INTELLIHACK 5.0

Task 03 - Technical report

Team Cognic AI

Model Selection

For this project, we had to choose between **Qwen/Qwen2.5-3B (Base)** and **Qwen/Qwen2.5-3B-Instruct**. We selected **Qwen2.5-3B-Instruct** because it is fine-tuned specifically for instruction-following tasks. Unlike the base model, which is a raw pretrained language model without specific task adaptation, the instruct variant is optimized to generate more coherent and contextually relevant responses when given prompts in a conversational or instructional format. This makes it more suitable for applications that require structured responses, reasoning, and following specific guidelines.

To load the model efficiently, we used **FastLanguageModel**, which provides optimized methods for handling large models. This was particularly useful when setting load_in_4bit, allowing us to load the model in **4-bit quantization** to reduce memory usage while maintaining performance. By merging **LoRA (Low-Rank Adaptation) weights** with the base model, we further improved efficiency. Using **FastLanguageModel** helped ensure that the model was loaded efficiently without exceeding GPU memory limitations, making it more accessible for training and fine-tuning on our available hardware.

For saving and sharing the model, we used **GGUF format with q4_k_m quantization**, which helps reduce storage requirements while keeping a good balance between efficiency and accuracy.

Data Extraction & Dataset Creation

The first step in our dataset creation process was converting all relevant research papers into `.md` **(Markdown) format** for easy processing. One of the key papers, **DeepSeek-R1**, was originally in **PDF format**, which made direct text extraction challenging. While we could have used **OCR (Optical Character Recognition)** to extract plain text, research papers often contain **tables, equations, and graphs** that provide critical insights beyond just text.

To handle this, we leveraged **Agentic Document Extraction**, a tool built on **VisionAgent by Landing AI**, which was released recently. This allowed us to extract structured and meaningful information from the research papers, including formatted tables.

For example, below shows how a table benchmark comparison between **DeepSeek-R1-Zero** and OpenAI models was extracted meaningfully.

<table> <tr> <th>Model</th> <th colspan="2">AIME 2024</th> <th>MATH-500</th> <th>GPQA Diamond</th> <th>LiveCode Bench</th> <th>CodeForces</th> </tr> <tr> <td></td> <td>pass@1</td> <td>cons@64</td> <td>pass@1</td> <td>pass@1</td> <td>pass@1</td> <td>rating</td> </tr> <tr> <td>OpenAI-o1-mini</td> <td>63.6</td> <td>80.0</td> <td>90.0</td> <td>60.0</td> <td>53.8</td> <td>1820</td> </tr> <tr> <td>OpenAI-o1-0912</td> <td>74.4</td> <td>83.3</td> <td>94.8</td> <td>77.3</td> <td>63.4</td> <td>1843</td> </tr> <tr> <td>DeepSeek-R1-Zero</td> <td>71.0</td> <td>86.7</td> <td>95.9</td> <td>73.3</td> <td>50.0</td> <td>1444</td> </tr> </table>

Generating a Q&A Dataset

Once we had the `.md` files, the next step was creating a **Q&A dataset** for fine-tuning the model. Instead of manually crafting questions and answers, we used **RAGAS** with our **OpenAI API key** to generate a dataset automatically. Due to API costs, we limited the dataset size to **~500 samples** to balance quality and efficiency.

Here's an example of how we generated the dataset:

```python
def generate_Q_A_testset(docs, testset_size):
    generator_llm = LangchainLLMWrapper(ChatOpenAI(model="gpt-4o-mini"))
    generator_embeddings = LangchainEmbeddingsWrapper(OpenAIEmbeddings())

    generator = TestsetGenerator(llm=generator_llm, embedding_model=generator_embeddings)
    dataset = generator.generate_with_langchain_docs(docs, testset_size=testset_size)

    return dataset
```

Converting to Hugging Face Dataset

To efficiently fine-tune the model, we converted our dataset into the **Hugging Face dataset format**, which allows seamless integration with training pipelines. After conversion, we split the dataset into **80% training and 20% test sets** to ensure we had a separate evaluation set for future experiments.

Training Process

Hyperparameters and Justifications

We used the `SFTTrainer` from `trl` for fine-tuning while leveraging `Unsloth` for memory-efficient training. The following hyperparameters were chosen:

Batch Size & Gradient Accumulation: Small batch size (**2**) with gradient accumulation (**4**) to optimize memory use.

Learning Rate & Optimizer: Set to **1e-4** with **AdamW 8-bit optimizer** for stability and efficiency.

Precision & Hardware Optimization: Used **BF16 or FP16** depending on GPU support for faster training.

Regularization & Scheduling: Applied **0.01 weight decay** and a **cosine learning rate scheduler** to prevent overfitting.

Warmup Steps & Epochs: 5 warmup steps to stabilize early training and trained for **4 epochs** for balance.

To optimize training, we used **SFTTrainer** with **multi-processing (8 workers)** for faster tokenization. **Packing was disabled** to keep long sequences intact, improving model understanding.

During training, we **logged loss at every step** but initially faced an issue where **validation loss was missing**, which was later fixed.

```
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    data_collator=DataCollatorForSeq2Seq(tokenizer=tokenizer),
    dataset_num_proc=8,
    packing=False,
    args=TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=5,
        num_train_epochs=4,
        learning_rate=1e-4,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        logging_steps=1,
        optim="adamw_8bit",
        weight_decay=0.01,
        lr_scheduler_type="cosine",
        seed=3407,
        output_dir="outputs",
        report_to="none",
        evaluation_strategy="epoch",
    ),
)
```

Tokenization & Prompt Formatting

- Used `get_chat_template` from `Unsloth` to apply the correct **Qwen2.5 chat format**.
- Standardized dataset using `standardize_sharegpt` to match structured conversation styles.
- Ensured each conversation in the dataset followed a proper **user-assistant exchange format** before tokenization.

```
dataset[5]["conversations"]
```

```
[{'content': 'What advancements does DeepSeek-R1-Zero bring to reasoning capabilities in language models?',
  'role': 'user'},
 {'content': 'DeepSeek-R1-Zero exhibits super performance on reasoning benchmarks, with a pass@1 score on AIME 2024 increasing from 15.6% to 71.0%, and further improving to 86.7% with majority voting, matching the performance of OpenAI-o1-0912. However, it also faces challenges such as poor readability and language mixing.',
  'role': 'assistant'}]
```

Since `Qwen2.5-3B-Instruct` is already **instruction-tuned**, we didn't require additional pre-training. Instead, we performed **supervised fine-tuning (SFT)** on our custom dataset, allowing the model to specialize in **domain-specific conversations and reasoning tasks**.

Uploading to Hugging Face

After fine-tuning, we uploaded the model to **Hugging Face Hub** for easy access and future use. Since we trained the model using **4-bit quantization (QLoRA)**, we needed to **merge LoRA weights with the base model** before saving.

The fine-tuned model is now **publicly accessible** on Hugging Face at:

🔗 [AkinduH/Qwen2.5-3B-Instruct-Fine-Tuned-on-Deepseek-Research-Papers]()

Implementing a RAG (Retrieval-Augmented Generation) System

After fine-tuning the model, we also developed a **Retrieval-Augmented Generation (RAG) system** as a backup to enhance responses using research paper data. This ensures that even if the fine-tuned model lacks specific details, the RAG system can retrieve and provide **context-rich answers**.

Embedding Model Selection

We used **sentence-transformers/all-MiniLM-L6-v2** from **Hugging Face Embeddings** to convert research paper text into vector embeddings. This model was chosen because:

Creating the Vector Store with FAISS

To store and retrieve embeddings efficiently, we used **FAISS (Facebook AI Similarity Search)** as our **vector database**.

Integrating Retrieval with the Model

During inference, we wrapped the fine-tuned model with the **retrieval system**:

- The input query was first **matched** against the FAISS index.
- **Top-k relevant passages** were retrieved and added as context.
- The **context-rich prompt** was sent to the fine-tuned model for final response generation.

This **hybrid approach** significantly improved response accuracy, especially for **complex reasoning and research-related queries**.