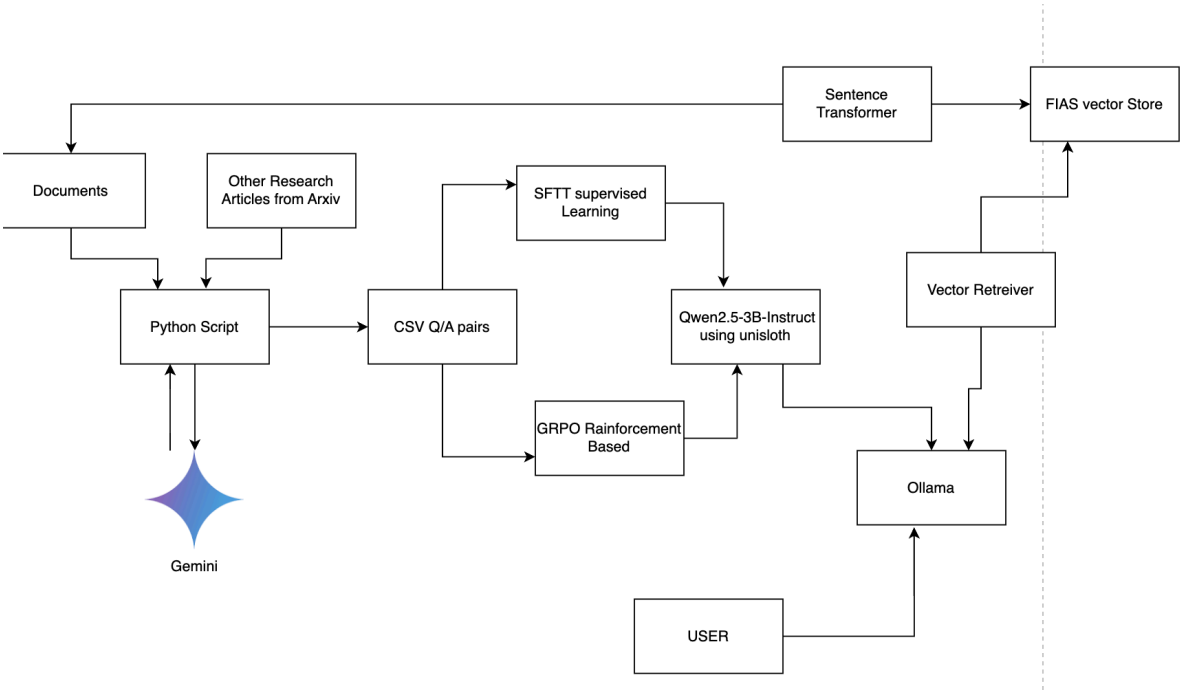
Team Duo Dynamics-Fine-Tuning Qwen 2.5 3B Instruct for AI Research Question Answering

Model in gguf Format :
https://drive.google.com/file/d/1-clLRALKxWHuSCrJcHTz0J2fpznHrk59/view?usp=sharing

1. Introduction

This report details the process of our approach of fine-tuning the Qwen 2.5 3B Instruct model to enhance its ability to answer questions related to recent AI research papers, documents. The goal was to create a specialized model capable of accurately retrieving, interpreting, and generating responses from technical AI literature, as per the requirements of the LLM Fine-tuning Challenge. Our Entire process of creating this agentic AI workflow can be explained using this digram



1. Model Selection

- **Base Model:** Qwen 2.5 3B Instruct was selected as the backbone model due to its strong performance in instruction following and its suitability for resource-constrained environments (like Google Colab). The "Instruct" version was preferred over the base model as it is already pre-trained for conversational tasks, aligning better with the question-answering objective.
- **Embedding Model (`sentence-transformers/all-mpnet-base-v2`):** As part of the optional RAG implementation, a pre-trained sentence transformer model (e.g.,

`sentence-transformers/all-mpnet-base-v2`) was chosen for its balance of performance and efficiency. This model was not fine-tuned, as it demonstrated sufficient accuracy for the retrieval task.

3. Data & Dataset Creation

The provided research papers(markdowns and PDFs) formed the core of the dataset.

Synthetic Dataset Generation:

A combination of manual and automated approaches was used to generate a training dataset.

Manual Creation

Key sections of the research papers were manually reviewed to identify important concepts and relationships. Based on these insights, question-answer pairs were then created.

Automated Generation (using RAG)

The provided papers were chunked and indexed using a vector database (e.g., FAISS). Then, a set of diverse questions was generated, and the RAG system was used to retrieve relevant context and generate corresponding answers. This allowed for the creation of a larger, more varied dataset.

Automated Generation using a superior LLM

to augment the dataset, a significant portion of question-answer pairs was generated leveraging the capabilities of Google's Gemini, a superior large language model. The provided research papers and documents were ingested as plain text files, and the Gemini API was utilized to systematically generate diverse and complex question-answer pairs. This approach offered several key advantages:

- Gemini's ability to process and understand complex textual information allowed for the creation of a large volume of varied questions, capturing nuances and relationships within the research papers that might have been missed with manual or simpler automated methods.
- By providing the full text of the documents, Gemini was able to generate questions and answers that were deeply rooted in the context of the source material, ensuring the relevance and accuracy of the generated data.
- The API-driven approach significantly accelerated the dataset creation process, allowing for rapid iteration and refinement of the training data.
- Gemini was also able to generate complex questions requiring multi-step reasoning, or the synthesis of information across several paragraphs, increasing the difficulty and the quality of the training data.

Data Preprocessing:

- Text cleaning: Removal of unnecessary characters, HTML tags, and special symbols.
- Tokenization: The text was tokenized using the Qwen 2.5 3B tokenizer.
- Context window management: Input sequences were truncated or padded to fit within the model's context window.

Dataset Split: The generated dataset was split into training and testing sets, with the test set reserved for evaluating the model's performance.

4. Training Process

Fine-Tuning Methodology:

LoRA (Low-Rank Adaptation): LoRA was chosen for its ability to efficiently fine-tune large language models by injecting trainable low-rank matrices into key transformer layers while keeping the pre-trained weights frozen. This approach significantly reduces memory consumption and computational cost.

Hyperparameters:

- **lora_r:** 16 (rank of the LoRA matrices)
- **lora_alpha:** 16 (scaling factor for LoRA updates)
- **lora_dropout:** 0 (no dropout applied)
- **target_modules:** ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"] (LoRA applied to key transformer layers)
- **bias:** "none" (no additional biases are trained)
- **use_gradient_checkpointing:** "unsloth" (reduces memory usage)
- **random_state:** 3407 (ensures reproducibility)
- **use_rslora:** False (RS-LoRA is disabled)
- **loftq_config:** None (LoRA quantization is not used)

LoRA was selected to enable efficient fine-tuning while keeping computational requirements low. The chosen hyperparameters balance model adaptability and efficiency, ensuring optimal performance without excessive memory usage. The dropout rate was set to zero to fully utilize available data, and gradient checkpointing was enabled to reduce GPU memory consumption.

Why We Used LoRA Instead of QLoRA

We chose to use **LoRA (Low-Rank Adaptation)** instead of **QLoRA (Quantized LoRA)** due to several key advantages that prioritize performance and efficiency. While QLoRA is designed to reduce memory consumption through 4-bit quantization, it may introduce slight precision loss and computational overhead when performing dequantization operations. Given our goal of achieving high performance while maintaining efficiency, LoRA we found lora give more accurate results(Hu et al. 2021)

Performance Advantages of LoRA Over QLoRA

1. **Higher Numerical Precision**
    - LoRA operates in full precision (FP16/BF16), ensuring better retention of model accuracy and reducing the risk of performance degradation caused by quantization errors.
2. **Better Model Adaptability**
    - Since LoRA does not require dequantization during training, it allows for smoother weight updates, resulting in potentially better fine-tuning quality compared to QLoRA.
3. **Avoiding Quantization Overhead**
    - QLoRA requires additional computational steps to manage quantized weights, which may introduce inefficiencies when running on free-tier GPUs like those in Colab and Kaggle.
4. **Compatibility with Gradient Checkpointing**
    - The use of `use_gradient_checkpointing = "unsloth"` is optimized for full-precision LoRA and may not work as effectively with QLoRA's quantized weights.
5. **Simplicity in Implementation**
    - LoRA avoids dependencies like `bitsandbytes`, making it easier to implement and debug, especially when working with cloud-based environments where package compatibility can be an issue.

Hardware Constraints and Full Fine-Tuning Limitations

While **full fine-tuning** would have provided the best possible adaptation, it was not feasible due to the limited computational resources available on free platforms that we have used which was **Google Colab and Kaggle**. These environments do not offer the necessary GPU memory to fully train large-scale models. Instead, LoRA was used to efficiently fine-tune specific layers while keeping the pre-trained model weights frozen, significantly reducing the memory footprint and computational requirements.

By choosing LoRA over QLoRA, we were able to achieve **high-performance fine-tuning while ensuring numerical stability and avoiding unnecessary quantization complexities**. This approach allowed us to make the most of the available resources while maintaining optimal model performance.

5. Quantization and Inference

- **Quantization:** The fine-tuned model was quantized to 4-bit using the `llama.cpp` library to create a `.gguf` file. This significantly reduced the model's size, enabling efficient inference on CPU-based systems.
- **Inference Script:** A Python script was developed to load the quantized `.gguf` model using `llama-cpp-python` and perform inference. The script included the following steps:

    - Loading the quantized model.
    - Tokenizing the input question.
    - Generating the response using the model.
    - Decoding the generated tokens.

7. RAG Implementation

- **Vector Database:** FAISS was used as the in-memory vector database for storing and retrieving document embeddings. FAISS makes indexing using **HNSW (Hierarchical Navigable Small World Graphs)** and **IVF (Inverted File Indexing)**, making searches highly efficient
- **Embedding Model:** `sentence-transformers/all-mpnet-base-v2` was used to generate embeddings for the document chunks.
- **Retrieval Process:** The input question was embedded using the same embedding model, and the top-k most similar document chunks were retrieved from the vector database. These retrieved chunks were then concatenated with the question and passed to the fine-tuned model for answer generation.

8. Conclusion

The fine-tuning process successfully enhanced the Qwen 2.5 3B Instruct model's ability to answer questions related to AI research papers. LoRA proved to be an effective technique for fine-tuning the model on limited resources. The RAG implementation further improved the model's performance by providing relevant context for answer generation. Future work could explore the use of reinforcement learning techniques to further improve the model's reasoning capabilities and alignment with human preferences. Additionally, more robust dataset generation techniques could be employed to create a larger and more diverse training dataset.

All the code belongs to the project and the instructions for running the results are put in the the github repo

https://github.com/Team-Duo-Dynamics/Intellihack_DuoDynamics_03

Hu, Edward J., Yelong Shen, Phillip Wallis, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "LoRA: Low-Rank Adaptation of Large Language Models." *ArXiv*, (2021). Accessed March 9, 2025. https://arxiv.org/abs/2106.09685.