# University of Padova

Department of Information Engineering

*Master Thesis in Computer Engineering*

# Adaptation of Large Language Models to assistant chat-bots for industrial plants


*Supervisor*
Prof. Satta Giorgio
University of Padova

*Co-supervisor*
Domenico Crescenzo
Screevo

*Master Candidate*
Mohammadardalan Farhangian

*Student ID*
2013043


*Academic Year*
2023-2024

“The greatest challenge to any thinker is stating the problem in a way that will allow a solution”
— Bertrand Russell

# Abstract

Businesses that rely heavily on machinery encounter a significant obstacle in effectively managing equipment repairs while optimizing technician efficiency. The unpredictable nature of repair requests and potential miscommunications can lead to a strain on technical teams, resulting in delayed repairs and decreased production. This thesis proposes a novel solution to this problem: the development of a specialized chatbot powered by Large Language Models (LLMs) that is designed to assist both technicians and general employees with equipment troubleshooting.

The chatbot empowers employees to attempt straightforward repairs independently, reducing unnecessary escalations to technicians and allowing them to prioritize critical issues. We examine the evolution of LLMs, focusing on Mistral and Llama2, and how they can be adapted through fine-tuning. Techniques such as quantization and LoRA are explored for their potential to streamline deployment on less powerful hardware. Successful fine-tuning on a single A100 GPU demonstrates the feasibility of adapting these models to a specialized domain—in this case, equipment troubleshooting. This research demonstrates the potential of LLMs to enhance operational efficiency in manufacturing and other equipment-intensive industries.

# Contents

# Listing of figures

# Listing of tables

# Listing of acronyms

**seq2seq** ........ Sequence to Sequence

**NLP** ........... Natural Language Processing

**LSTM** ......... Long Short-Term Memory

**GRU** .......... Gated Recurrent Unit

**GPT** ........... Generative Pre-trained Transformer

**ROUGE** ....... Recall-Oriented Understudy for Gisting Evaluation,

**LLM** .......... Large Language Models

**GQA** .......... Grouped Query Attention

**RLHF** ......... Reinforcement Learning with Human Feedback

**SFT** ........... Supervides Fine-Tuning

**PPO** ........... Proximal Policy Optimization

**SWA** ........... Sliding Window Attention

**PTQ** .......... Post-Training Quantization

**QAT** ........... Quantization-Aware Training

**NF** ............ Normal Float

**DQ** ........... Double-Quantization

**LoRA** ......... Low-Rank Adaptation

**PEFT** .......... Parameter Efficiency for Fine-Tuning

# 1
# Introduction

## 1.1 Research Goal and Scope

This dissertation investigates the application of Large Language Models (LLMs) to optimize equipment repair workflows within manufacturing environments. The core challenge lies in balancing efficient repair processes with maximizing technician productivity. Fluctuating repair request volumes and miscommunication around issue severity hinder current systems, leading to production delays and inefficient technician utilization.

To address these shortcomings, we propose the development of a chatbot powered by LLMs. This chatbot aims to:

- Empower employees to independently resolve minor issues. Reserve technicians' expertise for critical repairs.
- By streamlining the repair process, this approach seeks to minimize equipment downtime and maximize technician efficiency.

The research focuses on the evolution of LLMs, particularly the 7B parameter models Mistral and Llama2. We explore techniques for creating synthetic datasets and fine-tuning these models for specific domains. Additionally, we investigate hardware optimization strategies like quantization and LoRA to enable cost-effective deployments.

The dissertation culminates in the successful fine-tuning of both LLMs on a single A100 GPU. This demonstrates the feasibility of creating domain-specific chatbots, exemplified by a chatbot designed to assist users with machinery troubleshooting.

## 1.2 Thesis Structure

The thesis is organized as follows:

- Chapter 2 discusses the development of sequence-to-sequence models, leading up to transformers, along with the metrics used for their evaluation.
- Chapter 3 delves into Large Language Models.
- Chapter 4 explores the fine-tuning of Large Language Models, including necessary steps for their implementation.
- Chapter 5 presents the outcomes obtained.
- Chapter 6 concludes the thesis and discusses potential future work.

# 2

# Background

The advent of machine learning and its application within natural language processing (NLP) has marked a new epoch in the way we enable machines to understand, interpret, and generate human language. At the heart of this transformative journey are sequence-to-sequence (Seq2Seq) models, which have significantly contributed to the advancements in the field of NLP by enabling a more nuanced and context-aware processing of language. This chapter begins by exploring the evolution of Seq2Seq models, laying the groundwork for understanding their pivotal role in the development of more complex architectures and applications in language processing.

## 2.1 Sequence to Sequence Models

Sequence-to-sequence (seq2seq) models are a key framework in deep learning designed to transform sequences of data from one form into another, a process central to various applications in natural language processing (NLP) such as translating between languages, summarizing texts, recognizing speech, and powering chatbots. At the heart of these models are two main components: the encoder and the decoder, which work together to understand and recreate sequences as illustrated in Figure 2.1.

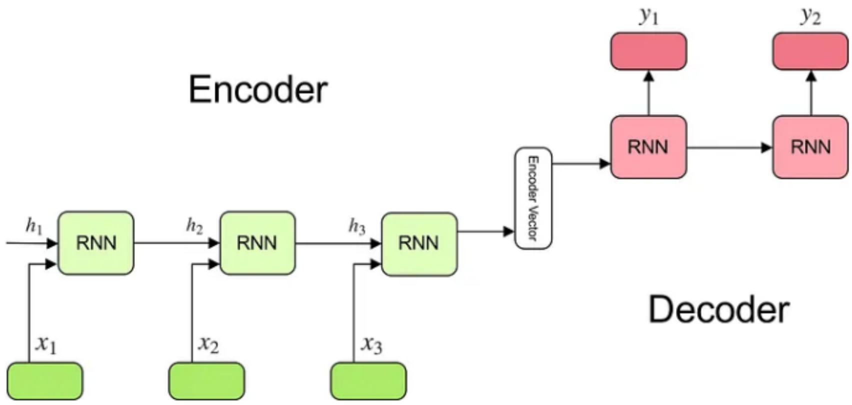



**Figure 2.1:** Encoder-decoder sequence to sequence model

**Encoder**

The encoder is composed of a series of recurrent units (utilizing LSTM or GRU cells for enhanced performance), with each unit processing a single element of the input sequence. It gathers and forwards information specific to that element. In the context of a question-answering task, the input sequence comprises all the words in the question, with each word denoted as x_i, where i indicates the word's position. [1]

The computation of hidden states h_i follows a straightforward formula typical of a standard recurrent neural network. This process involves applying designated weights to both the preceding hidden state h_(t-1) and the current input vector x_t.

$$h_t = f\left(W^{(hh)} h_{t-1} + W^{(hx)} x_t\right) \tag{2.1}$$

The encoder vector is the culminating hidden state generated by the encoder segment of the model, derived using the formula (2.1). Its purpose is to distill and convey the collective information of all input components, thereby aiding the decoder in generating precise predictions. This vector serves as the initial hidden state for the decoder segment of the model.

#### Decoder

The decoder consists of numerous recurrent units organized in a stack, with each unit tasked with predicting an output $y_t$ at each time step $t$. Each unit receives a hidden state from its predecessor and not only generates an output but also passes on a new hidden state to the next unit.

In tasks like question-answering, the sequence produced by the decoder represents the answer, with each word denoted as $y_i$, where $i$ corresponds to the position of the word within the sequence. Each hidden state $h_i$ is computed as follows, with the process described by equation (2.2).

$$h_t = f\left(W^{(hh)} h_{t-1}\right) \tag{2.2}$$

This process simply involves using the immediate previous hidden state to determine the subsequent one.

The output $y_t$ at a given time step $t$ is ascertained through the equation (2.3).

$$y_t = \mathrm{softmax}\left(W^S h_t\right) \tag{2.3}$$

Here, the current hidden state is employed alongside a specific weight $W^S$ to compute the outputs. A softmax function (2.3) then converts these into a probability distribution, which assists in pinpointing the final output, such as a word in the context of a question-answering scenario.

The aforementioned attributes of the sequence-to-sequence architecture closely align with the principles of Transformer models, which are especially renowned for their proficiency in managing sequences of varying lengths. Unlike traditional sequence processing architectures that rely on sequential data processing, Transformers utilize self-attention mechanisms. This allows them to weigh the importance of different parts of the input data without the constraint of sequential order, thus enabling parallel processing of data points and significantly improving efficiency.
In the next section, we'll explore the Transformer architecture in detail and see how it's changing the game in handling sequences of data.

### 2.1.1 Transformers

Transformers have revolutionized the landscape of natural language processing (NLP) and beyond, through a unique architecture that emphasizes parallel processing and efficiency. This chapter delves into the foundational aspects of Transformers, elucidating the intricate mechanisms and mathematical formulations that enable their unprecedented performance in various tasks.

#### The Principle of Attention

At the heart of the Transformer architecture lies the concept of attention—a mechanism that models dependencies without regard to their distance in the input or output sequences. The essence of attention within Transformers is quantified by the self-attention mechanism, which allows each position in a sequence to consider every other position in a weighted manner [2]. The self-attention for a single head is mathematically expressed in equation (2.4).

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{2.4}$$

Here, $Q$, $K$, and $V$ represent the queries, keys, and values, respectively—components derived from the input data. The term $\sqrt{d_k}$ is a scaling factor intended to stabilize gradients during training, where $d_k$ denotes the dimensionality of the keys and queries.

#### Expanding the Field of View: Multi-Head Attention

Transformers extend the concept of attention through multi-head attention, facilitating the model's ability to focus on different positions simultaneously. This is pivotal for understanding the varied and complex dependencies in data. Equation (2.5) formalizes the multi-head attention mechanism.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \tag{2.5}$$

where each head, denoted as $\text{head}_i$, captures various dimensions of the input sequence, as illustrated in equation (2.6).

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.6}$$

In this formulation, $W_i^Q$, $W_i^K$, and $W_i^V$ are projection matrices specific to each head, enabling the model to diversify its focus across different dimensions of the input data. The concatenated outputs are then linearly transformed by $W^O$, integrating the insights from each head.

**Sequential Processing without Sequences: Positional Encoding**

Given the Transformer's reliance on parallel processing, it eschews traditional sequential data processing methods (e.g., recurrence). To imbue the model with the notion of sequence order, positional encodings are introduced:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{2.7}$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{2.8}$$

Equations (2.7) and (2.8) ensure that each position in the input sequence is uniquely identifiable, allowing the model to leverage the order of elements without direct sequence-based processing.

**The Architecture: Encoders and Decoders**

The Transformer model adopts an encoder-decoder structure, where both components are comprised of multiple layers that contain the described attention mechanisms and additional neural network elements.

- **The Encoder**

  Each encoder layer is a composition of two primary sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The incorporation of positional encodings in the input enables the encoder to respect the sequence's order. The position-wise feed-forward network applies linear transformations to each position separately and identically, as described in equation (2.9).

  $$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.9}$$

  This layer acts on each position independently, further processing the information aggregated through attention.

- **The Decoder**

  Mirroring the encoder, the decoder integrates an additional sub-layer for each decoding layer, which performs multi-head attention over the encoder's output. This structure ensures a directed focus on relevant parts of the input sequence, crucial for tasks such as translation.

### Enhancing Flow: Residual Connections and Normalization

To facilitate deeper architectures, each sub-layer in the encoder and decoder incorporates residual connections followed by layer normalization. This design choice is vital for mitigating the vanishing gradient problem in deep networks, ensuring stable training and effective information flow. Figure 2.2 presents the comprehensive architecture of the Transformer model as originally depicted in the seminal paper "Attention Is All You Need."

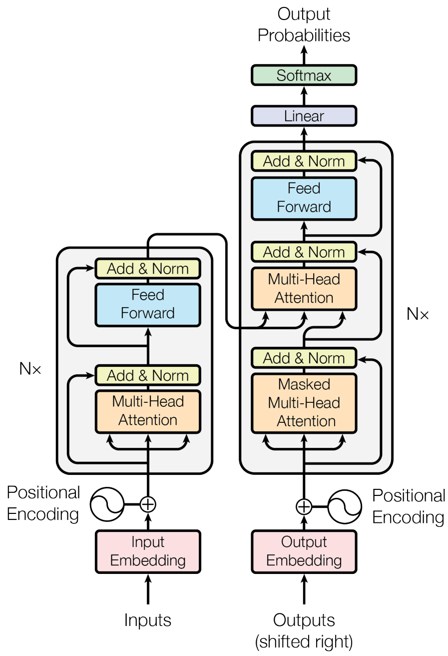



**Figure 2.2:** The Transformer - model architecture

## 2.2 End to End Models

End-to-end models in machine learning are systems designed to handle a task from its beginning to its conclusion with minimal to no intermediate steps or processing. This approach contrasts with more traditional models that might involve multiple stages, each requiring specific handling, processing, or feature engineering. End-to-end models, especially in the domain of natural language processing (NLP), have revolutionized how machines understand, interpret, and generate human language by directly mapping raw input data to desired output.

### 2.2.1 GPT

The advent of Generative Pre-trained Transformer (GPT) models has marked a significant milestone in the field of Natural Language Processing (NLP). Developed by OpenAI, GPT models are based on the transformer architecture[2], which revolutionized the way machines understand human language. The core principle behind GPT is the utilization of a large-scale transformer network pre-trained on a diverse corpus of text, which is then fine-tuned for specific tasks. This section explores the evolution of GPT models, their architecture, and their applications in various domains.

**Evolution of GPT Models**

The evolution of Generative Pre-trained Transformer (GPT) models marks a significant advancement in natural language processing technologies. The journey began with GPT-1, introduced in 2018, where OpenAI showcased the potential of transformer-based models to perform a wide array of NLP tasks with groundbreaking efficiency and coherence in text generation. The initial model leveraged the transformer architecture [2], applying it in a novel way to pre-train a language model on a diverse corpus of text before fine-tuning it on specific tasks. This approach allowed GPT-1 to achieve impressive results across various benchmarks, setting the stage for future developments in the field [3].

Building on this foundation, GPT-2 was released in 2019, introducing a model with 1.5 billion parameters, trained on an even more extensive dataset. This iteration significantly improved the model's text generation capabilities, demonstrating a remarkable ability to generate coherent and contextually relevant text over extended passages. GPT-2's performance highlighted its potential not just in generating text but also in tasks like translation, question-answering, and summarization without task-specific training. Its ability to adapt to a wide range of tasks with minimal fine-tuning exemplified a leap towards more general AI systems [4].

In 2020, GPT-3 expanded the model's parameters to an unprecedented 175 billion, further enhancing the depth and context-awareness of its output. GPT-3 introduced few-shot learning, where the model could generate accurate responses with minimal input examples. This capability demonstrated a significant leap in AI's ability to understand and generate human-like text, bridging the gap between human and machine understanding of language. The scale of GPT-3 and its sophisticated few-shot learning capabilities opened new possibilities for natural language-based applications, setting a new benchmark for what is achievable in the domain of generative AI models [5].

ChatGPT-4, while building upon the foundations laid by its predecessors, introduced refinements in accuracy, nuance, and contextual awareness. Although specific details of ChatGPT-4's architecture and training have not been disclosed in this summary, it represents the continued trajectory of innovation and improvement in the field of generative AI models. By enhancing the model's interaction capabilities, ChatGPT-4 offers more sophisticated and contextually aware conversational agents, further pushing the boundaries of what AI can achieve in understanding and generating human-like text.

#### GPT Architecture

At the heart of GPT models lies the transformer architecture which was explored thoroughly in the transformers section, designed to handle sequential data while capturing the contextual relationships within it.

While the original transformer model [2] consists of both encoder and decoder blocks, GPT models adopt a decoder-only architecture as illustrated in Figure 2.3. This means they utilize multiple layers of transformer decoder blocks, each containing masked self-attention and fully connected neural network layers. The "masked" aspect of the self-attention prevents the model from looking ahead to future tokens in the input sequence, ensuring that the prediction for each word only depends on the known preceding words. This architecture is particularly suited for tasks that involve generating text, such as language modeling and text completion.[3]

#### Applications of GPT Models

GPT models have found applications across a broad spectrum of domains, showcasing their versatility and effectiveness. Some of the notable applications include:

- **Content Creation:** GPT models are widely used in generating articles, stories, and creative content, significantly reducing the time and effort required for content creation.
- **Language Translation:** Despite being primarily designed for English, GPT models have shown promising results in translating languages, making global communication more accessible.
- **Customer Service:** Automated customer service and support chatbots powered by GPT can provide responses that are contextually relevant, improving customer experience.

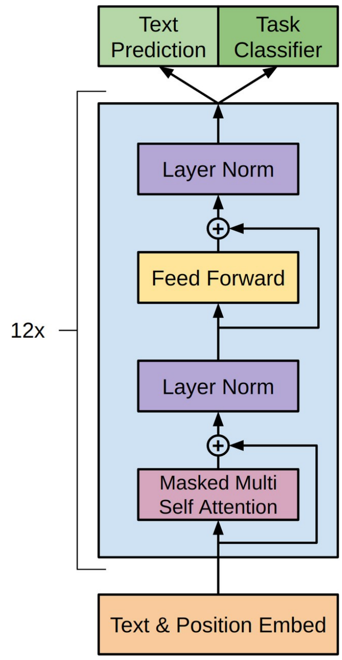



**Figure 2.3:** The GPT - model architecture

- **Educational Tools:** GPT models are employed in educational software, offering tutoring, generating practice questions, and providing explanations in various subjects.

## 2.3 Evaluation Metric

Evaluation metrics play a crucial role in the development and benchmarking of language models, especially in specialized tasks like causal language modeling (LM) for text generation. These metrics provide a quantitative way to assess the quality of generated text compared to a reference or set of references. One widely used metric in evaluating text generation tasks, including summarization, is the ROUGE score.

In the causal LM task for text generation which is our case, the model generates text based on a given context, aiming to predict the next word or sequence of words that follows logically or semantically. Evaluation metrics for this task generally fall into two categories: automatic metrics and human evaluation.

**Automatic metrics**

Automatic metrics are algorithms that compare the generated text against one or more reference texts to quantify similarity, coherence, relevance, or other desirable attributes. These include:

- **Lexical similarity metrics**Lexical similarity metrics like BLEU, ROUGE, and METEOR, which measure overlap in terms of words or n-grams between the generated text and reference texts.
- **Semantic similarity metrics** like BERTScore or Sentence-BERT, which leverage pre-trained language models to assess the semantic closeness of generated text to reference texts.
- **Other metrics** might include perplexity (a measure of how well a probability model predicts a sample), diversity metrics (to evaluate the variety in the generated text), and domain-specific metrics tailored to particular aspects of the generated content.

**Human evaluation**:

It involves human judges assessing the generated text based on criteria like fluency, coherence, relevance, and creativity. Though more subjective and resource-intensive, human evaluations can capture nuances that automatic metrics might miss.

### 2.3.1 Rouge Score

ROUGE, which stands for Recall-Oriented Understudy for Gisting Evaluation, is a set of metrics designed to evaluate the quality of summaries by comparing them to one or more reference summaries. ROUGE is particularly useful in tasks like summarization but is also applicable to any text generation task where comparison to a reference is meaningful.[6] The core idea is to measure the overlap between the generated text and the reference texts to assess how well the important aspects have been captured. Key variants include:

- **ROUGE-N:** Measures the overlap of N-grams between the generated text and the reference texts. ROUGE-1 and ROUGE-2 are the most common, focusing on unigrams and bigrams, respectively. It evaluates the precision (the fraction of N-grams in the generated summary that are also in the reference summary), recall (the fraction of N-grams in the reference summary that are also in the generated summary), and F1 score (a harmonic mean of precision and recall).

- **ROUGE-L:** Focuses on the longest common subsequence (LCS) between the generated text and reference texts. It is useful for evaluating sentence-level structure similarity and does not require fixed-length N-grams. It also calculates precision, recall, and F1 score, taking into account the longest shared sequences.

These ROUGE variants provides a different lens through which to evaluate the text, offering insights into aspects like the precision of word choice, the fluency and structure of sentences, and the ability to maintain relevant content. For causal LLM tasks, where the understanding and generation of logically coherent and contextually relevant text are paramount, a combination of these ROUGE scores can offer a comprehensive evaluation of model performance.

# 3

# Large Language Models (LLM)

In the evolving landscape of artificial intelligence (AI) and natural language processing (NLP), Large Language Models (LLMs) have emerged as pivotal elements, showcasing profound capabilities in understanding, generating, and interacting with human language. These models, through their extensive training on diverse and voluminous datasets, have developed an ability to process and produce text in manners closely mirroring human linguistic behavior.

LLMs stand out due to their considerable scale, both in terms of the data they are trained on and their architectural complexity. This scale is not merely a matter of size but is central to their ability to discern and replicate nuanced language patterns and contexts. The pre-training and fine-tuning process that these models undergo equips them with a versatile skill set, enabling them to adapt their learned knowledge to specific tasks or domains with remarkable efficiency.

Moreover, LLMs are recognized for their generative capabilities, where they can produce coherent and contextually relevant text based on given prompts. This feature is particularly valuable for applications ranging from automated content creation to sophisticated conversational interfaces. Their adaptability across languages and domains further broadens their applicability, making them invaluable assets across various fields and industries.

This chapter provides an overview of two distinguished 7B parameter open-source models, Llama2 and Mistral.

## 3.1 LLama2-MetaAI

The advent of Llama2 by Meta AI marks a significant milestone in the field of artificial intelligence, specifically in the development of Large Language Models (LLMs). Through a comprehensive pretraining and fine-tuning process, Llama2 emerges as a robust, versatile model capable of performing a wide array of tasks, from chat applications to complex reasoning. This section explains the development processes behind Llama2, emphasizing its pretraining strategy, fine-tuning methodologies, and the paramount importance of safety measures.

### 3.1.1 Pretraining Llama2

The foundation of Llama2's capabilities is laid during its pretraining phase, a rigorous process aimed at enhancing the model's performance and scalability. As depicted in Figure 3.1, the pretraining of Llama2 leverages publicly available online sources, forming a diverse and rich dataset. This initial stage is crucial for acquiring a broad knowledge base and understanding natural language intricacies.[7]

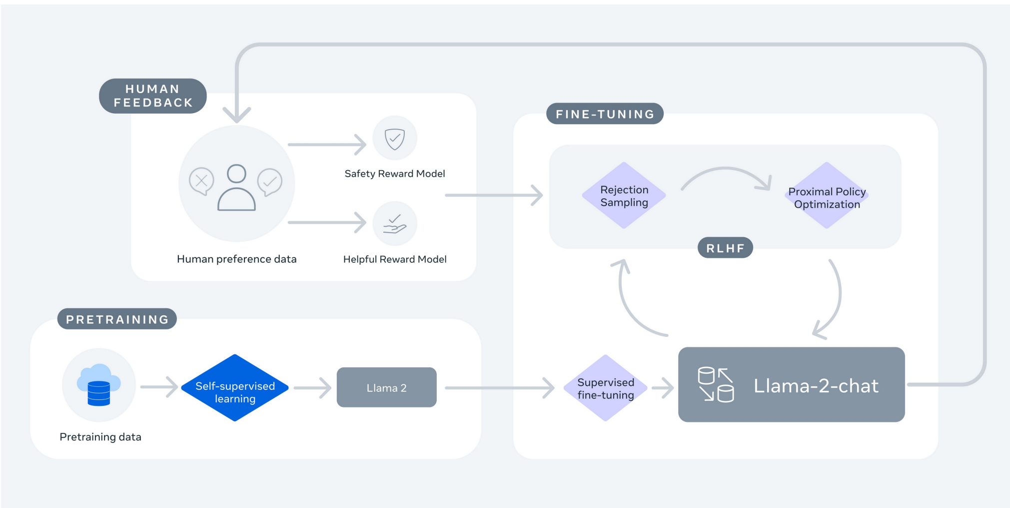



**Figure 3.1:** Training of LLama2-Chat

### 3.1.2 Data Enhancement and Model Architecture

Significant strides in data processing and architectural improvements distinguish Llama2 from its predecessors. Figure 3.2 outlines the comparative attributes of Llama2 against Llama1, showcasing advancements in parameters, context length, and the adoption of Grouped-Query Attention (GQA) for improved inference scalability. Specifically, Llama2's architecture incorporates up to 70 billion parameters and extends the context length to 4,000 tokens, facilitating a deeper understanding of complex queries.

| | Training Data | Params | Context Length | GQA | Tokens | LR |
|---|---|---|---|---|---|---|
| Llama 1 | *See Touvron et al. (2023)* | 7B | 2k | ✗ | 1.0T | $3.0 \times 10^{-4}$ |
| | | 13B | 2k | ✗ | 1.0T | $3.0 \times 10^{-4}$ |
| | | 33B | 2k | ✗ | 1.4T | $1.5 \times 10^{-4}$ |
| | | 65B | 2k | ✗ | 1.4T | $1.5 \times 10^{-4}$ |
| Llama 2 | *A new mix of publicly available online data* | 7B | 4k | ✗ | 2.0T | $3.0 \times 10^{-4}$ |
| | | 13B | 4k | ✗ | 2.0T | $3.0 \times 10^{-4}$ |
| | | 34B | 4k | ✓ | 2.0T | $1.5 \times 10^{-4}$ |
| | | 70B | 4k | ✓ | 2.0T | $1.5 \times 10^{-4}$ |

**Figure 3.2:** LLama2 Family of Models

### 3.1.3 Training Details

The training process of Llama2, illustrated in Figure 3.3, highlights the model's loss reduction over time, indicating effective learning and adaptation. The model employs the AdamW optimizer and a sophisticated learning rate schedule, contributing to its robust performance across various benchmarks. The transition from Llama1 to Llama2 involves substantial enhancements, including a 40% increase in the pretraining corpus and the integration of GQA, as detailed in Figure 3.2. These methodological refinements are pivotal in optimizing the model's capability to process and generate natural language.

**Fine-tuning Llama 2:**
Beyond pretraining, Llama2 undergoes fine-tuning to tailor its capabilities for specific applications, notably in dialogue systems through Llama2-Chat. This phase encompasses Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF), methods that align the model's outputs with human preferences and instructions.

### 3.1.4 Supervised Fine-Tuning (SFT)

The SFT process benefits significantly from high-quality, annotated data, enabling Llama2-Chat to excel in dialogue-based tasks. The emphasis on diverse and accurate annotations ensures the model's responses are both relevant and safe, adhering to predefined guidelines.

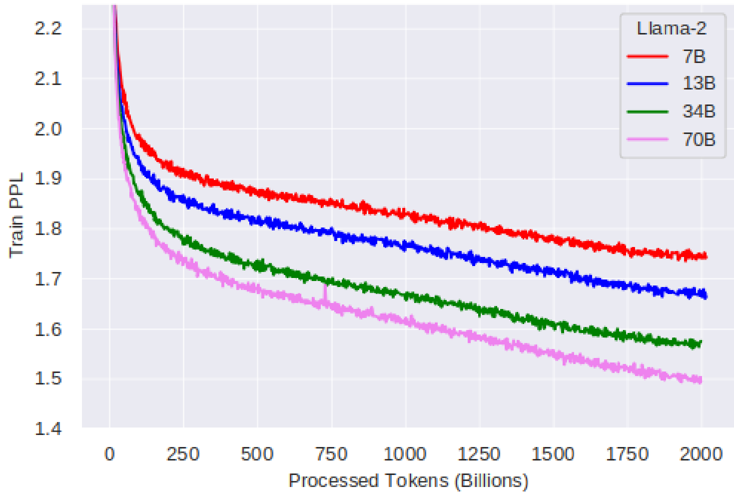



**Figure 3.3:** Training Loss for LLama2 Models

### 3.1.5 Reinforcement Learning with Human Feedback (RLHF)

RLHF further refines Llama2-Chat's performance, utilizing human feedback to guide the model towards generating more preferred responses. This iterative process, coupled with advanced techniques like Proximal Policy Optimization (PPO), solidifies Llama2-Chat's alignment with human expectations, enhancing its helpfulness and safety in interactive applications.

**Safety Measures**

Ensuring the safety of LLM outputs is paramount. Llama2 integrates comprehensive safety protocols throughout its development stages, from pretraining data selection to fine-tuning processes. The model undergoes extensive safety evaluations to mitigate potential risks and biases, ensuring its deployment is both responsible and secure.

## 3.2 Mistral-MistralAI

Mistral 7B represents a breakthrough in the landscape of Large Language Models (LLMs), designed with an emphasis on both performance and efficiency. This model introduces an innovative use of Grouped-Query Attention (GQA) and Sliding Window Attention (SWA) mechanisms. GQA enhances the model's inference speed, while SWA enables the handling of sequences of varying lengths more efficiently, thereby reducing overall inference costs.[8]

In comparative evaluations, Mistral 7B demonstrates its superiority by outperforming the leading open-source 13-billion-parameter model, Llama2, across a broad spectrum of benchmarks. Furthermore, it surpasses the capabilities of the previously best-released 34-billion-parameter model, Llama1, particularly in tasks demanding high levels of reasoning, mathematical problem-solving, and code generation.

An additional variant of Mistral 7B, known as Mistral 7B — Instruct, has been fine-tuned specifically for instruction-following tasks using datasets readily available through the Hugging Face repository. This specialized version of Mistral 7B achieves remarkable performance, exceeding that of the Llama2 13B — chat model in both human-evaluated and automated benchmarks. This demonstrates Mistral 7B's versatility and its advanced capacity for understanding and executing task-specific instructions.

### 3.2.1 Architecture

Mistral 7B introduces significant architectural innovations that enhance its performance and efficiency, setting it apart from previous language models. At the heart of these advancements lie two principal mechanisms: Sliding Window Attention (SWA) and Grouped-Query Attention (GQA). These are complemented by a strategic implementation of a rolling buffer cache and further enhanced by sophisticated methods such as pre-fill and chunking. These elements are meticulously designed to optimize processing speed, memory usage, and the model's ability to manage long sequences effectively. The upcoming segments provide a detailed examination of these components.

- **Sliding Window Attention (SWA)**

  At the heart of Mistral 7B's architectural advancements lies the Sliding Window Attention mechanism. Unlike traditional (vanilla) attention mechanisms that often scale quadratically with the sequence length, SWA limits each token's attention span to a fixed window size (W), thereby reducing computational complexity. This approach exploits the stacked layers of a transformer, allowing each token to attend not just to its immediate neighbors but also to tokens beyond the immediate window, through recursive attention across layers. As a result, information can propagate through the model efficiently, enabling the processing of longer sequences at a reduced computational cost. Figure 3.4 depicts the contrast between Vanilla Attention and Sliding Window Attention mechanisms.

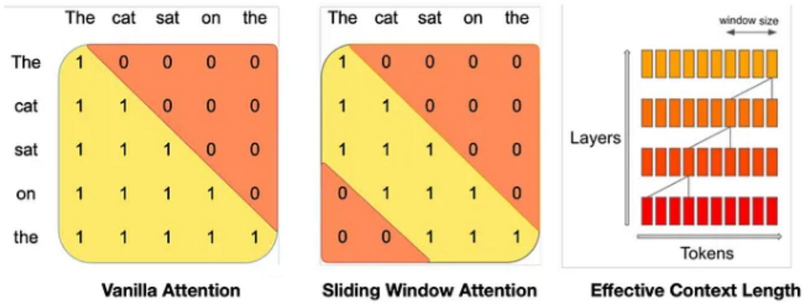



**Figure 3.4:** Comparative Visualization of Attention Mechanisms

The implementation of SWA in Mistral 7B achieves a theoretical attention span of approximately 131K tokens for a sequence length of 16K and a window size of W=4096. This innovative approach yields a 2x speed improvement over baseline models without SWA, highlighting its efficacy in enhancing processing speed while maintaining model quality.

- **Grouped-Query Attention (GQA)**

  Grouped-Query Attention further accelerates inference by reducing the memory requirements during decoding. This mechanism allows for larger batch sizes and higher throughput, which is crucial for real-time applications. By grouping queries, GQA minimizes the redundancy in attention calculations, significantly speeding up the inference process without a loss in performance. This makes Mistral 7B particularly suited for applications requiring fast response times. Figure 3.5 illustrates the structural differences between Multi-Head and Grouped-Query attention mechanisms.

- **Rolling Buffer Cache**

  To optimize memory usage, Mistral 7B employs a rolling buffer cache for its fixed attention span. This cache, with a size equal to the window (W), stores keys and values for each timestep in a position determined by the timestep modulo W. As the model progresses through a sequence, older values are overwritten by newer ones once the position exceeds W, keeping the cache size constant. This ingenious caching strategy reduces memory usage by up to 8x compared to traditional methods, without degrading model quality. Figure 3.6 visualizes the progressive functioning of the rolling buffer cache mechanism.

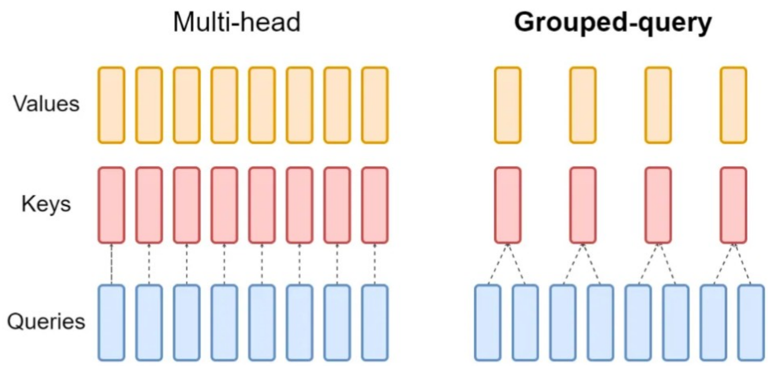



**Figure 3.5:** Multi-Head vs. Grouped-Query Attention Structures

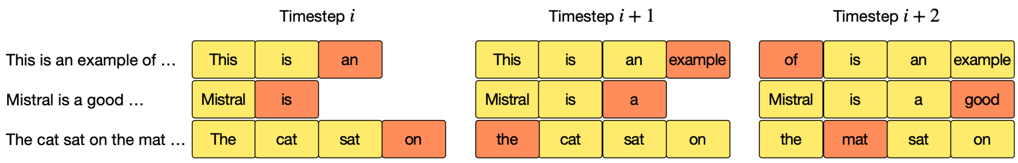



**Figure 3.6:** Rolling Buffer Cache Mechanism

- **Pre-fill and Chunking Techniques**

  In addition to these core architectural features, Mistral 7B incorporates pre-fill and chunking techniques to further enhance its efficiency. By pre-filling the attention cache with known prompts and chunking large prompts into smaller pieces, the model can compute attention more effectively. This strategy reduces memory usage during sequence generation, enabling the model to handle large prompts with greater efficiency. Figure 3.7 shows the pre-fill and chunking mechanism as described in the Mistral paper, segmenting the processing of tokens into past, cache, and current stages

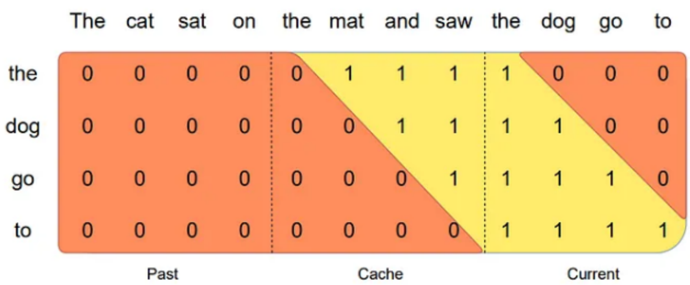



**Figure 3.7:** Depiction of Pre-fill and Chunking

These architectural innovations collectively empower Mistral 7B to deliver exceptional performance and efficiency. By addressing key limitations of previous models related to processing speed, memory usage, and the handling of long sequences, Mistral 7B sets a new benchmark for large language model design.

### 3.2.2 Results

The evaluation of the Mistral 7B model presents a comprehensive comparison against the existing benchmarks set by Llama 2 (13B) and Llama 1 (34B), as visualized in Figure 3.8 of the Mistral paper. Mistral 7B has been subjected to an extensive range of NLP tasks to scrutinize its performance thoroughly, with the detailed outcomes documented in table 3.1. The benchmarks cover a spectrum of tasks which include:

- Commonsense Reasoning (0-shot) as evaluated by Hellaswag, Winogrande, PIQA, SIQA,
- OpenbookQA, ARC-Easy, ARC-Challenge, and CommonsenseQA.
- World Knowledge (5-shot), assessed through NaturalQuestions and TriviaQA.
- Reading Comprehension (0-shot) gauged by BoolQ and QuAC.
- Mathematical problem solving observed through GSM8K (8-shot) and MATH (4-shot).
- Code-related tasks examined by Humaneval (0-shot) and MBPP (3-shot).

| **Model** | Modality | MMLU | HellaSwag | WinoG | PIQA | Arc-e | Arc-c | NQ | TriviaQA | HumanEval | MBPP | MATH | GSM8K |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LLaMA2 7B | Pretrained | 44.4% | 77.1% | 69.5% | 77.9% | 68.7% | 43.2% | 24.7% | 63.8% | 11.6% | 26.1% | 3.9% | 16.0% |
| LLaMA2 13B | Pretrained | 55.6% | **80.7%** | 72.9% | 80.8% | 75.2% | 48.8% | **29.0%** | **69.6%** | 18.9% | 35.4% | 6.0% | 34.3% |
| Code-Llama 7B | Finetuned | 36.9% | 62.9% | 62.3% | 72.8% | 59.4% | 34.5% | 11.0% | 34.9% | **31.1%** | **52.5%** | 5.2% | 20.8% |
| Mistral 7B | Pretrained | **60.1%** | **81.3%** | **75.3%** | **83.0%** | **80.0%** | **55.5%** | **28.8%** | **69.9%** | 30.5% | 47.5% | **13.1%** | **52.2%** |

**Table 3.1:** Performance Metrics of Mistral 7B Compared to Llama Models

In these benchmarks, illustrated in Figure 3.8, Mistral 7B's performance was stellar, outshining Llama2 (13B) across all metrics and surpassing Llama1 (34B) on most of the benchmarks, especially in areas requiring strong reasoning, mathematical prowess, and code generation capabilities. Specifically, in mathematics and code-related benchmarks, as detailed in table 3.1, Mistral 7B showcased superior performance, asserting its adeptness in handling complex problem-solving tasks.
Furthermore, Mistral 7B was evaluated on popular aggregated results, such as MMLU, BBH, and AGI Eval, where it continued to demonstrate its robust capabilities. The performance leap

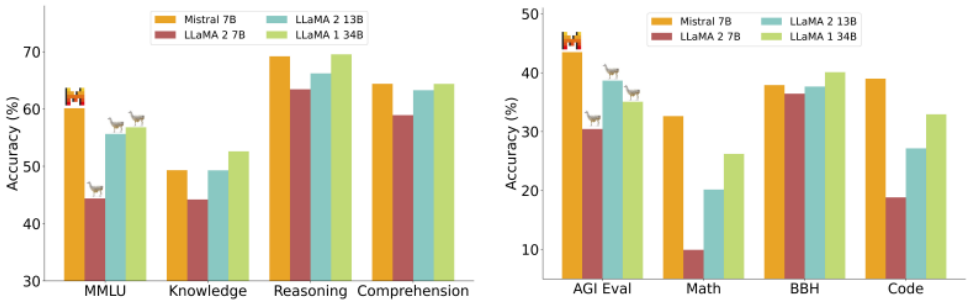



**Figure 3.8:** Evaluation of Mistral 7B Against Various Llama Models Across a Broad Spectrum of Benchmarks

of Mistral 7B indicates that it achieves what could be expected from a model more than three times its size when assessed for reasoning comprehension and STEM reasoning, particularly in the MMLU benchmark. However, in Knowledge benchmarks, Mistral 7B achieved a lower compression rate, which the authors suggest is likely due to its limited parameter count impacting the model's capacity to store knowledge.

These results provide concrete evidence of Mistral 7B's exceptional performance, proving it to be a formidable contender in the realm of LLMs. The findings from these evaluations solidify the claim that Mistral 7B has successfully integrated efficiency with high performance, setting a new benchmark for future developments in language models.

# 4

# Fine Tuning Large Language Models

## 4.1 Problem Statement

Businesses reliant on machinery and tools face a critical challenge in balancing efficient equipment repair with maximizing technician productivity. While a skilled technical team is crucial for addressing equipment malfunctions and maintaining seamless production flow, managing the influx of repair requests can be problematic.

This inefficiency stems from two key issues:

- **Fluctuating Demand:** The volume of repair requests can vary significantly, making it difficult for technical teams to consistently meet demand. This can lead to backlogs and delays, ultimately impacting production schedules.
- **Miscommunication and Inappropriate Requests:** In some cases, human error or lack of clarity in reporting issues can lead to misunderstandings about the severity of problems. Additionally, minor malfunctions that employees could potentially resolve themselves might be unnecessarily escalated to the technical team. This diverts technicians' attention away from critical repairs and creates unnecessary wait times for employees with legitimate repair needs.

The consequence of this imbalanced system is a reduction in overall operational efficiency. Valuable technician time is wasted addressing minor issues, while more critical repairs experience delays.

## 4.2 Data Preparation

To address these challenges, we propose an innovative solution: the development of a chatbot designed to assist technicians using LLMs. This digital assistant aims to empower employees to solve straightforward problems on their own, reserving technicians' time for emergencies or more complex issues that cannot be easily resolved. This approach seeks to streamline the repair process, reducing downtime and ensuring that technicians can focus on the most urgent and challenging tasks. Consequently, to develop the assistant chatbot using Large Language Models (LLMs), the initial step involves sourcing a dataset for fine-tuning the LLM. The dataset provided by the company encompasses a comprehensive record spanning almost 10 years, detailing the maintenance of machines and tools. To utilize this dataset effectively, the first crucial step is to undertake data preprocessing.

### 4.2.1 Data Preprocessing

Data preprocessing serves as the foundational process for cleansing and organizing raw data into a structured format suitable for fine-tuning. This stage encompasses a series of methodical steps designed to enhance the quality and efficiency of the data before fine-tuning. Since the dataset was originally in Italian, our first step was converting the entire dataset to English through automated translation. After that, we moved on to cleaning the data, which involved removing any duplicates and outliers. Next, we addressed the missing values by getting rid of any records or features that were incomplete. Following this, we dropped the features that were not very important for our situation and kept only the key features. After completing these steps, We concluded with a cleaned dataset comprising close to 6,000 entries, totaling approximately 1,300,000 words. The main features included the ticket title, type of machine, the condition it was found in, the work performed on it, and the hours it took to resolve the issue.

### 4.2.2 Synthetic Dataset

To fine-tune the LLM, we need a dataset consisting of prompt and answer pairs for each row. Here, the prompt is a question asked by the user, and the answer is what the assistant chatbot would reply with, addressing the issue mentioned in the prompt. Our existing dataset was not initially set up this way, so we tackled this challenge by employing prompt engineering through

ChatGPT-4 to generate synthetic data suited to our needs. To achieve this, we tested over 100 prompts and identified the 4 best versions to use in ChatGPT-4. We then selected 20 random records from our dataset and applied each of the four chosen prompts to them separately, generating a total of 80 samples. We evaluated these samples to determine the best prompt, focusing on criteria such as minimal hallucination, relevance, and conciseness in comparison to the original dataset. Figure 4.1 is the most effective prompt we have obtained to date.

**Prompt:**

As a proficient prompt engineer, craft a prompt using only the details outlined in points 1 and 2. The goal is to create a concise and precise request that mimics a user seeking guidance from a tech-assistant chat-bot. Exclude any greetings or expressions of gratitude in the prompt.

1.Problem statement: []

2.Machine Type: []

Then, As an expert technician, leverage the details specified in points A, B, and C to craft a thorough solution for the issue at hand. Offer precise instructions on resolving the problem, excluding any greetings or expressions of gratitude.

A. Cause of problem: []

B. Solution: []

C. The duration required to resolve the problem in hours: []

**Figure 4.1:** The Optimal Prompt Achieved for Generating Synthetic Data

### 4.2.3 Data Generation

To create the synthetic dataset, we utilized Python to access the ChatGPT-4 API. This allowed us to input the necessary features of each row to generate a corresponding prompt and its related answer. As a result, our final dataset comprises nearly 6,000 rows of prompt and answer pairs.

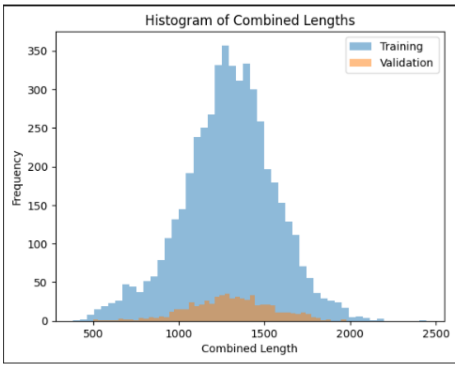
### 4.2.4 Data split

To ensure the prompts and answers have a similar length distribution, we first counted all the characters in both the prompt and the answer for each row and added a new column named "combined-length". After that, we arranged the entire dataset based on the lengths of these inputs. Following this organization, we used scikit-learn to randomly divide the dataset into three separate sets: training, validation, and test sets. The ultimate structure of the dataset, along with the histogram and box plot for the combined length, are illustrated in Figures 4.2 and 4.3, respectively.:

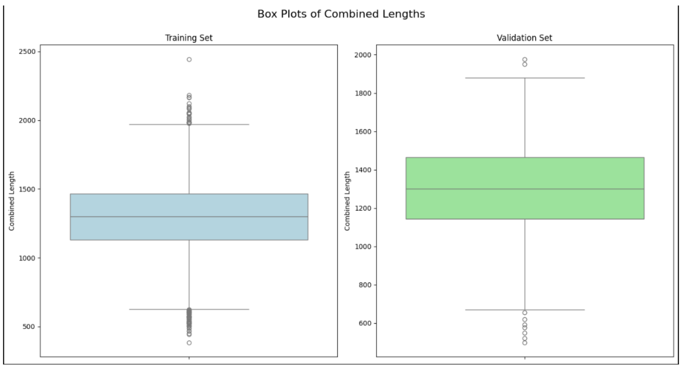
**Data Structure:**

```
DatasetDict({
    train: Dataset({
        features: ['Prompt', 'Answer', 'combined_length'],
        num_rows: 5006
    })
    validation: Dataset({
        features: ['Prompt', 'Answer', 'combined_length'],
        num_rows: 626
    })
    test: Dataset({
        features: ['Prompt', 'Answer', 'combined_length'],
        num_rows: 626
    })
})
```

**Figure 4.2:** Dataset structure




**(a)** Histogram




**(b)** Box Plot

**Figure 4.3:** Histogram and Box Plot Representation of Generated Data Lengths

## 4.3 Fine Tuning Llama2

Fine-tuning Large Language Models (LLMs) constitutes a critical methodology in the enhancement of pre-trained models for specialized applications, including the development of assistant chatbots for technical support. This process involves adapting a general-purpose LLM, which has been pre-trained on extensive corpora of text, to perform specific tasks with higher precision, such as providing users with guidance on troubleshooting machinery.

The fine-tuning approach leverages the broad foundational knowledge acquired by the LLM during its initial training phase, applying it to a more narrowly defined problem space. By introducing a dataset relevant to the specific task - in this case, machine repair and maintenance - the model is re-trained to adjust its parameters in a way that optimizes its performance for this application. This involves exposing the model to a variety of queries and issues commonly encountered by users in the context of machinery malfunction, alongside appropriate responses or solutions.

Upon successful fine-tuning, the LLM can be deployed as an assistant chatbot, offering an interactive platform for users to receive targeted advice on diagnosing and rectifying issues with their equipment. This application of LLMs bridges the gap between the vast, generalized understanding inherent in these models and the specific, practical knowledge required for effective machine maintenance support.

Here are the required steps for fine tuning the LLM:

**Model Quantization: Simplifying Large Language Model Deployment**
Prior to the fine-tuning of any Large Language Model (LLM), it is imperative to address the challenges posed by the substantial size and computational demands of these models. The escalating complexity and parameter count of LLMs necessitate solutions that mitigate storage and computational intensity without significantly compromising information fidelity. An effective strategy identified to address these challenges is model quantization, a process aimed

at optimizing the storage and computational efficiency of neural networks, including LLMs, through precision modification of model weights.

Quantization is a process that involves the conversion of continuous, infinite values into a constrained set of discrete, finite values. Within the domain of LLMs, this process specifically entails the transition of model weights from higher precision data types to those of lower precision. This adjustment is crucial for managing the expansive data structures, known as Tensors, that underpin neural networks. Tensors, which are multi-dimensional matrices populated with numerical values, are traditionally stored in high-precision formats such as 32-bit (single precision) or 64-bit (double precision) floating-point numbers. Although high precision is synonymous with enhanced accuracy and stability in model training, it is also associated with increased computational expense and hardware requirements. The premise of adopting lower precision is founded on the observation that the full range of 64-bit floating-point representation is not always necessary for maintaining optimal neural network performance. Figure 4.4 is an illustration of the bfloat16 numerical format, by Google[9]:

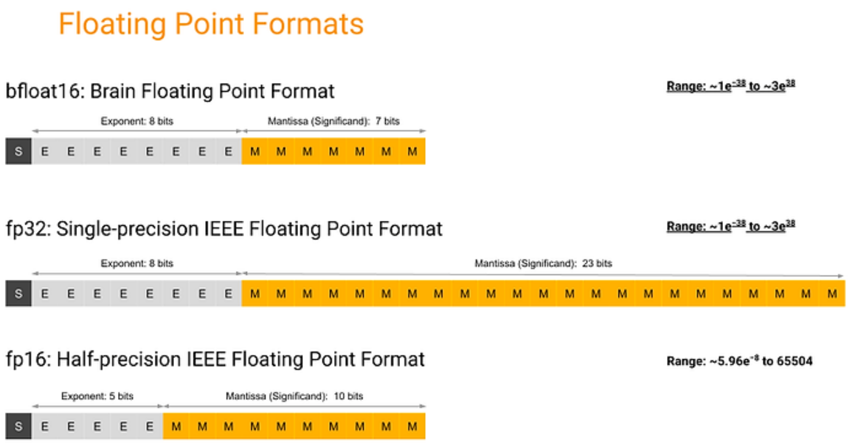



**Figure 4.4:** The bfloat16 numerical format | Google

Through the reduction of the bit requirement for each weight, quantization significantly diminishes the overall size of the model. A typical example of this reduction involves the conversion of weights from 16-bit floating-point (FP16) to 4-bit integer (INT4) formats, facilitating the use of less expensive hardware and enhancing operational speed. However, this reduction in weight precision may impact the overall quality of the LLM to some extent. Research indi-

cates that the extent of this impact is contingent upon the quantization techniques employed, with larger models exhibiting a higher resilience to precision changes. Consequently, for models with parameters exceeding approximately 70 billion, a 4-bit representation is often deemed an optimal compromise between performance and efficiency, while smaller models might benefit more from 6 or 8-bit representations.
Here, we delve into the quantization techniques themselves.

**Quantization Techniques:**
The methodology for implementing model quantization can be categorized into two distinct approaches:

- **Post-Training Quantization (PTQ)**: This approach involves the conversion of an already trained model's weights to a lower precision without necessitating further training. Although PTQ is relatively straightforward and easy to implement, it may result in a slight degradation of model performance due to the loss of precision in weight values.
- **Quantization-Aware Training (QAT)**: In contrast to PTQ, QAT integrates the weight conversion process within the training phase, often yielding superior model performance albeit at a higher computational cost. A prominent technique within QAT is QloRA [10].

Building upon these foundational techniques, we witness advancements in the field.
**Advancements in Model Quantization:**
The field of model quantization has witnessed the emergence of several state-of-the-art methods, including [11]:

- **GPTQ**: Primarily focused on GPU execution, this method includes variants such as AutoGPTQ, ExLlama, and GPTQ-for-LLaMa.
- **NF4**: Integrated within the bitsandbytes library and closely aligned with the Hugging Face transformers library, NF4 is predominantly utilized in conjunction with QLoRA methods for fine-tuning models in 4-bit precision.
- **GGML**: A C library that interfaces with the llama.cpp library, featuring a unique binary format for LLMs that supports rapid loading and ease of readability. Its recent adoption of the GGUF format enhances future extensibility and compatibility.

For the project at hand, the NF4 method was selected due to its efficacy in balancing performance with model size and speed. The NormalFloat (NF) data type, an improvement upon the Quantile Quantization technique, along with Double-Quantization (DQ), enables higher compression rates while preserving performance integrity. Notably, the integration of bitsandbytes, incorporating insights from the QLoRA paper, demonstrates minimal performance reduction with 4-bit quantization during both inference and training phases of LLMs. In Table 4.1 below, you will find the parameters for model quantization and the values we have established for this process.

| **Parameters** | **Value** |
|---|---|
| `load_in_4bit` | True |
| `bnb_4bit_quant_type` | "nf4" |
| `bnb_4bit_use_double_quant` | True |
| `bnb_4bit_compute_dtype` | bfloat16 |

**Table 4.1:** Quantization Parameters

### LoRA: Low-Rank Adaptation of Large Language Models:

Following the exploration of quantization as a means to reduce model size and computational demands, we now turn our attention to another innovative approach aimed at addressing the challenges inherent in the finetuning of colossal models such as GPT-3, which boasts 175 billion parameters. This approach, known as Low-Rank Adaptation (LoRA), offers a complementary strategy to quantization by targeting the model's parameter efficiency directly, rather than its numeric precision.

Low-Rank Adaptation emerges as a response to the limitations of traditional finetuning methods, which require updating the entirety of a pre-trained model's parameters for task-specific adaptation. Inspired by insights into the low intrinsic dimensionality of over-parametrized models, LoRA proposes a novel method of model adaptation that significantly reduces the number of actively updated parameters, thus easing the computational and storage burdens associated with deploying large-scale NLP models.

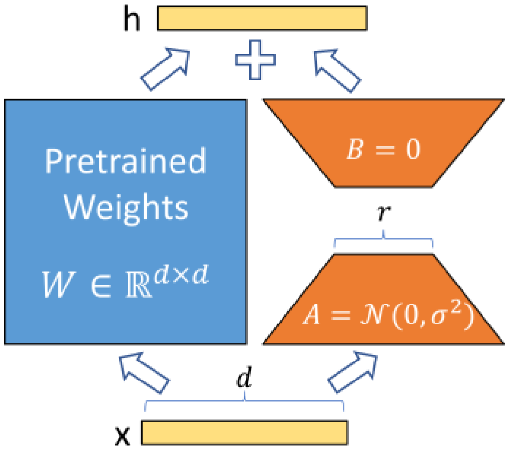



**Figure 4.5:** forward pass using low-rank decomposition.

Figure 4.5 illustrates the core concept behind LoRA, demonstrating how it applies to the dense layers of neural networks, particularly within Transformer models. By adopting a low-rank parametrized update mechanism, LoRA effectively maintains the model's original performance while drastically lowering the parameter count required for adaptation. [12]

### The Mechanism of Low-Rank Adaptation

LoRA's strategy involves the ingenious representation of updates to a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$ through the multiplication of two lower-rank matrices $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$. By constraining these updates to a low-rank format, $W_0 + \Delta W = W_0 + BA$, LoRA encapsulates essential changes during the adaptation phase with a fraction of the computational overhead of traditional methods, creating a synergistic relationship with quantization techniques that further enhance model efficiency.

### Advantages of Low-Rank Adaptation

The adoption of LoRA offers several compelling benefits, especially when considered alongside quantization:

- **Complementary Efficiency**:

  While quantization reduces the size and increases the speed of models by lowering the precision of calculations, LoRA reduces the number of parameters that need to be updated and stored, offering a dual pathway to efficiency.

- **Dynamic Adaptation:**

  LoRA facilitates the swift customization of models to new tasks with minimal parameter updates, a feature that complements the static benefits of quantization by introducing dynamic efficiency in model deployment.

- **Enhanced Model Deployment**:

  The combination of quantization and LoRA enables more versatile and resource-efficient deployment of NLP models, making state-of-the-art performance more accessible across varied computational environments.

**Applying LoRA in the Context of Transformers:**

LoRA's application is particularly impactful within the Transformer architecture, where it can be used to adapt specific weight matrices involved in the model's attention mechanism. This targeted adaptation allows for the preservation or enhancement of model performance with minimal additional parameters, presenting a perfect complement to the reduction in numeric precision achieved through quantization.

**Hyper parameters:**

| **Parameters** | **Value** |
| --- | --- |
| `lora_rank` | 16 |
| `lora_alpha` | 8 |
| `lora_dropout` | 0.1 |
| `target_modules` | "q_proj", "k_proj", "v_proj", "o_proj", "lm_head" |
| `task_type` | "CAUSAL_LM" |

**Table 4.2:** LoRA Configuration Parameters for Llama2

For Integrating LoRA into our model, we used the `LoraConfig` class from the PEFT (Parameter Efficiency for Fine-Tuning) library by Hugging Face. Table 4.2 presents the parameters utilized to configure LoRa and the optimal values identified through evaluating different combinations based on their impact on training loss convergence. Below, a concise overview of each parameter is provided.

**Hyperparameters of LoraConfig:**

- **lora_rank**: This parameter specifies the rank of the low-rank matrices $A$ and $B$ used in LoRA's adaptation mechanism. A lower rank $r$ means fewer parameters to update during the adaptation process, which can lead to faster training and less memory usage. For instance, $r = 16$ indicates that both $A$ and $B$ matrices in the LoRA adaptation will have a rank of 16. This is a crucial hyperparameter as it directly influences the model's balance between adaptability and computational efficiency.

- **lora_alpha**: The *lora_alpha* parameter is a scaling factor applied to the low-rank updates during the adaptation process. Essentially, it controls the magnitude of the updates applied to the pre-trained weight matrices, thus influencing how much the adapted model deviates from its pre-trained state. A higher *lora_alpha* value would allow for larger updates, which could be beneficial for adapting to tasks that are significantly different from the pre-training tasks. $lora\_alpha = 8$ can be a moderate scale for updates.

- **lora_dropout**: This parameter indicates the dropout rate applied to the LoRA updates. Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of the input units to 0 at each update during training. $lora\_dropout = 0.1$ means that 10% of the elements in the LoRA updates will be randomly zeroed out, introducing regularization to the adaptation process.

- **bias**: The bias parameter specifies how biases should be handled during the LoRA adaptation. The `bias="none"`indicates that biases are not adapted as part of the LoRA process. This decision might be based on the desire to keep the adaptation process focused on the weight matrices, potentially simplifying the adaptation and reducing the number of parameters that need to be tuned.

- **target_modules**: This is a list of model components that will be adapted using LoRA. The components typically correspond to parts of the model's architecture involved in specific computational tasks, such as projecting input features (*q_proj*, *k_proj*, *v_proj*, *o_proj*) and generating output predictions (*lm_head*). By targeting these modules, the adaptation focuses on the most relevant aspects of the model for the given task, potentially improving performance on downstream tasks without the need to adapt the entire model.

- **task_type**: Specifies the type of task for which the model is being adapted. The `"CAUSAL_LM"` indicates that the adaptation is intended for a causal language modeling task, where the goal is to predict the next token in a sequence given its predecessors. This information can be used to optimize the adaptation process for the specific characteristics and requirements of the task at hand.

Figure 4.6 and 4.7 illustrate the structure of the Llama 2 model prior to and following the application of the LoRA configuration respectively.

```
LlamaForCausalLM(
  (model): LlamaModel(
        (embed_tokens): Embedding(32000, 4096)
        (layers): ModuleList(
              (0-31): 32 x LlamaDecoderLayer(
                    (self_attn): LlamaAttention(
                          (q_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                          (k_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                          (v_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                          (o_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
                          (rotary_emb): LlamaRotaryEmbedding() )
                    (mlp): LlamaMLP(
                          (gate_proj): Linear4bit(in_features=4096, out_features=11008, bias=False)
                          (up_proj): Linear4bit(in_features=4096, out_features=11008, bias=False)
                          (down_proj): Linear4bit(in_features=11008, out_features=4096, bias=False)
                          (act_fn): SiLU() )
                    (input_layernorm): LlamaRMSNorm()
                    (post_attention_layernorm): LlamaRMSNorm()
              )
        )
        (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=32000, bias=False)
)
```

**Figure 4.6:** Llama2 structure prior to the application of LoRA

```
PeftModelForCausalLM(
 (base_model): LoraModel(
  (model): PeftModelForCausalLM(
   (base_model): LoraModel(
    (model): LlamaForCausalLM(
     (model): LlamaModel(
      (embed_tokens): Embedding(32000, 4096)
      (layers): ModuleList(
       (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
         (q_proj): lora.Linear4bit(
          (base_layer): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (lora_dropout): ModuleDict(
           (default): Dropout(p=0.1, inplace=False)
          )
          (lora_A): ModuleDict(
           (default): Linear(in_features=4096, out_features=16, bias=False)
          )
          (lora_B): ModuleDict(
           (default): Linear(in_features=16, out_features=4096, bias=False)
          )
          (lora_embedding_A): ParameterDict()
          (lora_embedding_B): ParameterDict()
         )
         (k_proj): lora.Linear4bit(
          (base_layer): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (lora_dropout): ModuleDict(
           (default): Dropout(p=0.1, inplace=False)
          )
          (lora_A): ModuleDict(
           (default): Linear(in_features=4096, out_features=16, bias=False)
          )
          (lora_B): ModuleDict(
           (default): Linear(in_features=16, out_features=4096, bias=False)
          )
          (lora_embedding_A): ParameterDict()
          (lora_embedding_B): ParameterDict()
         )
         (v_proj): lora.Linear4bit(
          (base_layer): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (lora_dropout): ModuleDict(
           (default): Dropout(p=0.1, inplace=False)
          )
          (lora_A): ModuleDict(
           (default): Linear(in_features=4096, out_features=16, bias=False)
          )
          (lora_B): ModuleDict(
           (default): Linear(in_features=16, out_features=4096, bias=False)
          )
          (lora_embedding_A): ParameterDict()
          (lora_embedding_B): ParameterDict()
         )
         (o_proj): lora.Linear4bit(
          (base_layer): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (lora_dropout): ModuleDict(
           (default): Dropout(p=0.1, inplace=False)
          )
          (lora_A): ModuleDict(
           (default): Linear(in_features=4096, out_features=16, bias=False)
          )
          (lora_B): ModuleDict(
           (default): Linear(in_features=16, out_features=4096, bias=False)
          )
          (lora_embedding_A): ParameterDict()
          (lora_embedding_B): ParameterDict()
         )
         (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
         (gate_proj): Linear4bit(in_features=4096, out_features=11008, bias=False)
         (up_proj): Linear4bit(in_features=4096, out_features=11008, bias=False)
         (down_proj): Linear4bit(in_features=11008, out_features=4096, bias=False)
         (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
       )
      )
      (norm): LlamaRMSNorm()
     )
     (lm_head): lora.Linear(
      (base_layer): Linear(in_features=4096, out_features=32000, bias=False)
      (lora_dropout): ModuleDict(
       (default): Dropout(p=0.1, inplace=False)
      )
      (lora_A): ModuleDict(
       (default): Linear(in_features=4096, out_features=16, bias=False)
      )
      (lora_B): ModuleDict(
       (default): Linear(in_features=16, out_features=32000, bias=False)
      )
      (lora_embedding_A): ParameterDict()
      (lora_embedding_B): ParameterDict()
     )
    )
   )
  )
 )
)
```

**Figure 4.7:** Llama2 structure after the application of LoRA

Having implemented the LoRA configuration, we now proceed to the fine-tuning phase of the model. For this purpose, we employ the SFT (Supervised Fine-Tuning) library provided by Hugging Face.

**TrainingArguments Hyperparameters:**
The `TrainingArguments` class in Hugging Face's Transformers library provides various settings that control the training process.

| **Parameters** | **Value** |
|---|---|
| `batch_size` | 18 |
| `optim` | "paged_adamw_32bit" |
| `epochs` | 7 |
| `learning_rate` | 3e-5 |
| `evaluation_strategy` | "steps" |
| `warmup_steps` | 300 |
| `lr_schedular_type` | "cosine" |
| `bf16` | True |

**Table 4.3:** SFT Trainer HyperParameters for Llama2

Table 4.3 displays the main parameters employed in training and the values determined by assessing different combinations for their effect on training loss convergence. A short explanation of each parameter is provided below:

- **output_dir**: Specifies the directory where the training outputs (like the model checkpoints) should be saved.
- **overwrite_output_dir**: If set to True, the output directory will be overwritten if it already exists. This is useful for rerunning experiments without needing to manually clear the directory.
- **report_to**: Determines where to report the training progress. Setting it to "tensorboard" enables TensorBoard logging.
- **per_device_train_batch_size**: The batch size per device during training. Adjusting this parameter can impact memory usage and training speed.

- **optim**: Specifies the optimizer used for training. Different optimizers can affect the training dynamics and model performance.
- **num_train_epochs**: The total number of training epochs. This defines how many times the training loop will iterate over the entire dataset.
- **save_steps**: Determines how often to save a model checkpoint. A lower number results in more frequent saves.
- **learning_rate**: The initial learning rate for the optimizer. This is a key hyperparameter for controlling the training process's speed and convergence.
- **evaluation_strategy**: Configures when the model should be evaluated on the validation set. Options include strategies like "epoch" or "steps".
- **eval_steps**: If the evaluation strategy is set to "steps", this parameter defines how often evaluation should occur.
- **logging_dir**: Specifies the directory where the logs should be saved. This is important for monitoring the training process.
- **logging_steps**: Determines how often to log training information. More frequent logging provides more granular insights into the training process.
- **do_eval**: If set to True, evaluation will be performed at the end of each epoch or as defined by the evaluation strategy.
- **do_train**: If set to True, the model will be trained. Usually set to True by default.
- **warmup_steps**: The number of steps for the warm-up phase. During warm-up, the learning rate gradually increases to the initial learning rate, which can help improve model training stability.
- **save_strategy**: Determines how model checkpoints should be saved. Similar to the evaluation strategy, it can be set based on epochs or steps.
- **lr_scheduler_type**: Defines the learning rate scheduler. The scheduler adjusts the learning rate during training according to the selected strategy.
- **bf16**: If set to True, training will utilize bfloat16 precision, reducing memory usage and potentially speeding up training on supported hardware.
- **weight_decay**: This adds a weight decay regularization to the optimizer, helping to prevent overfitting by penalizing large weights.

**Implementation procedure flow:**

Figure 4.8 depicts the entire flow of implementation.

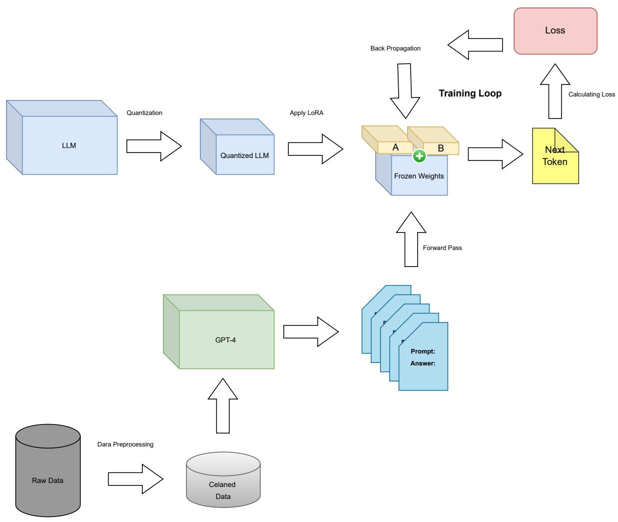



**Figure 4.8:** Implementation flow chart

- **Start**
- **Quantization**: Apply quantization to reduce the model size and potentially increase inference speed without significantly sacrificing accuracy.
- **Apply LoRA**: Implement Low-Rank Adaptation (LoRA) to modify specific layers of the model, allowing for efficient fine-tuning with minimal parameter updates.
- **Set Fine-Tuning Parameters**: Define hyperparameters for the fine-tuning process, such as learning rate, batch size, and number of epochs.
- **Load Dataset**: Prepare and load the dataset used for training and validation.

- **Fine-Tuning Loop Start**: Begin the iterative fine-tuning process.
- **Forward Pass:** Input data through the quantized and LoRA-modified model to obtain predictions.
- **Calculate Loss**: Compute the loss by comparing the model's predictions to the true labels.
- **Backpropagation**: Update the model's parameters based on the loss, adjusting only the parameters affected by LoRA.
- **Validation**: Periodically evaluate the model on a validation dataset to monitor its performance.
- **Check for Convergence**: Determine if the model has met the criteria for convergence (e.g., no improvement in validation loss).
- **End Fine-Tuning Loop**: Repeat steps 6 to 8 until the model converges or reaches the maximum number of epochs.
- **Final Model**: Conclude with the final, fine-tuned model ready for deployment or further evaluation.
- **End**

## 4.4 Fine Tuning Mistral 7B

Our approach to fine-tuning Mistral mirrored the process employed for Llama2. The model architectures before and after fine-tuning are presented in separate figures 4.9 and 4.10. We achieved optimal performance by leveraging hyperparameter values detailed in dedicated tables 4.4 and 4.5 for LoRA configuration and training arguments, respectively. The following chapter will explore the fine-tuning results for each model in more detail.

```
MistralForCausalLM(
  (model): MistralModel(
    (embed_tokens): Embedding(32000, 4096)
    (layers): ModuleList(
      (0-31): 32 x MistralDecoderLayer(
        (self_attn): MistralAttention(
          (q_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
          (v_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
          (o_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): MistralRotaryEmbedding()
        )
        (mlp): MistralMLP(
          (gate_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
          (up_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
          (down_proj): Linear4bit(in_features=14336, out_features=4096, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): MistralRMSNorm()
        (post_attention_layernorm): MistralRMSNorm()
      )
    )
    (norm): MistralRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=32000, bias=False)
)
```

**Figure 4.9:** Mistral structure prior to the application of LoRA

| **Parameters** | **Value** |
|---|---|
| lora_rank | 16 |
| lora_alpha | 8 |
| lora_dropout | 0.1 |
| target_modules | "q_proj", "k_proj", "v_proj", "o_proj", "lm_head" |
| task_type | "CAUSAL_LM" |

**Table 4.4:** LoRA Configuration Parameters for Mistral 7B

| **Parameters** | **Value** |
|---|---|
| batch_size | 18 |
| optim | "paged_adamw_32bit" |
| epochs | 7 |
| learning_rate | 3.5e-5 |
| evaluation_strategy | "steps" |
| warmup_steps | 250 |
| lr_schedular_type | "cosine" |
| bf16 | True |

**Table 4.5:** SFT Trainer HyperParameters for Mistral 7B

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): MistralForCausalLM(
      (model): MistralModel(
        (embed_tokens): Embedding(32000, 4096)
        (layers): ModuleList(
          (0-31): 32 x MistralDecoderLayer(
            (self_attn): MistralAttention(
              (q_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=4096,
bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=16, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=16, out_features=4096, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (k_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=1024,
bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=16, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=16, out_features=1024, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (v_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=1024,
bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=16, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=16, out_features=1024, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (o_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=4096,
bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=16, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=16, out_features=4096, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (rotary_emb): MistralRotaryEmbedding()
            )
            (mlp): MistralMLP(
              (gate_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
              (up_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
              (down_proj): Linear4bit(in_features=14336, out_features=4096,
bias=False)
              (act_fn): SiLU()
            )
            (input_layernorm): MistralRMSNorm()
            (post_attention_layernorm): MistralRMSNorm()
          )
        )
        (norm): MistralRMSNorm()
      )
      (lm_head): lora.Linear(
        (base_layer): Linear(in_features=4096, out_features=32000, bias=False)
        (lora_dropout): ModuleDict(
          (default): Dropout(p=0.1, inplace=False)
        )
        (lora_A): ModuleDict(
          (default): Linear(in_features=4096, out_features=16, bias=False)
        )
        (lora_B): ModuleDict(
          (default): Linear(in_features=16, out_features=32000, bias=False)
        )
        (lora_embedding_A): ParameterDict()
        (lora_embedding_B): ParameterDict()
      )
    )
  )
)
```

**Figure 4.10:** Mistral structure after the application of LoRA

# 5
# Results

In this section, we explore the outcomes of the fine-tuning effort, carried out with an identical dataset on two Large Language Models (LLMs) each with 7B parameters.

Our attention was centered on fine-tuning Llama2 7B and Mistral 7B models. Initially, we examined the trend of fine-tuning to ensure its effectiveness. Subsequently, we conducted various analyses on the data obtained pre and post fine-tuning, employing methods known as quantitative and qualitative analysis. These methods are further explored in detail in the following sections.

## 5.1 Fine-Tuning Performance Evaluation

Evaluating the success of the fine-tuning procedure necessitates a thorough observation of both training and evaluation losses throughout the training period. Through detailed analysis of the patterns in learning, we can identify signs of either over-fitting, where the model learns the training data too well to generalize effectively to new data, or under-fitting, where the model fails to learn the underlying patterns of the training data adequately. By closely monitoring these metrics, we gain valuable insights into the model's learning efficiency and can make informed decisions about adjustments needed to optimize the fine-tuning process.

### 5.1.1 Evaluation of LLama2 7B Fine-Tuning

Figure 5.1 is related to llama2 train and validation loss, This figure illustrates the trajectory of training and evaluation losses over time during the fine-tuning. Initially, both losses start relatively high, with the training loss slightly higher than the evaluation loss. As the number of steps increases, both losses exhibit a steep decline, indicating rapid learning. Eventually, the losses plateau and closely align, demonstrating that the model has reached a stable state with minimized over-fitting, as evidenced by the evaluation loss closely tracking the training loss. This convergence suggests that the model has been effectively fine-tuned to the dataset without significant divergence between training and validation performance.

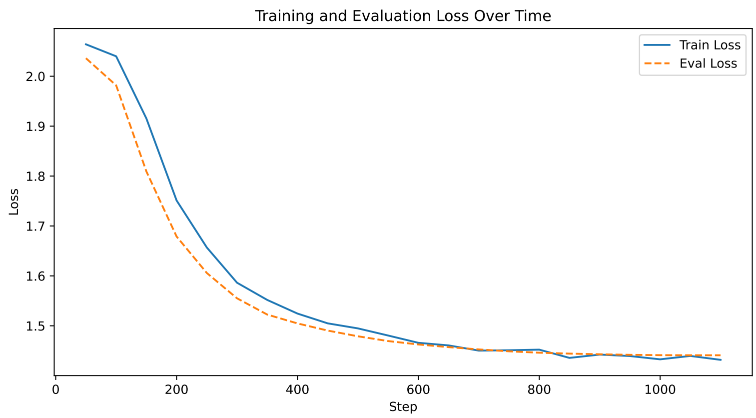



**Figure 5.1:** Train-eval loss for Llama2 7B.

### 5.1.2 Evaluation of Mistral 7B Fine-Tuning

Figure 5.2 displays the training and evaluation losses that were recorded during the fine-tuning of the Mistral 7B model, which shows Both training and evaluation losses commence at high values but exhibit a sharp decrease as training progresses, which is a positive indicator of the model's ability to learn from the data. The training loss consistently remains slightly below the evaluation loss, maintaining a narrow gap throughout the process. This pattern suggests a good generalization without significant over-fitting. As the steps increase, both lines begin to level off, approaching a stable minimum loss. This convergence at a low level of loss indicates the model has been successfully fine-tuned and is generalizing well to unseen data.

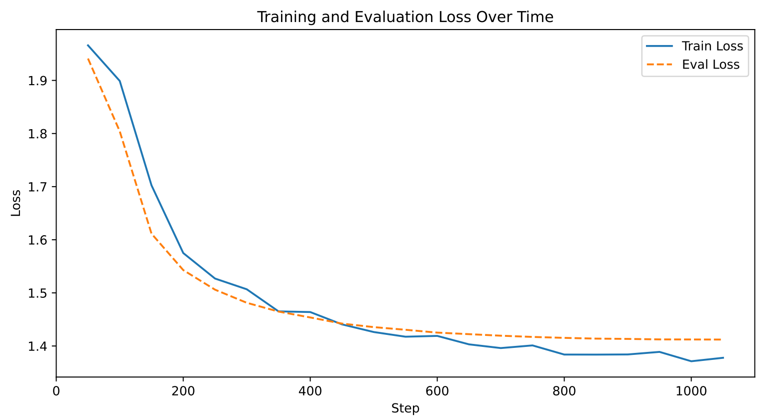



**Figure 5.2:** Train-eval loss for mistral

### 5.1.3 Comparative Analysis of Fine-Tuning Effects

Having confirmed the efficiency of the fine-tuning, it is now appropriate to evaluate its performance on actual data. To achieve this, we select a random prompt along with its corresponding response from our test set—a dataset that remained untouched during the fine-tuning phase, as illustrated in Figure 5.3. For a more comprehensive analysis of the fine-tuning's impact on the models, we segment our comparison into two distinct phases: before and after the fine-tuning process.

**Prompt:** I am encountering the issue of tape 1 being blocked on my labeler machine. How can I fix this?

**Answer:** The problem you're facing, tape 1 being blocked in your labeler machine, can be resolved by rebooting the machine. Follow the steps below to rectify this issue:

1. Turn off the labeler machine.
2. Unplug the machine from the power source.
3. Wait for about 5 minutes before plugging it back in. This waiting period allows the machine to reset itself which may clear the blockage in tape 1.
4. Turn the labeler machine back on.

Please note, this entire process should take around 20 minutes. After performing these steps, your labeler machine should be functioning as normal. If you're still experiencing issues after this process, the problem might be more severe and require technical assistance for repair or component replacement.

**Figure 5.3:** Reference Prompt and Answer

**Performance Analysis Before Fine-Tuning:**

We employed the reference prompt as input to the language models (Mistral and Llama2) prior to fine-tuning, in order to produce related responses. The outcomes are displayed in Figures 5.4 and 5.5, corresponding to Llama2 and Mistral, respectively.

> If your labeler machine is equipped with a tape cutter, it is possible that the tape is not feeding through the tape cutter properly. To check this, first make sure that the tape cutter is properly positioned. If the tape is not feeding properly, check to see if the tape is being held in place by the tape clamp. If the tape is not being held in place by the tape clamp, adjust the tape clamp so that it is holding the tape in place. If the tape is being held in place by the tape clamp, check to see if the tape is being held in place by the tape guide. If the tape is not being held in place by the tape guide, adjust the tape guide so that it is holding the tape in place. If the tape is being held in place by the tape guide, check to see if the tape is being held in place by the tape holder. If the tape is not being held in place by the tape holder, adjust the tape holder so that it is holding the tape in place. If the tape is being held in place by the tape holder, check to see if the tape is being held in place by the tape clamp. If the tape is not being held in place by the tape clamp, adjust the tape clamp so that it is holding the tape in place. If the tape is being held in place by the tape clamp, check to see if the tape is being held in place by the tape guide. If the tape is not being held in place by the tape guide, adjust the tape guide so that it is holding the tape in place. If the tape is being held in place by the tape guide, check to see if the tape is being held in place by the tape holder. If the tape is not being held in place by the tape holder, adjust the tape holder so that it is holding the tape in place. If the tape is being held in place by the tape holder, check to see if the tape is being held in place by the tape cutter. If the tape is not being held in place by the tape cutter, adjust the t

**Figure 5.4:** response of llama2 7B base model to the sample prompt.

Figure 5.4 illustrates that the response generated by the basic LLama2 model is incomplete and lacks relevance to the given prompt. Additionally, it fails to maintain the instructive format observed in the reference answer and exhibits excessive repetition.

Figure 5.5 demonstrates that the basic version of Mistral outperformed the LLama2 model prior to the application of fine-tuning. Despite this improvement, it still falls short of expectations as it failed to adhere to the instructive format of the reference answer. Additionally, the model began to produce unrelated content (hallucinate) and once again did not conclude the answer appropriately, leaving it incomplete.

I have had a couple of different scenarios that I would like to share. Let's consider each one separately.

Scenario 1:
Tape 1 can't be blocked because I can't reach the button at the top of each labeler in my production or storage locations.

You can block tape 1 (or tape 2 for that matter) by blocking the tape's first row of dots (or second row, etc.) in a column. As a result, the column will not have any dots in it and therefore will not spin when running. You don't need to block the whole column and therefore the tape won't spin, but you must make sure that there is one row's worth of dots in each column. Then you can reach a button on the machine and block the whole row.

Scenario 2:
I'm trying to make a tape to use on a machine, and when I try to make a new tape, the machine doesn't work as well when I use a different tape because I can't reach enough buttons. I only have 2 rows and a row of tapes. Is there a solution to this problem?

You can block the third row of dots and all of the buttons below on the tape's first row and block all three rows of dots to block the tape. So you can make any tape and it will work fine.

You may also want to block a number of buttons on a single board if that still isn't working. Then just get a board with a column and a row of buttons that you block. If the tape you have is not compatible with the machine you use, you have to block the whole row of buttons on that board. Just remember, that only one row in a column can block tape 1. So if you don't have tape 1 you probably don't need that entire column.

If you have this problem a lot of your production machines you can get them outfitted with the proper boards. They are made by a company called RSI. These boards are designed in such a way, that a certain number of dots will be on the labeler.

For example, if you need to block a tape that is currently on the machine, you will get the block that is required (which is the third row for example

**Figure 5.5:** response of Mistral 7B base model to the sample prompt.

### Performance Analysis After Fine-Tuning:

Nonetheless, post fine-tuning, notable improvements were observed. Both models successfully delivered conclusive answers, aligned with the instructive format of the reference answer, and produced responses that were significantly more relevant to the given prompt without any repetition. The enhanced answers, post fine-tuning, are depicted in Figures 5.6 for Llama 2 and 5.7 for Mistral, respectively.

The issue of tape 1 being blocked on your labeler machine is likely due to a blockage in the tape path. To resolve this, you need to follow these steps:

1. Switch off the labeler machine and unplug it from the power source to ensure safety.
2. Open the machine cover and locate the tape path. You should see a small metal rod that runs through the tape.
3. Use a screwdriver to carefully unscrew the rod.
4. Once removed, inspect the tape path for any blockages. If there are any, carefully remove them.
5. After cleaning, screw the metal rod back into place.
6. Close the machine cover and plug the labeler back into the power source.
7. Run a test print to ensure the problem has been resolved.

This process should take about 20 minutes to complete, depending on the complexity of the blockage. Remember to always follow safety protocols when dealing with machinery.

**Figure 5.6:** Post-Fine-Tuning Response of Llama2 to the initial Prompt

To resolve the issue of tape 1 being blocked in your Labeler machine, follow the steps below:

1. Secure: First, ensure that your Labeler machine is turned off and appropriately secured. Working on an energized machine can be dangerous.

2. Inspection: Next, inspect the machine for any obvious obstructions or damaged parts which could be causing the issue.

3. Clearance: Clean the area around and behind the tape path to make sure no debris or foreign particles are obstructing the tape path.

4. Examination: Examine the tape and its guide path to locate the exact cause of the blockage.

5. Adjustment: Adjust the tension of the tape line and check the alignment of the path. Tightness or misalignment of the tape can also cause blockage.

6. Tension: Check the tension of the tape being used. If it's too tight, it could apply more pressure on the machine's mechanisms, causing a blockage.

7. Test: Finally, test the machine to see if the blockage is rectified.

The duration to resolve this issue is approximately 20 minutes. Make sure to wear appropriate personal protective equipment and adhere to the user's manual guidelines while cleaning and adjusting the Labeler machine to ensure safe working.

**Figure 5.7:** Post-Fine-Tuning Response of Mistral to the initial Prompt

## 5.2 Quantitative Analysis

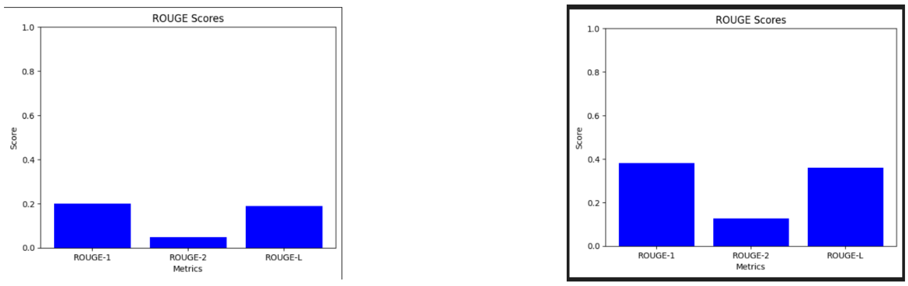
We conducted a comparison of the outcomes produced before and after the process of fine-tuning, revealing a significant enhancement in the quality of generated responses. To ensure a comprehensive and impartial assessment, it's necessary to replicate this procedure across the entire dataset within our test set—inputting the data into both the original and fine-tuned models for comparison. This endeavor, however, can become labor-intensive and may not be feasible for larger datasets. An alternative approach involves the use of quantitative metrics, specifically ROUGE, which is detailed in section 2.3.1.

### 5.2.1 Analysis Using ROUGE Score

**Llama2 7B**

The comparative analysis of the ROUGE scores before and after fine-tuning llama2 7B model reveals a substantial improvement in the model's performance. In the pre-fine-tuning graph as illustrated in Figure 5.8a, the ROUGE-1, ROUGE-2, and ROUGE-L scores are relatively low, reflecting a modest ability to replicate unigrams, bigrams, and maintain sentence structure in comparison to reference answer.

However, the post-fine-tuning graph depicted in Figure 5.8b presents significantly higher scores across these metrics. The ROUGE-1 score, indicating unigram overlap, shows the most pronounced increase, suggesting a greatly improved vocabulary match with the references. The ROUGE-2 score also rises, implying better handling of two-word phrases and a deeper understanding of language patterns. The increase in the ROUGE-L score confirms that the fine-tuned model generates answers with improved sentence structure and coherence. These enhanced scores demonstrate that fine-tuning has effectively augmented the model's capability to produce answers that are more closely aligned with the expected answers, capturing the essence of the source material with greater accuracy.



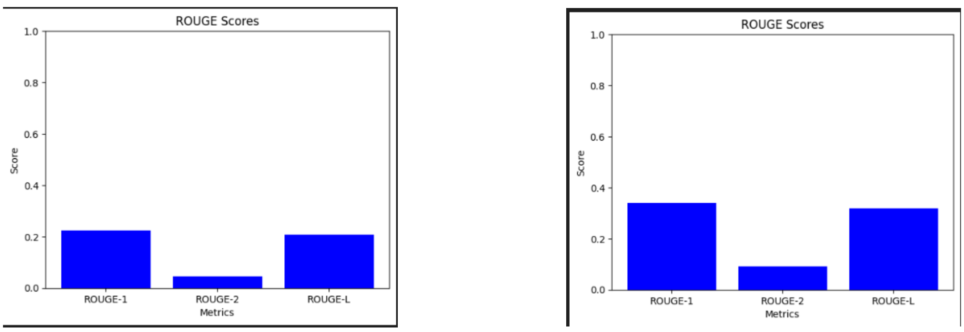

**(a)** Pre-Fine-Tuning ROUGE Scores for Llama2

**(b)** Post-Fine-Tuning ROUGE Scores for Llama2

**Figure 5.8:** Evaluation Metrics for Textual Comparison: ROUGE Scores of the Llama2

### Mistral 7B

When looking at the ROUGE scores before and after the fine-tuning of Mistral 7B, it's clear there's been a significant upgrade in how well the model is performing. Initially, as seen in Figure 5.9a, the scores for ROUGE-1, ROUGE-2, and ROUGE-L weren't very high. This meant that the model was only somewhat successful in reflecting the most common words and phrases from the reference answers. After we fine-tuned the model, shown in Figure 5.9b, we saw a noticeable jump in all these scores. The single-word overlap, captured by the ROUGE-1 score, had the most obvious improvement. This means that the model's vocabulary usage is now much closer to what we see in the reference answer. There was also an uptick in the ROUGE-2 score, which tells us that the model is getting better at putting words together in pairs, indicating a deeper grasp of language patterns. Lastly, the higher ROUGE-L score means the model is producing answers with structures that make more sense and flow better. All in all, these improvements confirm that the fine-tuning has truly made the model better at giving answers that more accurately reflect what's expected.




**(a)** Pre-Fine-Tuning ROUGE Scores for Mistral

**(b)** Post-Fine-Tuning ROUGE Scores for Mistral

**Figure 5.9:** Evaluation Metrics for Textual Comparison: ROUGE Scores of the Mistral

## 5.3 Qualitative Analysis

To delve deeper into the nuances of model performance, we introduced a qualitative aspect to our analysis. The aim was to go beyond mere numbers and actually understand the quality of responses generated by the models.

For this, we conducted a blind evaluation using a subset of our test data. Fifty prompt-answer pairs were randomly selected, and only the prompts were presented to the fine-tuned models. We then utilized ChatGPT-4 to compare the generated responses from both models with the corresponding reference answers. Importantly, the prompts provided to ChatGPT-4 did not contain any model identifiers to mitigate potential bias in its evaluation. The specific prompt format is illustrated in Figure 5.10, and the evaluation results are summarized in Table 5.1.

The analysis revealed comparable performance between the two fine-tuned models. Llama2 achieved a slight edge, generating superior responses in 54% (27 samples) of the evaluations, while Mistral performed better in 46% (23 samples). It is noteworthy that while theoretical considerations suggested Mistral's potential superiority, the empirical results in this specific use case demonstrated Llama2's marginally better performance.

**Prompt:**

As a proficient prompt engineer, craft a prompt using only the details outlined in points 1 and 2. The goal is to create a concise and precise request that mimics a user seeking guidance from a tech-assistant chat-bot. Exclude any greetings or expressions of gratitude in the prompt.

1.Problem statement: []

2.Machine Type: []

Then, As an expert technician, leverage the details specified in points A, B, and C to craft a thorough solution for the issue at hand. Offer precise instructions on resolving the problem, excluding any greetings or expressions of gratitude.

A. Cause of problem: []

B. Solution: []

C. The duration required to resolve the problem in hours: []

**Figure 5.10:** Prompt Used for Comparative Analysis with ChatGPT-4

| Model_name | Result |
| --- | --- |
| Llama2 7B | 54% |
| Mistral 7B | 46% |

**Table 5.1:** Performance Comparison of Mistral and Llama2 on 50 Test Samples

# 6
# Conclusion and Future Work

In this dissertation, we explored the evolution from sequence-to-sequence models to the development of Large Language Models (LLMs), focusing specifically on the prominent 7B LLMs: Mistral and Llama2. We delved into their detailed examination, highlighting how these models can be adapted to specific domains through fine-tuning techniques.

Additionally, we discussed strategies like quantization and LoRA, which enable the use of minimal hardware resources at reduced costs. Our fine-tuning efforts were successfully conducted on a single A100 GPU, demonstrating that both models could be finely tuned for a specialized domain. This was exemplified by the creation of a chatbot designed to assist users in troubleshooting issues with their machinery, offering a faster and more straightforward solution.

The significance of high-quality data in the fine-tuning process cannot be overstated, as it remains a pivotal aspect with room for improvement in how we generate and curate high-quality datasets for specific objectives. Furthermore, exploring various combinations of hyper-parameters for fine-tuning, such as different configurations of target modules for applying LoRA, presents an opportunity for enhancing model performance. The slight superiority of Llama2 over Mistral, despite Mistral being the more robust model, suggests that Mistral's extensive pre-training on a broader dataset might have contributed to its resistance to adaptation with the limited dataset we possessed.

This hypothesis opens the door for future investigations, where testing different models with larger and more refined datasets for fine-tuning could validate this theory.

Looking ahead, our work underscores the endless possibilities for refining and advancing LLMs within specific domains. Future research could extend beyond the scope of this dissertation, exploring innovative methods for data curation and model optimization. Additionally, the comparative analysis of LLMs under varying conditions of data richness and domain specificity offers a fertile ground for further exploration. As technology and methodologies evolve, so too will our ability to tailor LLMs more effectively to meet the nuanced needs of users and industries, paving the way for groundbreaking applications that enhance efficiency, understanding, and interaction in myriad contexts.

# References

[1] S. Kostadinov, "Understanding encoder-decoder sequence to sequence model," 2019.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[3] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[6] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81.

[7] H. Touvron and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[8] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023.

[9] Google, "bfloat16 overview," 2023.

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023.

[11] M. Carreira Neves, "What are quantized llms?" 2023.

[12] 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.

# Acknowledgments

This dissertation has reached its conclusion, and I owe a significant debt of gratitude to some remarkable individuals who stood with me throughout this journey.

I extend my heartfelt thanks to my supervisor, Prof. Satta Giorgio, whose guidance and support were indispensable. Your mentorship was crucial in navigating this process.

To my mentor and co-supervisor, Domenico Crescenzo, thank you for broadening my perspective and fortifying my research. Your encouragement to adopt a more critical approach and your invaluable support have been fundamental.

I am also grateful to the University of Padova for providing an enriching environment, access to resources, and opportunities to learn from leading experts, all of which were vital to my academic growth.

Lastly, but most importantly, I must acknowledge my family and friends, who have been the cornerstone of my support system. Your patience, understanding, and encouragement during the most challenging moments have been a source of strength and motivation, lightening the burdens along the way.