Intel Unnati Industrial Training Program 2024

PROBLEM STATEMENT -04

Introduction to GenAI and Simple LLM Inference on CPU and finetuning of LLM Model to create a Custom Chatbot

PROJECT REPORT

Team: 5_Nearest_Neighbours

Team members:

Deependra Singh Rao

Janet Jomy

M Pooja

Nandana Krishnan S

Prashanth R J

July 2024

Abstract

This project investigates the enhancement and refinement of a CPU-powered chatbot through the utilization of two primary approaches: Retrieval-Augmented Generation (RAG) and Fine tuning. The creation of the chatbot involves incorporating advanced capabilities like inputting text, processing PDF documents and using RAG to enhance conversational responses to facilitate user interaction and engagement.

Fine-tuning is essential for improving the chatbot's effectiveness after it has been developed. This step requires fine-tuning model parameters and training on particular datasets to improve precision and agility. Evaluation metrics like loss, perplexity and runtime are used as benchmarks to evaluate performance in both methodologies.

RAG strengthens its base by utilizing external information sources for responses that are relevant to the context, and then improves these abilities through data-driven optimization. Reduced loss and perplexity metrics indicate better predictive accuracy and conversational coherence, essential for successful user interactions. Runtime metrics measure efficiency on CPU platforms, ensuring the best possible performance for real-time applications.

This summary focuses on the iterative process of creating a CPU-based chatbot, showcasing the ongoing improvements informed by evaluation metrics. It demonstrates how the combination of RAG for the beginning setup and fine-tuning for optimization can work together to improve user experience and operational efficiency in real-world scenarios.

Introduction

The demand for intelligent, responsive chatbots that can comprehend and communicate with consumers across multiple domains is rising in the contemporary digital world. These chatbots must be extremely configurable to meet specific business requirements and run efficiently on readily available technology, such as CPUs.

Businesses and individuals in today's digital world are depending more and more on chatbots for a variety of purposes, such as personal assistants and customer support. These chatbots must be perceptive, clever and able to comprehend a range of user inputs.

CPU based chatbots can be advantageous in various contexts due to reasons such as cost efficiency and accessibility. Small scale organizations in need of chatbots can prefer using CPU taking their limited budget into account. Availability of CPUs in most computing devices, even personal laptops, is quite helpful. Moreover, while GPUs offer superior performance for highly demanding tasks, CPUs provide a more cost effective and accessible option for deploying chatbots.

Nevertheless, there are a lot of obstacles in the way of developing such effective chatbots, especially when it comes to the computational power needed. The core of contemporary chatbots is Large Language Models (LLMs), which usually require a significant amount of computing power, frequently requiring the usage of GPUs. This can be a significant obstacle for a lot of apps that require regular CPU hardware to function.

Objectives:

- Develop a custom chatbot using Llama 3 models that can perform efficiently on CPUs.
- Utilize Intel's extension libraries to optimize the performance of transformer models on Intel CPUs.
- Finetune the chatbot model using the Alpaca dataset to improve its contextual understanding and response accuracy.
- Implement a Retrieval-Augmented Generation (RAG) approach for effective real-time information retrieval.
- Design an intuitive and user-friendly interface using Streamlit, allowing users to upload PDFs, input queries, and receive responses effortlessly.
- Evaluate the performance and effectiveness of this approach to determine best practices for building customizable, high-performing chatbots on CPU hardwares.

Challenges

- Large Language Models (LLMs) usually need a lot of processing power, and GPUs are often needed for best results. For many applications that require regular CPU hardware to function, this could be a barrier.
- To create a chatbot that understands and replies to domain-specific queries, LLMs must be fine-tuned based on relevant facts. This procedure must be quick and easy to follow in order to guarantee that the model fits the particular use case perfectly.
- Chatbots require real-time retrieval of relevant information from multiple sources, including papers and databases, to be genuinely useful. This necessitates the inclusion of powerful retrieval methods within the language model.

Methodology:

RAG Approach:

To develop a custom chatbot, we implemented the retrieval approach: integrating Retrieval Augmented Generation (RAG) using the Llama 3 model of 8 billion parameters. For this purpose, a curated dataset of questions and answers related to **Intel technologies (services and products)** was used. The data contains **39 queries and their expected responses**, with all the information collected from the internet.

The method of RAG basically implements indexing, retrieval and generation. The process of indexing begins with the extraction and cleaning of raw data from a variety of forms, including Word, HTML, PDF, and Markdown. The data is then transformed into a standard plain text format. Text is divided into more manageable sections, known as chunks, in order to satisfy language models' constraints about context. After that, chunks are kept in vector databases after being converted into vector representations using an embedding model. Here we have used FAISS to store and retrieve chunks. In order to provide effective similarity searches during the retrieval phase that follows, this step is essential.

Retrieval: When the RAG system receives a user query, it converts it into a vector representation using the same encoding methodology that was used for indexing. The similarity scores between the query vector and the chunk vector within the indexed corpus are then calculated. The top K chunks that show the highest resemblance to the query are retrieved by the system after it has prioritized them. These sections are then utilized in prompt as the enlarged context.

Generation: A big language model is tasked with creating a response to a logical prompt that is created from the stated question and chosen documents. Depending on task-specific criteria, the model's method of responding can change, enabling it to either use its built-in parametric knowledge or limit its answers to the data in the given documents. If a dialogue is already in progress, it is possible to incorporate any previous conversations into the prompt, which will allow the model to participate in multi-turn dialogue interactions efficiently.

RAG combines the merits of retrieval-based and generative models, resulting in accuracy and contextual relevance. While it can be applied to various natural language processing tasks making it versatile, its merits include demonstrating a precise understanding of prompts and helping reduce bias and spread of misinformation as compared to purely generative models.

As compared to fine tuning, RAG systems are less likely to create details that aren't factual, which makes them more reliable. Moreover, RAG offers a higher level of transparency.

In this approach, the generative model is built using Groq, which is a Python-based open-source framework for interacting with and querying graph data structures. It gives programmers an easy-to-use and expressive approach to interact with graphs, enabling them to query, filter, and aggregate data among other activities. Working with large-scale graph data, like that found in recommender systems, knowledge graphs, and social networks, is where Groq excels.

The generative model was the end result of breaking documents into chunks, storing and retrieving them using FAISS and embedding the text data. A user interface for interactive testing was created using Streamlit, through which the performance of the model was evaluated on three different systems of different specifications of processors and RAM size. While the model was able to curate responses with high precision, the increased time for retrieval and the need for more powerful hardware was demanding.

Fine tuning approach:

Fine tuning is a significant capability that can boost the suitability of the large language model for your specific use case. By training the model with a task specific data set, it facilitates a pretrained model to accomplish better on that specific domain. Since it equips itself with each and every nuance of the data, the accuracy and relevance are immeasurably improved.

As for this approach, we developed the chatbot using the **Llama 3 model with 8B parameters** and then fine tuning it with a **custom data set extracted from the Alpaca data set**. This process involved using the Intel Extension for Transformers to enhance the performance and efficiency of transformer models on Intel CPUs.

A chatbot was constructed with mixed precision optimization to improve performance. The fine-tuning process involved optimizing the model with 3 epochs and a batch size of 4 per device, using the reduced Alpaca dataset with 1% validation split.

The fine-tuned model was evaluated using metrics such as epoch, loss, perplexity, runtime, and processing rates. The model achieved a loss of 2.9922 and a perplexity of 19.96, demonstrating improved performance.

Process Flow:

RAG Approach

1) Environment Setup: Set up dependencies (Streamlit, LangChain, PyPDF2, FAISS, dotenv) and import environment variables.
2) Define Functions: Create functions to:
   * Perform Information extraction on texts and documents with the use of pdfminer.six and PyPDF2.
   * Character Text splitter (Split the text into chunks that can be penetrated by the human mind effortlessly).
   * Design the vector store (FAISS, OllamaEmbeddings)
3) To do so, the user must set up a conversational chain, or a LangChain or RAG. Handle user input.
4) Initialize Streamlit App: Designate title and icon of the app as well as its layout, and other session state related parameters.
5) Build User Interface: With these it entails including features such as the header section, text input bar, chat history viewer, and PDF uploading feature.
6) Process PDFs: Summarize text, process it on a sentence level, build a vector store, and start the conversational retrieval chain with Llama 3, also known as ChatGroq and RAG.
7) Handle Queries: Handle user inputs with the conversational chain and update the chat history: LangChain, ConversationBufferMemory.
8) Deploy and Test: Install the app to a relevant device/assets and do the functionality testing of the selected app.

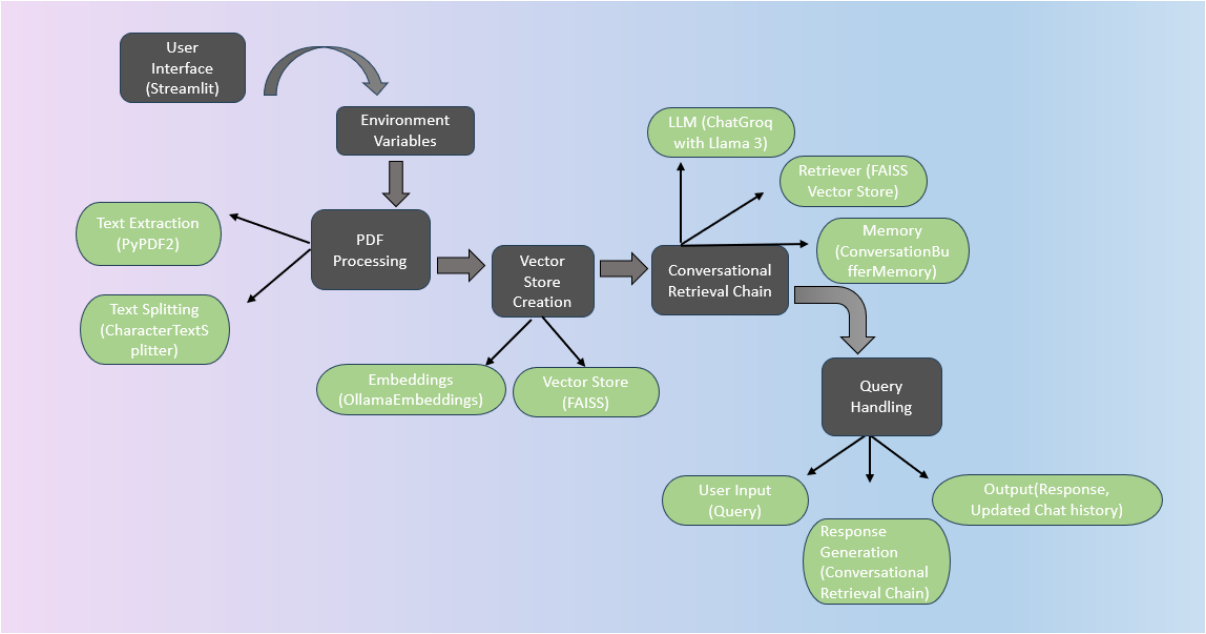
Fine tuning approach

1) Environment Setup: Install and import necessary libraries and packages. Add the intel_extension_for_transformers library for enhancing optimization functions in transformer models and then import the needed functions and classes.
2) Set up the chatbot:
   - Instantiate a configuration object (PipelineConfig) and configure it to utilize mixed precision optimization (MixedPrecisionConfig).
   - Construct the chatbot utilizing the provided configurations.

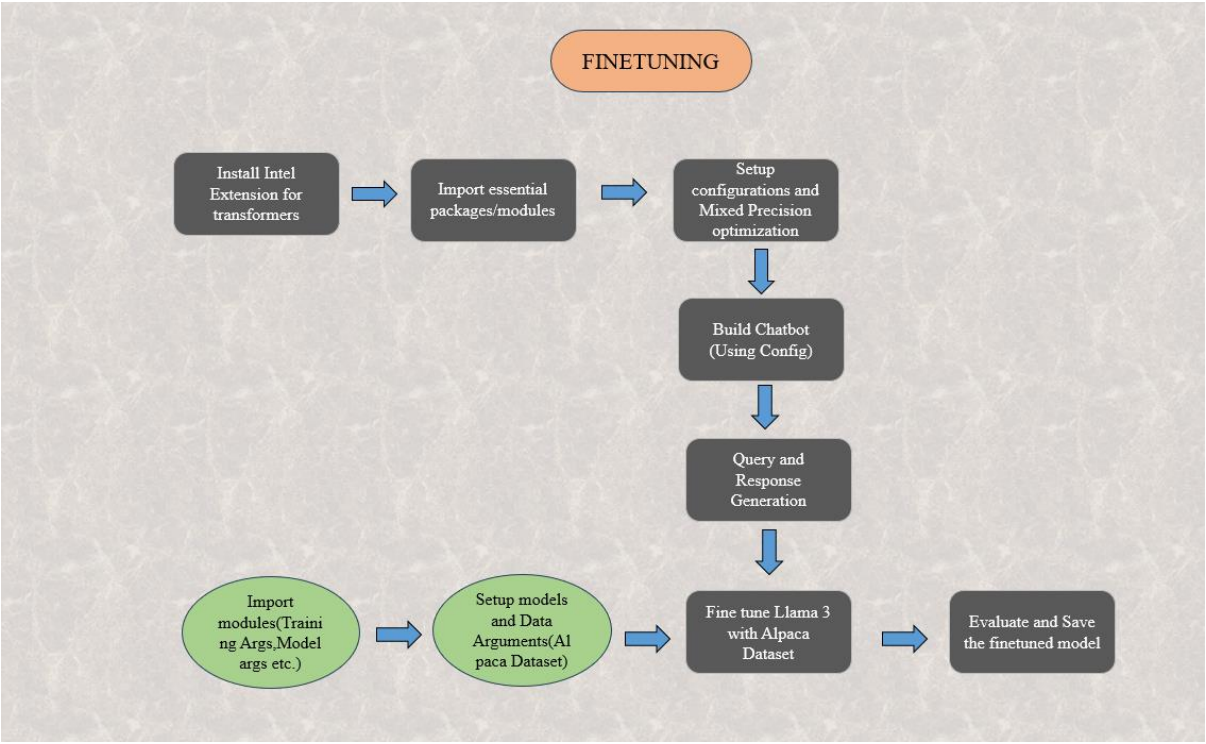
3) Make predictions and display the responses: Utilize the constructed chatbot to handle and answer various inquiries. The output of chatbot's responses for every question is provided to assess its performance and the accuracy of its replies.
4) Importing necessary modules: Import required modules (TrainingArguments, ModelArguments, DataArguments, FinetuningArguments, TextGenerationFinetuningConfig) containing necessary classes. Also import the finetune_model to carry out fine tuning.
5) Configuration:
    - Start the ModelArguments with the model path "meta-llama/Meta-llama-3-BB"
    - Create DataArguments for training with "reduced_alpaca_data_corrected.json" and splitting 1% for validation.
    - Set up TrainingArguments with specified output directory, 3 epochs, batch sizes and logging.
6) Fine tuning process:
    - Merge arguments in TextGenerationFinetuningConfig
    - Implement fine tuning with the help of finetune_model
7) Training process: Begin with 163 instances for training, utilizing 3 epochs, with a batch size of 4 per device and a total batch size of 8. Also implement 60 optimization steps along with 2 gradient accumulation steps. Utilize half precision (amp) for training on CPU.
8) Evaluation and metrics: Assess with 2 instances, batch size of 4. The metrics consists of epoch (2.93), loss (2.9922), perplexity (19.96), runtime (31.38 seconds) and processing rates (~0.064 samples per second)
9) Completion: Finish training successfully. Store model checkpoint and tokenizer configuration in the ./tmp folder.

Architecture Diagram

1) RAG approach



2) Fine tuning



Results

- The results for finetuning on the utilized CPU:
  - **Finetuning Time:** 8 mins 19 secs
  - **Average response time:** 1 min 13 secs.

- The RAG results for different CPUs are listed below:

System 1	System 2	System 3
Processor: AMD Ryzen 7 5800U with Radeon Graphics 1.90 GHz **Installed RAM:** 16.0 GB (15.3 GB usable) **System type:** 64-bit operating system, x64-based processor	**Processor**: 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz **Installed RAM:** 16.0 GB (15.7 GB usable) **System type:** 64-bit operating system, x64-based processor	**Processor**: AMD Ryzen 5 5600H with Radeon Graphics 3.3 GHz **Installed RAM:** 8.0 GB (7.34 GB usable) **System type:** 64-bit operating system, x64-based processor
Average Processing Time: 315.15 secs **Average Response Time:** 4.31 secs	**Average Processing Time:** 46.32 secs **Average Response Time:** 4.9 secs	**Average Processing Time:** 88.46 secs **Average Response Time:** 4.54 secs

Conclusion

- The overall time for finetuning and the response generated exceeded 10 mins in case of a very small dataset indicating the disadvantage of finetuning over RAG.
- On the basis of response time, the performance of RAG model was found to be better than that of finetuned model.
- The Average Response Time of 4.31 seconds was found to be the minimum for AMD Ryzen 7 5800U.
- The Average Processing Time of 46.32 seconds was found to be the minimum for 12th Gen Intel(R) Core(TM) i5-12450H.
- The overall performance of the Intel CPU was found to be the best with a very low processing time and a decent response time compared to the other systems ensuring great performance.

References

1. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997.
2. Ovadia, O., Brief, M., Mishaeli, M., & Elisha, O. (2023). Fine-tuning or retrieval? comparing knowledge injection in llms. *arXiv preprint arXiv:2312.05934*.
3. Pandya, K., & Holia, M. (2023). Automating Customer Service using LangChain: Building custom open-source GPT Chatbot for organizations. arXiv preprint arXiv:2310.05421.
4. Alghisi, S., Rizzoli, M., Roccabruna, G., Mousavi, S. M., & Riccardi, G. (2024). Should We Fine-Tune or RAG? Evaluating Different Techniques to Adapt LLMs for Dialogue. arXiv preprint arXiv:2406.06399.
5. Dong, G., Zhu, Y., Zhang, C., Wang, Z., Dou, Z., & Wen, J. R. (2024). Understand What LLM Needs: Dual Preference Alignment for Retrieval-Augmented Generation. arXiv preprint arXiv:2406.18676.
6. Basilico, M. (2024). Design, Implementation and Evaluation of a Chatbot for Accounting Firm: A Fine-Tuning Approach With Two Novel Dataset (Doctoral dissertation, Politecnico di Torino).