitud variable, atendiendo a la distribución de valores de 8b no tiene el efecto esperado y los resultados son negativos.

4.2. Alternativas

A la vista de los resultados obtenidos, se ha llegado a las siguientes conclusiones. No ha sido posible mejorar el rendimiento general del sistema utilizando los mecanismos sencillos propuestos, pero esto no quiere decir que no sea posible mejorar el rendimiento utilizando la técnica de cuantización por rangos no equivalentes. El principal problema que las cuantizaciones alternativas no han sido capaz de abordar ha sido la redistribución de la precisión del modelo al final. Es posible que un mecanismo más complejo sea capaz de repartir mejor la precisión.

5. Conclusiones y trabajo futuro

El objetivo de este trabajo ha sido estudiar el funcionamiento de la cuantización y su implementación en llama.cpp implementando cambios para evaluar la viablidad del uso de técnicas de cuantización sencillas que aprovechen las ventajas de la refactorización y las técnicas de *sparsity* para mejorar el rendimiento de LLMs durante la inferencia.

Para ello, se han logrado implementar funciones de cuantización propias mediante un análisis, tanto de los fundamentos teóricos en los que se basa, como del flujo de ejecución a través de las diferentes partes de GGML y llama.cpp, llegando a identificar los elementos más importantes para la modificación de la cuantización. Al final se han llevado a cabo 2 acercamientos a este problema, los cuales han ayudado a mejorar el entendimiento, no solo de la herramienta *software* que se ha utilizado, sino de lo que supone trabajar a este nivel de detalle sobre un *framework* tan complejo.

Adicionalmente, se han desarrollado herramientas de prueba automatizadas para acelerar la toma de medidas y el procesamiento de los datos. Se ha llegado a vincular el funcionamiento de la librería propia a un fichero de configuración que facilita el control de la ejecución mediante *scripts* y los resultados de las pruebas se han exportado en formato *json* y analizados por otro conjunto de *scripts* escritos en *python* para construir las gráficas.

A nivel teórico, también se ha mejorado el entendimiento de en qué consiste la arquitectura *transformer* y el impacto que ha tenido en la industria, así como las particularidades de su implementación en Llama. Gracias a ello, se ha tenido la oportunidad de complementar la formación del grado profundizando en una tecnología muy reciente desde un punto de vista práctico.

Sobre los resultados de las pruebas, con relación a la técnica de la reducción de tamaño, el objetivo era comprimir el modelo a partir de una transformación basada en técnicas de cuantización y *sparsity*. Sin embargo, a la vista de los resultados, aunque hay una cantidad significativa de ceros, parece que la distribución de estos valores es aleatoria, lo que impide la implementación de un algoritmo de compresión eficiente. Teniendo esto en cuenta, la reducción de valores alrededor del centro de la distribución a 0 ha demostrado aumentar el grado de agrupación general, a costa de un empeoramiento importante de la precisión del modelo.

Con relación a la técnica de refactorización de valores, el objetivo era reducir el tamaño del modelo mediante una segunda cuantización de Q8_0 a Q4_0. Tras la realización de las pruebas, aunque este sistema consigue reducir el tamaño del modelo, lleva a cabo una simplificación demasiado severa de los valores originales de Q8_0, empeorando significativamente la precisión.

En cuanto a posibles líneas futuras, en todos los casos la cuantización se ha realizado utilizando tablas de traducción sobre las que recae la refactorización de los valores. Para este trabajo se ha comprobado la eficacia de técnicas simples, pero es posible que el uso de tablas más complejas consigan mejores resultados.

En el caso de la cuantización de Q8_0, la pérdida de rendimiento viene ocasionada por la sobresimplificación al refactorizar del rango de valores original de 8b al de destino de 4b.

6. Bibliografía

[1] T.Brown et. al. “Language Models are Few-Shot Learners”, Advances in Neural Information Processing Systems 33 (NeurIPS 2020).

[2] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. https://chat.openai.com/chat

[3] OpenAI. (2023). GPT-4 Technical Report (4 Mar 2024 version). https://arxiv.org/abs/2303.08774

[4] Anthropic Claude (2023) https://claude.ai

[5] Google Bard (2023) https://bard.google.com

[6] Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context (2024) https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf

[7] Hugo Touvron, et. al. “LLaMA: Open and Efficient Foundation Language Models” https://arxiv.org/abs/2302.13971v1

[8] Hugo Touvron, et. al. “Llama 2: Open Foundation and Fine-Tuned Chat Models” https://arxiv.org/abs/2307.09288

[9] Rohan Taori, et. al. “Stanford Alpaca: An Instruction-following LLaMA model”, https://github.com/tatsu-lab/stanford_alpaca

[10] Chiang, Wei-Lin, et. al. “Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90\%* ChatGPT Quality”, https://lmsys.org/blog/2023-03-30-vicuna/

[11] Amir Gholami, et. al. “A Survey of Quantization Methods for Efficient Neural Network Inference”, https://arxiv.org/abs/2103.13630v3

[12] Hao Wu, et. al. “Integer quantization for deep learning inference: Principles and empirical evaluation”, https://arxiv.org/abs/2004.09602

[13] Mathew S. Smith “The case for Running AI on CPUs Isn´t Dead Yet”, IEEE Spectrum news, June 2023.

[14] “perf: linux profiling with performance counters.” [Online]. Available: https://perf.wiki.kernel.org/

[15] “perf: linux profiling with performance counters.” [Online]. Available: https://perf.wiki.kernel.org/

[16] Ashish Vaswani, et. al. “Attention is all you need”, https://arxiv.org/abs/1706.03762

[17] Joshua Ainslie, et. al. “GQA: Training generalized Multi-Query transformer models from Multi-Head checkpoints”, https://arxiv.org/abs/2305.13245

[18] Biao Zhang, Rico Sennrich “Root mean square layer normalization”, https://arxiv.org/abs/1910.07467

[19] Jianlin Su, et. al. “RoFormer: enhanced transformer with rotary position embedding”, https://arxiv.org/abs/2104.09864

[20] Muhammad Usman Hadi, et. al. “Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects”, https://d197for5662m48.cloudfront.net/documents/publicationstatus/217266/preprint_pdf/4191f2ca2927adfc73d23922bdbcc359.pdf

[21] P. Khadka (2024, abr 10). LLaMA Explained! [Online] Available: https://pub.towardsai.net/llama-explained-a70e71e706e9

[22] Wikipedia (2024, nov 12). History of artificial intelligence. [Online] Available: https://en.wikipedia.org/wiki/History_of_artificial_intelligence

[23] llama.cpp fork (2024) https://github.com/PabloGSJ/llama.cpp_TFG2024

[24] S. J. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. 4ª edición. Hoboken: Pearson, 2021. ISBN 978-0-13-461099-3.

[25] llama.cpp repository: https://github.com/ggerganov/llama.cpp