Universidad de Murcia

Facultad de Informática

Trabajo Fin de Grado

Quantization as a democratization tool for Large Language Models

Autor:
David Fernández Expósito

Tutor:
Juan Antonio Botía Blaya

Cotutor:
Eduardo Martínez Gracia

May 28, 2025

Álvaro, siempre estarás presente en todo lo que hago. Espero que estés orgulloso de este trabajo.

Contents

Page

List of Figures **iv**

List of Tables **v**

Resume **1**

Extended Abstract **2**

1 Introduction **7**

2 Background **8**
2.1 AI Deep Learning Models . . . 8
2.1.1 Neural Networks . . . 8
2.1.2 The Transformer . . . 10
2.1.3 Large Language Models . . . 12
2.1.4 LLaMA: Meta's Open-Source Large Language Model . . . 14
2.1.5 Fine-Tuning of LLMs . . . 15
2.1.6 Environmental Impact of LLMs and Emission Measurement with CodeCarbon . . . 16
2.1.7 Measuring Large Language Model Performance and Perplexity . . . 17
2.2 Quantization . . . 18
2.2.1 Numerical Representation and the Need for Quantization . . . 18
2.2.2 Symmetric vs. Asymmetric Quantization and Range Mapping . . . 19
2.2.3 Quantization Methods . . . 22
2.2.4 Challenges and Considerations in Quantization . . . 23
2.3 State-of-the-art quantization methods . . . 23
2.3.1 BitsAndBytes . . . 23
2.3.2 GPTQ . . . 25
2.3.3 Quanto . . . 27
2.3.4 HQQ . . . 28

3 Objectives and Methodology **30**
3.1 Objectives . . . 30
3.2 Methodology . . . 30
3.2.1 Revise essential background concepts and select state-of-the-art quantization methods and measures for LLM performance . . . 31
3.2.2 Select a common use Large Language Model (LLM) as base model . 32
3.2.3 Creation of a working environment . . . 33

3.2.4 Prepare the base model for sentiment analysis . . . 34
3.2.5 Quantize the model and run experiments on sentiment analysis . . . 36
3.2.6 Quantize the model and run experiments on text generation . . . 36
3.2.7 Analyze the results and extract conclusions . . . 37

4 Design of the Solutions 38
4.1 Preparation of the base model for sentiment analysis . . . 38
4.2 Experiments on sentiment analysis . . . 38
4.3 Experiments on text generation . . . 39

5 Results 41
5.1 Evaluating sentiment analysis . . . 41
5.1.1 Accuracy and F1 score results . . . 41
5.1.2 Memory use results . . . 41
5.1.3 Inference and quantization speed results . . . 41
5.1.4 Inference and quantization emissions results . . . 41
5.2 Evaluating text generation . . . 44
5.2.1 Perplexity results . . . 44
5.2.2 Inference speed results . . . 44
5.2.3 Memory use results . . . 45
5.2.4 Emissions results . . . 45
5.2.5 All together . . . 49
5.2.6 A New Metric: Balancing Model Performance and Sustainability . . . 49

6 Discussion and limitations of the study 53

7 Conclusions 55

8 Credits 57

9 Bibliography 58

10 Annex 61
10.1 Results obtained from experiments . . . 61
10.1.1 Results for sentiment analysis . . . 61
10.1.2 Results for text generation . . . 63
10.2 Code . . . 64
10.2.1 Example of BitsAndBytes 4 bit quantization and sentiment analysis test . . . 64
10.2.2 Example of GPTQ 4 bit quantization and sentiment analysis test . . . 66
10.2.3 Example of HQQ 4 bit quantization and sentiment analysis test . . . 68
10.2.4 Example of Quanto 4 bit quantization and sentiment analysis test . . . 70

10.2.5 Example of BitsAndBytes quantization and text generation test . . 73
10.2.6 Example of GPTQ quantization and text generation test 75
10.2.7 Example of HQQ quantization and text generation test 76
10.2.8 Example of Quanto quantization and text generation test 78
10.3 Zero-shot experiment 81
10.4 LoRA training 83

Acronyms **85**

List of Figures

1 A simple 2-layer feedforward network, with one hidden layer, one output layer and one input layer. Source: [15] . . . 9
2 Architecture of a Transformer. Source: [15] . . . 10
3 Diagram of LoRA. Source: [10]. . . . 16
4 Example of symmetric quantization. Source: [9] . . . 20
5 Example of asymmetric quantization. Source: [9] . . . 21
6 Pipeline for sentiment analysis experiments . . . 36
7 Pipeline for text generation experiments . . . 37
8 Accuracy and F1 score results . . . 42
9 Memory use results for sentiment analysis experiments . . . 42
10 Inference speed results for sentiment analysis experiments . . . 43
11 Quantization speed results for sentiment analysis experiments . . . 43
12 Inference emissions results for sentiment analysis experiments . . . 43
13 Quantization emissions results for sentiment analysis experiments . . . 44
14 Perplexity results . . . 46
15 Inference speed results for text generation experiments . . . 46
16 Quantization speed results for text generation experiments . . . 47
17 Memory use results for text generation experiments . . . 47
18 Inference emissions results for text generation experiments . . . 48
19 Quantization emissions results for text generation experiments . . . 48
20 8 bit models radar chart . . . 49
21 4 bit models radar chart . . . 50
22 3 bit models radar chart . . . 50
23 Comparing models in perplexity and emissions . . . 51
24 New metric results . . . 52

List of Tables

1 Results obtained with LLaMA 3 following the same experiments as with LLaMA 2 in this work. 32
2 Zero-shot sentiment classification performance 35
4 Results obtained for quantization in sentiment analysis experiments. 61
3 Results obtained for sentiment analysis experiments. 62
5 Results obtained for text generation experiments. 63
6 Results obtained for quantization in text generation experiments. 63

Resume

Transformers have become one of the most important innovations in artificial intelligence in recent years. Their capacity to model natural language has made them the foundation of most modern Large Language Models (LLMs), enabling advances in a wide range of Natural Language Processing (NLP) tasks, from text generation to classification. Among these models, LLaMA 2 ([21]) has emerged as one of the most versatile and widely used architectures in both research and industry, although newer versions have been released in recent months.

At the same time, the growing size of LLMs has raised concerns about their computational and environmental costs. Quantization has emerged as a promising solution to reduce the memory requirements and energy consumption of these models, allowing their deployment on more modest hardware while maintaining competitive performance.

This work focuses on the study and evaluation of quantization techniques applied to LLaMA 2. A series of state-of-the-art quantization methods have been selected and tested in both sentiment analysis and text generation tasks. This project analyzes the effects of quantization on accuracy, perplexity, inference speed, memory use and energy emissions, and proposes a new metric to evaluate LLMs. Experimental results were obtained using a dedicated laboratory setup. A GitHub repository ([7]) has been prepared with the code, notebooks and results, ensuring the reproducibility of the experiments and providing a resource for future research. The results obtained reveal that 8-bit quantization can be a promising tool to reduce LLMs size while maintaining an acceptable level of performance. However, more aggressive quantization configurations, such as 4-bit and 3-bit, can lead to bigger losses in performance.

Extended Abstract

En los últimos diez años, el campo de la inteligencia artificial (IA) ha experimentado una transformación sustancial, situándose como uno de los pilares más influyentes en el mundo tecnológico de la actualidad. Estos avances incluyen desde desarrollos algorítmicos hasta innovaciones en hardware especializado cuyas aplicaciones han influido de manera notable en sectores como la sanidad, la industria automotriz, las finanzas, la educación y el entretenimiento, impactando de forma directa en la vida cotidiana de millones de personas. La integración de la IA en actividades del día a día es ya una realidad que todos podemos observar. Esto ha generado no solo entusiasmo, sino que también ha provocado que cada vez cobre más importancia el debate sobre sus implicaciones éticas, sociales y medioambientales, y los peligros que estas conllevan.

Una de las principales razones de este progreso ha sido la creciente disponibilidad de grandes volúmenes de datos debido a la presencia cada vez mayor de las tecnologías en las actividades cotidianas. La combinación de estos datos con técnicas de aprendizaje automático, y en particular con el aprendizaje profundo (*deep learning*), ha posibilitado la creación de modelos con una gran capacidad de generalización. Estos modelos son entrenados sobre grandes conjuntos de datos que capturan patrones complejos y han permitido resolver tareas que estaban fuera del alcance incluso de los sistemas más avanzados hasta hace pocos años.

El procesamiento del lenguaje natural (NLP por sus siglas en inglés) se ha situado como una de las ramas de la IA que más se han beneficiado por estos avances. Dentro de las ciencias de la computación, uno de los mayores desafíos ha sido siempre comprender y generar lenguaje humano, debido a la ambigüedad, la riqueza semántica y la variabilidad contextual que lo caracteriza. Durante décadas, los enfoques tradicionales basados en reglas o modelos estadísticos mostraban limitaciones severas a la hora de capturar estas complejidades. Sin embargo, con la aparición de arquitecturas neuronales avanzadas se empezó a entrever una solución que superaba estos retos de mejor forma. En este sentido, la introducción del modelo Transformer en 2017 representó un punto de inflexión histórico. A diferencia de las redes neuronales recurrentes (RNNs) o las redes LSTM, el Transformer se basa en mecanismos de atención que permiten procesar secuencias en paralelo y modelar relaciones a largo plazo de forma mucho más eficiente. Esta arquitectura ha servido como base para el desarrollo de una nueva generación de modelos capaces de generar texto con un grado de coherencia y naturalidad mucho mayor a las soluciones anteriores.

Sobre la base del Transformer surgieron los modelos de lenguaje a gran escala (LLM por sus siglas en inglés), que han revolucionado la forma en que las máquinas interactúan con el lenguaje humano. Estos modelos son entrenados sobre corpus inmensos que incluyen desde obras literarias y artículos científicos hasta publicaciones en redes sociales y foros de discusión. El resultado es un modelo que es capaz de abordar tareas muy diversas como

la traducción automática, la redacción de textos creativos, la clasificación semántica, la respuesta a preguntas y la generación de código fuente. En este contexto, ChatGPT se ha convertido en una de las herramientas más conocidas y utilizadas a nivel mundial, ya que ha generado un alto nivel de atención mediática y popular.

Entre los modelos más recientes y relevantes dentro de esta corriente destaca LLaMA 2, desarrollado por Meta. Este modelo ha conseguido posicionarse como una alternativa competitiva frente a modelos propietarios como GPT-3 o Falcon. Una de las claves de su éxito es su filosofía de apertura, ya que LLaMA 2 ha sido diseñado con una política de accesibilidad que facilita su uso en contextos académicos e industriales. Esta decisión ha permitido a investigadores y desarrolladores experimentar con modelos de última generación sin las barreras de acceso impuestas por soluciones comerciales cerradas. Así, LLaMA 2 se presenta como una herramienta idónea para universidades con recursos limitados o para pequeñas y medianas empresas interesadas en incorporar capacidades lingüísticas avanzadas a sus productos o servicios.

Sin embargo, los LLM, a pesar de todas sus ventajas y avances, conllevan también desafíos importantes, especialmente en términos de coste computacional y sostenibilidad energética. El entrenamiento de modelos como LLaMA 2 requiere infraestructuras de alto rendimiento con clústeres de GPUs que operan durante días o semanas, generando un consumo energético muy elevado. Incluso su despliegue en tareas de inferencia (la fase en la que el modelo ya entrenado se utiliza para producir resultados) implica un gran uso de recursos. Esto puede hacer que el uso de LLMs sea inasumible para ciertas aplicaciones en entornos con limitaciones de hardware. Estas exigencias han motivado una reflexión profunda en la comunidad investigadora sobre cómo optimizar el rendimiento de los modelos y democratizar el acceso a los LLMs, permitiendo su uso eficiente en hardware modesto sin comprometer excesivamente su rendimiento. Además, este esfuerzo se enmarca también en un contexto global marcado por la urgencia de reducir la huella de carbono de las tecnologías emergentes.

Entre las distintas estrategias posibles que existen para mejorar la eficiencia de los LLMs, la cuantización es una de las técnicas más interesantes. Esta técnica consiste en reducir la precisión numérica con la que se representan los parámetros del modelo, como las activaciones y los pesos, pasando de formatos habituales como *float32* a representaciones con menos bits como *int8*, *int4* o incluso *int3*. Este cambio conlleva, en un principio, una disminución en el tamaño del modelo, una reducción en el uso de memoria RAM y VRAM, y una disminución de los tiempos de respuesta. Por lo tanto, todo esto a su vez teóricamente se traduce en un menor consumo energético. Sin embargo, una cuantización excesiva puede deteriorar el rendimiento del modelo, lo que puede afectar a su precisión y utilidad práctica. Por tanto, uno de los grandes retos que conlleva la cuantización de modelos de lenguaje consiste en identificar el punto óptimo en el que se maximicen las ganancias en eficiencia sin comprometer la calidad de los resultados.

Este trabajo se propone precisamente analizar este equilibrio, evaluando el impacto de diversas técnicas de cuantización aplicadas al modelo LLaMA 2 en dos tipos de tareas particularmente representativas: el análisis de sentimientos y la generación automática de texto. Para ello, se han seleccionado cuatro métodos de cuantización de última generación: BitsAndBytes ([5]), GPTQ ([8]), Quanto ([12], [13]) y HQQ ([2], [17]). Cada uno de estos métodos presenta particularidades técnicas que lo hacen más o menos adecuado en función del escenario de uso. Por ejemplo, mientras algunos métodos aplican la cuantización de manera global tras el entrenamiento, otros adoptan esquemas más refinados que tienen en cuenta la sensibilidad de cada capa del modelo, buscando minimizar la pérdida de precisión allí donde más impacto puede tener.

Los experimentos llevados a cabo para esta investigación se desarrollaron en un entorno controlado con acceso a unidades de procesamiento gráfico (GPUs) de alto rendimiento. En particular se utilizó una tarjeta gráfica NVIDIA A100 con 40GB de VRAM. Para garantizar una evaluación precisa y reproducible de los modelos se han empleado herramientas especializadas, como Weights & Biases para la gestión y seguimiento de experimentos y CodeCarbon para la estimación del consumo energético y la huella de carbono asociada a cada configuración. Con estas herramientas se han calculado métricas de interés a la hora de comparar diferentes modelos. Estas métricas incluyen la precisión, puntuación F1, *perplexity*, uso de memoria y emisiones estimadas de CO_2.

Uno de los hallazgos más relevantes del estudio ha sido la confirmación de que la cuantización a 8 bits representa un buen compromiso entre eficiencia y rendimiento. En las tareas evaluadas, los modelos cuantizados en este formato mantuvieron un nivel de desempeño muy cercano al del modelo original en coma flotante, al tiempo que redujeron significativamente el uso de recursos computacionales, especialmente de memoria. Esta constatación resulta especialmente importante para escenarios en los que se requiera modelos en dispositivos con restricciones de memoria o potencia, como puede ocurrir en aplicaciones móviles, sistemas embebidos o plataformas de computación en el borde (*edge computing*), y acerca el objetivo de democratizar el uso de herramientas de IA como son los LLMs.

Sin embargo, al aplicar una cuantización más agresiva con representaciones de 4 y 3 bits se observó una disminución en la calidad de los resultados, tanto en términos de precisión como de coherencia textual en las tareas generativas. Esta degradación fue más acusada en el caso de los modelos de 3 bits, que mostraron síntomas claros de pérdida de información y errores semánticos que los hacen menos adecuados para aplicaciones que requieren un alto grado de fiabilidad.

Cabe señalar, además, que los diferentes métodos de cuantización no ofrecieron resultados homogéneos. BitsAndBytes destacó por su robustez y consistencia, proporcionando mejoras sustanciales en eficiencia sin comprometer de forma notable la precisión del modelo, sobre todo en su versión de 8-bit. Por el contrario, GPTQ, Quanto y HQQ presen-

taron un comportamiento más variable, con diferencias marcadas según la tarea y el tipo de cuantización aplicada. En ciertos casos, incluso se detectaron incrementos inesperados en el consumo energético o en la utilización de memoria, lo que pone de manifiesto que una implementación deficiente puede anular los beneficios teóricos de la cuantización. Estos resultados subrayan la importancia de evaluar cada método en función de las características específicas del problema y del entorno de ejecución.

Además, en este trabajo también se presenta una nueva métrica específicamente creada para combinar el rendimiento del modelo en términos de *perplexity* con las emisiones generadas por el mismo. Esta nueva métrica viene motivada del conflicto existente entre tener modelos con un buen rendimiento, que normalmente necesitan mayores recursos y generan mayores emisiones, y modelos más sostenibles, que normalmente tienen rendimientos más pobres. De este modo, con esta métrica no solo se consideran los aspectos puramente computacionales a la hora de comparar los diferentes modelos, sino también la sostenibilidad y la eficiencia, proporcionando una herramienta valiosa para la selección de modelos y métodos de cuantización que maximicen ambas dimensiones. Con esto se pretende avanzar hacia un escenario de democratización y reducción de emisiones asociadas a los modelos de lenguaje.

Además de la evaluación cuantitativa, este trabajo incita a una reflexión sobre las implicaciones del uso de la cuantización en grandes modelos de lenguaje. Aunque la cuantización puede ser vista como una solución técnica a un problema, su impacto trasciende lo meramente ingenieril. En un contexto de emergencia climática global, la posibilidad de reducir el consumo energético asociado a la inteligencia artificial adquiere una dimensión política y moral, ya que no se trata solo de hacer modelos más rápidos o baratos, sino de avanzar hacia un paradigma de desarrollo tecnológico que sea compatible con los límites planetarios y con una distribución equitativa de los recursos.

Sin embargo, esta eficiencia mejorada no debe ser entendida como una justificación para el uso indiscriminado de modelos cada vez más grandes. Existe un riesgo real de efecto rebote en el que las mejoras tecnológicas acaben alimentando una expansión descontrolada del uso de herramientas como los LLMs, con el consiguiente aumento agregado de emisiones y demanda energética. Por ello, resulta esencial combinar las soluciones técnicas con una planificación estratégica del uso de estas tecnologías, guiada por principios de sostenibilidad y justicia social. La cuantización, en este sentido, no es un fin en sí mismo, sino una herramienta dentro de un conjunto más amplio de estrategias orientadas a la responsabilidad tecnológica.

Un componente fundamental del trabajo ha sido el desarrollo de un *pipeline* reproducible que abarca todas las fases del proceso experimental. Este flujo de trabajo incluye scripts automatizados para la configuración de los modelos, la ejecución de los procesos de carga y cuantización de los mismos, la recogida de métricas y la generación de informes

visuales sobre ellas. El objetivo ha sido no solo facilitar la realización de experimentos adicionales, sino también promover la transparencia científica mediante la publicación del código fuente, los datos y los resultados en un repositorio de GitHub accesible ([7]). Por tanto, este trabajo sirve como documento aclarativo sobre todo el proyecto, en el cual en los anexos se podrá encontrar parte del material generado, mientras que el repositorio de GitHub sirve como punto de acceso general a su totalidad.

1 Introduction

In recent years, artificial intelligence has experienced a remarkable progress, bringing technological innovation across countless fields. One of the most important developments has been the emergence of large language models (LLMs), which have revolutionized tasks such as text generation, translation, summarization and sentiment analysis. These models, built on the Transformer architecture, have shown an unprecedented ability to process, comprehend and generate human language, opening up new possibilities for applications and tasks that were once considered impossible.

However, this progress has come with important challenges. As LLMs have grown in size and complexity, their computational and environmental costs have risen. Training and deploying these models require massive hardware resources, leading to concerns about accessibility, scalability and sustainability. Addressing these challenges has become a central focus in the AI research community. For example, the latest Meta available model, LLaMA 4 Maverick, has up to 400 billion parameters. To run a model like this would require servers equipped with multiple Graphics Processing Units (GPUs), such as 8×NVIDIA A100 with 80 GB VRAM each, or even higher-end H100 GPUs, with costs exceeding 100000$.

Quantization has emerged as a promising solution to this problem. By reducing the numerical precision of model parameters and activations, quantization significantly lowers memory requirements, accelerates inference and reduces energy consumption, all while preserving competitive performance. This helps to reduce the environmental impact LLMs have and makes it possible to run them on more modest hardware, helping to democratize the access to these powerful tools.

This work explores the application of state-of-the-art quantization techniques to LLaMA 2, one of the most important open-source LLMs available today. We analyze the impact of four leading quantization methods (BitsAndBytes, GPTQ, Quanto and HQQ) on tasks such as sentiment analysis and text generation. By evaluating these methods across metrics such as accuracy, *perplexity*, inference speed, memory usage and energy emissions, we aim to provide a clear picture of the trade-offs involved in quantization and to identify the most promising strategies for balancing performance and efficiency.

Beyond the empirical analysis, this study contributes a reproducible experimental pipeline that can serve as a foundation for future research. By sharing the code, data and results, we hope to advance the collective understanding of LLM quantization and support the development of more sustainable, accessible and responsible artificial intelligence.

2 Background

This section aims to provide the foundational concepts and context necessary to understand the research and experiments conducted.

2.1 AI Deep Learning Models

Artificial Intelligence (AI) is a field of study dedicated to developing computational systems that can reason or act similarly to human beings. Within AI, Machine Learning (ML) focuses specifically on enabling these systems to learn from experience and improve their performance over time as they process more data.

To train these models, specific datasets are used. Initially, a training dataset is provided to help the model learn patterns and adjust its internal parameters. Later, a test dataset is used to evaluate its performance on previously unseen data, ensuring that the learning process has been effective. In some cases, a third dataset, called a validation dataset, is introduced to compare different model configurations before finalizing the optimal structure and hyperparameter settings. These training experiments are configured using hyperparameters, which are predefined values that influence their behavior and overall performance. Finding the right combination of hyperparameters and training data is crucial for achieving accurate and generalizable results.

One of the most advanced branches of Machine Learning is Deep Learning. This approach relies on models composed of multiple transformation layers, where data evolve through both linear and nonlinear processes. Thanks to these techniques, highly sophisticated systems have been developed, capable of recognizing complex patterns in images, text and even real-time data sequences. This section explores the main Deep Learning concepts related to this work, drawing extensively from [1] and [15].

2.1.1 Neural Networks

Neural networks are widely used ML computing tools. They receive their name because their design was originally inspired by biological neurons, and are used for all kind of tasks, including classification, prediction and feature learning tasks.

A neural network consists of multiple layers of interconnected units called neurons, also known as perceptrons. These layers are created by vectorizing the input and operations of groups of neurons. With these layers as base units, the primary components of neural networks include an **input layer**, which receives raw data; **hidden layers**, where intermediate computations and transformations occur; and an **output layer**, which produces the final predictions or classifications. Each connection between neurons is associated with **weights and biases**, which determine the importance of each input feature. An example of a simple 2-layer is shown in Figure 1.

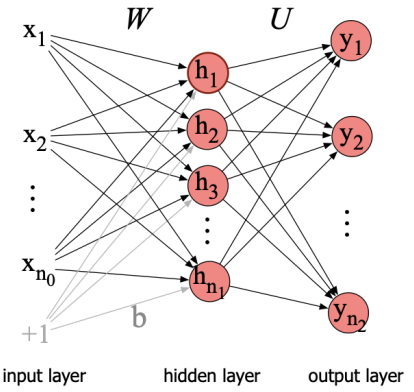

Figure 1: A simple 2-layer feedforward network, with one hidden layer, one output layer and one input layer. Source: [15]

An essential aspect of neural networks is the use of **activation functions**, which introduce non-linearity into the model, allowing it to capture complex relationships. Common activation functions include the sigmoid function, which maps values to the range (0,1); the hyperbolic tangent (tanh), which outputs values between -1 and 1; and the Rectified Linear Unit (ReLU), which allows only positive values to pass through while setting negative values to zero. The output of an activation function is called an **activation**, and represents the signal that is passed to the next layer.

Neural networks training work by processing input data through a series of transformations. This process follows four key steps:

- **Forward Propagation**: Each neuron computes a weighted sum of its inputs, adds a bias term, and applies an activation function to determine its output. This process propagates through the network until a final prediction is obtained.
- **Loss Calculation**: The predicted output is compared to the actual target value using a loss function. For classification tasks, cross-entropy loss is commonly used, while Mean Squared Error (MSE) is preferred for regression tasks.
- **Backpropagation**: The network calculates gradients of the loss function with respect to each weight using the chain rule of differentiation. These gradients indicate the direction in which each weight should be adjusted to minimize errors.
- **Optimization**: An optimization algorithm, such as Stochastic Gradient Descent (SGD) or Adam, updates the weights iteratively. The learning rate determines the magnitude of these updates to balance convergence speed and model accuracy.

Neural networks are highly flexible models capable of approximating complex functions by decomposing problems into smaller subproblems. The multi-layered structure enables

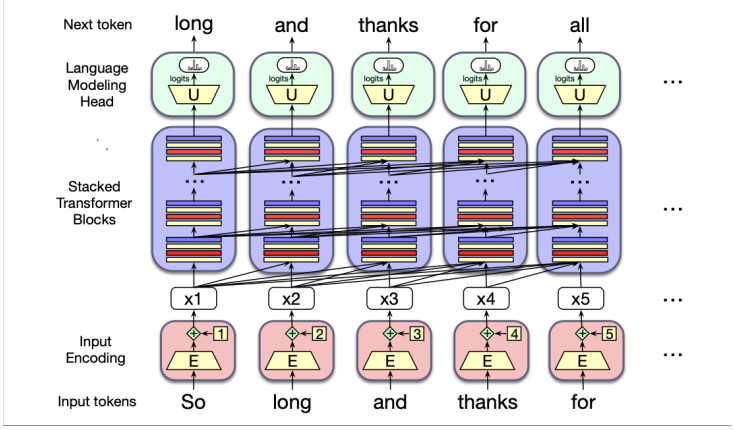

Figure 2: Architecture of a Transformer. Source: [15]

the extraction of hierarchical features from the data, making them particularly effective for applications involving large datasets.

Neural networks have applications in a variety of fields, including natural language processing, image recognition and time-series forecasting. They are extensively used in sentiment analysis to classify text into positive, neutral, or negative categories, in language modeling to predict word sequences and in computer vision to classify objects in images. Their ability to learn from raw data without the need for extensive feature engineering has contributed to their success in these domains.

2.1.2 The Transformer

Transformers are a class of Deep Learning models that have revolutionized the field of Natural Language Processing (NLP). Introduced by Vaswani et al. [22] in their 2017 paper *"Attention Is All You Need"*, the Transformer architecture has become the backbone of state-of-the-art NLP models, including Large Language Models (LLMs). Unlike previous sequential models, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, Transformers use a unique mechanism known as **self-attention** to process entire sequences of text in parallel. This innovation significantly improves the efficiency and scalability of NLP models, enabling them to handle vast amounts of text data while capturing complex relationships between words.

At its core, the Transformer model consists of an **encoder-decoder architecture**, where the encoder processes input sequences into meaningful representations, and the decoder generates an output based on that representation. However, many modern implementations of Transformers use only the encoder (e.g., BERT) or only the decoder (e.g., Generative Pretrained Transformer (GPT)), depending on the intended application.

One of the key innovations of Transformers is the **self-attention** mechanism, which allows the model to weigh the importance of different words in a sequence relative to one another. Unlike traditional RNNs, which process text sequentially and struggle with long-range dependencies, self-attention enables Transformers to better capture contextual relationships between words regardless of their position in the sequence.

The self-attention mechanism works by computing three key components for each token in a sentence:

- **Query (Q)**: Represents the token that is "asking" for information.
- **Key (K)**: Represents the token that is being "queried".
- **Value (V)**: Represents the actual information associated with each token.

The model calculates attention scores between all tokens in a sequence using the **scaled dot-product attention** formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where d_k is the dimensionality of the key vectors and the *softmax* function ensures that attention scores sum to 1. This mechanism enables the model to focus on the most relevant words when making predictions.

To further enhance its capabilities, Transformers employ **multi-head attention**, which runs multiple self-attention mechanisms in parallel. This allows the model to capture different aspects of the relationships between words, improving its ability to understand complex sentence structures.

Unlike RNNs and LSTMs, which process text in a sequential manner, Transformers lack an inherent notion of word order. To address this, the model incorporates **positional encodings**, which are added to the input embeddings to provide information about the position of each token in the sequence. These encodings are computed using *sine* and *cosine* functions:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d}) \tag{2}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d}) \tag{3}$$

where pos is the position of the token and i represents the embedding dimension. By adding these values to the input embeddings, the model retains information about word order while still benefiting from parallel processing.

The Transformer consists of two main components: the **encoder** and the **decoder**. Each component is composed of multiple identical layers, with each layer containing two key sublayers:

- **Multi-Head Self-Attention Layer**: Allows the model to attend to different words in a sequence, capturing dependencies across the input.
- **Feed-Forward Neural Network**: A fully connected network that processes the output of the self-attention layer to extract higher-level features.

Each sublayer is followed by **layer normalization** and **residual connections**, which help stabilize training and improve gradient flow. A diagram of the Transformer architecture is shown in Figure 2.

Transformers have largely replaced earlier sequence-processing models like RNNs and LSTMs due to several advantages:

- **Parallelization**: Unlike RNNs, which process text sequentially, Transformers can process entire sequences at once, significantly improving training efficiency.
- **Better Long-Range Dependencies**: Self-attention allows Transformers to capture relationships between words regardless of distance, while RNNs struggle with long-range dependencies.
- **Scalability**: The Transformer architecture can scale to much larger datasets and parameter sizes, making it ideal for training large models such as BERT and GPT.

These advantages have made Transformers the foundation for modern NLP, enabling the development of powerful LLMs that can perform complex language understanding and generation tasks.

2.1.3 Large Language Models

Large Language Models (LLMs) are a class of deep learning models that are particularly specialized in handling tasks related to Natural Language Processing (NLP). These models are designed to process and understand human language at scale, enabling them to perform a wide variety of tasks during inference, such as text generation, translation, summarization or question answering. LLMs are called "large" due to the sheer volume of data they are trained on, typically involving billions of words from diverse datasets, and the enormous number of parameters they possess, often numbering in the billions or even trillions. This way, these models learn to capture the complexities of language.

At the core of LLMs lies a deep neural network architecture, often based on the Transformer model seen before (see section 2.1.2), which enables them to process sequential

data, such as text, efficiently. The primary goal of LLMs is to learn to represent the meaning of words and sentences in a way that can be effectively used for a wide range of NLP tasks. The underlying mechanism involves mapping textual data into high-dimensional vector spaces, allowing the model to reason about language through mathematical operations. These vector representations are known as **embeddings** and play a critical role in how LLMs understand and manipulate language.

Before a large language model can begin its processing, the input text must be transformed into a form that the model can understand. This transformation is known as **tokenization**. It involves breaking down a sequence of text into smaller units, called **tokens**, which can be individual words, sub-words, or even characters. The goal of tokenization is to convert human-readable text into numerical representations that the model can process. Each token corresponds to a unique identifier in the model's vocabulary, which is a list of all possible tokens that the model is capable of recognizing.

For instance, a sentence like "LLMs are powerful" might be tokenized into individual words: "LLMs", "are", and "powerful". However, in certain cases, tokenization might involve breaking down words into smaller sub-word units to handle complex or rare words more effectively. For example, "unhappiness" could be tokenized as "un", "happi", and "ness", based on the model's vocabulary and the frequency of these sub-word units in its training data. This approach, known as sub-word tokenization, allows LLMs to efficiently handle out-of-vocabulary words or morphologically complex language structures.

Once the text has been tokenized, each token is mapped to a vector representation in a high-dimensional space (embeddings). These are continuous representations of the tokens, where similar words or tokens are placed closer together in the vector space. The embeddings allow the model to capture semantic relationships between words based on their context in the text. For example, the words "king" and "queen" might be placed close to each other in the vector space due to their shared semantic properties, while "king" and "dog" might be farther apart.

The process of tokenization and embedding is a crucial first step in the operation of an LLM, as it enables the model to convert text into a numerical format that can be processed by its neural network layers.

There are two primary types of LLM architectures that have emerged: encoders and decoders. Each serves a different purpose depending on the specific NLP task at hand.

- **Encoder-based Models:** Encoder-based LLMs are designed to generate a vector representation of the input text, where the goal is to map sentences with similar meanings into vectors that are close to each other in the vector space. These models typically operate in an autoencoding manner, where the model is trained to map the input sequence to a fixed-length vector (the encoding) and then attempt to

reconstruct the original input from this representation. Encoder-based models are often used in tasks such as text classification or sentiment analysis, where the model's output is a summary or representation of the input text rather than a generated sequence of words.

- **Decoder-based Models:** Decoder-based LLMs, also known as autoregressive models, are designed to generate text. These models are trained to predict the next token in a sequence based on the tokens that preceded it. Given an initial input, the model predicts the next word, appends it to the sequence, and then predicts the following word based on the updated sequence. This process continues until the model generates a complete output, such as a sentence, paragraph, or even an entire article. The most famous example of a decoder-based model is GPT (Generative Pretrained Transformer), which uses a unidirectional (left-to-right) approach to generate text. Unlike encoder models that focus on generating representations of input text, decoder models are typically used for text generation tasks such as machine translation, creative writing, dialogue systems, and more.

Training an LLM is an intensive process that requires vast amounts of data and computational resources, often involving the use of high-performance GPUs or specialized hardware like Tensor Processing Units (TPUs) during weeks or even months. The models are trained using supervised or unsupervised learning techniques, depending on the task they are designed for. In supervised learning, models are trained on a dataset consisting of input-output pairs, where the goal is to learn a mapping from the input to the correct output. For example, in a sentiment analysis task, the input could be a sentence, and the output would be a label indicating whether the sentiment is positive or negative. Unsupervised learning, on the other hand, involves training a model on large corpora of text without explicit labels. This type of training is often used for pretraining LLMs, where the goal is for the model to learn the structure and patterns of language without any task-specific labels. The model can then be fine-tuned on a smaller, labeled dataset to adapt it for specific tasks, such as question answering or text summarization.

2.1.4 LLaMA: Meta's Open-Source Large Language Model

The Large Language Model Meta AI (LLaMA) series ([20], [21]), developed by Meta, represents a major milestone in the development of open-source large language models (LLMs). Released in 2023, the original LLaMA models were designed to open the access to state-of-the-art natural language processing (NLP) systems by providing models that combine competitive performance with high efficiency, while being available under a research-friendly open license. The LLaMA family was introduced with models of varying parameter sizes, like 7 billion (7B), 13B, 30B, and 65B parameters; trained on publicly available datasets. This design decision marked a significant change from many previous LLMs, which frequently relied on proprietary data or restricted access.

The LLaMA models are built upon the Transformer decoder-only architecture, originally popularized by GPT models, but they include several critical architectural optimizations. Among these improvements are the use of rotary positional embeddings (RoPE) instead of absolute positional embeddings (which allows the model to better generalize to longer sequences by encoding relative positional information in a more flexible and scalable way), the implementation of pre-normalization (applying LayerNorm before the attention and feedforward sublayers), and efficient training strategies leveraging mixed-precision arithmetic (like float16). These optimizations enable LLaMA to achieve high performance with fewer parameters and reduced computational cost compared to models like GPT-3, making them particularly attractive for academic and industrial research groups with limited hardware resources.

A central result highlighted in the LLaMA paper was that smaller LLaMA models often outperformed much larger models in multiple NLP benchmarks when trained sufficiently and on high-quality data. For example, LLaMA-13B was shown to match or exceed the performance of GPT-3 (175B) across several tasks, despite using an order of magnitude fewer parameters. This finding reinforced the idea that model quality and data curation can often outweigh sheer model size, especially when combined with architectural refinements.

The release of LLaMA 2 in 2023 marked a significant evolution of the series. LLaMA 2 models were trained on a corpus approximately 40% larger than that used for the original LLaMA model, including a curated mix of web data, code and academic publications. Meta also introduced fine-tuned variants, notably the LLaMA 2-Chat models, which were specifically optimized for dialogue and instruction-following tasks using reinforcement learning from human feedback (RLHF). The LLaMA 2 models maintained the same general architecture but included refinements in training strategies, optimization algorithms and scaling laws, leading to notable gains in robustness, generalization and downstream task performance.

2.1.5 Fine-Tuning of LLMs

Fine-tuning refers to the process of adapting a pre-trained language model, such as LLaMA 2, to perform a specific downstream task (e.g. sentiment classification) by training it on a smaller and task specific dataset. This approach allows to leverage the general linguistic and world knowledge already learned by the model, avoiding the need to train from scratch. However, full fine-tuning still requires updating all of the model's parameters, which is computationally expensive and demands significant memory and time, especially with large models like LLaMA 2.

To address these limitations, several parameter efficient fine-tuning methods have been proposed. One of the most widely adopted is Low-Rank Adaptation (LoRA) ([10]),

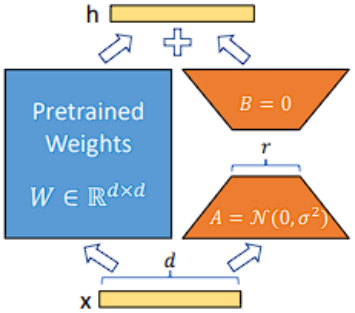

Figure 3: Diagram of LoRA. Source: [10].

which significantly reduces the computational and memory demands of fine-tuning. LoRA achieves this by inserting small trainable matrices into specific layers of the model and keeping the original weights frozen. As a result, only a small number of additional parameters are updated during training. This makes LoRA particularly well-suited for adapting large models in resource-constrained environments. In our case, it will allow us to fine-tune LLaMA 2 for sentiment analysis without the need to retrain all 7 billion parameters. In fact, only 4206592 parameters from the total of 6611562496 (0.0636%) were trained. This shows the reduction that can be achieved by using this powerful method. Figure 3 shows a diagram of LoRA.

2.1.6 Environmental Impact of LLMs and Emission Measurement with CodeCarbon

The revolution of natural language processing (NLP) seen in recent years thanks to LLMs (see section 2.1.3) have come at a considerable environmental cost. Training large models from scratch typically requires massive computational resources, including thousands of GPUs running for weeks or months, resulting in significant energy consumption and associated carbon dioxide (CO_2) emissions. Even the fine-tuning and deployment phases, although less computationally intensive than training, can carry a substantial environmental impact when performed at scale.

Several studies have raised concerns about the carbon footprint of AI systems. For example, Strubell et al. [19] estimate that training a single large transformer model can emit as much CO_2 as five American cars in their entire lifetimes. This highlights the conflict between advancing AI capabilities and ensuring environmental sustainability. As AI continues to expand into more domains, there is a growing recognition that improving the efficiency of these models is not only an engineering challenge but also an environmental imperative.

Given this context, accurately measuring and reporting the energy use and emissions

of machine learning experiments has become an increasingly important aspect of responsible AI research. One notable tool that addresses this need is **CodeCarbon** ([4]), an open-source software package designed to estimate the CO_2 emissions associated with model training and inference. CodeCarbon monitors hardware usage, such as GPU and CPU utilization, and calculates energy consumption based on power draw and runtime. Importantly, it also takes into account the carbon intensity of the local electricity grid, using geolocation data to provide region-specific estimates. This allows researchers and engineers to obtain granular and interpretable metrics about the environmental impact of their computational workloads.

CodeCarbon is designed to integrate easily into Python based machine learning pipelines and can provide real-time feedback as well as summary reports. By logging emissions at each stage of an experiment, it enables researchers to compare the environmental cost of different approaches, optimizers, architectures or quantization schemes. This kind of insight is critical for making informed decisions when developing models, particularly when balancing accuracy, speed and sustainability.

2.1.7 Measuring Large Language Model Performance and Perplexity

The evaluation of LLMs is a fundamental aspect of NLP research, as it allows researchers to compare the capabilities of models across various tasks. As LLMs continue to scale in terms of parameters and training data, robust evaluation metrics have become really important for assessing their quality.

LLM performance can be measured using a variety of metrics depending on the nature of the task. Broadly, evaluation can be divided into two categories: **intrinsic evaluation** and **extrinsic evaluation**. Intrinsic evaluation focuses on assessing the model's core language modeling abilities, often through metrics such as next-word prediction accuracy, masked language modeling loss, and most prominently, **perplexity**. Extrinsic evaluation, in contrast, assesses how well the model performs on downstream tasks, including text classification, summarization, machine translation, and question answering, often using task-specific metrics such as accuracy, F1 score or human preference ratings.

Perplexity is one of the most widely used metrics in language modeling. It measures the uncertainty of a probabilistic model in predicting the next word in a sequence. Formally, for a sequence of tokens $w_1, w_2, \ldots, w_N$, and a language model that assigns probabilities $p(w_i|w_1, \ldots, w_{i-1})$, the perplexity is defined as:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log p(w_i|w_1, \ldots, w_{i-1})\right). \tag{4}$$

Intuitively, perplexity can be interpreted as the model's average branching factor or the effective number of choices it considers at each prediction step. A lower perplexity

indicates that the model assigns higher probability to the observed tokens, suggesting better predictive performance. Early foundational work, such as that by Jelinek et al. [14], established perplexity as a central metric in the evaluation of language models and speech recognition systems.

Despite its widespread use, perplexity has several limitations. It is sensitive to the domain and distribution of the evaluation data, meaning that perplexity values are only meaningful when models are evaluated on the same dataset or under comparable conditions. Moreover, perplexity does not always correlate with downstream task performance. For example, a language model may achieve low perplexity and underperform in tasks such as summarization or translation, where other characteristics of the model also matter. Therefore, perplexity is often complemented by extrinsic evaluations tailored to specific applications.

2.2 Quantization

Quantization is a model compression technique aimed at reducing the computational complexity and memory footprint of neural networks. By lowering the numerical precision of parameters and activations, quantization enables efficient deployment on resource-constrained hardware while maintaining acceptable levels of model accuracy. Therefore, it is an interesting tool for the democratization of LLMs. Beyond hardware efficiency, quantization also plays an important role in reducing the environmental footprint of AI models, helping to address the growing concern over the energy demands and carbon emissions associated with large-scale deep learning models. This section explores the principles, methodologies and challenges associated with quantization in deep learning, drawing extensively from [9].

2.2.1 Numerical Representation and the Need for Quantization

Neural networks traditionally operate using high-precision floating-point representations, most commonly 32-bit floating-point (32-bit Floating Point (FP32)). While this representation provides high accuracy and dynamic range, it incurs substantial memory and computational overhead. Alternative representations such as 16-bit Floating Point (FP16) (16-bit floating point) and 8-bit integer (INT8) (8-bit integer) offer a trade-off between precision and efficiency. The key motivations for quantization include:

- **Memory Efficiency:** Reducing parameter size allows larger models to fit within limited memory, facilitating deployment on edge devices and helping democratize the use of LLMs.
- **Computation Speedup:** Integer operations can be significantly faster than floating-point operations on specialized hardware, leading to reduced inference latency. How-

ever, using reduced precision data types also require additional processing that can affect this.

- **Energy Efficiency:** If done right, lower precision computations consume less power, which is critical for mobile and embedded applications. In addition to this, this also reduces the overall energy consumption and carbon emissions of large-scale AI systems, helping to improve the sustainability of AI.

For example, a model with 7 billion parameters in FP32 requires approximately 28 GB of memory. Converting it to INT8 reduces this to around 7 GB, making deployment feasible on devices with constrained resources.

2.2.2 Symmetric vs. Asymmetric Quantization and Range Mapping

When quantizing neural network models, one of the key challenges is how to represent continuous (floating-point) values using a reduced number of bits (such as INT8). Two fundamental strategies to achieve this are **symmetric** and **asymmetric quantization**, both of which rely on techniques like **range mapping** and **clipping** to ensure efficient and accurate representation.

2.2.2.1 Symmetric Quantization In symmetric quantization, the range of real numbers is centered around zero. This means that the minimum and maximum quantization range are equal in magnitude but opposite in sign (i.e. $[-S, S]$) and the quantized value for zero in the original real number space is still zero when quantized. The quantization process maps the floating-point values into integers using a **scale factor** without a zero-point shift. This approach is especially effective when the distribution of the data is roughly symmetric around zero, such as weights in many neural network layers.

A common example for this type of quantization is *absolute maximum quantization*, where the highest absolute value is used as range to perform the mapping. For example, if we have weights in the range $[-3.0, 3.0]$ and want to quantize them to INT8, the symmetric range would be mapped linearly into $[-127, 127]$. An example of symmetric quantization can be found on Figure 4.

To calculate the scale factor, the following formula can be used, where b is the number of bits we want to quantize to:

$$\mathrm{s} = \frac{2^{b-1} - 1}{\max(|\mathrm{min}|, |\mathrm{max}|)}.$$

With this scale factor, quantization is easily performed with the following formula:

$$x_{quantized} = \mathrm{round}(s \cdot x).$$

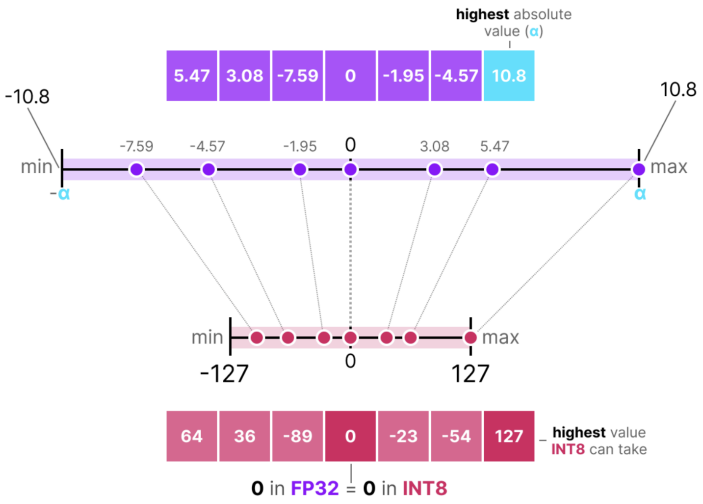

Figure 4: Example of symmetric quantization. Source: [9]

To perform a dequantization, the opposite calculation is needed:

$$x_{dequantized} = \frac{x_{quantized}}{s}.$$

The simplicity of symmetric quantization allows for faster computations, as it avoids extra additions (zero-point corrections), but it may not be ideal when the data distribution is skewed or offset from zero.

2.2.2.2 Asymmetric Quantization Asymmetric quantization allows the quantization range to shift away from zero. It introduces a **zero-point offset** to align the floating-point zero with an integer value within the quantized range. This is useful when the data's minimum and maximum are not centered around zero (for example, if the activation outputs of a ReLU layer range from 0 to some positive value). An example of asymmetric quantization can be found on Figure 5.

In this case, the formula for quantization becomes:

$$x_{quantized} = \text{round}(s \cdot x + z),$$

where s is the scale factor

$$\text{s} = \frac{\text{qmax} - \text{qmin}}{\text{max} - \text{min}}.$$

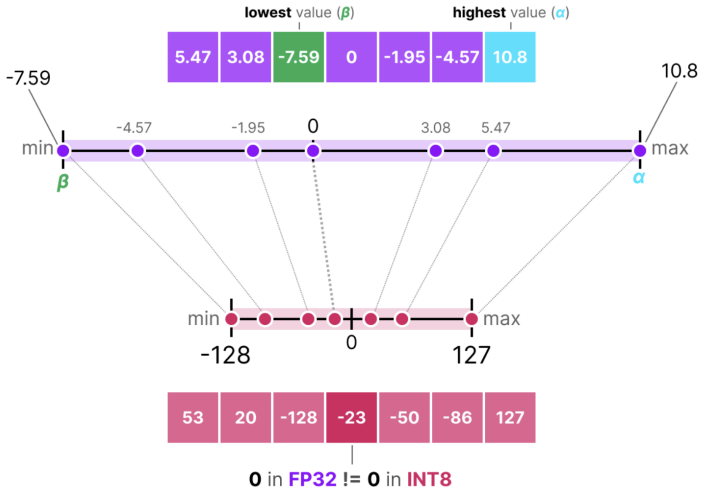

Figure 5: Example of asymmetric quantization. Source: [9]

In this formula, *max* and *min* are the maximum and minimum values in the real space, while *qmax* and *qmin* are the maximum and minimum values possible when quantized. The *zero-point* can be computed as:

$$z = \text{round}\left(-\text{min} \cdot s\right).$$

To perform a dequantization, the following calculation is needed:

$$x_{dequantized} = \frac{x_{quantized} - z}{s}.$$

This approach helps preserve accuracy in layers with non-centered distributions but comes at the cost of slightly more complex computations.

2.2.2.3 Range Mapping and Clipping Before applying symmetric or asymmetric quantization, it is critical to map the floating-point range to the target quantization range. This is called **range mapping** and typically involves two steps:

1. **Determining the min and max range**: Instead of naively using the absolute min and max of the data (which may contain outliers), many quantization strategies compute the optimal range that minimizes quantization error.
2. **Clipping**: Once the range is set, any value falling outside it is *clipped* to the nearest bound. For example, if the range is set to $[-3, 3]$ and a value is 4.2, it will be clipped to 3. This ensures that extreme outliers do not disproportionately stretch

the quantization range, which would otherwise degrade the resolution of the more frequent (central) values.

This careful balance of range selection and clipping is critical, as it directly influences the precision and performance of the quantized model.

2.2.3 Quantization Methods

Quantization can be applied at different stages of a neural network's lifecycle, leading to various methodologies:

2.2.3.1 Post-Training Quantization (PTQ) Post-Training Quantization (PTQ) converts a trained model's weights and activations from high-precision floating point to lower-bit representations, after the model has been trained. There are different approaches within PTQ for the quantization of the activations, since they require inference of the model to obtain their potential distribution.

- **Dynamic Quantization:** Each time data passes through a layer, its activations are collected and used to obtain the distribution. With this, the *zero-point* and *scaling factor* are calculated. This process is repeated each time data passes through a new layer, resulting in different quantization schemes for each layer.
- **Static Quantization:** Both *zero-point* and *scaling factor* are calculated beforehand, not during inference. For this, a calibration dataset and techniques are used, such as histogram-based scaling.

While PTQ is straightforward and requires minimal changes to the training pipeline, it may lead to a loss in accuracy if not carefully tuned, because it does not consider the actual training process.

2.2.3.2 Quantization Aware Training (QAT) Quantization Aware Training (QAT) incorporates quantization effects directly into the training process. Instead of training a model in full precision and quantizing it afterwards, QAT simulates low-precision arithmetic during forward and backward propagation. This allows the model to learn quantization-induced errors and adapt to them, often resulting in superior performance compared to PTQ.

QAT typically employs techniques such as:

- **Fake Quantization:** Floating-point computations simulate integer precision using scaling factors to approximate the behavior of lower-precision arithmetic.
- **Learnable Scaling Factors:** The model learns appropriate scaling factors for each layer, optimizing the quantization process dynamically.

The biggest downside with QAT is that, while it improves model robustness under quantization constraints, it demands additional training time and computational resources. For this reason, all quantization methods explored in this work are PTQ.

2.2.3.3 Mixed-Precision Quantization Rather than using a single precision level across all layers, mixed-precision quantization assigns different bit widths to different network components based on their sensitivity to quantization. For instance, early convolutional layers may be quantized more aggressively (e.g., 4-bit integer (INT4)), while fully connected layers retain higher precision (e.g., FP16). Mixed precision is particularly beneficial for transformer-based architectures where the most important attention layers may require higher precision to maintain expressivity.

2.2.4 Challenges and Considerations in Quantization

Despite enabling efficient deep learning inference, particularly in edge computing and real-time applications, some considerations need to be made when deciding to use quantization.

Firstly, reducing numerical precision inevitably introduces approximation errors. Certain layers or operations may be more susceptible to quantization-induced loss, necessitating careful tuning of quantization parameters.

In addition to this, determining optimal scaling factors for quantized values is critical. Poorly chosen quantization parameters can lead to saturation or excessive rounding errors, significantly degrading model performance. For example, for weights, which are static and known, calibration techniques for choosing the range include optimizing the mean squared error (MSE) between the original and quantized weights, and minimizing entropy (KL-divergence) between the original and quantized values.

Finally, hardware support needs to be considered in order to achieve efficient execution of quantized models. Modern accelerators such as NVIDIA Tensor Cores, Google's Edge TPU, and custom Application-Specific Integrated Circuits (ASICs) like Apple's Neural Engine offer optimized support for lower-precision computations. However, compatibility issues may arise when deploying quantized models across different hardware platforms.

2.3 State-of-the-art quantization methods

In this section, four state-of-the-art quantization methods are introduced: BitsAndBytes, GPTQ, Quanto and HQQ.

2.3.1 BitsAndBytes

BitsAndBytes ([5]) is an optimized library designed for quantizing large-scale deep learning models, developed by Tim Dettmers, a researcher known for his work on efficiency and

optimization in deep learning. The primary objective of BitsAndBytes is to enhance the computational efficiency of deep learning models by using lower precision arithmetic, thus reducing memory usage and accelerating both the training and inference processes.

BitsAndBytes operates within the realm of post-training quantization (PTQ) (see section 2.2.3.1), a technique where a pre-trained model is quantized after training to reduce its size and computational cost. Specifically, BitsAndBytes offers support for 8-bit, 4-bit, and mixed-precision quantization. The core idea behind this approach is to leverage lower precision operations while maintaining model accuracy, which is crucial for large neural networks with massive parameter sizes.

One of the standout features of the BitsAndBytes quantization method is its handling of outliers. In deep learning models, certain weights in the network may take on extreme values (either very large or very small) compared to the rest of the weights. These values are known as outliers, and they can have a significant impact on the model's performance if not handled properly. BitsAndBytes identifies and retains these outliers in higher precision (16-bit floating-point format), while the majority of the weights, which are closer to zero, are quantized into 8-bit or 4-bit integer formats. This approach ensures that the outliers are represented with sufficient precision, thereby preventing them from degrading the model's accuracy, while the common weights are reduced to lower precision to save memory and computational cost.

For the 4-bit quantization variant, BitsAndBytes introduces an innovative data type known as the 4-bit NormalFloat (NF4). This format is specifically designed for the task of representing normally distributed weights in deep learning models. By using NF4, BitsAndBytes achieves a significant reduction in memory usage while maintaining a high level of model performance.

The effective identification and management of outliers are central to the performance of BitsAndBytes. By default, the outlier threshold in the 8-bit quantization configuration is set to 6 (in absolute value), which means any weight with a magnitude greater than or equal to 6 is considered an outlier and preserved in 16-bit precision. This threshold can be adjusted by modifying the `llm_int8_threshold` parameter when configuring the quantization process, allowing users to tailor the quantization process to their specific model and use case.

Empirical studies suggest that outliers typically account for less than 1% of the total number of weights in a model, although this proportion can vary depending on the model architecture and the specific task. While outliers are relatively rare, they are crucial to the model's performance, particularly in deep layers where small changes in weight values can drastically impact the output. Therefore, it is essential to perform a layer-by-layer analysis to determine the exact number and distribution of outliers in a given model. This analysis will also help in fine-tuning the outlier threshold for the best performance.

The process of inference in a model that has undergone quantization with BitsAndBytes is optimized for efficiency. Inference is performed using a combination of low-precision integers (8-bit or 4-bit) for the majority of weights, and higher precision floating-point operations (16-bit) for the outliers. To facilitate this, BitsAndBytes employs specialized CUDA kernels (i.e., small GPU programs designed to execute parallel operations efficiently) designed to handle these mixed-precision matrices. These kernels ensure that the normal weights and the outliers are processed separately, and then appropriately combined at the end of the inference process to produce the final result.

2.3.2 GPTQ

Generalized Post-Training Quantization (GPTQ) ([8]) is an advanced quantization method designed specifically for fine-tuning and optimizing LLMs. It was developed to address the need for efficient deployment of these models without significant loss in accuracy. By applying quantization techniques at the post-training stage (therefore a PTQ method), GPTQ reduces the precision of model weights while preserving critical characteristics of the original model, making it ideal for scenarios where memory and computational resources are limited.

The core idea behind GPTQ is gradient-guided quantization, which uses gradient-based metrics to prioritize the importance of different weights in the model. Unlike naive quantization methods that treat all weights equally, GPTQ identifies weights that have the greatest impact on the model's performance and assigns them higher precision values. Typically, these critical weights are left at higher precision, such as 16-bit floats, while less influential weights are quantized to lower precision formats like 8-bit or 4-bit integers. This selective approach minimizes the quantization error and helps maintain model accuracy. These weights are identified using a second-order approximation based on the Hessian matrix. This approach allows the measurement of the model's sensitivity to changes in its weights, minimizing accuracy loss.

First, the squared error introduced by quantizing each weight individually is computed using the metric:

$$\|WX - \hat{W}X\|^2,$$

where W represents the original weights, $\hat{W}$ are the quantized weights, and X corresponds to the layer inputs. Subsequently, an approximation of the Hessian matrix is calculated to estimate each weight's contribution to the overall error. Using this information, weights are ranked by their impact, and those with the highest influence on accuracy are processed first.

Once identified, weights are processed through an iterative and adaptive approach that adjusts the remaining weights to compensate for the error introduced by quantization. This treatment involves the following steps:

1. **Adaptive Quantization:** Each weight is quantized to the desired precision level. Then, the remaining weights are dynamically updated to minimize the cumulative error caused by the quantization of the processed weight.
2. **Global Approach:** Instead of separating weights into categories of high and low impact, GPTQ adopts a holistic strategy, optimizing all weight interactions within the same layer.
3. **Block-wise Processing:** In large-scale models, weights are divided into blocks (e.g., column blocks). Each block is quantized independently to enhance computational efficiency, significantly reducing the cost of processing very large matrices.

A key feature of GPTQ is its iterative approach to quantization. Instead of quantizing all weights at once, GPTQ applies quantization layer by layer, recalculating gradients at each step. This allows the method to adjust for quantization-induced errors dynamically and ensures that the final model remains robust. The process results in a model with a significantly smaller memory footprint while retaining most of its predictive performance.

Layer-by-layer compression in GPTQ relies on an independent optimization strategy for each layer. This ensures that the impact of quantization on one layer does not propagate uncontrollably to the rest of the model. The procedure follows these steps:

1. **Input and Output of Each Layer:** A small dataset is used to calculate the inputs (X) and outputs for each layer of the model. These inputs and outputs serve as a reference to assess the quantization impact.
2. **Local Optimization:** For each layer, an optimization problem is solved to minimize the difference between the original output (with full-precision weights) and the output generated with quantized weights. This ensures each layer is locally adjusted to preserve the model's overall accuracy.
3. **Sequential Processing:** The model is quantized sequentially, layer by layer. The inputs to each layer are updated using the outputs from already quantized layers. This approach ensures efficient quantization while maintaining consistency across the model.

Inference with models quantized using GPTQ incorporates key features to optimize performance:

1. **Dynamic De-quantization:** Weights stored in low precision (e.g., 3 or 4 bits) are dynamically converted to higher precision during runtime. This enables accurate computations without significantly increasing storage requirements.
2. **Optimized Kernels:** Custom kernels are used to perform matrix-vector multiplications, where the matrix is quantized, and the vector remains in high precision.

These kernels are designed to minimize memory movement, resulting in significant acceleration of inference time.

2.3.3 Quanto

Quanto (Quantization for Transformers Optimization) ([12], [13]) is a cutting-edge quantization method specifically designed to optimize the deployment of large language models while minimizing the trade-off between efficiency and accuracy. Developed as a PTQ technique, Quanto allows for the efficient scaling of Transformer-based models to hardware with limited computational and memory resources. By leveraging advanced techniques for weight distribution analysis and precision tuning, Quanto preserves the essential properties of the model while significantly reducing its memory footprint and computational demands. This makes it ideal for environments where large-scale deployment of Transformer models is necessary, but computational and memory resources are limited.

The core principle of Quanto lies in precision-aware quantization, which uses statistical and structural insights into the layers of Transformer models to determine the optimal precision for different components of the model. This approach is similar to GPTQ but, while GPTQ relies heavily on second-order information (e.g., Hessian approximations) and gradient-based error metrics to prioritize important weights, Quanto adopts a statistical and structural approach. It analyzes the distribution of activations and weights, quantization noise and layer sensitivity to decide which components require higher precision and which can tolerate aggressive quantization. This allows Quanto to perform mixed-precision quantization, similar to GPTQ's selective quantization of high-impact weights.

The quantization workflow in Quanto can be broken down into the following steps:

1. **Statistical Analysis and Sensitivity Scoring:** A small calibration dataset is used to analyze the activation ranges and weight distributions across the layers of the model. Sensitivity scoring is then performed to rank the importance of individual weights and activations within each layer. This analysis helps determine which weights are critical to the model's performance and which can tolerate a reduction in precision.

2. **Precision Allocation:** Based on the sensitivity scores, Quanto allocates mixed-precision configurations to each layer and component. Critical weights are assigned higher precision (e.g., 8-bit or 16-bit), while less critical weights are reduced to lower precision (e.g., 4-bit or ternary). Quantization scaling factors are computed to minimize the difference between the full-precision and quantized outputs, ensuring that the quantized model maintains the original model's performance.

3. **Layer-Wise Quantization:** Quanto processes each Transformer layer sequentially,

applying quantization decisions layer by layer. This approach ensures that errors introduced by the quantization of one layer do not propagate uncontrollably to subsequent layers. The outputs of the quantized layers are recalculated dynamically and used as inputs for the next layer's quantization process.

4. **Block-Wise Optimization:** For very large models, Quanto applies block-wise quantization, dividing large weight matrices into smaller, more manageable submatrices or blocks. This reduces memory and computation overhead while ensuring that critical components are quantized with higher precision. This block-wise approach allows Quanto to scale efficiently to models with billions of parameters without compromising performance.

Once a model is quantized with Quanto, several techniques are used to optimize its performance during inference:

- **Dynamic Weight Reconstruction:** The quantized weights are stored in low-precision formats (e.g., 3-bit or 4-bit) to save memory. However, during inference, these weights are dynamically reconstructed to higher precision when needed, balancing memory efficiency with computational accuracy. This dynamic reconstruction ensures that the model can perform complex computations without significant performance degradation.
- **Custom Kernels for Speed:** Quanto leverages optimized inference kernels designed specifically for hardware accelerators, such as NVIDIA GPUs. These custom kernels are optimized to minimize memory transfer overhead and enable efficient execution of quantized matrix operations. By reducing the time spent on memory transfers and ensuring that matrix multiplications are handled efficiently, Quanto accelerates inference speed.

2.3.4 HQQ

Half-Quadratic Quantization (HQQ) ([2], [17]) is an advanced post-training quantization (PTQ) method that optimizes large language models, minimizing both memory requirements and computational cost. Unlike traditional gradient-based quantization methods, such as GPTQ, HQQ leverages a semi-quadratic optimization approach, which does not require any calibration data or fine-tuning. This enables HQQ to quantize models significantly faster (up to 100 times faster than traditional techniques), making it highly suitable for environments with limited computational resources.

The primary goal of HQQ is to minimize the quantization error between the original model weights and their quantized counterparts. HQQ formulates quantization as an optimization problem that focuses on reducing the quantization error between the original and quantized weights. This approach involves using a sparse-aware loss function, which

aims to preserve important weight structures while effectively handling outliers. By focusing on weight errors rather than activations, HQQ can achieve efficient quantization without requiring additional training data, thus improving the speed and accessibility of the process.

One of the distinguishing features of HQQ is that it uses a semi-quadratic optimization technique, which provides a mathematical formulation that results in a closed-form solution for quantization. This avoids the need for expensive iterative processes, which are commonly used in other quantization methods. The quantization process in HQQ can be broken down into the following steps:

1. **Problem Formulation:** In HQQ, quantization is treated as an optimization problem, explicitly minimizing the error between the original weights and their quantized versions by defining an objective function based on the weight error.
2. **Semi-Quadratic Solution:** The optimization problem is solved using a semi-quadratic method, which provides a closed-form solution. This method enables the algorithm to find the optimal quantization parameters for each weight while minimizing the error introduced by the quantization process. This mathematical approach results in an efficient quantization process without compromising the accuracy of the model.

3 Objectives and Methodology

3.1 Objectives

The objective of this work is to experiment with quantization and analyze the impact it has on a common use Large Language Model (LLM), that will work as our base model, in order to advance towards the creation of a pipeline to help democratize the use of LLMs and reduce their environmental impact. Specifically, this work evaluates how different quantization techniques affect the model's performance, efficiency and accuracy in sentiment classification tasks and in common text generation. More precisely, aims of this work can be listed as follows:

1. Analyze state-of-the-art quantization methods suitable for this work and choose the most interesting ones.
2. Compare the effect each chosen quantization method has in our base model's sentiment classification performance.
3. Compare the effect each chosen quantization method has in our base model's text generation performance.
4. Discuss results obtained for each quantization method considering accuracy, perplexity, time, size and emissions results.

3.2 Methodology

The methods used to reach all objectives could be divided in the following steps:

1. Revise essential background concepts and select state-of-the-art quantization methods and measures for LLM performance.
2. Select a common use LLM as base model.
3. Decide where to run the experiments and create a working environment.
4. Prepare the base model for sentiment analysis.
5. Quantize the model and run experiments on sentiment analysis.
6. Quantize the model and run experiments on text generation.
7. Analyze the results and extract conclusions.

Each step is described in a separate section down below.

3.2.1 Revise essential background concepts and select state-of-the-art quantization methods and measures for LLM performance

To establish a strong foundation for this work, several papers and online resources were read and examined in order to gather enough information about quantization techniques and evaluation of LLMs. The basics about Machine Learning were already known, so this process was focused on these two topics.

Regarding quantization, the first approach was to understand the basic concepts related to it, such as common data types used or the difference between symmetric and asymmetric quantization. After this, numerous papers were read about quantization techniques. Some of them stood out and were selected for this work due to their prevalence in the industry and ease of use for this project: BitsAndBytes ([5]), GPTQ ([8]), Quanto ([12], [13]) and HQQ ([2], [17]) (see sections 2.3.1, 2.3.2, 2.3.3 y 2.3.4). Other quantization methods such as Activation-Aware Weight Quantization (AWQ) ([16]) or AQLM ([6]) were also considered due to being widely used. However, their hardware requirements were too high and out of the scope of this work.

Secondly, a search to find a suitable performance measure for LLMs was made. Perplexity was the chosen measure after reading several resources and papers ([3], [15]). Although it has limitations, its natural intrinsic meaning and the fact that this work will perform experiments with comparable models make this metric suitable for this work (see section 2.1.7).

However, before attempting to compute perplexity or evaluate the models in generative settings, a simpler and more interpretable downstream task was selected in order to verify the correct functioning of the models and the quantization pipeline. Sentiment analysis was chosen because it is a well-established and widely studied task in natural language processing, with clearly defined evaluation metrics such as accuracy and F1-score. Moreover, it allows for a straightforward assessment of the impact that quantization has on the model's performance in classification scenarios.

For this purpose, the TweetEval dataset ([18]) was used. TweetEval is a benchmark specifically designed for evaluating NLP models on Twitter data across multiple tasks, including sentiment analysis. It provides a standardized dataset with a balanced distribution of sentiments and a variety of linguistic features, including informal language, slang, emojis and hashtags, which makes it especially useful for testing the generalization capabilities of LLMs. Its accessibility through the Hugging Face *Datasets* library and the availability of established baselines also contributed to its suitability for this project.

3.2.2 Select a common use LLM as base model

The previous revision of the state-of-the-art, although focused on different topics, also gave some ideas about commonly used LLMs. Nowadays, LLMs are widely applied in various tasks, such as text classification, question answering and natural language generation. Among the various available options, LLaMA 2 7B ([21]) was selected as the base model for this study for several key reasons.

Firstly, LLaMA 2, developed by Meta, has shown competitive performance in a wide range of natural language processing (NLP) benchmarks, making it a strong candidate for a general-purpose LLM. The 7B variant, with its balance between model size and computational efficiency, allows for meaningful analysis without excessive resource requirements, making it particularly suitable for the scope of this research.

Furthermore, LLaMA 2 has been extensively evaluated in various settings, providing a solid foundation for benchmarking its performance against different quantization methods. For this reason, LLaMA 3 or LLaMA 4 were not selected for this work, since the primary goal is to analyze the impact of quantization rather than to evaluate the latest available model. LLaMA 2 provides a well-documented and extensively studied baseline, and using a newer model would not necessarily provide further insights into the core objective of this study. Additionally, LLaMA 3 did not provide an improvement in accuracy (results shown in Table 1) big enough in order to justify its use, which requires extra computational power.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Negative	0.75	0.75	0.75	3972
Neutral	0.74	0.71	0.73	5937
Positive	0.69	0.77	0.73	2375
Accuracy			0.73	12284
Macro avg	0.73	0.74	0.73	12284
Weighted avg	0.74	0.73	0.73	12284

Table 1: Results obtained with LLaMA 3 following the same experiments as with LLaMA 2 in this work.

In addition to this, given that LLaMA 2 is publicly available and widely used in the research community, it also allows for reproducibility of results, which is essential in the context of comparing different quantization approaches.

Other popular LLMs, such as OpenAI's GPT models (e.g., ChatGPT) and Google's PaLM, were also considered for this study. However, these models were ultimately not selected due to limitations in their availability and licensing. While these models are

widely recognized for their impressive performance across various NLP tasks, they are not fully open-source and often come with usage restrictions, making them less suitable for reproducible research and experimentation in an open-source environment. In contrast, LLaMA 2 7B is open-source and freely accessible, which enables greater flexibility in evaluating and modifying the model. This ensures that the results of this study can be easily reproduced and extended by other researchers, aligning with the goals of transparency and open research in the field of machine learning. Finally, is it worth noting that this work is previous to the release of DeepSeek models, so these models were not considered.

3.2.3 Creation of a working environment

It was needed to decide where the experiments conducted in this work were going to be made. This decision needed to consider both, hardware and software requirements. All the tests, trainings and inferences were made in a server managed by the University of Murcia, with the following hardware specifications:

- Hard-Disk Storage
  - Total Capacity: 3.8 TiB
- RAM Storage
  - Total Capacity: 984 GiB
- CPUs:
  - Number of CPUs: 80
  - CPU Type: Intel(R) Xeon(R) Silver 4316
  - CPU Base Freq: 2.30 GHz
- GPUs:
  - Number of GPUs: 4
  - GPU Types: 1 x NVIDIA A100 80GB, 1 x NVIDIA A100 40GB, 2 x NVIDIA L4 24GB

The connection to this server and use of it was done through SSH, specifically using *Visual Studio Code* and its extension for this purpose.

All the programming was done in Python and standard Python virtual environments were used, so that each environment had the correct libraries and versions discrepancies

were avoided. A virtual environment was created for each quantization method, following the usual steps:

```
python3 -m venv .env
source .env/bin/activate
```

For each method, all the required libraries were installed using *pip*. These libraries include:

- PyTorch
- Transformers
- scikit-learn
- Accelerate
- Datasets
- Codecarbon
- Wandb
- Bitsandbytes
- Auto-gptq
- Hqq
- Optimum-quanto

Each library can be installed with the same command:

```
pip install [LIBRARY-NAME]
```

It is also worth noting that libraries for GPU management are not required separately because *PyTorch*, a deep learning framework, includes a precompiled CUDA version within its installation.

3.2.4 Prepare the base model for sentiment analysis

After the base model was chosen and the working environment was all set, the next step was to prepare the base model for the sentiment analysis task.

The first approach for this was to conduct a zero-shot experiment with the base model. This approach consisted of prompting the model directly with a tweet and asking it to classify the sentiment as positive, negative, or neutral without any task-specific training.

While this method is efficient in terms of time and resources, it heavily relies on the base model's general knowledge and ability to understand and reason about sentiment without being explicitly trained for the task. Code used for this approach can be found in annex 10.3 and [7].

However, the results of the zero-shot evaluation were not satisfactory, as shown in Table 2. This behavior is expected given that the model was originally trained for general language modeling and not for sentiment classification specifically.

Table 2: Zero-shot sentiment classification performance

Class	**Precision**	**Recall**	**F1-score**	**Support**
Negative	0.52	0.38	0.44	3339
Neutral	0.52	0.24	0.33	5033
Positive	0.27	0.74	0.40	2050
Accuracy		0.38 (on 10422 samples)		
Macro avg	0.44	0.45	0.39	
Weighted avg	0.47	0.38	0.38	

As shown in Table 2, the overall accuracy is only 38%, which is barely above chance level for a three-class classification task. The model struggles especially with the *neutral* class, achieving a recall of just 24%. Although the recall for the *positive* class is relatively high (74%), the precision is extremely low (27%), suggesting that the model over-predicts this class, leading to a high number of false positives.

Due to these results, it was decided to adapt the base model to the sentiment analysis task by adding a classification head (a simple feedforward layer) on top of the model's output representations. This classification layer maps the output embeddings of the model to a set of predefined sentiment classes. By training this extended model on a labeled sentiment analysis dataset, the model could learn task-specific patterns and significantly improve its accuracy and robustness in classification.

This approach leverages the representational power of the pre-trained language model while allowing it to specialize through fine-tuning, making it more suitable for the domain-specific task of sentiment analysis. This fine-tuning process also facilitates a more direct comparison between the performance of the quantized and non-quantized versions of the model under realistic inference conditions.

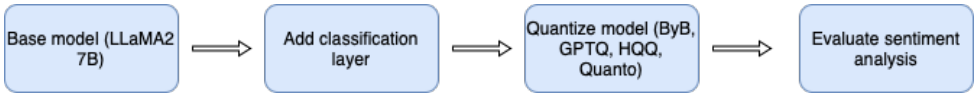

Figure 6: Pipeline for sentiment analysis experiments

3.2.5 Quantize the model and run experiments on sentiment analysis

After the working environment and the base model were all set, the next step was to quantize the base model with the classification layer and run experiments in sentiment analysis with the TweetEval dataset.

The base model was quantized with the selected methods (BitsAndBytes, GPTQ, Quanto and HQQ) using their respective libraries and the *Transformer* library from HugginFace (except with Quanto, where the own Quanto library methods were used to quantize the model).

After this, the experiments were conducted. Metrics such as accuracy, F1 score, inference time, maximum GPU memory consumption and emissions were recorded. To help with these tasks, the Weights and Biases (Wandb) and CodeCarbon libraries were used. The first one is a powerful library designed to track, visualize and manage machine learning experiments. It allows researchers and developers to log metrics such as loss, accuracy, model parameters, gradients and system resource usage in real time, providing interactive dashboards to monitor training progress and compare different runs. In this work, Wandb was used to ensure rigorous experiment tracking, making it easy to reproduce results, analyze the performance and detect potential issues during training and evaluation. Its integration into the workflow helped with experiment transparency and accelerated all the experiments. The second one is a library that estimates CO_2 emissions associated with model training and inference, as seen in section 2.1.6. Its integration into the workflow helped to study the potential reduction in the environmental impact that quantization has.

A diagram of the pipeline created for the sentiment analysis experiments is shown in Figure 6.

3.2.6 Quantize the model and run experiments on text generation

The process followed to run experiments on text generation was similar to the one followed for sentiment analysis, although the base model without the classification layer was used.

This base model was quantized with BitsAndBytes, GPTQ, HQQ and Quanto using their respective libraries and the *Transformer* library from HugginFace (except with Quanto, where the own Quanto library methods were used to quantize the model). However, only the 8-bit quantization option could be used with Quanto, since the 4-bit option

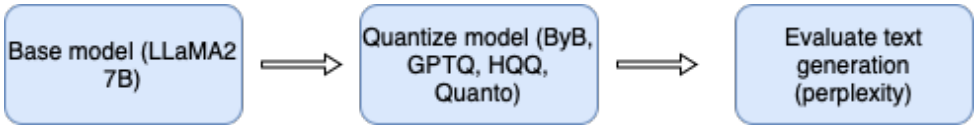

Figure 7: Pipeline for text generation experiments

has known problems when applied to big models in text generation tasks.

After this, the experiments were conducted. The main metric computed in these experiments was perplexity, as stated before. Other important metrics such as inference time, maximum GPU memory consumption and emissions were also recorded. To help with these tasks, the Weights and Biases (Wandb) and CodeCarbon libraries were used.

A diagram of the pipeline for text generation experiments is shown in Figure 7.

3.2.7 Analyze the results and extract conclusions

In this stage, the experimental outcomes will be evaluated. The quantized models' performance will be assessed against their non-quantized counterparts, focusing on metrics like accuracy, perplexity, inference speed, memory requirements and power consumption. The discussion will center on weighing the advantages of quantization, such as lower resource demands, against potential downsides, like reduced accuracy. Based on this comparison, insights will be formed about the real-world applicability and value of quantization techniques.

4 Design of the Solutions

This section details the specific implementation steps followed to carry out the experiments proposed in the methodology (see Section 3.2). It is structured around the two main tasks under evaluation: sentiment analysis and text generation.

Firstly, we describe the preparation and fine-tuning of the base LLaMA 2 model for sentiment classification. Then, we present the procedure for evaluating various quantized versions of the model on the same task. Finally, we outline the experimental design for text generation.

4.1 Preparation of the base model for sentiment analysis

To fine-tune the LLaMA 2 model for sentiment analysis, the TweetEval dataset was used, focusing on the sentiment classification task on its training subset. The dataset was tokenized using the LLaMA tokenizer, with padding and truncation applied to match the maximum token length found in the training and validation sets. The pre-trained `meta-llama/Llama-2-7b-hf` model was adapted for sequence classification by adding a classification head with three output labels. This new classification layer needed to be fine-tuned with our sentiment dataset TweetEval. To make the fine-tuning more efficient and lightweight, Low-Rank Adaptation (LoRA) ([10]) (see Section 2.1.5) was applied to the attention layers using the PEFT library. This allowed us to reduce the number of trainable parameters to only 0.0636% and carry out the fine-tuning on a single NVIDIA A100 40GB GPU, using the Hugging Face *Trainer* API with a learning rate of 2×10^{-5}, a batch size of 8, and a total of 3 epochs.

The code used for this training can be found on annex 10.4 and [7].

4.2 Experiments on sentiment analysis

For the evaluation of the quantized models, the environment was configured to restrict visibility to a single GPU (the one used: NVIDIA A100 40GB GPU) by setting the appropriate CUDA environment variables. This ensured controlled and reproducible usage of the computational resources.

The LoRA fine-tuned base model and its associated tokenizer were loaded from a local directory and quantized using the *Transformer* library from HugginFace. Quantization was made using different variants of each model. For BitsAndBytes and Quanto, the 8 and 4 bit options were tested, while for GPTQ and HQQ the 3 bit option was also used. To monitor the environmental impact of both quantizing the model and its inference stage, CodeCarbon was employed, generating emissions logs for each process separately.

After this, the evaluation pipeline was implemented. Firstly, the model was loaded

onto the GPU using "device_map="auto"", to distribute the memory usage efficiently, and configured for sequence classification with three output labels.

To manage the dataset, the Hugging Face *datasets* library was used. The dataset was formatted into PyTorch tensors and organized in batches using DataLoader with batch sizes of 32 and 64. Each batch of text samples was tokenized, padded to the maximum sequence length observed in the test set to maintain consistency, and then fed into the model for prediction. It is important to mention that with the base model only batches of size 32 were used, due to the impossibility of running the experiment on the selected GPU. The reason for this is that the memory requirements for this configuration exceeded the GPU memory capacity.

During inference, memory usage was actively monitored by resetting and recording the peak GPU memory allocated at each batch, allowing precise tracking of resource consumption. Predictions were compared to ground truth labels to compute standard classification metrics such as precision, recall and F1-score, and printed via "classification_report" from the *scikit-learn* library.

The total inference time was measured from start to finish, providing an indication of the speed of the quantized model and key performance indicators, such as maximum memory usage, were logged to Weights & Biases (wandb) for later visualization and analysis.

This structured approach allowed not only a technical evaluation in terms of accuracy and speed, but also a broader analysis including memory optimization and environmental impact, offering a comprehensive view of the benefits and trade-offs introduced by model quantization.

Codes for quantization in 4 bits using BitsAndBytes, GPTQ, HQQ and Quanto and experiment run can be found in annexes 10.2.1, 10.2.2, 10.2.3 and 10.2.4. For all the codes and configurations, the repository ([7]) is available.

4.3 Experiments on text generation

For the evaluation of quantized models on text generation, the *WikiText-2* dataset was employed, focusing on measuring the perplexity over its test split.

The experimental environment was configured to limit computation to a single GPU (the one used: NVIDIA A100 40GB GPU) by setting the appropriate CUDA environment variables, ensuring controlled and reproducible execution. The `meta-llama/Llama-2-7b-hf` model and its tokenizer were loaded using the Hugging Face *Transformers* library, with quantization applied using different variants of each model. For BitsAndBytes, the 8 and 4 bit options were tested, while for GPTQ and HQQ the 3 bit option was also used. For Quanto, only the 8 bit variant was used, since the 4bit option has known problems when

applied to big models in text generation tasks, as stated before.

Environmental impact during both the model quantization and evaluation phases was monitored using CodeCarbon, generating separate emissions logs for each stage.

Once loaded, the model was evaluated using a custom perplexity computation pipeline that accounted for long sequences by splitting the inputs into overlapping chunks, processed using a sliding window approach with a stride of 512 tokens and a maximum context length of 1024 tokens. This allowed precise calculation of the negative log-likelihood (NLL) across the entire dataset, ensuring accurate perplexity estimation ([11]).

During inference, GPU memory usage was actively monitored by recording peak memory consumption at each batch. Weights & Biases (wandb) was used to log both the final perplexity scores and the maximum memory usage for subsequent visualization and comparison across runs.

This structured evaluation not only measured generation quality using perplexity, but also provided insights into the efficiency and environmental cost of running quantized large language models, offering a comprehensive view of the trade-offs introduced by low-bit precision.

The code for quantization with BitsAndBytes, GPTQ, HQQ and Quanto and the corresponding experiment run can be found in annexes 10.2.5, 10.2.6, 10.2.7 and 10.2.8. For all the codes and configurations, the repository ([7]) is available.

5 Results

In this section, results obtained for sentiment analysis and text generation using the chosen quantization methods are shown. Firstly, results for sentiment analysis experiments are presented. Then, results for text generation experiments are shown. Finally, all these results are combined and a new metric to evaluate LLMs is introduced.

5.1 Evaluating sentiment analysis

5.1.1 Accuracy and F1 score results

Results obtained for accuracy and F1 score for each model are shown in Figure 8. These results show how quantized models, particularly 8-bit models, perform very similarly to the base model. It is also interesting to notice how 3-bit models are the worst performers, with special mention to the poor results obtained with the 3-bit HQQ model.

5.1.2 Memory use results

Results obtained for the maximum GPU memory needed for each model are shown in Figure 9. These results show how quantized models require less memory than the base model, and this requirement is lower with a lower bit count. BitsAndBytes and GPTQ seem to be the best performing methods. Additionally, batch size also affects in GPU memory requirements, with larger batches requiring more memory.

5.1.3 Inference and quantization speed results

Results obtained for inference and quantization speed for each model are shown in Figures 10 and 11. These results show how BitsAndBytes and GPTQ models are the fastest ones. HQQ and Quanto models seem to perform even worse than the base model, although differences are small. Batch sizes also seem to affect, with larger batch sizes reducing inference time. Regarding quantization speed, GPTQ models need a lot of time to be quantized, with a huge difference with the other methods. Between the rest, BitsAndBytes is the fastest method to quantize. Regarding differences between bit count, it does not seem that there is a general relation between bit count and quantization time.

5.1.4 Inference and quantization emissions results

Results obtained for inference and quantization emissions for each model are shown in Figures 12 and 13. These results show how, following the same trend shown in speed results, BitsAndBytes and GPTQ models are the ones with the lower emissions, while HQQ and Quanto models have even higher emissions than the base model. The same way, higher batch sizes reduce emissions. This is coherent with the way CodeCarbon works

(see section 2.1.6), where emissions are calculated in a way that they are "proportional" to time. Regarding quantization emissions, results also follow the same trend seen before with quantization time results. GPTQ models have higher quantization emissions and between the rest BitsAndBytes has the lowest emissions. Regarding differences between bit count, it does not seem that there is a general relation between bit count and quantization emissions.

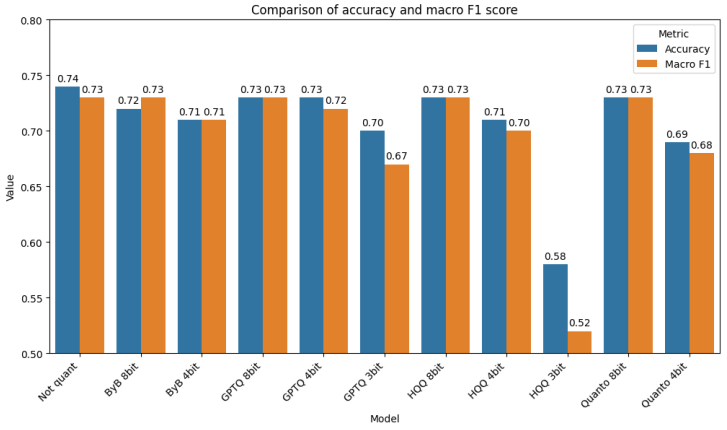

Figure 8: Accuracy and F1 score results

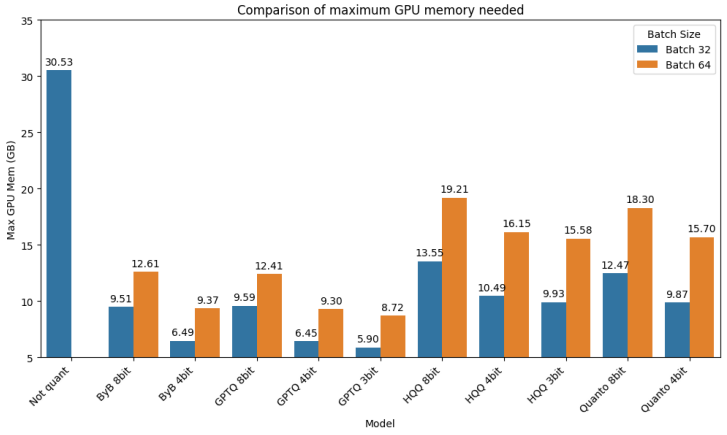

Figure 9: Memory use results for sentiment analysis experiments

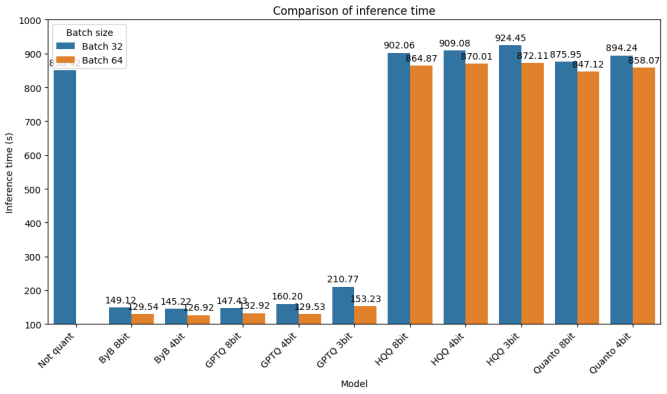

Figure 10: Inference speed results for sentiment analysis experiments

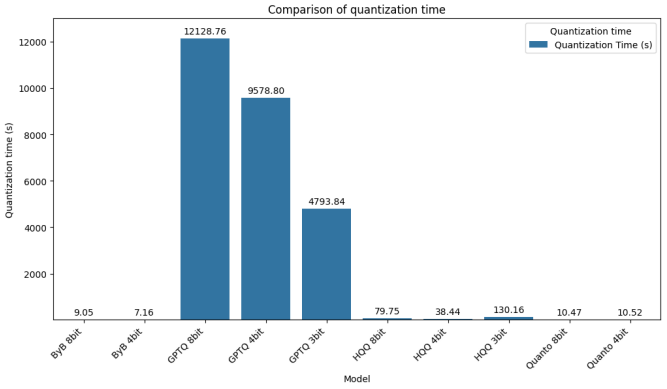

Figure 11: Quantization speed results for sentiment analysis experiments

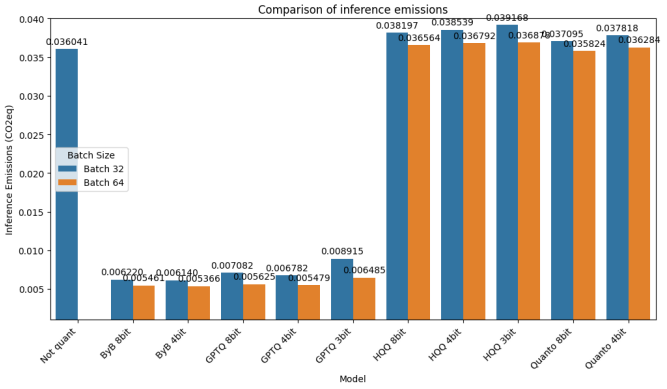

Figure 12: Inference emissions results for sentiment analysis experiments

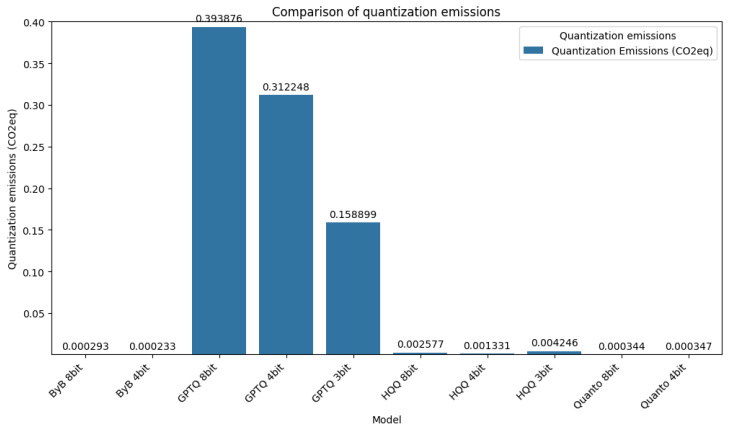

Figure 13: Quantization emissions results for sentiment analysis experiments

5.2 Evaluating text generation

5.2.1 Perplexity results

Results obtained for perplexity with each of the four quantization methods are shown in Figure 14.

We observe how the best perplexity is achieved by the not quantized model, followed very closely by the four 8-bit models. In fact, the 8-bit HQQ quantized model achieves the same perplexity than the not quantized model. Regarding the 4-bit models, they don't perform as good but they are not too far away from the others, specially the 4-bit GPTQ model. Lastly, the 3-bit models are the worst ones, obtaining substantially higher perplexities.

5.2.2 Inference speed results

Results obtained for inference time and quantization time are shown in Figures 15 and 16 respectively.

For inference time, we see how GPTQ obtains the worst results by far. Regarding HQQ, it looks like the lower the bit count the slower is the model, contrary to what would be expected. BitsAndBytes obtains good results, with both models being faster than the base model and with the 4-bit model being the fastest one. Finally, Quanto seems to obtain very promising results, since its 8-bit model performs closely to the fastest one while having a higher bit count.

For quantization time, we see how BitsAndBytes and Quanto are, by far, the fastest methods to quantize, while GPTQ is the slowest by a huge margin. HQQ obtains acceptable results. Regarding the difference between bits, with BitsAndBytes we don't see a

significant difference, while with GPTQ the 3-bit model is noticeably faster. Finally, with HQQ it seems like the lower the bit count, the slower is the quantization.

5.2.3 Memory use results

Results obtained for the maximum GPU memory needed during inference are shown in Figure 17.

We observe how all quantized models perform better than the base model by a big difference. It also seems like the lower the bit count, the lower memory requirements, as it was initially expected.

5.2.4 Emissions results

Results obtained for emissions produced by inference and quantization are shown in Figures 18 and 19 respectively.

We see how, following the same trend shown in speed results, GPTQ obtains the worst results by far. Regarding HQQ, it looks like the lower the bit count, the more emissions, contrary to what would be expected. Finally, BitsAndBytes and Quanto obtain similar results to the ones seen with inference speed, with the first one obtaining good results and the latter showing some promising results with its 8-bit model.

Quantization emissions results also follow the same trend seen before with quantization time results. We see how BitsAndBytes and Quanto are, by far, the best methods to quantize regarding emissions, while GPTQ is the worst one by a huge margin. HQQ obtains acceptable results, specially with its 8 and 4 bits variants. Regarding the difference between bits, with BitsAndBytes we don't see a significant difference, while with GPTQ the 3-bit model is noticeably more efficient. Finally, with HQQ it seems like the lower the bit count, the more emissions.

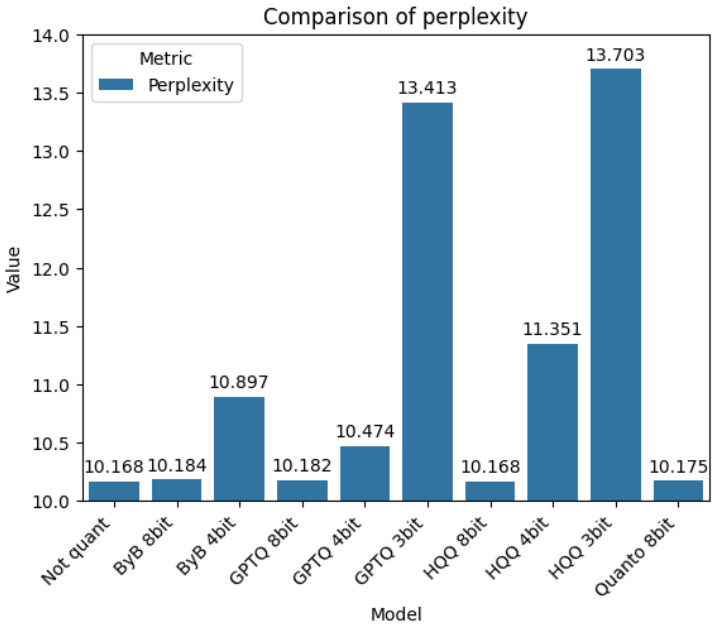

Figure 14: Perplexity results

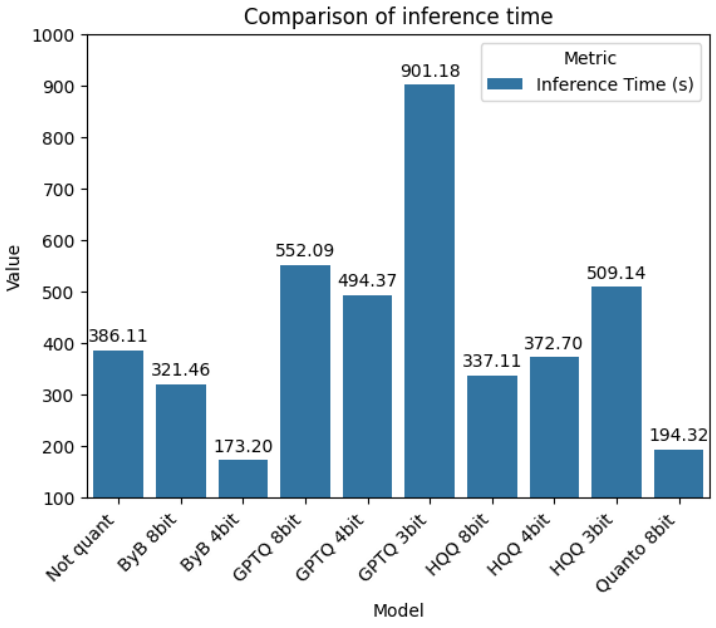

Figure 15: Inference speed results for text generation experiments

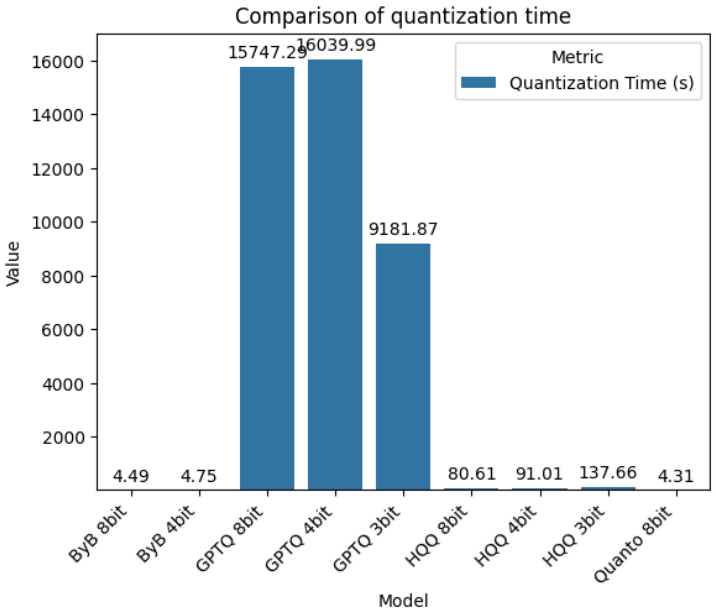

Figure 16: Quantization speed results for text generation experiments

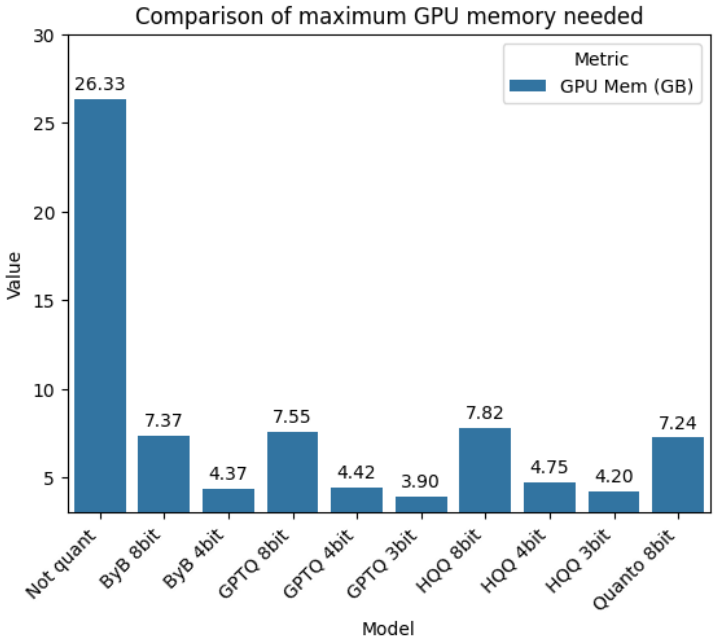

Figure 17: Memory use results for text generation experiments

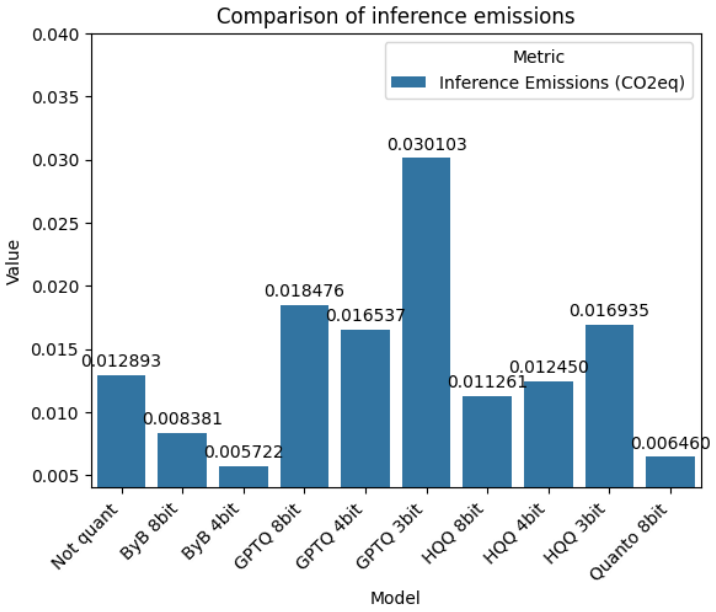

Figure 18: Inference emissions results for text generation experiments

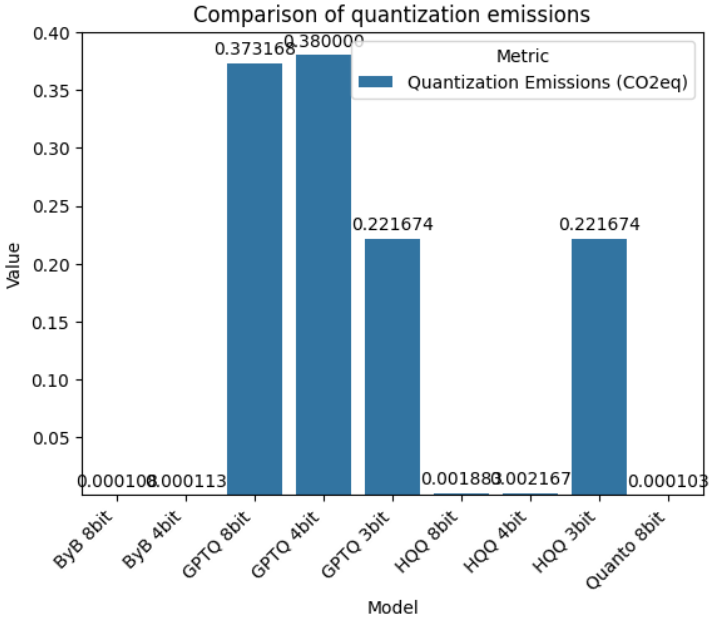

Figure 19: Quantization emissions results for text generation experiments

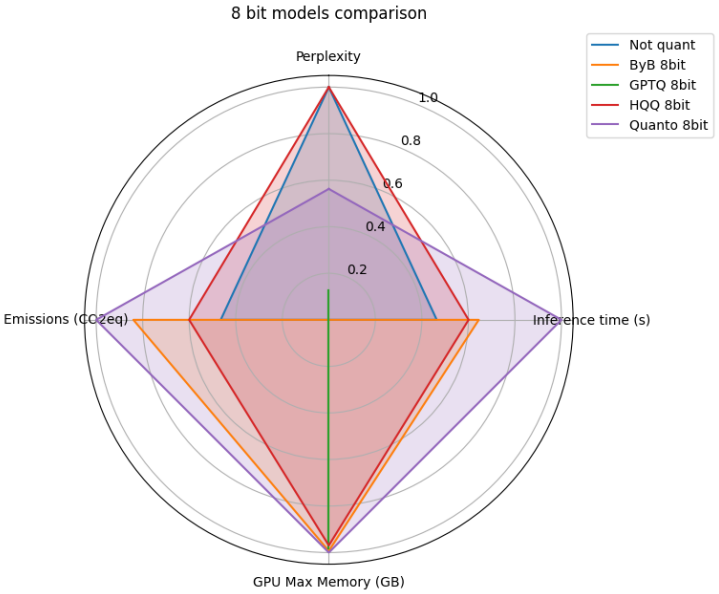

Figure 20: 8 bit models radar chart

5.2.5 All together

We now show these results together to have a better view of how each model performs. We show three radar charts comparing the 8-bit models, the 4-bit models and the 3-bit models in Figures 20, 21 and 22, respectively. These charts help showing which model performs better in each metric. For example, between the 8-bit models, the Quanto model is the best one regarding time and emissions, while the HQQ model matches the base model perplexity results. Between the 4-bit models, BitsAndBytes obtains the best results for time, emissions and memory, while GPTQ performs better in perplexity results. Finally, between the two 3-bit models, HQQ has the best performance in terms of memory, while GPTQ performs better in the other metrics.

5.2.6 A New Metric: Balancing Model Performance and Sustainability

As part of our study, we aim to evaluate models not only by their predictive quality, but also by their environmental impact. To do this, we introduce a new metric that jointly captures model performance (via perplexity) and carbon emissions (measured in CO_2eq), enabling a more holistic comparison across different model configurations.

Traditionally, lower perplexity is used as a signal of better model performance. Moreover, a lower carbon footprint is preferable from a sustainability standpoint. However, these two objectives often trade off against each other, since improving performance can come at the cost of significantly higher emissions.

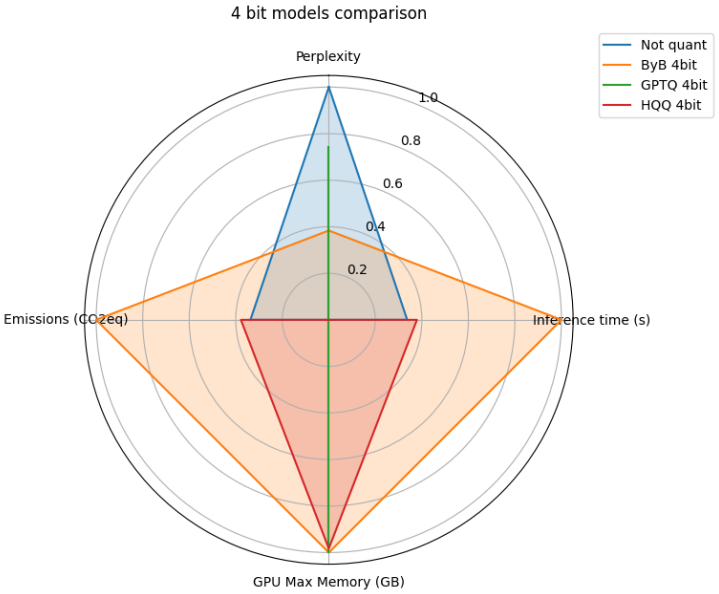

Figure 21: 4 bit models radar chart

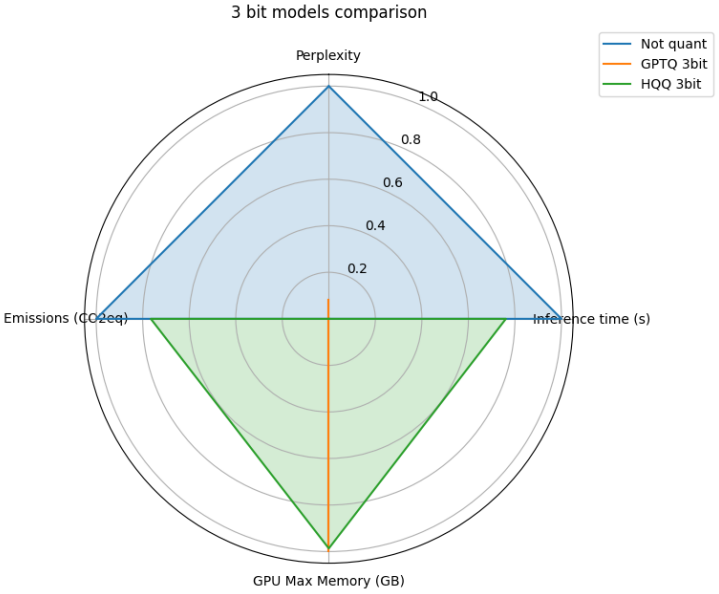

Figure 22: 3 bit models radar chart

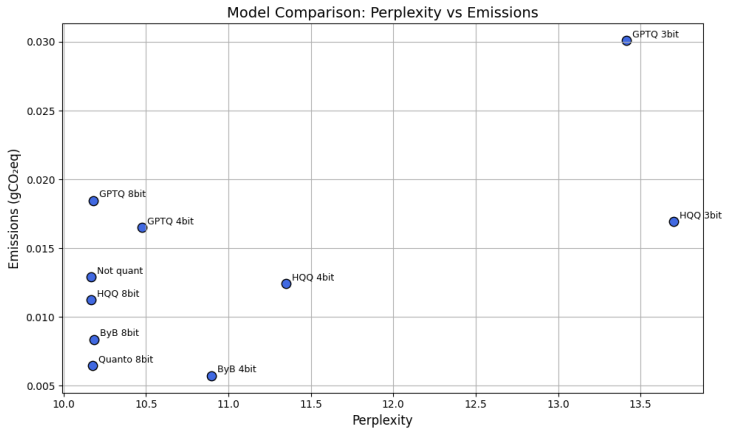

Figure 23: Comparing models in perplexity and emissions

To visualize this, we include the scatter plot shown in Figure 23, where each point represents a different model run, with perplexity on the x-axis and CO_2 emissions on the y-axis.

This conflict motivates the need for a single metric that reflects both aspects simultaneously.

Let p_i be the perplexity and c_i the carbon emissions of model i. We define a normalized perplexity score via the softmax function across all models:

$$\hat{p}_i = \frac{e^{p_i}}{\sum_j e^{p_j}} \tag{5}$$

This softmax transformation ensures that higher perplexity values are penalized exponentially, while preserving comparability within the current experiment.

We then define our combined score as:

$$\text{Score}_i = \hat{p}_i \times c_i \tag{6}$$

This metric directly couples performance and sustainability. Lower scores indicate better trade-offs, either due to excellent performance, low emissions, or both.

- A low score means the model has both low perplexity (relative to others) and low emissions.
- A high score reflects either poor performance, high emissions, or both.

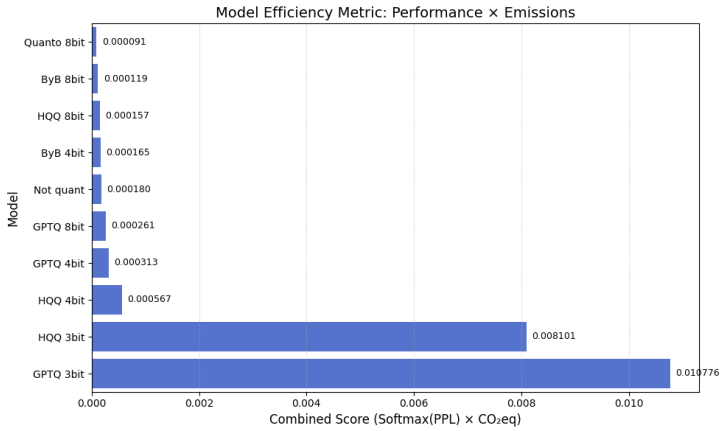

Figure 24: New metric results

- The metric does not require knowing the absolute range of perplexity or CO_2 values, and it adapts to the dynamic scale of any given experiment.

This approach provides a practical and interpretable way to assess models beyond accuracy alone.

In Figure 24 we see how, with this new metric, we are able to define which models are the best. Both 3-bit models are the worst ones, while the 8-bit versions of Quanto, BitsAndBytes and HQQ are the best options. The 4-bit option of BitsAndBytes also gets a good result. The not quantized base model also obtains a good result, being in fifth place. It seems like the good perplexity of the base model and the fact that some quantized models do not perform better in terms of speed and emissions help this model. Lastly, Figure 24 suggest that results obtained with this new metric may follow a logarithmic curve.

6 Discussion and limitations of the study

The experimental results obtained in this work offer valuable insights into the impact of quantization on LLMs such as LLaMA 2. Quantization has emerged as one of the most promising approaches to address the computational and environmental challenges posed by the increasing deployment of LLMs, and this study provides a detailed assessment of its implications across multiple dimensions.

From the perspective of model performance, the experiments demonstrate that quantization, particularly at 8-bit precision, allows models to retain much of their predictive capability in both sentiment analysis and text generation tasks. The 8-bit quantized models achieve accuracy, F1 score and perplexity metrics close to those of the original non-quantized model, indicating that the reduction in numerical precision does not necessarily imply a big loss in task performance. This is an encouraging finding, as it shows that substantial reductions in model size and computational cost can be achieved without sacrificing the quality of the predictions.

However, the situation becomes more complex when considering lower bit-width quantization, such as 4-bit and 3-bit. While 4-bit quantized models still maintain acceptable levels of performance in many cases (especially with methods such as BitsAndBytes and GPTQ), the 3-bit models exhibit notable performance degradation, both in terms of accuracy for classification tasks and perplexity for text generation. This highlights a critical trade-off between compression and performance: although aggressive quantization leads to greater savings in memory and storage, it can also compromise the practical utility of the model. Therefore, these results highlights that a careful selection of the quantization level is essential to balance efficiency gains against potential losses in predictive power. However, more experiments and deeper studies about 3-bit quantization would need to be made. This option is the one that saves more memory, so studying whether these poor results are a consequence of poor hardware support or other specific issues related with the LLM used or the experiments conducted would be very beneficial to the purpose of this project.

In terms of computational efficiency, quantization consistently reduces memory requirements, with a clear trend where lower bit-width models demand less GPU memory. This translates into the ability to deploy LLMs on less powerful hardware, enabling wider accessibility and reducing the barrier to entry to these powerful tools. Inference speed also generally improves with quantization, particularly with BitsAndBytes, which achieve faster execution times compared to both the baseline model and other quantization methods like HQQ and Quanto. Our study also observes that batch size plays a significant role in both speed and memory consumption, suggesting that optimization of batch processing can further enhance the benefits of quantization.

An specially relevant topic explored in this work is the environmental impact of quanti-

zation. By measuring emissions during both inference and quantization stages, the experiments reveal that certain quantization configurations, not only improve computational efficiency, but also lead to considerable reductions in energy consumption and associated carbon emissions. This positions quantization as a promising tool for reducing the environmental footprint of LLM deployments. However, it is important to note that not all methods yield the same environmental benefits in every situation: some quantization methods, such as GPTQ, show limited or even adverse effects depending on the task and bit-width used. This underlines the importance of evaluating quantization strategies for each task and particular case.

This project also introduces a new evaluation metric that jointly accounts for performance and environmental cost. The results obtained for this metric confirm all the conclusions extracted from the results previously shown. Although more testing, study and tuning for this metric are obviously needed, it opens the door to more balanced and holistic model assessment frameworks.

Finally, this work has some potential limitations that could be addressed in a possible future work. One of them is the lack of comparison against lighter architectures natively designed for classification tasks, which could have provided deeper insight into the trade-offs between starting from a large general purpose model and using specialized, smaller models from the beginning. Additionally, this work only explores the use of one LLM and two tasks, so using and comparing more LLMs and evaluation tasks could expand the research conducted. Lastly, more quantization methods should be explored in order to have a deeper insight to this technique.

7 Conclusions

It has been possible to adapt and optimize the deployment of LLaMA 2 models through quantization techniques, significantly reducing memory requirements and energy consumption without incurring major losses in performance. Using selected state-of-the-art quantization methods, models originally intended for high-resource environments have been made accessible to systems with more modest computational capabilities, thus contributing to the democratization of large language model research.

Through experimentation with the TweetEval dataset for sentiment classification, it has been validated that LLaMA 2, when complemented with a simple classification head, can maintain competitive accuracy even after undergoing aggressive quantization. Although the degradation in performance is not negligible in some cases, careful selection of the quantization method and parameters (such as bit width) proves crucial to balancing resource savings and task-specific quality.

In the text generation experiments, it was confirmed that LLaMA 2 retains a strong generative capacity even under aggressive quantization, particularly when using 8-bit or 4-bit representations. While perplexity scores do exhibit some degradation as precision decreases, the model's ability to generate coherent and contextually appropriate outputs remains mostly intact at moderate quantization levels. However, with 3-bit quantization the quality of the generated text deteriorates noticeably. As with sentiment classification, the choice of quantization method and configuration plays a critical role in minimizing performance loss while maximizing gains in efficiency and sustainability.

It has also been highlighted inference settings, like batch size, can have a strong influence on the observed behavior of the models. Wider hyperparameter searches, especially during fine-tuning and quantized evaluation, could have yielded further optimizations.

Additionally, this project has emphasized the importance of monitoring environmental impact during machine learning experiments. The use of CodeCarbon has demonstrated that quantized models can meaningfully reduce energy consumption and emissions, providing not only technical but also ethical advantages to their adoption.

Results obtained in this work invite reflection on the role of quantization as part of a more general strategy for sustainable AI. Quantization alone cannot solve all the challenges related to the resource-intensive nature of modern LLMs, but it represents an effective and immediately deployable solution. Combined with other techniques, such as adaptive computation or hardware-aware optimization, quantization can contribute to create more responsible and scalable AI systems. Future research directions could explore hardware specific optimizations, more sophisticated quantization-aware training techniques and evaluating models on a broader range of tasks to assess generalization capabilities post-quantization. Further integration of energy tracking and environmental

impact metrics should also become a standard in model evaluation, encouraging more sustainable AI practices.

To conclude, this work underlines that quantization is not merely a necessity for constrained environments, but a promising path towards efficient, responsible and accessible AI development.

8 Credits

I would like to thank my professors Juan Antonio Botía Blaya and Eduardo Martinez Gracia. They have given me all the support needed during this work. I have learned a lot from them, not just about computer science, but also about life and how to approach challenges.

9 Bibliography

[1] J. Alammar and M. Grootendorst. *Hands-On Large Language Models: Language Understanding and Generation*. O'Reilly Media, 2024. ISBN 9781098151076. https://www.oreilly.com/library/view/hands-on-large-language/9781098151083/.

[2] Y. Belkada and Shaji. Half-quadratic quantization of large machine learning models. https://mobiusml.github.io/hqq_blog/, 2024. Accessed: 2025-03-29.

[3] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003. https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.

[4] CodeCarbon. Codecarbon: A tool to estimate the carbon footprint of machine learning models, 2025. https://github.com/mlco2/codecarbon, Accessed: 2025-05-03.

[5] T. Dettmers and Bitsandbytes Foundation. Bitsandbytes: 8-bit and 4-bit Compression Techniques for AI Models, 2024. https://github.com/bitsandbytes-foundation/bitsandbytes, Accessed: 2024-03-29.

[6] V. Egiazarian, A. Panferov, D. Kuznedelev, E. Frantar, A. Babenko, and D. Alistarh. Extreme compression of large language models via additive quantization. 2024. URL https://arxiv.org/abs/2401.06118.

[7] D. F. Expósito, J. A. B. Blaya, and E. M. Gracia. Llama2_quantization. https://github.com/davidferex/LLaMA2_Quantization, 2025. Accessed: 2025-04-27.

[8] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. 2023. https://arxiv.org/abs/2210.17323.

[9] M. Grootendorst. A visual guide to quantization, 2023. https://www.maartengrootendorst.com/blog/quantization/.

[10] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. 2021. https://arxiv.org/abs/2106.09685.

[11] HuggingFace. Compute Perplexity — Hugging Face Transformers, 2024. https://huggingface.co/docs/transformers/perplexity.

[12] HuggingFace. Quanto: a PyTorch Quantization Backend for Optimum, 2025. https://huggingface.co/blog/quanto-introduction.

[13] HuggingFace. Optimum Quanto: A PyTorch Quantization Backend for Optimum, 2025. https://github.com/huggingface/optimum-quanto, Accessed: 2025-03-29.

[14] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63, 1977. https://doi.org/10.1121/1.2016299.

[15] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2025. https://web.stanford.edu/~jurafsky/slp3/, Online manuscript released January 12, 2025.

[16] MIT HAN Lab. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration, 2023. https://github.com/mit-han-lab/llm-awq, Accessed: 2025-03-29.

[17] MobiusML. HQQ: Half-Quadratic Quantization, 2025. https://github.com/mobiusml/hqq, Accessed: 2025-03-29.

[18] S. Rosenthal, N. Farra, and P. Nakov. Semeval-2017 task 4: Sentiment analysis in Twitter, 2017. https://arxiv.org/abs/1912.00741.

[19] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1355. https://aclanthology.org/P19-1355.

[20] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. LLaMA: Open and efficient foundation language models, 2023. https://arxiv.org/abs/2302.13971.

[21] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. https://arxiv.org/abs/2307.09288.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser,

and I. Polosukhin. Attention Is All You Need. 2017. https://arxiv.org/abs/1706.03762.

10 Annex

This section contains to all the code used in the experiments described above, as well as data obtained and support data visualizations. It is not meant to be read at once: the above sections include references to the annex, so that it is not neccesary to check every section.

10.1 Results obtained from experiments

10.1.1 Results for sentiment analysis

Model	**Quantization time (s)**	**Quantization emissions (CO2eq)**
ByB 8bit	9.05	0.000293
ByB 4bit	7.15	0.000233
GPTQ 8bit	12128.76	0.393876
GPTQ 4bit	9578.80	0.312248
GPTQ 3bit	4793.84	0.158899
HQQ 8bit	79.74	0.002577
HQQ 4bit	38.43	0.001331
HQQ 3bit	130.16	0.004246
Quanto 8bit	10.46	0.000344
Quanto 4bit	10.51	0.000347

Table 4: Results obtained for quantization in sentiment analysis experiments.

Model	Batch	Accuracy	F1 score	Inference time(s)	Max GPU mem (GB)	Emissions (CO2eq)
Base model	32	0.74	0.73	850.42	30.53	0.036041
ByB 8bit	32	0.72	0.73	149.12	9.51	0.006220
ByB 8bit	64	0.72	0.73	129.54	12.61	0.005461
ByB 4bit	32	0.71	0.71	145.22	6.49	0.006140
ByB 4bit	64	0.71	0.71	126.92	9.37	0.005366
GPTQ 8bit	32	0.73	0.73	147.43	9.59	0.007082
GPTQ 8bit	64	0.73	0.73	132.92	12.41	0.005625
GPTQ 4bit	32	0.73	0.72	160.20	6.45	0.006782
GPTQ 4bit	64	0.73	0.72	129.53	9.30	0.005479
GPTQ 3bit	32	0.70	0.67	210.77	5.90	0.008915
GPTQ 3bit	64	0.70	0.67	153.23	8.72	0.006485
HQQ 8bit	32	0.73	0.73	902.06	13.55	0.038197
HQQ 8bit	64	0.73	0.73	864.87	19.21	0.036564
HQQ 4bit	32	0.71	0.70	909.08	10.49	0.038539
HQQ 4bit	64	0.71	0.70	870.01	16.15	0.036792
HQQ 3bit	32	0.58	0.52	924.45	9.93	0.039168
HQQ 3bit	64	0.58	0.52	872.11	15.58	0.036878
Quanto 8bit	32	0.73	0.73	875.95	12.47	0.037095
Quanto 8bit	64	0.73	0.73	847.12	18.30	0.035824
Quanto 4bit	32	0.69	0.68	894.24	9.87	0.037818
Quanto 4bit	64	0.69	0.68	858.07	15.70	0.036284

Table 3: Results obtained for sentiment analysis experiments.

10.1.2 Results for text generation

Model	Perplexity	Inference time(s)	Maximum GPU memory (GB)	Emissions (CO2eq)
Base model	10.168	386.11	26.33	0.012893
ByB 8bit	10.184	321.46	7.37	0.008381
ByB 4bit	10.897	173.20	4.37	0.005722
GPTQ 8bit	10.182	552.09	7.55	0.018476
GPTQ 4bit	10.474	494.37	4.42	0.016537
GPTQ 3bit	13.413	901.18	3.90	0.030103
HQQ 8bit	10.168	337.11	7.82	0.011261
HQQ 4bit	11.351	372.70	4.75	0.012450
HQQ 3bit	13.703	509.14	4.20	0.016935
Quanto 8bit	10.175	194.32	7.24	0.006460

Table 5: Results obtained for text generation experiments.

Model	Quantization time (s)	Quantization emissions (CO2eq)
ByB 8bit	4.49	0.000108
ByB 4bit	4.75	0.000113
GPTQ 8bit	15747.29	0.373168
GPTQ 4bit	16039.99	0.380000
GPTQ 3bit	9181.87	0.221674
HQQ 8bit	80.61	0.001883
HQQ 4bit	91.01	0.002167
HQQ 3bit	137.66	0.221674
Quanto 8bit	4.31	0.000103

Table 6: Results obtained for quantization in text generation experiments.

10.2 Code

10.2.1 Example of BitsAndBytes 4 bit quantization and sentiment analysis test

```
import os
# Las GPUs se identifican como con nvidia-smi
os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
# La GPU visible al script es solo la primera
os.environ['CUDA_VISIBLE_DEVICES'] = '0'

import torch
from transformers import LlamaTokenizer, LlamaForSequenceClassification, LlamaConfig, BitsAndBytesConfig
from datasets import load_dataset
import time

from codecarbon import OfflineEmissionsTracker
import wandb

# Para imprimir la matriz de confusion
from sklearn.metrics import classification_report

# Barra de progreso de la inferencia
from torch.utils.data import DataLoader
from tqdm import tqdm

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_quant_batch64_ByB_LORA.csv", gpu_ids=[0])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_eval_batch64_ByB_LORA.csv", gpu_ids=[0])

wandb.init(project="ByB_Lora", name="4bit_batch64")

# Imprimo el nombre de GPU's disponibles para el script
num_gpus = torch.cuda.device_count()
for i in range(num_gpus):
    print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

# Cargar del dataset TweetEval para analisis de sentimientos
dataset = load_dataset("tweet_eval", "sentiment")

# Nombre del modelo
model_name = './results_lora'

# Clave Huggingface
hf_token = "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"

# Dispositivo: solo se indica cuda, en lugar de cuda:0, ya que solo esto visible el 0
device = 'cuda'

# Cargar el tokenizador
tokenizer = LlamaTokenizer.from_pretrained(
    model_name,
    token=hf_token)
tokenizer.pad_token = tokenizer.eos_token

tracker1.start()
# Cargar el modelo, indicando con LlamaConfig la clasificacion (por LoRA)
config = LlamaConfig.from_pretrained(model_name, num_labels=3)
bnb_config = BitsAndBytesConfig(
```

```
    load_in_4bit=True,     # Cuantizacion en 8 bits
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
)
model = LlamaForSequenceClassification.from_pretrained(
    model_name,
    token=hf_token,
    config=config,
    device_map="auto",
    quantization_config=bnb_config)
model.config.pad_token_id = model.config.eos_token_id
tracker1.stop()

# Enviar modelo a la GPU en una espera activa hasta que haya memoria
#loaded = False
#num_tries = 0
#while not loaded:
#    try:
#        model.to("cuda")
#        loaded = True
#    except torch.OutOfMemoryError:
#        num_tries += 1
#        print('New try:', num_tries)
#        time.sleep(5)


def evaluate_model(model, tokenizer, dataset, batch_size=8):
    # Poner el modelo en modo de evaluacion
    model.eval()

    start_time = time.time()
    true_labels = []
    pred_labels = []
    max_memory_usage = 0

    # Configura el dataset para que devuelva tensores de PyTorch, para usar DataLoader de PyTorch
    dataset.set_format("torch")
    test_dataset = dataset["test"]
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # Calculamos el tamano maximo de tokens
    max_length = max([len(tokenizer.tokenize(tweet)) for tweet in test_dataset["text"]])

    for batch in tqdm(test_loader, desc="Test"):
        # Sincronizar y medir la memoria antes
        torch.cuda.synchronize()
        torch.cuda.reset_peak_memory_stats()

        # Tokenizar el batch
        inputs = tokenizer(batch['text'], return_tensors="pt",
                           padding="max_length", truncation=True, max_length=max_length).to(device)

        # Inferencia. Desactivar el calculo de gradientes
        with torch.no_grad():
            outputs = model(**inputs)
            pred_batch = torch.argmax(outputs.logits, dim=-1).cpu().tolist()

        # Medir memoria utilizada despu s de la inferencia
        memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
        max_memory_usage = max(max_memory_usage, memory_used)
```

```
        true_labels.extend(batch['label'])
        pred_labels.extend(pred_batch)

    end_time = time.time()
    inference_time = end_time - start_time

    # Imprimir la matriz de confusion
    print(classification_report(true_labels, pred_labels, target_names=["Negative", "Neutral",
        "Positive"]))

    wandb.log({"Max_memory_usage_MB": max_memory_usage})

    return inference_time

tracker2.start()
# Evaluar modelo
time_no = evaluate_model(model, tokenizer, dataset, 64)
tracker2.stop()

# Mostrar resultados
print(f"Modelo LoRA cuantizado byb 4bit - Tiempo de inferencia: {time_no:.2f}s")
```

10.2.2 Example of GPTQ 4 bit quantization and sentiment analysis test

```
import os
# Las GPUs se identifican como con nvidia-smi
os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
# La GPU visible al script es solo la primera
os.environ['CUDA_VISIBLE_DEVICES'] = '0'

import torch
from transformers import LlamaTokenizer, LlamaForSequenceClassification, LlamaConfig, GPTQConfig
from datasets import load_dataset
import time

from codecarbon import OfflineEmissionsTracker
import wandb

# Para imprimir la matriz de confusion
from sklearn.metrics import classification_report

# Barra de progreso de la inferencia
from torch.utils.data import DataLoader
from tqdm import tqdm

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_quant_batch64_GPTQ_LORA.csv", gpu_ids=[0])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_eval_batch64_GPTQ_LORA.csv", gpu_ids=[0])

wandb.init(project="GPTQ_Lora", name="4bit_batch64")

# Imprimo el nombre de GPU's disponibles para el script
num_gpus = torch.cuda.device_count()
for i in range(num_gpus):
    print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

# Cargar del dataset TweetEval para analisis de sentimientos
```

```
dataset = load_dataset("tweet_eval", "sentiment")

# Nombre del modelo
model_name = '../bitsandbytes/results_lora'

# Clave Huggingface
hf_token = "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"

# Dispositivo: solo se indica cuda, en lugar de cuda:0, ya que solo esta visible el 0
device = 'cuda'

# Cargar el tokenizador
tokenizer = LlamaTokenizer.from_pretrained(
    model_name,
    token=hf_token)
tokenizer.pad_token = tokenizer.eos_token

tracker1.start()
# Cargar el modelo, indicando con LlamaConfig la clasificacion (por LoRA)
config = LlamaConfig.from_pretrained(model_name, num_labels=3)
gptq_config = GPTQConfig(bits=4, dataset="c4", tokenizer=tokenizer)
model = LlamaForSequenceClassification.from_pretrained(
    model_name,
    token=hf_token,
    config=config,
    device_map="auto",
    quantization_config=gptq_config)
model.config.pad_token_id = model.config.eos_token_id
tracker1.stop()

# Enviar modelo a la GPU en una espera activa hasta que haya memoria
loaded = False
num_tries = 0
while not loaded:
    try:
        model.to("cuda")
        loaded = True
    except torch.OutOfMemoryError:
        num_tries += 1
        print('New try:', num_tries)
        time.sleep(5)


def evaluate_model(model, tokenizer, dataset, batch_size=8):
    # Poner el modelo en modo de evaluacion
    model.eval()

    start_time = time.time()
    true_labels = []
    pred_labels = []
    max_memory_usage = 0

    # Configura el dataset para que devuelva tensores de PyTorch, para usar DataLoader de PyTorch
    dataset.set_format("torch")
    test_dataset = dataset["test"]
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # Calculamos el tamano maximo de tokens
    max_length = max([len(tokenizer.tokenize(tweet)) for tweet in test_dataset["text"]])

    for batch in tqdm(test_loader, desc="Test"):
```

```
        # Sincronizar y medir la memoria antes
        torch.cuda.synchronize()
        torch.cuda.reset_peak_memory_stats()

        # Tokenizar el batch
        inputs = tokenizer(batch['text'], return_tensors="pt",
                           padding="max_length", truncation=True, max_length=max_length).to(device)

        # Inferencia. Desactivar el calculo de gradientes
        with torch.no_grad():
            outputs = model(**inputs)
            pred_batch = torch.argmax(outputs.logits, dim=-1).cpu().tolist()

        # Medir memoria utilizada despues de la inferencia
        memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
        max_memory_usage = max(max_memory_usage, memory_used)

        true_labels.extend(batch['label'])
        pred_labels.extend(pred_batch)

    end_time = time.time()
    inference_time = end_time - start_time

    # Imprimir la matriz de confiusion
    print(classification_report(true_labels, pred_labels, target_names=["Negative", "Neutral",
         "Positive"]))

    wandb.log({"Max_memory_usage_MB": max_memory_usage})

    return inference_time

tracker2.start()
# Evaluar modelo
time_no = evaluate_model(model, tokenizer, dataset, 64)
tracker2.stop()

# Mostrar resultados
print(f"Modelo LoRA cuantizado gptq 8bit - Tiempo de inferencia: {time_no:.2f}s")
```

10.2.3 Example of HQQ 4 bit quantization and sentiment analysis test

```
import os
# Las GPUs se identifican como con nvidia-smi
os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
# La GPU visible al script es solo la primera
os.environ['CUDA_VISIBLE_DEVICES'] = '0'

import torch
from transformers import LlamaTokenizer, LlamaForSequenceClassification, LlamaConfig, HqqConfig
from datasets import load_dataset
import time

# Para imprimir la matriz de confusion
from sklearn.metrics import classification_report

from codecarbon import OfflineEmissionsTracker
import wandb

# Barra de progreso de la inferencia
```

```
from torch.utils.data import DataLoader
from tqdm import tqdm

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_quant_batch64_HQQ_LORA.csv", gpu_ids=[0])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_eval_batch64_HQQ_LORA.csv", gpu_ids=[0])

wandb.init(project="HQQ_Lora", name="4bit_batch64")

# Imprimo el nombre de GPU's disponibles para el script
num_gpus = torch.cuda.device_count()
for i in range(num_gpus):
    print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

# Cargar del dataset TweetEval para analisis de sentimientos
dataset = load_dataset("tweet_eval", "sentiment")

# Nombre del modelo
model_name = '../bitsandbytes/results_lora'

# Clave Huggingface
hf_token = "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"

# Dispositivo: solo se indica cuda, en lugar de cuda:0, ya que solo esta visible el 0
device = 'cuda'

# Cargar el tokenizador
tokenizer = LlamaTokenizer.from_pretrained(
    model_name,
    token=hf_token)
tokenizer.pad_token = tokenizer.eos_token

tracker1.start()
# Cargar el modelo, indicando con LlamaConfig la clasificacion (por LoRA)
config = LlamaConfig.from_pretrained(model_name, num_labels=3)
hqq_config = HqqConfig(nbits=4, group_size=64)
model = LlamaForSequenceClassification.from_pretrained(
    model_name,
    token=hf_token,
    config=config,
    device_map="auto",
    quantization_config=hqq_config)
model.config.pad_token_id = model.config.eos_token_id
tracker1.stop()

# Enviar modelo a la GPU en una espera activa hasta que haya memoria
'''
loaded = False
num_tries = 0
while not loaded:
    try:
        model.to("cuda")
        loaded = True
    except torch.OutOfMemoryError:
        num_tries += 1
        print('New try:', num_tries)
        time.sleep(5)
'''

def evaluate_model(model, tokenizer, dataset, batch_size=8):
```

```
    # Poner el modelo en modo de evaluacion
    model.eval()

    start_time = time.time()
    true_labels = []
    pred_labels = []
    max_memory_usage = 0

    # Configura el dataset para que devuelva tensores de PyTorch, para usar DataLoader de PyTorch
    dataset.set_format("torch")
    test_dataset = dataset["test"]
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # Calculamos el tamano maximo de tokens
    max_length = max([len(tokenizer.tokenize(tweet)) for tweet in test_dataset["text"]])

    for batch in tqdm(test_loader, desc="Test"):
        # Sincronizar y medir la memoria antes
        torch.cuda.synchronize()
        torch.cuda.reset_peak_memory_stats()

        # Tokenizar el batch
        inputs = tokenizer(batch['text'], return_tensors="pt",
                           padding="max_length", truncation=True, max_length=max_length).to(device)

        # Inferencia. Desactivar el calculo de gradientes
        with torch.no_grad():
            outputs = model(**inputs)
            pred_batch = torch.argmax(outputs.logits, dim=-1).cpu().tolist()

        # Medir memoria utilizada despues de la inferencia
        memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
        max_memory_usage = max(max_memory_usage, memory_used)

        true_labels.extend(batch['label'])
        pred_labels.extend(pred_batch)

    end_time = time.time()
    inference_time = end_time - start_time

    wandb.log({"Max_memory_usage_MB": max_memory_usage})

    # Imprimir la matriz de confiusion
    print(classification_report(true_labels, pred_labels, target_names=["Negative", "Neutral",
         "Positive"]))

    return inference_time

tracker2.start()
# Evaluar modelo
time_no = evaluate_model(model, tokenizer, dataset, 64)
tracker2.stop()

# Mostrar resultados
print(f"Modelo LoRA cuantizado gptq 4bit - Tiempo de inferencia: {time_no:.2f}s")
```

10.2.4 Example of Quanto 4 bit quantization and sentiment analysis test

```
import os
```

```
# Las GPUs se identifican como con nvidia-smi
os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
# La GPU visible al script es solo la primera
os.environ['CUDA_VISIBLE_DEVICES'] = '0'

import torch
from transformers import LlamaTokenizer, LlamaForSequenceClassification, LlamaConfig, QuantoConfig
from datasets import load_dataset
import time
from optimum.quanto import QuantizedModelForCausalLM, qint4, qint8

from codecarbon import OfflineEmissionsTracker
import wandb

# Para imprimir la matriz de confusion
from sklearn.metrics import classification_report

# Barra de progreso de la inferencia
from torch.utils.data import DataLoader
from tqdm import tqdm

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_quant_batch64_Quanto_LORA.csv", gpu_ids=[0])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_eval_batch64_Quanto_LORA.csv", gpu_ids=[0])

wandb.init(project="Quanto_Lora", name="4bit_batch64")

# Imprimo el nombre de GPU's disponibles para el script
num_gpus = torch.cuda.device_count()
for i in range(num_gpus):
    print(f"GPU {i}: {torch.cuda.get_device_name(i)}")

# Cargar del dataset TweetEval para analisis de sentimientos
dataset = load_dataset("tweet_eval", "sentiment")

# Nombre del modelo
model_name = '../bitsandbytes/results_lora'

# Clave Huggingface
hf_token = "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"

# Dispositivo: solo se indica cuda, en lugar de cuda:0, ya que solo esta visible el 0
device = 'cuda'

# Cargar el tokenizador
tokenizer = LlamaTokenizer.from_pretrained(
    model_name,
    token=hf_token)
tokenizer.pad_token = tokenizer.eos_token

tracker1.start()
# Cargar el modelo, indicando con LlamaConfig la clasificacion (por LoRA)
config = LlamaConfig.from_pretrained(model_name, num_labels=3)
model = LlamaForSequenceClassification.from_pretrained(
    model_name,
    token=hf_token,
    config=config,
    device_map="auto")
model.config.pad_token_id = model.config.eos_token_id
model = QuantizedModelForCausalLM.quantize(model, weights=qint4, exclude='lm_head')
```

```
tracker1.stop()

# Enviar modelo a la GPU en una espera activa hasta que haya memoria
loaded = False
num_tries = 0
while not loaded:
    try:
        model.to("cuda")
        loaded = True
    except torch.OutOfMemoryError:
        num_tries += 1
        print('New try:', num_tries)
        time.sleep(5)


def evaluate_model(model, tokenizer, dataset, batch_size=8):
    # Poner el modelo en modo de evaluacion
    model.eval()

    start_time = time.time()
    true_labels = []
    pred_labels = []
    max_memory_usage = 0

    # Configura el dataset para que devuelva tensores de PyTorch, para usar DataLoader de PyTorch
    dataset.set_format("torch")
    test_dataset = dataset["test"]
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # Calculamos el tamano maximo de tokens
    max_length = max([len(tokenizer.tokenize(tweet)) for tweet in test_dataset["text"]])

    for batch in tqdm(test_loader, desc="Test"):
        # Sincronizar y medir la memoria antes
        torch.cuda.synchronize()
        torch.cuda.reset_peak_memory_stats()

        # Tokenizar el batch
        inputs = tokenizer(batch['text'], return_tensors="pt",
                           padding="max_length", truncation=True, max_length=max_length).to(device)

        # Inferencia. Desactivar el calculo de gradientes
        with torch.no_grad():
            outputs = model(**inputs)
            pred_batch = torch.argmax(outputs.logits, dim=-1).cpu().tolist()

        # Medir memoria utilizada despues de la inferencia
        memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
        max_memory_usage = max(max_memory_usage, memory_used)

        true_labels.extend(batch['label'])
        pred_labels.extend(pred_batch)

    end_time = time.time()
    inference_time = end_time - start_time

    # Imprimir la matriz de confiusion
    print(classification_report(true_labels, pred_labels, target_names=["Negative", "Neutral",
         "Positive"]))

    wandb.log({"Max_memory_usage_MB": max_memory_usage})
```

```
    return inference_time

tracker2.start()
# Evaluar modelo
time_no = evaluate_model(model, tokenizer, dataset, 64)
tracker2.stop()

# Mostrar resultados
print(f"Modelo LoRA cuantizado quanto 4bit - Tiempo de inferencia: {time_no:.2f}s")
```

10.2.5 Example of BitsAndBytes quantization and text generation test

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from datasets import load_dataset
from tqdm import tqdm
from codecarbon import OfflineEmissionsTracker
import math
import os
import wandb

os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES'] = '1' # Cambia si usas otra GPU

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_ByB_quant_perplexity.csv", gpu_ids=[1])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_4bit_ByB_quant_eval_perplexity.csv", gpu_ids=[1])
wandb.init(project="Perplexity", name="4bit_ByB")

def load_llama2_model(model_id, quantization=None):
    if quantization == "4bit":
        quant_config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.float16,
            bnb_4bit_use_double_quant=True,
            bnb_4bit_quant_type="nf4",
        )
    elif quantization == "8bit":
        quant_config = BitsAndBytesConfig(load_in_8bit=True)
    else:
        quant_config = None

    tokenizer = AutoTokenizer.from_pretrained(model_id, use_fast=False)
    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        device_map="auto",
        quantization_config=quant_config,
        torch_dtype=torch.float16 if quantization else torch.float32
    )
    return tokenizer, model

def calculate_perplexity_precise(model, tokenizer, texts, max_length=1024, stride=512, device='cuda'):
    model.eval()

    max_memory_usage = 0
    nll_sum = 0.0
    n_tokens = 0
```

```
    for text in tqdm(texts, desc="Calculando perplexity"):
        encodings = tokenizer(text, return_tensors="pt", truncation=False)
        input_ids = encodings.input_ids.to(device)
        seq_len = input_ids.size(1)
        prev_end_loc = 0

        for begin_loc in range(0, seq_len, stride):
            end_loc = min(begin_loc + max_length, seq_len)
            trg_len = end_loc - prev_end_loc

            input_ids_chunk = input_ids[:, begin_loc:end_loc]
            target_ids = input_ids_chunk.clone()
            target_ids[:, :-trg_len] = -100 # solo los ultimos tokens contribuyen a la loss

            with torch.no_grad():
                torch.cuda.synchronize()
                torch.cuda.reset_peak_memory_stats()

                outputs = model(input_ids_chunk, labels=target_ids)
                neg_log_likelihood = outputs.loss

            memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
            max_memory_usage = max(max_memory_usage, memory_used)

            valid_tokens = (target_ids != -100).sum().item()
            effective_tokens = valid_tokens - target_ids.size(0) # ajustar por el shift interno
            nll_sum += neg_log_likelihood.item() * effective_tokens
            n_tokens += effective_tokens

            prev_end_loc = end_loc
            if end_loc == seq_len:
                break

    avg_nll = nll_sum / n_tokens
    ppl = math.exp(avg_nll)
    wandb.log({"Perplexity": ppl,
    "Max_memory_usage_MB": max_memory_usage})
    return ppl

if __name__ == "__main__":
    model_id = "meta-llama/Llama-2-7b-hf" # Asegurate de tener acceso
    quantization = "4bit" # Cambia a "8bit" o "4bit" si lo deseas

    print(f"Cargando modelo {model_id} con cuantizaci0n: {quantization or 'sin cuantizar'}")
    tracker1.start()
    tokenizer, model = load_llama2_model(model_id, quantization)
    tracker1.stop()

    print("Cargando WikiText-2...")
    dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="test[:100%]")
    texts = [sample["text"] for sample in dataset if sample["text"].strip()]

    print("Calculando perplexity (preciso)...")
    tracker2.start()
    ppl = calculate_perplexity_precise(model, tokenizer, texts)
    tracker2.stop()
    print(f"\n Perplexity precisa ({quantization or 'no quant'}): {ppl:.2f}")
```

10.2.6 Example of GPTQ quantization and text generation test

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, GPTQConfig
from datasets import load_dataset
from tqdm import tqdm
from codecarbon import OfflineEmissionsTracker
import math
import os
import wandb

os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES'] = '1' # Cambia si usas otra GPU

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_GPTQ_quant_perplexity.csv", gpu_ids=[1])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_GPTQ_quant_eval_perplexity.csv", gpu_ids=[1])
wandb.init(project="Perplexity", name="8bit_GPTQ")

def load_llama2_model(model_id, quantization=None):
    tokenizer = AutoTokenizer.from_pretrained(model_id, use_fast=False)
    if quantization == "4bit":
        quant_config = GPTQConfig(bits=4, dataset="c4", tokenizer=tokenizer)
    elif quantization == "8bit":
        quant_config = GPTQConfig(bits=8, dataset="c4", tokenizer=tokenizer)
    elif quantization == "3bit":
        quant_config = GPTQConfig(bits=3, dataset="c4", tokenizer=tokenizer)
    else:
        quant_config = None

    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        device_map="auto",
        quantization_config=quant_config,
        torch_dtype=torch.float16 if quantization else torch.float32
    )
    return tokenizer, model

def calculate_perplexity_precise(model, tokenizer, texts, max_length=1024, stride=512, device='cuda'):
    model.eval()

    max_memory_usage = 0
    nll_sum = 0.0
    n_tokens = 0

    for text in tqdm(texts, desc="Calculando perplexity"):
        encodings = tokenizer(text, return_tensors="pt", truncation=False)
        input_ids = encodings.input_ids.to(device)
        seq_len = input_ids.size(1)
        prev_end_loc = 0

        for begin_loc in range(0, seq_len, stride):
            end_loc = min(begin_loc + max_length, seq_len)
            trg_len = end_loc - prev_end_loc

            input_ids_chunk = input_ids[:, begin_loc:end_loc]
            target_ids = input_ids_chunk.clone()
            target_ids[:, :-trg_len] = -100 # solo los ultimos tokens contribuyen a la loss

            with torch.no_grad():
```

```
                torch.cuda.synchronize()
                torch.cuda.reset_peak_memory_stats()

                outputs = model(input_ids_chunk, labels=target_ids)
                neg_log_likelihood = outputs.loss

            memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
            max_memory_usage = max(max_memory_usage, memory_used)

            valid_tokens = (target_ids != -100).sum().item()
            effective_tokens = valid_tokens - target_ids.size(0) # ajustar por el shift interno
            nll_sum += neg_log_likelihood.item() * effective_tokens
            n_tokens += effective_tokens

            prev_end_loc = end_loc
            if end_loc == seq_len:
                break

    avg_nll = nll_sum / n_tokens
    ppl = math.exp(avg_nll)
    wandb.log({"Perplexity": ppl,
    "Max_memory_usage_MB": max_memory_usage})
    return ppl

if __name__ == "__main__":
    model_id = "meta-llama/Llama-2-7b-hf" # Asegurate de tener acceso
    quantization = "8bit" # Cambia a "8bit" o "4bit" si lo deseas

    print(f"Cargando modelo {model_id} con cuantizacion: {quantization or 'sin cuantizar'}")
    tracker1.start()
    tokenizer, model = load_llama2_model(model_id, quantization)
    tracker1.stop()

    print("Cargando WikiText-2...")
    dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="test[:100%]")
    texts = [sample["text"] for sample in dataset if sample["text"].strip()]

    print("Calculando perplexity (preciso)...")
    tracker2.start()
    ppl = calculate_perplexity_precise(model, tokenizer, texts)
    tracker2.stop()
    print(f"\n Perplexity precisa ({quantization or 'no quant'}): {ppl:.2f}")
```

10.2.7 Example of HQQ quantization and text generation test

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, HqqConfig
from datasets import load_dataset
from tqdm import tqdm
from codecarbon import OfflineEmissionsTracker
import math
import os
import wandb

os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES'] = '1' # Cambia si usas otra GPU

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_HQQ_quant_perplexity.csv", gpu_ids=[1])
```

```
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_HQQ_quant_eval_perplexity.csv", gpu_ids=[1])
wandb.init(project="Perplexity", name="8bit_HQQ")

def load_llama2_model(model_id, quantization=None):
    if quantization == "4bit":
        quant_config = HqqConfig(nbits=4, group_size=64)
    elif quantization == "8bit":
        quant_config = HqqConfig(nbits=8, group_size=64)
    elif quantization == "3bit":
        quant_config = HqqConfig(nbits=3, group_size=64)
    else:
        quant_config = None

    tokenizer = AutoTokenizer.from_pretrained(model_id, use_fast=False)
    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        device_map="auto",
        quantization_config=quant_config,
        torch_dtype=torch.float16 if quantization else torch.float32
    )
    return tokenizer, model

def calculate_perplexity_precise(model, tokenizer, texts, max_length=1024, stride=512, device='cuda'):
    model.eval()

    max_memory_usage = 0
    nll_sum = 0.0
    n_tokens = 0

    for text in tqdm(texts, desc="Calculando perplexity"):
        encodings = tokenizer(text, return_tensors="pt", truncation=False)
        input_ids = encodings.input_ids.to(device)
        seq_len = input_ids.size(1)
        prev_end_loc = 0

        for begin_loc in range(0, seq_len, stride):
            end_loc = min(begin_loc + max_length, seq_len)
            trg_len = end_loc - prev_end_loc

            input_ids_chunk = input_ids[:, begin_loc:end_loc]
            target_ids = input_ids_chunk.clone()
            target_ids[:, :-trg_len] = -100 # solo los ultimos tokens contribuyen a la loss

            with torch.no_grad():
                torch.cuda.synchronize()
                torch.cuda.reset_peak_memory_stats()

                outputs = model(input_ids_chunk, labels=target_ids)
                neg_log_likelihood = outputs.loss

            memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
            max_memory_usage = max(max_memory_usage, memory_used)

            valid_tokens = (target_ids != -100).sum().item()
            effective_tokens = valid_tokens - target_ids.size(0) # ajustar por el shift interno
            nll_sum += neg_log_likelihood.item() * effective_tokens
            n_tokens += effective_tokens

            prev_end_loc = end_loc
            if end_loc == seq_len:
```

```
            break

    avg_nll = nll_sum / n_tokens
    ppl = math.exp(avg_nll)
    wandb.log({"Perplexity": ppl,
    "Max_memory_usage_MB": max_memory_usage})
    return ppl

if __name__ == "__main__":
    model_id = "meta-llama/Llama-2-7b-hf" # Asegurate de tener acceso
    quantization = "8bit" # Cambia a "8bit" o "4bit" si lo deseas

    print(f"Cargando modelo {model_id} con cuantizacion: {quantization or 'sin cuantizar'}")
    tracker1.start()
    tokenizer, model = load_llama2_model(model_id, quantization)
    tracker1.stop()

    print("Cargando WikiText-2...")
    dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="test[:100%]")
    texts = [sample["text"] for sample in dataset if sample["text"].strip()]

    print("Calculando perplexity (preciso)...")
    tracker2.start()
    ppl = calculate_perplexity_precise(model, tokenizer, texts)
    tracker2.stop()
    print(f"\n Perplexity precisa ({quantization or 'no quant'}): {ppl:.2f}")
```

10.2.8 Example of Quanto quantization and text generation test

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, QuantoConfig
from optimum.quanto import QuantizedModelForCausalLM, qint4, qint8
from datasets import load_dataset
from tqdm import tqdm
from codecarbon import OfflineEmissionsTracker
import math
import os
import wandb

os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES'] = '1' # Cambia si usas otra GPU

tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_Quanto_quant_perplexity.csv", gpu_ids=[1])
tracker2 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_8bit_Quanto_quant_eval_perplexity.csv", gpu_ids=[1])
wandb.init(project="Perplexity", name="8bit_Quanto")

def load_llama2_model(model_id, quantization=None):
    tokenizer = AutoTokenizer.from_pretrained(model_id, use_fast=False)
    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        device_map="auto",
        torch_dtype=torch.float16 if quantization else torch.float32
    )
    if quantization == "4bit":
        model_quantized = QuantizedModelForCausalLM.quantize(model, weights=qint4, exclude='lm_head')
    elif quantization == "8bit":
        model_quantized = QuantizedModelForCausalLM.quantize(model, weights=qint8, exclude='lm_head')
```

```
    else:
        model_quantized = model
    return tokenizer, model_quantized

def calculate_perplexity_precise(model, tokenizer, texts, max_length=1024, stride=512, device='cuda'):
    model.eval()
    max_memory_usage = 0

    nll_sum = 0.0
    n_tokens = 0

    for text in tqdm(texts, desc="Calculando perplexity"):
        encodings = tokenizer(text, return_tensors="pt", truncation=False)
        input_ids = encodings.input_ids.to(device)
        seq_len = input_ids.size(1)
        prev_end_loc = 0

        for begin_loc in range(0, seq_len, stride):
            end_loc = min(begin_loc + max_length, seq_len)
            trg_len = end_loc - prev_end_loc

            input_ids_chunk = input_ids[:, begin_loc:end_loc]
            target_ids = input_ids_chunk.clone()
            target_ids[:, :-trg_len] = -100 # solo los ultimos tokens contribuyen a la loss

            with torch.no_grad():
                torch.cuda.synchronize()
                torch.cuda.reset_peak_memory_stats()

                outputs = model(input_ids_chunk, labels=target_ids)
                neg_log_likelihood = outputs.loss

            memory_used = torch.cuda.max_memory_allocated() / (1024 ** 2) # Convertir a MB
            max_memory_usage = max(max_memory_usage, memory_used)

            valid_tokens = (target_ids != -100).sum().item()
            effective_tokens = valid_tokens - target_ids.size(0) # ajustar por el shift interno
            nll_sum += neg_log_likelihood.item() * effective_tokens
            n_tokens += effective_tokens

            prev_end_loc = end_loc
            if end_loc == seq_len:
                break

    avg_nll = nll_sum / n_tokens
    ppl = math.exp(avg_nll)
    wandb.log({"Perplexity": ppl,
    "Max_memory_usage_MB": max_memory_usage})
    return ppl

if __name__ == "__main__":
    model_id = "meta-llama/Llama-2-7b-hf" # Asegurate de tener acceso
    quantization = "8bit"

    print(f"Cargando modelo {model_id} con cuantizacion: {quantization or 'sin cuantizar'}")
    tracker1.start()
    tokenizer, model = load_llama2_model(model_id, quantization)
    tracker1.stop()

    print("Cargando WikiText-2...")
    dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="test[:100%]")
```

```
texts = [sample["text"] for sample in dataset if sample["text"].strip()]

print("Calculando perplexity (preciso)...")
tracker2.start()
ppl = calculate_perplexity_precise(model, tokenizer, texts)
tracker2.stop()
print(f"\n Perplexity precisa ({quantization or 'no quant'}): {ppl:.2f}")
```

10.3 Zero-shot experiment

```
    import os
from datasets import load_dataset
import time
import re
from sklearn.metrics import classification_report

# Especificar GPU
os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'

import torch
from transformers import pipeline

# Nombre del modelo en Hugging Face
model_name = "meta-llama/Llama-2-7b-hf"

hf_token = "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"

generator = pipeline("text-generation", model=model_name, device=0, token=hf_token)

# Cargar el dataset TweetEval para analisis de sentimientos
dataset = load_dataset("tweet_eval", "sentiment")

correct = 0
total = 0
invalid = 0
start_time = time.time()

true_labels = []
predicted_labels = []

label_code = ['negative', 'neutral', 'positive']
er_label = re.compile(r'(?i)(negative)|(neutral)|(positive)')

for sample in dataset["test"]:
    tweet = sample["text"]
    prompt = f'Analyze the sentiment of the following tweet. ' \
             f'Respond with ONLY one word: "positive", "negative", or "neutral". Do not generate anything
                 else.\n\n' \
             f'Tweet: "{tweet}"\n' \
             f'Sentiment:'

    result = generator(prompt, max_new_tokens=5, truncation=True, return_full_text=False)

    # Obtener la etiqueta predicha
    m = er_label.search(result[0]['generated_text'])
    if m:
        predicted_label = label_code[m.lastindex-1]
        true_label = label_code[sample["label"]]

        # Comparar con la etiqueta real
        if predicted_label == true_label:
            correct += 1
        total += 1
        predicted_labels.append(predicted_label)
        true_labels.append(true_label)

        print(classification_report(
            true_labels, predicted_labels,
            labels=["negative", "neutral", "positive"]
```

```
        ))
    else:
        invalid += 1
```

10.4 LoRA training

```
import torch
from peft import LoraConfig, get_peft_model, TaskType
from transformers import AutoTokenizer, LlamaForSequenceClassification, LlamaTokenizer, Trainer,
    TrainingArguments
from datasets import load_dataset
import os
import time
import wandb
from codecarbon import OfflineEmissionsTracker
from huggingface_hub import login
hf_token = "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"
login(token=hf_token)

run = wandb.init(project="LLaMA_LoRA_training")


os.environ['CUDA_DEVICE_ORDER'] = 'PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES'] = '0'



# Cargar el dataset TweetEval para analisis de sentimientos
dataset = load_dataset("tweet_eval", "sentiment")

# Cargar el tokenizer del modelo Llama
model_name = "meta-llama/Llama-2-7b-hf"
tokenizer = LlamaTokenizer.from_pretrained(model_name, token=hf_token)
tokenizer.pad_token = tokenizer.eos_token

# Calcular el tamao maximo de los tweets en tokens
max_length_train = max([len(tokenizer.tokenize(tweet)) for tweet in dataset['train']['text']])
max_length_validation = max([len(tokenizer.tokenize(tweet)) for tweet in dataset['validation']['text']])
max_length = max(max_length_train,max_length_validation)


# Funcion para generar tokens con el dataset
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=max_length)


# Generacion de los datasets de entrenamiento y validacion
tokenized_datasets = dataset.map(tokenize_function, batched=True)
train_dataset = tokenized_datasets["train"].remove_columns(["text"])
validation_dataset = tokenized_datasets["validation"].remove_columns(["text"])

# Carga del modelo
model = LlamaForSequenceClassification.from_pretrained(model_name, num_labels=3, token=hf_token)
model.config.pad_token_id = model.config.eos_token_id

# Configuracion LoRA
lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS, # Tarea objetivo: Sequence Classification
    r=8,                        # Dimensin low-rank (ajustable)
    lora_alpha=16,              # Factor de escalado
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj"] # Aplicar LoRA a las capas attention
)
```

```
tracker1 = OfflineEmissionsTracker(country_iso_code="ESP", allow_multiple_runs = True, output_file=
    "./emissions_LoRA.csv")
tracker1.start()
# Aplicar LoRA al modelo
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()

# Enviar modelo a la GPU
loaded = False
num_tries = 0
while not loaded:
    try:
        model.to("cuda")
        loaded = True
    except torch.OutOfMemoryError:
        num_tries += 1
        print('New try:', num_tries)
        time.sleep(5)

# Entrenamiento
training_args = TrainingArguments(
    output_dir="./results_lora",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=50,
    report_to="wandb",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    tokenizer=tokenizer
)

# Entrenar el modelo
trainer.train()

# Evaluar el modelo
trainer.evaluate()

# Grabar el modelo
trainer.save_model("./results_lora")

# Grabar el modelo LoRA
model.save_pretrained("./results_lora")

# Grabar el tokenizer
tokenizer.save_pretrained("./results_lora")

tracker1.stop()
```

Acronyms

AI	Artificial Intelligence.
ASIC	Application-Specific Integrated Circuit.
AWQ	Activation-Aware Weight Quantization.
FP16	16-bit Floating Point.
FP32	32-bit Floating Point.
GPT	Generative Pretrained Transformer.
GPTQ	Generalized Post-Training Quantization.
GPU	Graphics Processing Unit.
HQQ	Half-Quadratic Quantization.
INT4	4-bit integer.
INT8	8-bit integer.
LLM	Large Language Model.
LoRA	Low-Rank Adaptation.
LSTM	Long Short-Term Memory.
ML	Machine Learning.
MSE	Mean Squared Error.
NLP	Natural Language Processing.
PTQ	Post-Training Quantization.
QAT	Quantization Aware Training.
RNN	Recurrent Neural Network.
SGD	Stochastic Gradient Descent.
TPU	Tensor Processing Unit.