**Romanian Open-Source Chatbot**

**Candidate: Andrei - Răzvan, JOLDEA**

**Scientific coordinator: Associate Professor Cosmin, CERNĂZANU**

Session: June 2024

**Table of Contents**

**(Arial 20 pt, Bold, Uppercase, Center)**

The abstract is intended to inform about the content of the paper through a brief description of the research of up to one page, the procedures/methods, as well as its results or conclusions. The abstract in Romanian becomes mandatory for works edited in languages other than Romanian and will be written in 12 pt Arial fonts. It will start two blank lines after the heading "ABSTRACT". Before the title, there will be three blank 12 pt. lines.

**ABSTRACT**

**(Arial 20 pt, Bold, Uppercase, Center)**

The English abstract will be on the third page of the manuscript and will synthetically present the paper work. The maximum length of the abstract is one page written with Arial fonts, size 12 pt. The abstract text will begin after two blank lines (size 12pt.) from the “ABSTRACT” title. Before the title there will be left three 12 pt blank lines.

# INTRODUCTION

Over the past few years, the technological landscape has suffered an immense shift due to the emergence of large language models (LLMs). Fuelled by advancements in artificial intelligence (AI) domains such as machine learning (ML), deep learning (DL) and natural language processing (NLP), these models are prevalent in a wide variety of fields, namely healthcare, business intelligence, legal analytics, and even creative arts, encompassing music and literature [1].

While it is true that Large Language Models have impressive capabilities, most of the powerful models are only fluent in the English language, with a few exceptions regarding their multilingual capabilities.

My goal for this thesis was to create a large language model capable of understanding and speaking in Romanian, by finetuning a base model on high quality Romanian data.

A screenshot of a computer

Description automatically generated

Table 1: GPT-4 benchmarks (source: [2]).

A screenshot of a computer screen

Description automatically generated

Table 2: Gemini performance on text benchmarks with external comparisons (source: [3]).

A screenshot of a computer

Description automatically generated

Table 3: Meta Llama 3 Pre-trained model performance (source: [4]).

A screenshot of a computer screen

Description automatically generated

Table 4: Comparison of Mixtral with Llama (source: [5]).

The Massive Multi-task Language Understanding (MMLU) is one of the most well rounded and popular benchmarks when it comes to large language models. It allows researchers to gain insights into the capabilities and limitations of various LLMs, by measuring the performance of these models using a plethora of tasks and evaluation methods [6].

According to tables 1, 2, 3, 4, we can conclude that the best LLMs on the market right now are OpenAI’s GPT-4 and Google’s Gemini Ultra, each with a 5-shot MMLU benchmark score of 86.4% and 83.7%, respectively. It is worth noting that both these large language models, as well as other highly rated models such as GPT-3.5, PaLM 2L or Claude 2 are closed-source models. On the other hand, Meta’s Llama models and Mistral AI’s Mistral and Mixtral models are open source. This is a relevant aspect that we will discuss further in our analysis. Another detail that is of utmost importance for my task is that these results have been achieved on mainly English tasks.

A graph with numbers and a number of text

Description automatically generated with medium confidence

Figure 1: GPT-4 3-shot accuracy on MMLU across languages (source: [2])

From figure 1, one can infer the fact that the performance of GPT-4 decreases steadily in multilingual scenarios, based on the amount of training data that it had been exposed to during training. This is general amongst all other models.

When choosing between a closed source and an open source LLM, one must take into consideration the following criteria:

* Cost: Closed source models often involve subscriptions in order to have access to the model itself or its API.
* Customization: Open source LLMs are highly customazible for any particular task, due to the fact that the underlying architecture and source code are available to the users.
* Technical expertise: Closed source LLMs are typically more user-friendly, providing an easy to use UI
* Transparency: Users can better understand open source models due to their availability
* Collaboration: Open source communities contribute to the development of the field through shared resources and expertise

Another factor of utmost importance is the computation power available. In this case, I have had at my disposal a virtual machine (VM) with 3 NVIDIA Tesla T4 GPUs, each with approximately 16GB of GDDR6 memory, 16 8-core CPUs, 128GB of RAM and over 2T of storage to complete this task.

As stated before, all the afore mentioned models perform best on English use cases, making a Romanian customization difficult. For instance, if we wanted to create a chatbot specialized in a specific field such as medicine, law or business, a retrieval augmented generation (RAG) using text data from the particular field should be enough. However, since our chatbot should be able to communicate in the Romanian language, this process becomes much more difficult, as it must first be ensured that the LLM is consistent in understaning and providing qualitive answers in Romanian. With this in mind, an open source model would be the most logical choice, since it is highly customizable and would not require additional costs.

A table of numbers and letters

Description automatically generated

Table 5: Llama 2 language distribution in pretraining data with percentage >= 0.005% (source: [7])

In table 5, it can be observed the language distribution of the pretraining data for Llama 2. The Romanian language had a 0.03% distribution, which may seem extremely low (and it is) compared to English, for instance, with almost 90%. Even so, my goal could be worse if I was Bulgarian, Danish, Slovenian or Croat, each with a distribution lower than 0.03%.

With all these aspects in mind, it can be concluded that the base model to be used for the finetuning process should respect the following constraints:

* It should be open source.
* It must fit within the available computing resources.
* It must have been exposed to some Romanian data during pretraining

That being said, the large language models that fit these criteria are: Llama-2 7B, Llama-2 13B, Llama-3 8B and Mistral 7B. Altough the official research paper has not yet been released for Llama 3, we can take figure 3 as a reference point and assume that the Romanian language distribution is at least equal to that of Llama 2, but the reasoning capabilities should be much greatear due to its exposure to 15 trillion tokens, compared to Llama 2’s 2 trillion tokens. Therefore, on paper, Llama-3 8B should be the best candidate for this task.

In this paper, I am going to present the techniques, comparisons and results for each of these models, in order to fulfill the task of creating a chatbot that cand follow and express itself in a natural manner in the Romanian language.

1. **Theoretical Foundation**
   1. **The Transformer**
      1. **State of the art**

Recurrent neural networks (RNNs) form a diverse class of artifical neural networks characterized by internal feedback loops. These loops allow for information to flow both forward and backward, unlike traditional feedforward networks, enabling RNNs to process sequential data and model time dependencies [8].

Long Short-Term Memory (LSTM) networks are a specialized type of recurrent neural networks, designed to address the vanishing gradient problem and capture long-term dependencies in data more effectively. Memory cells represent the base of the innovation associated with LSTM. These memory cells are able to retain information over extended periods of time, by selectively remembering or forgetting information based on signals received from input data. Each LSTM unit is composed out of four main components: cell state, input gate, forget gate and output gate [9].

Recurrent neural networks (RNNs), Long Short-Term Memory (LSTM) and gated recurrent units (GRUs) in particular, have been leading approaches in the field of natural language processing (NLP) tasks such as language modelling and machine translation. However, these architectures face a significant challenge when it comes to handling long sequences, namely: as new elements are incorporated, their ability to retain information from the initial elements decreases. This is because the hidden state in each step of the encoder often reflects the most recent input element. If the decoder only relies on the final hidden state, it misses crucial contextual information from the beginning of the sequence. To adress this issue, the attention mechanism was introduced [10, 11].

The attention mechanism allows the decoder to examine all encoder states at each step, instead of focusing only on the last state. What this does, is it enables access to information about every element in the input sequence. The main idea behind attention lies in aggregating information from the whole sequence, by calculating a weighted sum of all past encoder states [11]. In this manner, the decoder is able to assign varying degrees of importance to different parts of the input sequence when predicting each output element. This is an iterative process, allowing the model to learn to focus on the most relevant parts of the input in order to provide an accurate prediction of the corresponding output element [10, 11]. However, a major bottleneck remains: the sequential nature of processing. Both the encoder and the decoder must complete processing the previous step, before moving on to the current step [11]. This approach becomes time consuming and computationally expensive when dealing with large datasets. The Transformer model aims to address this very issue.

The Transformer model architecture follows a unique approach in order to extract features for each word in a sentence. Unlike other traditional models that rely on recurrent units, it leverages a mechanism called „self-attention” [10] which analyses the importance of every other word in the sequence relative to the word being examined. Since the Transformer doesn’t use any recurrent units in order to obtain these features, the activations and weighted sums are very parallelizable, increasing efficiency.

**2.1.2 Model Architecture**

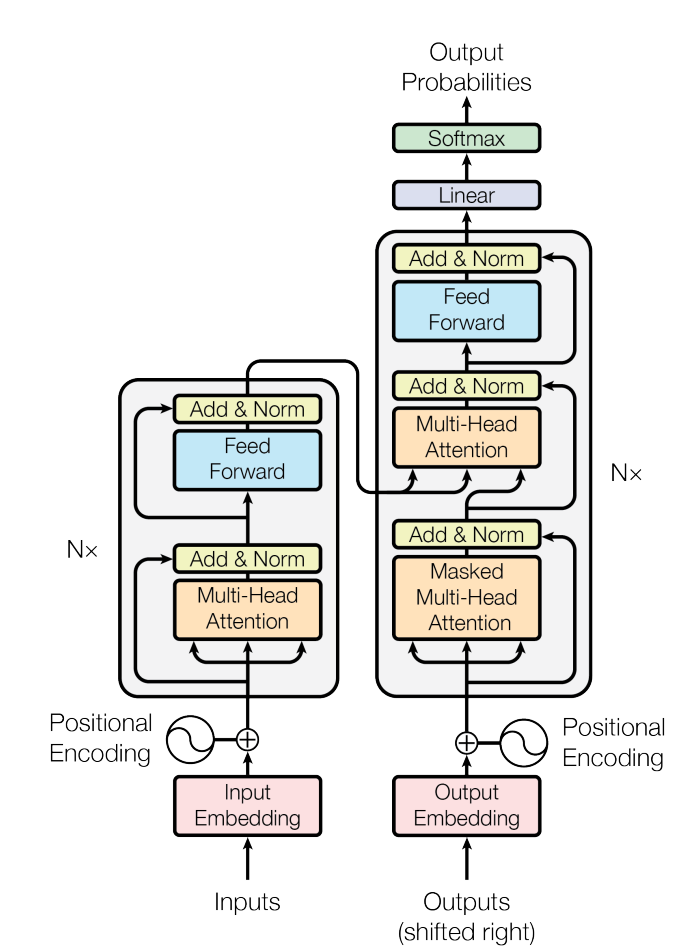


Figure 2: The Transformer - model architecture (source: [10])

Figure 2 depicts the Transformer model architecture, with an encoder model on the left side and a decoder on the right. Many state-of-the-art neural models for sequence translation encompass an encoder-decoder architecture. The encoder acts as an information extractor, processing an input sequence of symbol representations and mapping them to a continuous representation sequence [10, 12]. The decoder, provided with the encoded representation, is tasked with generating an output sequence, one symbol at a time. At every step, the model takes the generated output as additional input for the next one [10, 13].

The encoder model consists of six identical layers, each processing the input through two sub-layers: a multi-head self-attention mechanism that analyses relationships between different parts of the input sequence and a position-wise fully connected feed-forward network which adds non-linearity to the model learning capacity, which is represented by two linear transformations with a ReLU activation function, according to (1).

( 1 )

The decoder also uses six identical layers, each with the same two sub-layers as the encoder, with an additional sub-layer: a multi-head attention mechanism that is mainly focused on the output of the encoder, allowing the decoder to understand the context of the input.

Both the encoder and decoder layers incorporate residual connections around each sub-layer. This ensures efficient information flow through the network. In addition, layer normalization is applied after each sub-layer and residual connection for improved training stability.

To prevent information leakage during decoding, a masking mechanism is applied to the decoder’s self-attention sub-layer. This ensures each prediction at the current position considers only the information from preceding positions. This, combined with a one-position shift in the output embeddings, guarantees the model relies solely on previously generated outputs when predicting the next element in the sequence.

„Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence” [10]. The self attention mechanism of the Transformer model introduces three components: queries, values and keys, each corresponding to a distinct role within the self-attention mechanism. As a query, the vector is compared to others to identify relevant information. In its role as a key, it is compared against queries to determine its contribution to the final output based on the weights calculated during the comparison stage.

( 2 )

( 3 )

( 4 )

Each role corresponds to a k×k weight matrix, computed according to equations (2), (3) and (4).

A diagram of a diagram

Description automatically generated

Figure 3: Illustration of the self-attention with key, query and value transformations. (source: [14])

Commonly denoted as , and , these matrices are obtained by applying linear transformations to a single encoded input. This shared origin allows the model to perform self-attention, where the attention mechanism focuses on the input vector itself and its relationship with other elements within the sequence [11, 14].

The Scaled Dot-Product Attention mechanism operates on queries and keys of dimension , and values of dimension . It calculates the compatibility between each query and all keys using the dot product. To address potential instability during training, this value is scaled by the square root of . Furthermore, a softmax function is applied to these scaled values, resulting in weights that represent the relative importance of each value [10, 11, 14].

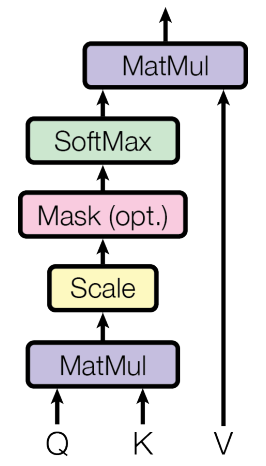


Figure 4: Scaled Dot-Product Attention (source: [10])

In other words, the mechanism employs the , , and matrices to compute attention scores. These scores indicate the significance of other positions (or words) within the sequence in relation to a specific word’s position. For instance, to determine the attention score for the first position (word), the model calculates the dot product of the first query vector with each key vector , , , etc [10, 11].

A crucial aspect of this approach is the scaling factor introduced before the softmax function. This scaling mitigates vanishing gradients, a phenomenon that can hinder training by making it difficult for the model to learn effectively. After obtaining the weights through softmax, the mechanism multiplies them by the value matrix . This selectively emphasizes relevant words (those with greater weights) and reduces the influence of irrelevant words (whose weights and corresponding values in become insignificant) [11, 14].

The matrix of outputs is computed according to equation (5):

( 5 )

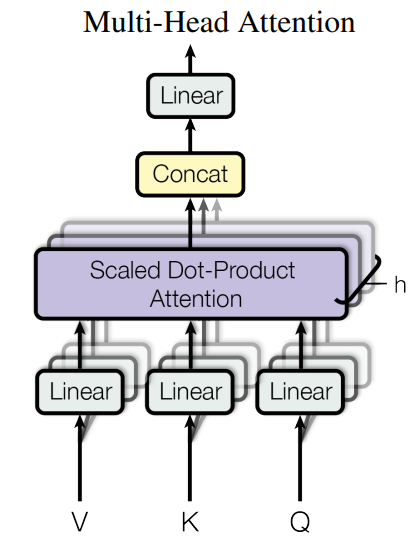


Figure 5: Multi-Head Attention (source: [10])

where

( 6 )

A potential limitation of the previously described approach is its inability to distinguish between sentences with identical words but different word order. Since attention scores are computed for the entire sentence at once, such sentences might receive similar scores. To address this challenge and enhance the model’s ability to differentiate between sentence structures, the Transformer architecture uses multi-head self-attention [11, 14].

Multi-head self-attention involves splitting the word vectors into separate groups, where represents the number of attention heads. Each head independently performs self-attention using its corresponding sub-matrices from the original, , and matrices. This process essentially allows the model to focus on different aspects of the sentence simultaneously through these separate heads, resulting in different output matrices containing attention scores [10, 11, 14].

However, the subsequent layer in the architecture, the feed-forward layer, requires a single input vector for each word. To adjust this, the outputs from each head (individual attention score matrices) are concatenated into a single larger matrix, according to equation (6). Finally, this combined matrix is multiplied by a learnable weight matrix to consolidate the information captured by all the attention heads [11, 14].

One issue that occurs within the self-attention mechanism is its inherent permutation invariance. This means shuffling the order of words in a sentence wouldn't affect the final output of the network. Since word order plays a crucial role in understanding sentence meaning, addressing this limitation is essential [10, 11].

A resolution for this issue is represented by incorporating positional encodings into the input embeddings at the beginning of both the encoder and decoder stacks. These encodings, with the same dimensionality as the embeddings, are added together to provide the model with information about the relative position of each word within the sequence. There are various approaches to implement positional encodings [10, 11].

One approach utilizes a function that maps each word's position in the sentence to a real-valued vector. This allows the network to learn and utilize this positional information during the training process. An alternative approach involves using position embeddings, similar to word embeddings. Here, a unique vector is assigned to each possible position in a sentence. However, this method requires the model to be trained on sentences with all possible lengths beforehand. In contrast, positional encoding empowers the model to generalize to sequences even longer than those encountered during training [11].

( 7 )

( 8 )

The sinusoidal functions from (7) and (8) are applied, according to [10].

It's crucial to incorporate different masks during the training process. These masks serve to selectively exclude irrelevant information from the attention calculations. The encoder mask acts as a filter, preventing padding tokens from influencing the attention calculations within the encoder. The first decoder mask combines two functionalities. It addresses padding tokens similar to the encoder mask, and additionally implements a look-ahead mask. The look-ahead mask ensures the decoder only attends to information from previous positions in the sequence, mimicking a left-to-right reading process and preventing information leakage from future words that could influence the prediction of the current word. The decoder mask is constructed by taking the maximum value between the padding mask and the look-ahead mask. The second mask is a separate padding mask specifically applied within the encoder-decoder attention layer. It prevents the decoder from attending to irrelevant padded tokens within the encoder's output.

After defining the essential components – the encoder, decoder, and final linear-softmax layer – we can now assemble them to form the complete Transformer architecture.

* 1. **Llama 2**

Building upon the success of its predecessor, in February 2023, Meta released Llama 2, an upgraded iteration boasting an increased parameter range of up to 70B and greater logical and thinking capabilities.

As LLM technology advances, researchers are engaging in efforts to enhance the capabilities of these models. A common approach has been to increase model size, under the assumption that larger models equate to better performance. However, recent studies have challenged this notion, emphasizing the critical role of training on vast and diverse datasets rather than solely prioritizing model size.Llama 2 exemplifies this shift in focus by achieving optimal performance through the exclusive use of publicly available datasets [7, 15].

* + 1. **Pretraining**

**A diagram of a diagram

Description automatically generated**

Figure 6: Pretraining of Llama 2 and training of Llama 2-Chat (source: [7])

The foundation of LLaMA 2 lies in its highly curated training dataset. This data is gathered from publicly accessible sources and then preprocessed accordingly. To protect user privacy, the selection process eliminates websites known for collecting large amounts of personal information. To balance out performance and computation efficiency, the corpus leverages approximately 2 trillion tokens, double the context length compared to its predecessor, LLaMA 1, as it can be seen from table 6. Furthermore, to increase the model's knowledge base and mitigate the potential for generating inaccurate information, factual sources are prioritized during the data up-sampling process [7, 16].

Llama 2 builds upon the existing architecture of Llama 1, retaining the base pretraining settings and sticking to the standard transformer architecture. This framework incorporates pre-normalization using RMSProp and the SwiGLU activation function, while also integrating rotary positional embeddings (RoPE) [7, 15, 16]. What differentiates Llama 2 from its predecessor is the inclusion of augmented context length and the adoption of grouped-query attention (GQA) [16]. These features enable Llama 2 to handle extensive contextual information more efficiently during generation.

A table with numbers and text

Description automatically generated

Table 6: Llama 2 family of models (source: [7])

* + 1. **Finetuning**

The fine-tuning process for LLaMA 2 is constructed for dialogue applications. It starts with supervised fine-tuning to establish a foundational version of Llama-2-chat. Afterwards, iterative refinement is implemented using Reinforcement Learning from Human Feedback (RLHF). This RLHF process incorporates techniques like rejection sampling and proximal policy optimization (PPO) to achieve optimal performance in dialogue scenarios [7, 15, 16].

Optimizing LLMs for conversations depends on getting qualitative and diverse third-party SFT data [16]. Additionally, Reinforcement Learning with Human Feedback (RLHF) plays a crucial role. This technique involves gathering data about human preference through binary comparisons to ensure that the finetuned model aligns with human preferences and instructions.

The reward model within the RLHF system is essential in optimizing Llama 2-Chat’s performance in dialogue scenarios. The model outputs scalar scores that indicate the quality of the generated text in terms of helpfulness and safety. To address the potential trade-off between these two aspects, separate reward models are trained for each [7, 16]. These models use a combination of newly collected data and existing open-source datasets in order to enhance generalization and prevent reward errors.

A graph showing different colored lines

Description automatically generated

Figure 7: Training Loss for Llama 2 models (source: [7])

In terms of tokenization, Llama 2 uses the same tokenizer and adopts the same approach as Llama 1, meaning that it utilizes a byte-pair encoding (BPE) algorithm implemented via SentencePiece [7]. Furthermore, all numbers are segmented into individual digits, and bytes are used to decompose unknown UTF-8 characters. This approach maintains a total vocabulary size of 32k tokens, ensuring seamless communication within the context of dialogue applications.

* + 1. **Evaluation**

A table with numbers and text

Description automatically generated

Table 7: Overall performance on grouped academic benchmarks compared to open-source base models (source: [7])

Table 7 showcases the clear performance advantage of Llama 2 models compared to their Llama 1 predecessors. Notably, the LLaMA 2 70B model demonstrates significant improvements in both Massive Multi-task Language Understanding (MMLU) and BIG-Bench Hard (BBH) benchmarks. These improvements translate to roughly 5 and 8 point gains, respectively, when compared to the LLaMA 1 65B model. Furthermore, comparisons between LLaMA 2 models (7B and 34B) and similarly sized MPT models reveal that LLaMA 2 outperforms across various categories, with the exception of coding benchmarks. Similarly, when compared to Falcon models (both 7B and 40B), LLaMA 2 models (7B and 34B) achieve superior performance across all benchmark categories.

A table with numbers and text

Description automatically generated

Table 8: Comparison to closed-source models (source: [7])

Beyond the open-source landscape, table 8 provides some insights regarding the comparison of Llama 2 models with closed-source LLMs. The results indicate that LLaMA 2 70B closely rivals GPT-3.5 on MMLU and GSM8K benchmarks. However, a notable performance gap persists on coding benchmarks. In comparison to PaLM (540B) by Chowdhery and others [17], LLaMA 2 70B demonstrates either comparable or superior performance across most benchmarks. It is important to acknowledge, however, that a substantial performance difference remains between LLaMA 2 70B and more advanced models like GPT-4 and PaLM-2-L [18].

LLama 2 emerges as a leading collection of open-source foundational language models, ranging from 7 billion to 70 billion parameters. These models have garnered recognition for their exceptional performance, scalability and transparency. By relying on publicly available data for training and open-sourcing these models, Meta favorizes advancements within the rapidly evolving landscape of large language models. The open and adaptable nature of the LLaMA 2 framework empowers researchers and developers to accelerate advancements in AI and language modeling.

* 1. **Mistral**

Mistral 7B demonstrates that smaller LLMs can achieve remarkable results, sometimes even better than their larger competitors. Mistral 7B surpasses the previously leading 13-billion parameter model (Llama 2) on a variety of benchmarks. Furthermore, it even rivals the performance of the best 34-billion parameter model (LLaMa 34B) in specific tasks such as mathematical reasoning and code generation.

Mistral 7B achieves this performance through the incorporation of innovative attention mechanisms, including grouped-query attention (GQA) and sliding window attention (SWA). GQA specifically accelerates inference speed and reduces memory consumption during the decoding stage, enabling higher throughput – a crucial factor for real-time applications. SWA tackles a common limitation of LLMs by efficiently handling longer sequences at a lower computational cost. The combined effect of these attention mechanisms is a significant enhancement in both the performance and efficiency of Mistral 7B [19].

**2.3.1 Architecture**

**A table with numbers and text

Description automatically generated with medium confidence**

Table 9: Mistral 7B Architecture Parameters (source: [19])

Similarly to Llama, Mistral is based off of the transformers architecture. However, some changes are introduced, which can be seen from table 9.

A yellow and orange squares with black numbers

Description automatically generated

Figure 8: Sliding Window Attention (source: [19])

Large language models often employ transformer architectures with stacked layers. However, the standard attention mechanism within a single layer can only attend to a limited window of the input sequence. Sliding Window Attention (SWA) addresses this limitation by leveraging the hierarchical nature of transformers. In SWA, the hidden state at position 𝑖 in layer 𝑘 (denoted as ) attends to hidden states from the previous layer within a window of size 𝑊. However, the impact of SWA extends beyond this immediate window. Due to the stacked nature of transformers, information from previous layers also contributes to the attention process. Each layer builds upon the context established in the prior layer. As a consequence of this, can effectively access information from tokens in the input layer up to a distance of 𝑊×𝑘 tokens. For example, with a window size of 𝑊=4096 at the final layer, the model can theoretically attend to a context of roughly 131,000 tokens in the input sequence [19].

A screenshot of a computer screen

Description automatically generated

Figure 9: Rolling Buffer Cache (source: [19])

While SWA enables access to a broader context, it introduces a potential memory bottleneck due to the need to store information for the entire window size. Rolling Buffer Cache (RBC) addresses this issue by working with SWA. By exploiting the fixed attention span of SWA, RBC allows for a cache with a fixed size of 𝑊. This cache stores keys and values relevant to the current processing step (timestep 𝑖) at a specific position within the cache determined by a modulo operation. As new information enters the processing window (as 𝑖 increases), older information exceeding the window size is discarded and replaced in the cache. This mechanism ensures that the cache size remains constant regardless of the overall sequence length, leading to significant memory savings. For instance, with a sequence length of 32,000 tokens and a window size of 4096, RBC can achieve an eight time reduction in cache memory usage compared to a standard approach, without sacrificing model performance.

A diagram of a number

Description automatically generated with medium confidence

Figure 10: Pre-fill and chunking (source: [19])

Sequence generation tasks involve predicting tokens one after another, while considering the influence of previously generated tokens. Two strategies, Pre-filling and Chunking, can improve the efficiency of this process.

Since the prompt (initial input) is known beforehand, a technique called Pre-filling can be utilized. This involves populating the (key, value) cache with the prompt information before the actual generation begins. This pre-filled cache allows the model to access relevant context from the prompt more quickly during the generation process, leading to faster inference.

When dealing with lengthy prompts, directly processing the entire prompt at once can be computationally expensive. Chunking addresses this issue by segmenting the prompt into smaller manageable units. Each chunk is then used to pre-fill the cache independently. This allows the model to focus on smaller portions of the prompt at a time, improving overall efficiency.

To effectively utilize both Pre-filling and Chunking, the window size of the attention mechanism can be strategically chosen to match the chunk size. This ensures that the model attends to both the pre-filled cache (containing the current chunk) and the actual chunk itself during the generation process. Figure 10 poses a visual representation of how the attention mask works.

**2.3.2 Results**

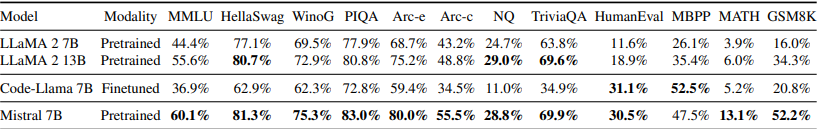


Table 10: Comparison of Mistral with Llama (source: [19])

Table 10 showcases the difference between Mistral and various Llama models. Mistral 7B outperforms Llama 2 13B across various benchmarks. Furthermore, it achieves performance on par with the code-oriented Code-Llama 7B, while maintaining strong performance in non-coding tasks.

* 1. **OkapiLlama**

Okapi offers instruction-tuned LLMs leveraging RLHF for multiple languages, including Romanian, using Supervised Fine-Tuning (SFT) and Reinforcement learning from human feedback (RLHF) [20].

In order to asssess the quality of their Romanian model, I have tested uonlp/okapi-ro-llama’s performance using text generation web UI.

A table with numbers and text

Description automatically generated

Table 11: Performance of the models on the translated HellaSwag dataset over different languages in Okapi. LLaMA 7B is used as the base LLM (source: [20])

A table of numbers and text

Description automatically generated

Table 12: Performance of the models on the translated ARC dataset over different languages in Okapi. LlaMA 7B is used as the base LLM (source: [20])

A table with numbers and text

Description automatically generated

Table 13: Performance of the models on the translated MMLU dataset over different languages in Okapi. LLaMA 7B is used as the base LLM. (source: [20])

Tables 11, 12 and 13 showcase the evaluation score of Okapi’s Llama models across various languages, for different benchmarking datasets. We can observe that for Romanian we have the following scores: 48.7 on the HellaSwag benchmark, 37.5 on the ARC benchmark and 30.9 for MMLU. However, we decide to test the LLM as well using some basic instructions.

A screenshot of a computer

Description automatically generated

Figure 11: Screenshot from Text Generation Web UI #1

From figure 11, we can see that the model performed pretty well, it responded correctly and it answered in Romanian. However, the instruction was quite straightforward with little difficulty presented.

A screenshot of a computer screen

Description automatically generated

Figure 12: Screenshot from Text Generation Web UI #2

Figure 12 showcases some understanding or logic limitation of the model, since the correct answer for the question “Care este reședința de județ a orașului Timișoara?” is “Timiș” not “Timișoara”. Let’s try to rephrase the question to see how the model performs.

A screenshot of a computer

Description automatically generated

Figure 13: Screenshot from Text Generation Web UI #3

As shown in figure 13, the answer this time is even further away from the truth, so the model clearly has some limitations when processing Romanian text and possibly even “reasoning” in Romanian.

Overall, the model can respond in Romanian to some basic questions, however it has some obvious logical and informational flaws, when dealing with Romanian input. There is a high possibility that this happens due to the fact that the Okapi models are finetuned based on the first version of Meta’s Llama 7B, inferior to its successors, Llama 2 and Llama 3.

* 1. **Chinese Llama Models**

Cui, Yang, and Yao introduced the first iteration, Chinese LLaMA. This model is an adaptation of the original LLaMA, incorporating a custom Chinese tokenizer built with SentencePiece. This tokenizer, with a vocabulary of 49,953 tokens, enhances the model's ability to handle Chinese characters effectively. Additionally, the researchers employed memory-efficient fine-tuning techniques (Hu et al., 2021) to optimize resource usage during training. Their experiments evaluated Chinese LLaMA 7B Plus, trained on a substantial corpus of approximately 120GB, equivalent to 30 billion Chinese tokens [21, 22].

Building upon Chinese LLaMA, Cui, Yang, and Yao introduced an upgraded version, Chinese LLaMA2. This model leverages the core advancements of LLaMA2, including its foundational model architecture and optimized vocabulary construction and code implementation. While utilizing the same training corpus and data as its predecessor (30 billion Chinese tokens), Chinese LLaMA2 demonstrates further improvements in performance [21, 22].

A diagram of a model

Description automatically generated with medium confidence

Figure 14: Overview of the proposed Chinese LLaMA and Chinese Alpaca models (based on Meta’s LLaMA and Llama-2) (source: [22])

As it can be seen from figure 14, the process for creating the Chinese models is almost identical, with a pretraining step in order to expose to models to a large, high-quality corpus of Chinese data, follow by a supervised fine-tuning step in order to construct the instruction Alpaca model.

A table with numbers and symbols

Description automatically generated

Table 14: Results on C-Eval valid and test sets (source: [22])

C-EVAL is an evaluation benchmark designed to evaluate the capabilities of base models in understanding and reasoning with Chinese language. This comprehensive benchmark specifically targets advanced knowledge and reasoning skills within a Chinese context [23].

According to table 14, the Chinese Llama and Alpaca finetuned models yield better results than their base model counterparts on the C-EVAL benchmark.

* 1. **vLLM**

Efficiently serving LLMs at high throughput represent a challenge due to issues with dynamic memory demands of the key-value cache (KV cache) for each LLM request. The KV cache size fluctuates based on the request, and poorly managed memory can lead to fragmentation and redundancy, ultimately restricting the number of requests processed simultaneously (batch size). vLLM addresses these issues by using an attention algorithm called PagedAttention, which efficiently manages the KV cache while minimizing memory waste [24].

A diagram of a computer

Description automatically generated

Figure 15: . vLLM system overview (source: [24])

As depicted in Figure 4, vLLM employs a centralized scheduler to coordinate the operations of distributed Graphics Processing Units (GPUs) acting as workers. This centralized approach ensures efficient task distribution and resource management.

A crucial component of vLLM is the KV cache manager. This manager leverages the PagedAttention algorithm to handle the key-value cache (KV cache) in a memory-efficient manner. In essence, the KV cache manager orchestrates the utilization of physical KV cache memory residing on the GPU workers, following directives received from the centralized scheduler. This collaboration between the scheduler, KV cache manager, and PagedAttention algorithm is key to vLLM's ability to process large language models at high throughput.

PagedAttention is an attention mechanism that facilitates efficient memory management within the vLLM serving system. Inspired by paging techniques used in operating systems, PagedAttention departs from traditional attention algorithms by enabling the storage of key-value (KV) data in non-contiguous memory locations.

A diagram of a key and value

Description automatically generated

Figure 16: Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory (source: [24])

In vLLM, PagedAttention structures the KV cache for each sequence into smaller units called KV blocks. These blocks hold key and value vectors for a predetermined number of tokens. During the attention computation process, PagedAttention retrieves and operates on individual KV blocks independently. This allows for the distribution of key and value vectors across non-contiguous memory segments, as illustrated in figure 16. The core advantage of PagedAttention lies in its ability to store KV blocks in a non-contiguous manner. This flexibility empowers vLLM to implement a more efficient paged memory management strategy, ultimately enhancing its overall performance [24].

1. **Proposed Solution And Development Methodology**
   1. **Project Overview**

A diagram of a process

Description automatically generated

Figure 3-1: High level project overview

Figure 3-1 displays a very high level overview of the project development phase. We can further group these parts into three main sections:

* Data gathering
* Base Model Finetuning
* LLM Evaluation
* Mobile Application Development

In the following chapters, we will dive into each of these phases in order to take a more granular look into the overview and implementation details for them.

* 1. **Data Gathering**

During this phase, we try to gather high quality Romanian data for pretraining and finetuning. In this respect, we would need to find a Romanian corpus of text for the pretraining process and a Romanian instruct dataset for the supervised finetuning phase. However, since there are practically no such datasets available in the Romanian language at this moment, we are tasked with creating these on our own.

Figure 3-2 showcases the process of gathering qualitative Romanian text for constructing a text corpus to be used in the pretraining phase, as well as the process of acquiring, processing and translating an instruction dataset to serve in the finetuning phase of our implementation.

A diagram of a process

Description automatically generated

Figure 3-2: Data Gathering and Processing Overview

* 1. **Base Model Finetuning**

A diagram of a diagram

Description automatically generated

Figure 3-2: Finetuning Process Overview

From figure 3-2, we can observe two approaches when it comes to finetuning a foundational model for Romanian language.

The first approach consists of a single supervised finetuning step on the Romanian instruction dataset created in the previous phase. During this step, we establish our finetuned model’s prompt and optionally a system message (we will discuss more of these in the implementation chapter).

The second approach introduces an additional step before the supervised finetuning, namely a pretraining step, responsible with exposing the base model to as much qualitative Romanian data as possible, before engaging in the actual finetuning. We will compare the results of these approaches in another chapter.

* 1. **Finetuned LLM Evaluation**

\*TO BE WRITTEN\*

* 1. **Mobile Application Development**

In order to showcase the functionality of the created Romanian Large Language Model, I have worked with 2 other colleagues, Vlad Sârbu and Diana Cernăzanu, in order to create a chatbot mobile application, to provide the user with a friendly UI to facilitate the communication with the LLM. As an extra feature, the user can either send their prompt in a traditional manner, by writing the prompt and sending it to the model, or, using Vlad’s finetuned speech-to-text Whisper model, they can speak their prompt which will then be converted into text and sent to the large language model. In this paper, I will only present and refer to the aspects that I have done regarding this shared mobile application project.

For the application we have decided to use a MERN stack. MERN stands for MongoDB, Express, React (Native) and Node. Altough MERN is typically a web application stack, we can easily leverage the advantages of this popular stack, with some minor changes for our mobile application.

MongoDB is a NoSQL database management system that utilizes a document-oriented data model. Unlike relational databases that store data in fixed tables with predefined schemas, MongoDB stores data in flexible, JSON-like documents. These documents, known as Binary JSON (BSON) documents, offer inherent flexibility as their structure can vary from one document to another. This characteristic makes MongoDB well-suited for storing and managing data that exhibits inherent heterogeneity [26].

Express.js is a popular application framework designed to streamline the development of web applications and APIs within the Node.js environment. It offers a comprehensive set of functionalities that make it easier for developers to construct applications, such as routing, middleware, error handling, integration with Node.js, etc [27].

React Native is an open-source framework for building native mobile apps using JavaScript and React.js. Key features and concepts include: cross-platform development for iOS and Android, JavaScript and React.js for web development skills, native components for UI development, hot reloading for faster development, native modules for accessing platform-specific APIs, active community and ecosystem

Node.js is a JavaScript runtime environment that allows developers to run JavaScript code on the server-side. It is suitable for building scalable and efficient server-side applications, APIs, and microservices that require high concurrency and performance [29].

* 1. **Development tools**

Jupyter Notebook offers an environment for creating interactive documents that combine code, mathematical expressions, visualizations, multimedia elements, and the resulting computational outputs. It is a popular tool, widely used in data science and machine learning projects [30].

Visual Studio Code (VS Code) is a popular source code editor created by Microsoft. It is free and open-source, it has cross-platform compatibility, it is lightweight and versatile and it offers access to various extensions [31].

Unsloth is an open-source software that significantly speeds up large language model training and finetuning and optimizes VRAM memory usage [32].

Expo is an open-source platform that facilitates the development and deployment of React Native applications. By providing a suite of tools, libraries, and services, Expo simplifies the development process and the deployment of mobile apps [33].

Expo Go is a companion app for Expo developers, available for both Android and iOS. It allows developers to test and preview the app on their mobile devices, by scanning the QR code generated by the Expo command line interface (Expo CLI).

Google Cloud Compute Engine is a cloud-based Infrastructure-as-a-Service (IaaS) offering from Google Cloud Platform (GCP). Virtual Machine (VM) instances in Google Cloud Compute Engine are virtualized computing environments that run on Google's infrastructure. We use such a VM instance in order to run the application’s Express server [34].

* 1. **Mobile Application Architecture**

A diagram of a cell phone

Description automatically generated

Figure 3-3: Architecture Overview

Figure 3-3 showcases the main architectural components of the mobile application.

On the client side, we have a React Native frontend which will provide a friendly and interactive user interface. From this frontend we will make requests to the three servers based on the action that we want to perform.

Using Google Cloud’s Compute Engine, we will be running the Express.js server so that it can be accessed remotely from anywhere. This server is responsible for handling requests related to user logic such as authentication, profile updating, creating and updating conversations, account deletion etc.

On the VM, two servers will be running, the Whisper Model backend and the large language model server using vLLM. From the frontend, the LLM server will send the prompt and expect the generated response so that it can be displayed to the user.

All the communication is tied together using the MongoDB Atlas database, where we store information about the users and the conversations.

We will dive deeper into these aspects during the implementation chapter of this paper.

* 1. **Work Breakdown Structure**

The Work Breakdown Structure (WBS) is a project management tool that systematically breaks down a project into smaller, manageable elements, facilitating effective planning, progress monitoring, and resource allocation [25].

A diagram of work breakdown structure

Description automatically generated

Figure 3-4: Work Breakdown Structure

Figure 3-4 displays the Work Breakdown Structure for this project, from the VM configuration, continuing with the data management, LLM environment for finetuning, mobile app development, testing and documentation of the whiole process.

1. **Implementation**

**Bibliography**

[1] Petroșanu, Dana-Mihaela, Alexandru Pîrjan, and Alexandru Tăbușcă. "Tracing the Influence of Large Language Models across the Most Impactful Scientific Works." Electronics 12.24 (2023): 4957.

[2] <https://openai.com/research/gpt-4>

[3] Team, Gemini, et al. "Gemini: a family of highly capable multimodal models." arXiv preprint arXiv:2312.11805 (2023).

[4] <https://ai.meta.com/blog/meta-llama-3/>

[5] Jiang, Albert Q., et al. "Mixtral of experts." arXiv preprint arXiv:2401.04088 (2024).

[6] Ono, Kensuke, and Akira Morita. "Evaluating Large Language Models: ChatGPT-4, Mistral 8x7B, and Google Gemini Benchmarked Against MMLU." Authorea Preprints (2024).

[7] GenAI, Meta. "Llama 2: Open foundation and fine-tuned chat models." arXiv preprint arXiv:2307.09288 (2023).

[8] Grossberg, Stephen. "Recurrent neural networks." Scholarpedia 8.2 (2013): 1888.

[9] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[10] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[11] Muñoz, Eduardo. "Attention is all you need: Discovering the Transformer paper." Towards Data Science (2020).

[12] Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." arXiv preprint arXiv:1406.1078 (2014).

[13] Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[14] Peter Bloem, "Transformers from scratch" blog post, 2019.

[15] Jayaseelan, Nivas. "LLaMA 2: The New Open Source Language Model."

[16] Roumeliotis, Konstantinos I., Nikolaos D. Tselikas, and Dimitrios K. Nasiopoulos. "Llama 2: Early Adopters' Utilization of Meta's New Open-Source Pretrained Model." (2023).

[17] Chowdhery, Aakanksha, et al. "Palm: Scaling language modeling with pathways." Journal of Machine Learning Research 24.240 (2023): 1-113.

[18] Anil, Rohan, et al. "Palm 2 technical report." arXiv preprint arXiv:2305.10403 (2023).

[19] Jiang, Albert Q., et al. "Mistral 7B." arXiv preprint arXiv:2310.06825 (2023).

[20] Lai, Viet Dac, et al. "Okapi: Instruction-tuned large language models in multiple languages with reinforcement learning from human feedback." arXiv preprint arXiv:2307.16039 (2023).

[21] Zhao, Jun, et al. "Llama beyond english: An empirical study on language capability transfer." arXiv preprint arXiv:2401.01055 (2024).

[22] Cui, Yiming, Ziqing Yang, and Xin Yao. "Efficient and effective text encoding for chinese llama and alpaca." arXiv preprint arXiv:2304.08177 (2023).

[23] Huang, Yuzhen, et al. "C-eval: A multi-level multi-discipline chinese evaluation suite for foundation models." Advances in Neural Information Processing Systems 36 (2024).

[24] Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." Proceedings of the 29th Symposium on Operating Systems Principles. 2023.

[25] Tausworthe, Robert C. "The work breakdown structure in software project management." Journal of Systems and Software 1 (1979): 181-186.

[26] <https://www.mongodb.com/resources/languages/mern-stack>

[27] <https://expressjs.com/>

[28] <https://reactnative.dev/>

[29] <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>

[30] <https://jupyter.org/>

[31] <https://code.visualstudio.com/docs>

[32] <https://unsloth.ai/>

[33] <https://expo.dev/>

[34] <https://cloud.google.com/compute/docs>