Group Project
Principal of Large Language Model

Group Members
Shriman
Juniad Kakar
Nguyen Tan Anh

Content:

1. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset
2. Distillation training on MiniLLM
3. Chatbot Agent with MCP interface, local deployment with vLLM

I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset

Llama 2 is a collection of pretrained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters.

Llama 2 was trained on **40% more data** than Llama 1, and has double the context length.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:** **Pretraining Tokens:** 2 Trillion **Context Length:** 4096	**Data collection for helpfulness and safety:** **Supervised fine-tuning:** Over 100,000 **Human Preferences:** Over 1,000,000
13B		
70B		

I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset

Model configuration

ning > models > finetune_model > {} config.json >


```json
{
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "eos_token_id": 2,
  "head_dim": 128,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 11008,
  "max_position_embeddings": 4096,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 32,
  "pad_token_id": 0,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_scaling": null,
  "rope_theta": 10000.0,
  "tie_word_embeddings": false,
  "torch_dtype": "float16",
  "transformers_version": "4.52.4",
  "use_cache": true,
  "vocab_size": 32000
}
```


Tokenizer configuration

ning > models > tokenizer > {} tokenizer_config.json > ...


```json
{
  "add_bos_token": true,
  "add_eos_token": false,
  "add_prefix_space": null,
  "added_tokens_decoder": {
    "0": {
      "content": "<unk>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "1": {
      "content": "<s>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "2": {
      "content": "</s>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    }
  },
  "bos_token": "<s>",
  "clean_up_tokenization_spaces": false,
  "eos_token": "</s>",
  "extra_special_tokens": {},
  "legacy": false,
  "model_max_length": 1000000000000000019884624838656,
  "pad_token": "</s>",
  "padding_side": "right",
  "sp_model_kwargs": {},
  "tokenizer_class": "LlamaTokenizer",
  "unk_token": "<unk>",
  "use_default_system_prompt": false
}
```

I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset

PEFT & QLoRA Quantization Setup

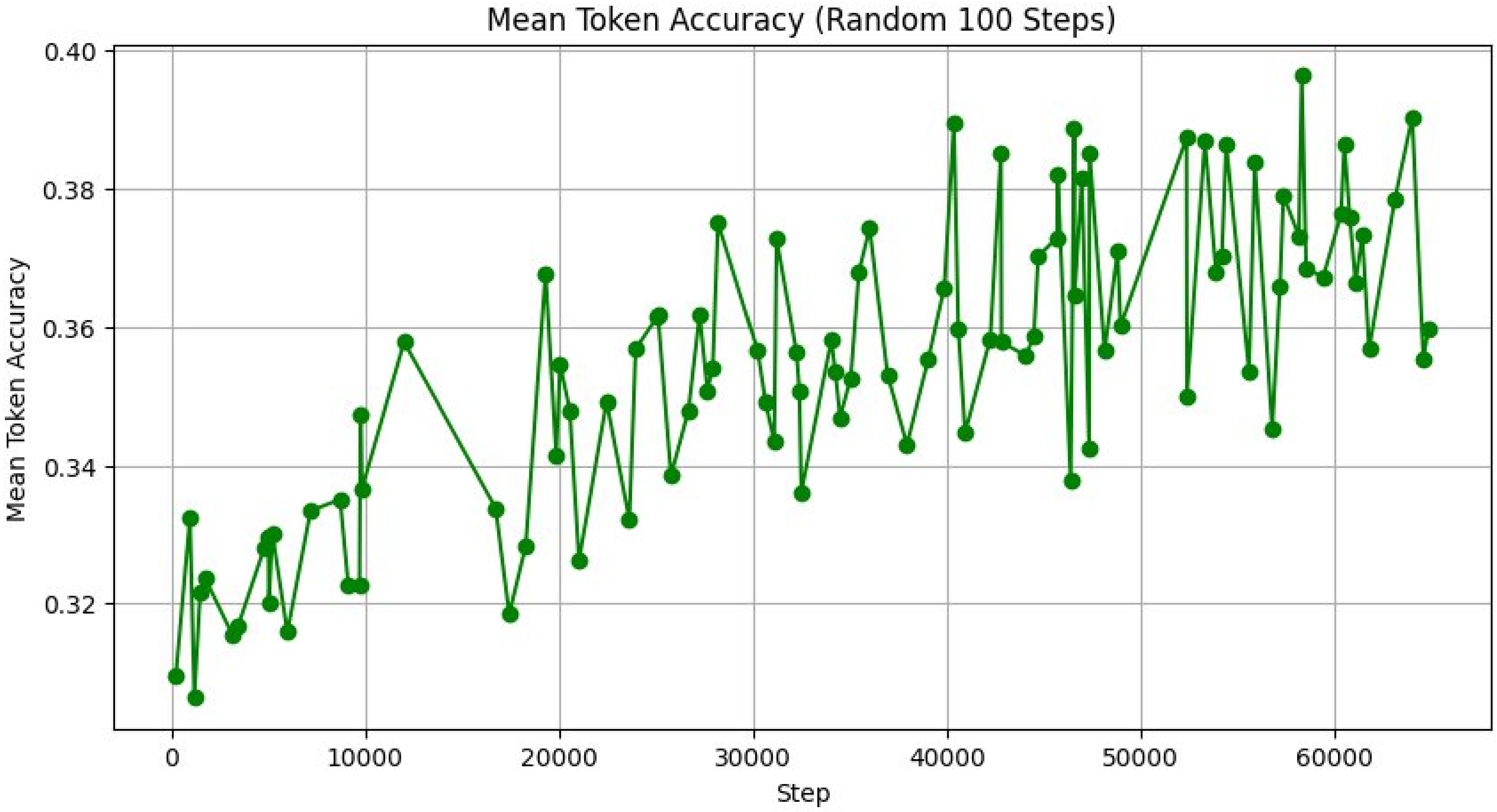
- PEFT (Parameter-Efficient Fine-Tuning):
  - Uses LoRA adapters to fine-tune only a small set of new parameters.
  - Reduces memory and compute requirements during training.
- QLoRA:
  - Loads the base model in 4-bit quantization to minimize GPU memory usage.
  - Applies LoRA adapters on top, enabling efficient fine-tuning of large language models even on limited hardware.
- Key configs:
  - 4-bit quantization (use_4bit=True, bnb_4bit_quant_type="nf4")
  - LoRA adapter parameters (lora_r=64, lora_alpha=16, lora_dropout=0.1)

```
model_name = "NousResearch/Llama-2-7b-chat-hf"
dataset_name = "rajpurkar/squad_v2"
finetune_model = "Llama-2-7b-chat-finetune"


# Output folder
output_dir = "./results"

# No of epochs
num_train_epochs = 1

# No change params
use_4bit, bnb_4bit_compute_dtype, bnb_4bit_quant_type, use_nested_quant = True, "float16", "nf4", False # To quantization
lora_r, lora_alpha, lora_dropout = 64, 16, 0.1
fp16, bf16 =  False, False
per_device_train_batch_size, per_device_eval_batch_size = 1, 1
gradient_accumulation_steps, gradient_checkpointing, max_grad_norm = 1, True, 0.3
learning_rate, weight_decay, optim = 2e-4, 0.001, "paged_adamw_32bit"
lr_scheduler_type, max_steps, warmup_ratio = "cosine", -1, 0.03
group_by_length, save_steps, logging_steps = True, 0, 25
max_seq_length, packing, device_map = 1024, False, {"": 0}
```

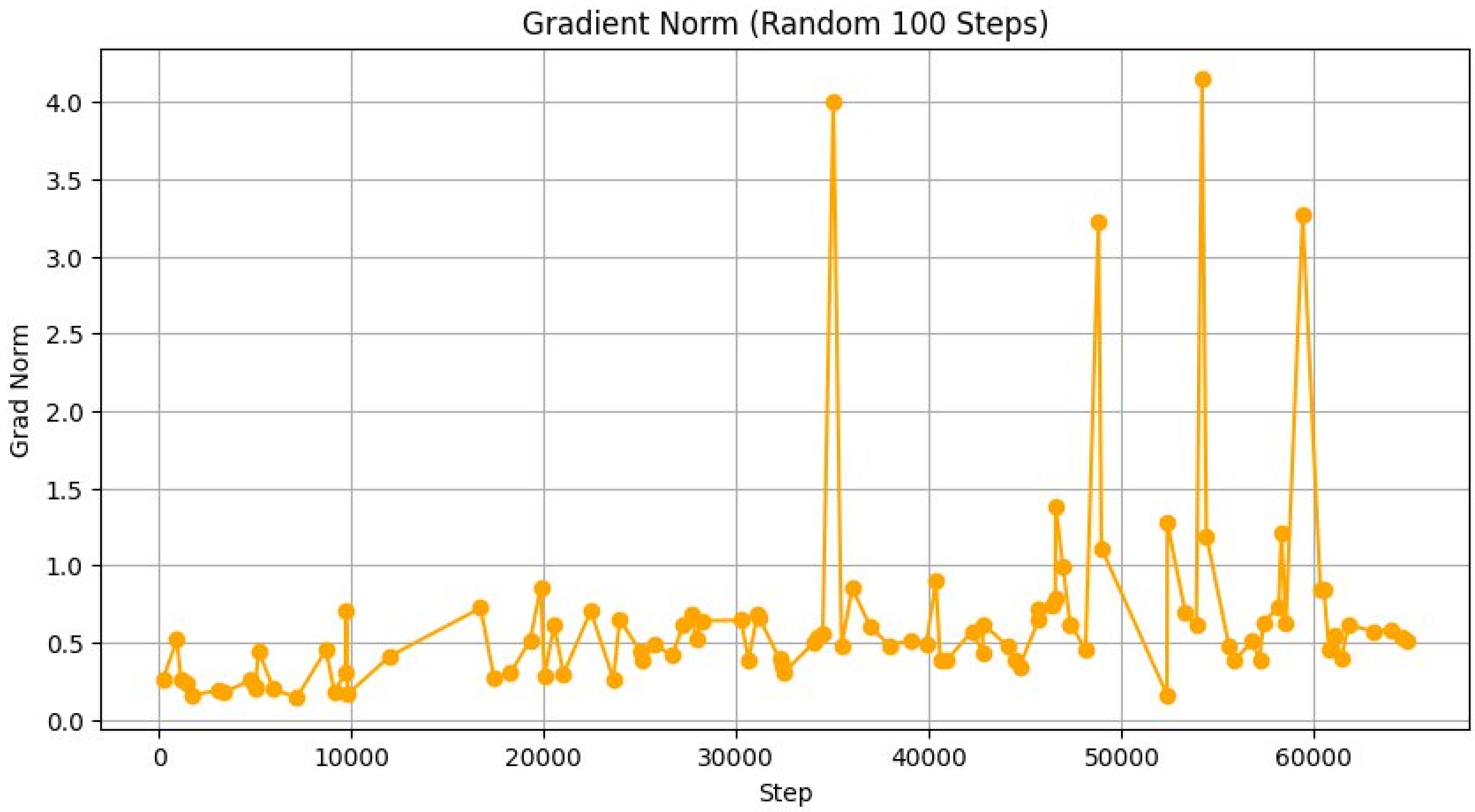
```
# Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

bnb_config = BitsAndBytesConfig(
    load_in_4bit=use_4bit,
    bnb_4bit_quant_type=bnb_4bit_quant_type,
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=use_nested_quant,
)
```

I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset

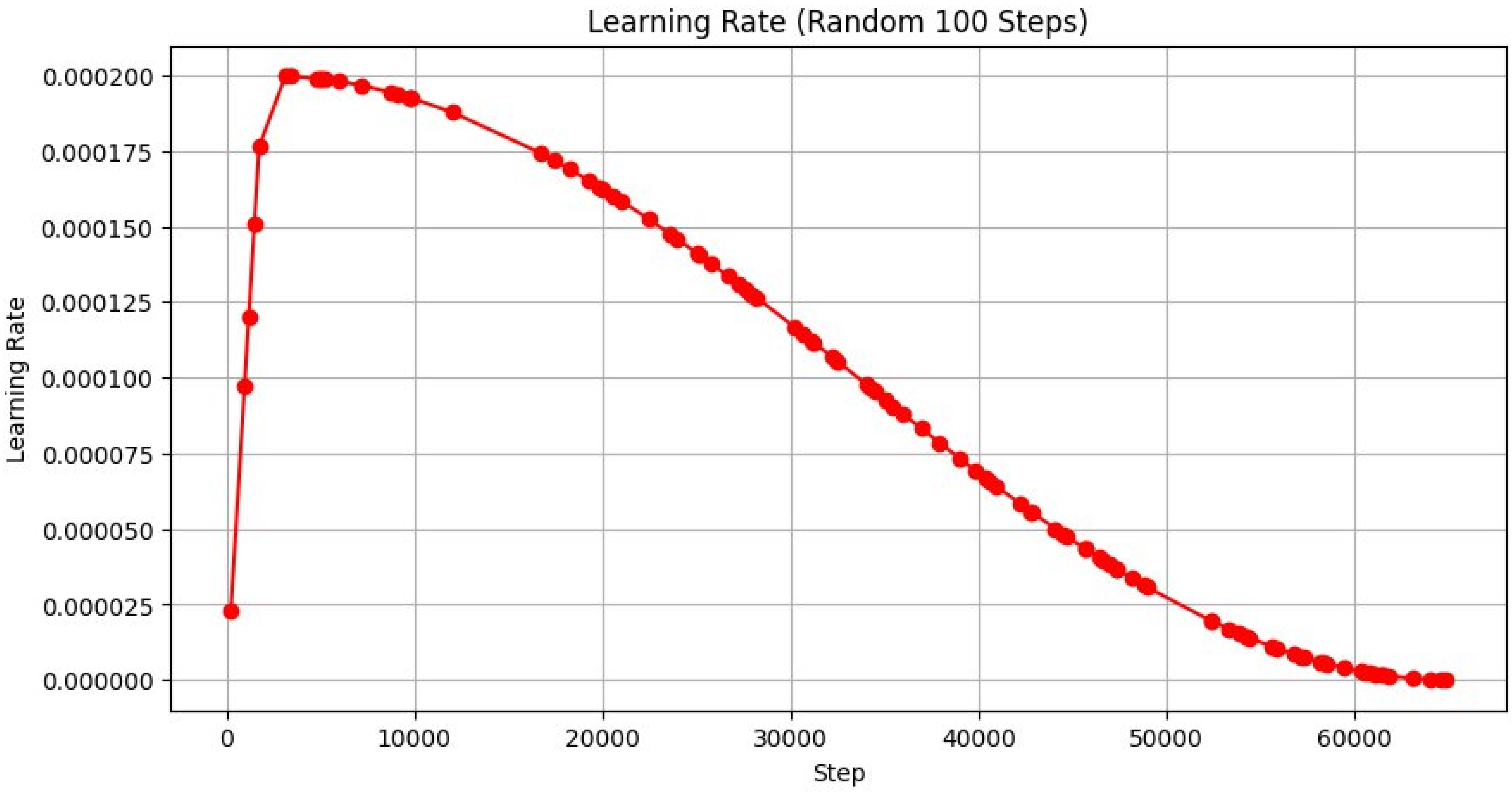
I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset



I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset



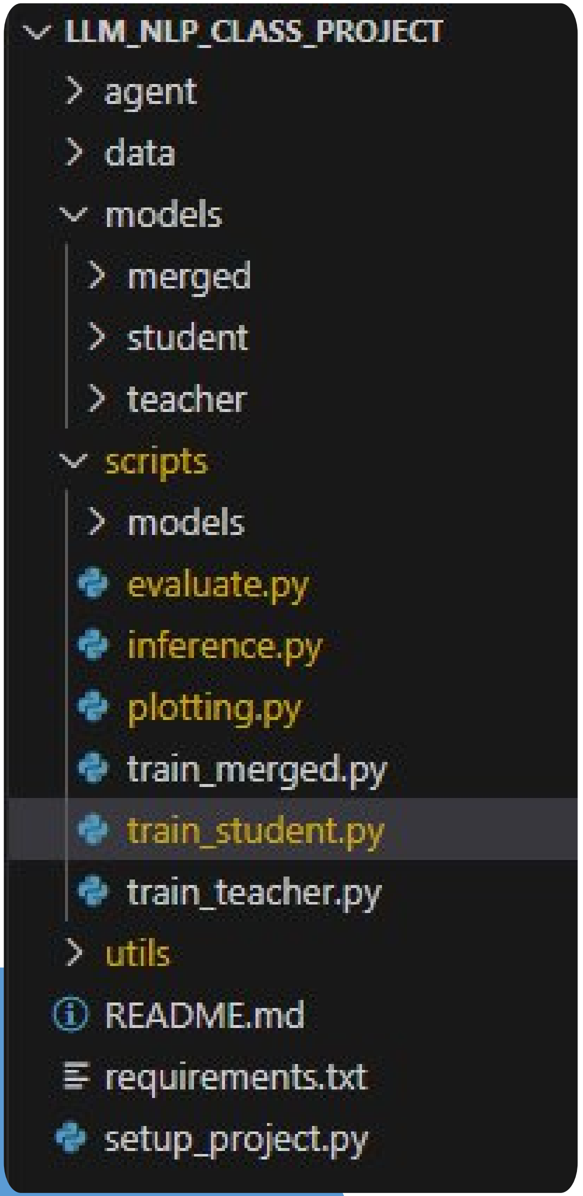
I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset



I. Fine-tuning LLama-2-7B on SQUAD 2.0 dataset

Metric	Value
Exact	75.0
F1	77.83
Total	100
HasAns Exact	73.33
HasAns F1	79.63
HasAns Total	45
NoAns Exact	76.36
NoAns F1	76.36
NoAns Total	55

Key Code Explanations



- **Goal:**

  Train, evaluate, and deploy a compact question answering model (MiniLM) using knowledge distillation, making it efficient for real-world applications while maintaining strong performance.

  Workflow

- **Prepare Data:**

  Use data_utils.py to load and preprocess datasets (SQuAD v2).
- **Train the Student Model:**

  Run train_student.py to train the MiniLM-based student model. Checkpoints and tokenizer files are saved in student.
- **Evaluate the Model:**

  Use evaluate.py to evaluate the model's predictions using SQuAD metrics.
- **Inference:**

  Use inference.py to make predictions on new or unseen data.
- **Visualization:**

  Use plotting.py to visualize training and evaluation results.

Key Code Explanations

```python
# scripts/train_student.py
dataset = load_squad_v2()
tokenizer, model = get_minilm_qa_model()

tokenized_datasets = dataset.map(
    lambda examples: preprocess_function(examples, tokenizer),
    batched=True,
    remove_columns=dataset["train"].column_names,
)

training_args = TrainingArguments(
    output_dir=os.path.join("models", "student"),
    evaluation_strategy="steps",
    eval_steps=500,
    save_strategy="steps",
    save_steps=500,
    save_total_limit=3,
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=4,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=100,
    push_to_hub=False,
    report_to="none",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    tokenizer=tokenizer,
    compute_metrics=partial(
        compute_metrics,
        tokenizer=tokenizer,
        eval_examples=dataset["validation"],
        eval_features=tokenized_datasets["validation"]
```

1. **Training Script:** train_student.py

Purpose: Trains the MiniLM student model on a QA dataset (SQuAD v2).

Key Steps:

Data Loading:

Uses load_squad_v2 to load and preprocess the dataset.

Model Loading:

Uses get_minilm_qa_model to load the MiniLM model and tokenizer.

Preprocessing:

The preprocess_function tokenizes questions and contexts, aligns answer spans, and prepares inputs for training.

TrainingArguments:

Sets up HuggingFace Trainer arguments (batch size, epochs, evaluation steps, etc.).

Trainer:

Initializes the HuggingFace Trainer with model, data, tokenizer, and metrics

Training:

Calls trainer.train() to start training and saves the final model.

Key Code Explanations

```python
def preprocess_function(examples, tokenizer):
    questions = [q.strip() for q in examples["question"]]
    inputs = tokenizer(
        questions,
        examples["context"],
        max_length=384,
        truncation="only_second",
        stride=128,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding="max_length",
    )
    # Maps answer start/end positions to token indices
    # Handles cases with no answer
    # Returns processed features for training
```

2. Preprocessing Function

Purpose:

Tokenizes and aligns the QA data for training.

Key Steps:

- Strips and tokenizes questions and contexts.
- Handles long contexts with sliding window and overflow.
- Maps answer spans to token positions.
- Handles cases with no answer (for SQuAD v2).

3. Evaluation Script: evaluate.py

Purpose:

Evaluates the trained model on the validation set using SQuAD metrics (Exact Match, F1).

Key Steps:

- Loads the trained model and tokenizer.
- Runs predictions on the validation set.
- Computes metrics using metrics.py.

Key Code Explanations

4. Inference Script: inference.py

Purpose: Runs the trained model on new context-question pairs for QA.

Key Steps:

- Loads the model and tokenizer from student.
- Accepts user input for context and question.
- Predicts and prints the answer.

```python
def main():
    model_dir = os.path.join("models", "student")
    device = "cuda" if torch.cuda.is_available() else "cpu"
    tokenizer, model = load_model_and_tokenizer(model_dir)
    model.to(device)

    print("MiniLM QA Inference")
    context = input("Enter context paragraph:\n")
    question = input("\nEnter question:\n")
    answer = predict_answer(context, question, tokenizer, model,
    print(f"\nPredicted Answer: {answer}")
```

5. Utilities

data_utils.py:
Functions for loading and preprocessing datasets (SQuAD v2).

model_utils.py:
Functions for loading MiniLM model and tokenizer.

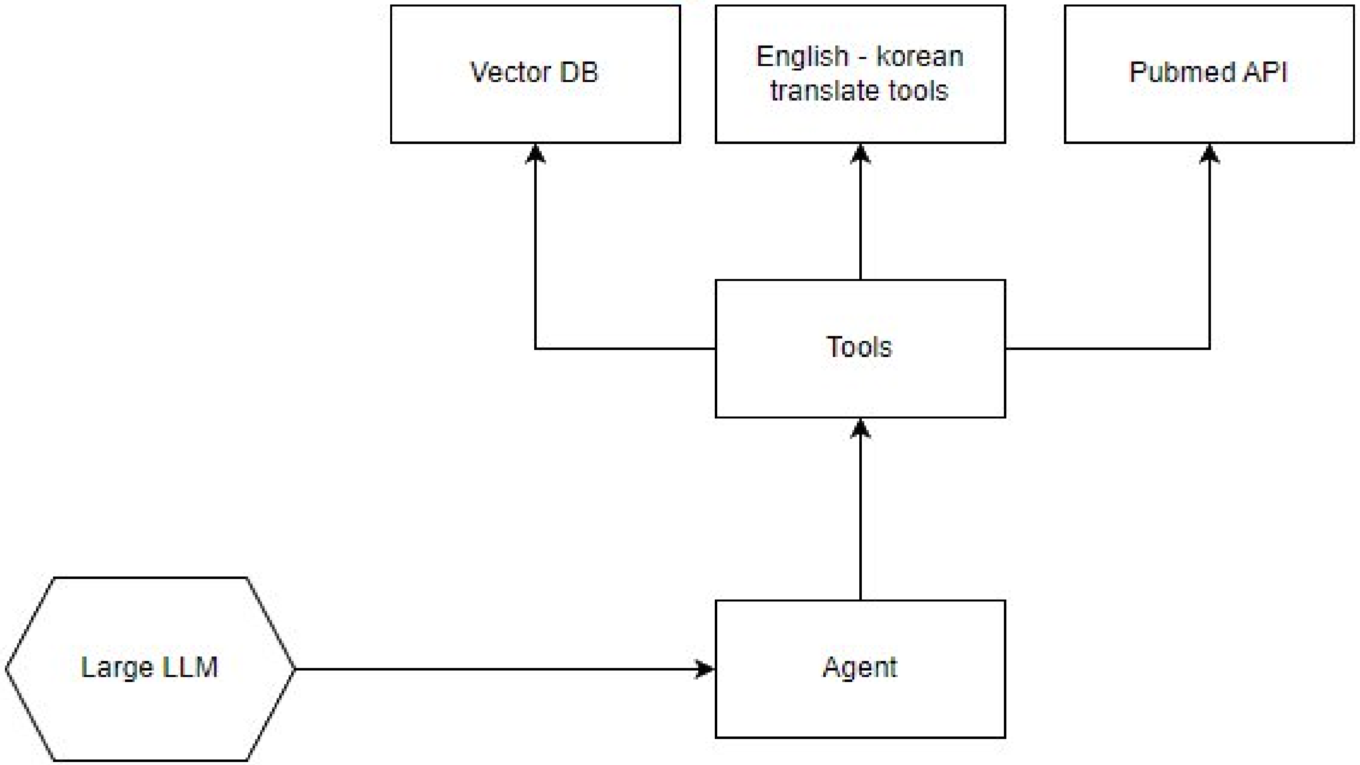
metrics.py:
Functions for computing SQuAD metrics (Exact Match (EM), F1).

Model Artifacts

- **Checkpoints:**
  Saved in models/student/checkpoint-xxxx/ during training.
- **Tokenizer & Vocab:**
  tokenizer.json, vocab.txt in student.
- **Config & Weights:**
  config.json, model.safetensors in student.

Chatbot Agent with MCP interface, local deployment with vLLM

Project Structure



Architecture Diagram

vLLM deploying local LLama-3-8B-Instruct

Agent receives user queries and routes them to the appropriate tool through MCP server

Tools module manages access to:

Vector DB (document/context retrieval)	sentence-transformers/all-MiniLM-L6-v2
English-Korean translation tools	facebook/nllb-200-distilled-600M & Google Deep Translate API
PubMed API (medical literature search)	https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi

Workflow Steps:

1.User Query Input:

Agent receives a query from the user, pass in to @mcp.prompt for prompt template.

2.Query Analysis & Routing:

Agent analyzes the query keywords to determine the required tool or model.

3.Tool Invocation:

Request to MCP server:

1. Translation → English-Korean translation tool
2. Medical/Scientific → PubMed API
3. Retrieval/Document → Vector DB + LLM (Retrieval-Augmented Generation, RAG)
4. Default → LLM with contex4

4. Final Output:

Agent returns the response to the user.

Key Modules & Code Highlights:

chatbot.py

- Using agent API from agents lib
- Asynchronous request to vLLM server and MCP server

mcp_server.py

- Run MCP server as an FastMCP API
- Provide tools for query papers from Arvix, Pubmed, save summary and return content
- @mcp.prompt for providing the prompt templates
- @mcp.source for providing source llm may use

translate_tool.py

- Use facebook/nllb-200-distilled-600M model pipeline provided by transformers Huggingface for local translating solution.
- Use deeptranslator API from Google for light weight remote translating solution.

pubmed_tool.py

- Calls PubMed API, fetches article titles for medical queries

large_llm.py

- Loads and runs a QA pipeline using a RoBERTa-based SQuAD2 model

vLLM deployment:

- Using KV cache

Code Explaination:

chatbot.py > run

```python
from agents.model_settings import ModelSettings

model= "llama-base"

async def run(mcp_server: MCPServer):
    agent = Agent(
        name="Assistant",
        model=model,
        instructions="Use the tools to answer the questions.",
        mcp_servers=[mcp_server],
        model_settings=ModelSettings(tool_choice="search_papers"),
    )

    # Run the `get_weather` tool
    message = "What is the Transformer architect? from research papers on arxiv."
    # message = "Translate to english arxiv의 연구 논문을 바탕으로 한 sLLM이란 무엇인가?"

    print(f"\n\nRunning: {message}")
    result = await Runner.run(starting_agent=agent, input=[{"role": "user", "content": message}], max_turns=10)#, tracing_disbale = True
    print(result)
    print(result.raw_responses)

    # Final turn
    new_input = result.to_input_list() + [{"role": "user", "content": message}]
    result = await Runner.run(agent, new_input)
    print("Final = ",result.final_output)

async def main():
    async with MCPServerSse(
        name="SSE Python Server",
        params={
            "url": "http://localhost:8000/sse",
        },
    ) as server:
        await run(server)


if __name__ == "__main__":
    # Let's make sure the user has uv installed
    if not shutil.which("uv"):
        raise RuntimeError(
            "uv is not installed. Please install it: https://docs.astral.sh/uv/getting-started/installation/"
        )

    asyncio.run(main())
```

Code Explaination:

```python
import sys
import os
import json
import arxiv
from loguru import logger
from typing import List
from datetime import datetime
# from Bio import Entrez, Medline

# Entrez.email = "202459045@jbnu.ac.kr"

# Add the project root to the Python path
project_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
sys.path.insert(0, project_root)

from mcp.server.fastmcp import FastMCP
# from pubmed_tool import PubMedSearchTool
from translate_tool import translate_en_to_ko, translate_ko_to_en

# pubmed_tool = PubMedSearchTool(max_results=5)

PAPER_DIR = os.getenv("PAPER_DIR", "papers")
RESEARCH_PORT = int(os.getenv("RESEARCH_PORT", "8001"))

# Port 8000
mcp = FastMCP(
    name="mcp-server",
    port=8000,
)

# @mcp.tool()
# def search_pubmed(query: str, max_results: int = 5) -> dict:
#     """Search PubMed research articles by query"""
```

```python
# Tools
# ---------------------------------------------------------------------------

@mcp.tool()
async def search_papers(topic: str, max_results: int = 5) -> List[str]:
    """Search arXiv for *topic* and persist metadata. Returns stored short IDs."""
    print(f"Searching arXiv for topic: {topic} with max results: {max_results}")
    logger.info("🔎 Searching arXiv for '%s' (%d results)", topic, max_results)

    client = arxiv.Client()
    search = arxiv.Search(
        query=topic,
        max_results=max_results,
        sort_by=arxiv.SortCriterion.Relevance,
    )

    stored_ids: List[str] = []
    async for paper in client.results_async(search):
        pid = paper.get_short_id()
        stored_ids.append(pid)
        meta = _load_topic(topic)
        meta[pid] = {
            "title": paper.title,
            "authors": [a.name for a in paper.authors],
            "summary": paper.summary,
            "pdf_url": paper.pdf_url,
            "published": paper.published.date().isoformat(),
            "saved_at": datetime.utcnow().isoformat() + "Z",
        }
        _save_topic(topic, meta)

    logger.success("✓ Stored %d papers for topic '%s'", len(stored_ids), topic)
    return "hi"


# @mcp.tool()
```

Code Explaination:

````markdown
README.md > ## Project Structure
## Deploy Llama Serve with vLLM

### Without Docker

```bash
vllm serve /home/tananh/llm_subject/finetuning/models/serve \
  --port 8001 \
  --api-key a \
  --served-model-name llama-finetune \
  --enable-auto-tool-choice \
  --tool-call-parser llama3_json
```

Or, to serve the Meta Llama 3.1-8B-Instruct model:

```bash
vllm serve NousResearch/Meta-Llama-3-8B-Instruct --dtype auto --api-key a --port 8001 --served-model-name llama-base --enable-auto-tool-choice
--tool-call-parser llama3_json
```

---

### With Docker

1. **Build the Docker image:**
   ```bash
   docker build -t llama_7b_finetune_serve .
   ```

2. **Run the Docker container:**
   ```bash
   docker run \
     -v /home/tananh/llm_subject/finetuning/models/serve:/weights \
     -p 8001:8001 \
     llama_7b_finetune_serve
   ```

---

## Run the MCP Server

Open a new terminal and run:

```bash
python agent/mcp_server.py
```
````

Demo Video:

https://drive.google.com/file/d/1o5DN_6TuTF4YVhlOMGHVaWtHd4koxtPA/view?usp=sharing