COMS 6998-013
LLM based GenAI
Instructor: Parijat Dube and Chen Wang
Due: Nov. 15, 2025

Homework 4

Problem 1: Fine-Tune a LLM Model for Dialogue Summarization (25 points)

In this assignment, you will fine-tune an existing Large Language Model (LLM) from Hugging Face for enhanced dialogue summarization. You will use the FLAN-T5 model, which provides high quality instruction tuning capabilities and can summarize text out of the box. You will explore both full fine-tuning and Parameter Efficient Fine-Tuning (PEFT) approaches, evaluate the results with ROUGE metrics, and compare their performance. Notebook

1.1 Load Required Dependencies, Dataset and LLM (5 points)

1.1.1 Set up Required Dependencies (1 point)

- Install required packages including `datasets`, `torch`, `transformers`, `evaluate`, *rouge_score*, `loralib`, and `peft` using pip
- Import necessary modules from these packages, particularly *load_dataset* from datasets, `AutoModelForSeq2SeqLM` and `AutoTokenizer` from transformers, and evaluation metrics from the `evaluate` package

1.1.2 Load Dataset and LLM (2 points)

- Use *load_dataset* function to load the DialogSum dataset from Hugging Face's "knkarthick/dialogsum" repository
- Initialize the FLAN-T5 model using $AutoModelForSeq2SeqLM.from\_pretrained()$ with appropriate data type settings
- Set up the tokenizer using $AutoTokenizer.from\_pretrained()$
- Use the provided utility function to analyze and display the model's trainable parameters

1.1.3 Test the Model with Zero Shot Inferencing (2 points)

- Create a test dialogue prompt following the specified format
- Use the tokenizer to encode the input text
- Generate a summary using the model's *generate*() method with appropriate generation parameters
- Compare the generated summary with the human baseline using qualitative analysis

1.2. Perform Full Fine-Tuning (10 points)

1.2.1 Preprocess the Dialog-Summary Dataset (2 points)

- Create a tokenization function that:

  Adds appropriate prompts to the dialogue Tokenizes both input dialogue and target summary Handles padding and truncation appropriately
- Apply the tokenization function to the dataset using the `map()` method
- Create a subsampled version of the dataset for efficient training

1.2.2 Fine-Tune the Model (3 points)

- Configure `TrainingArguments` with appropriate learning rate, epochs, and other hyperparameters
- Initialize the Hugging Face `Trainer` class with the model, training arguments, and datasets

- Execute the training process using `trainer.train()`
- Save the fine-tuned model checkpoint

1.2.3 Evaluate the Model Qualitatively (2 points)

- Generate summaries using both the original and fine-tuned models
- Compare outputs across models using the same test examples
- Analyze improvements in summary quality, coherence, and relevance

1.2.4 Evaluate the Model Quantitatively (3 points)

- Initialize the ROUGE metric using the `evaluate` library
- Generate summaries for a test set using both models
- Calculate ROUGE scores using the `compute()` method
- Analyze and compare performance metrics between models

1.3. Perform Parameter Efficient Fine-Tuning (PEFT) (10 points)

1.3.1 Setup PEFT/LoRA Model (2 points)

- Configure LoRA parameters using `LoraConfig` with appropriate rank, alpha, and target modules
- Initialize the PEFT model using $get\_peft\_model()$
- Verify the reduction in trainable parameters compared to full fine tuning

1.3.2 Train PEFT Adapter (3 points)

- Set up training arguments specific to PEFT, including higher learning rate
- Initialize and execute training using the Hugging Face Trainer
- Save the trained PEFT adapter weights
- Load a pre-trained PEFT model for comparison

1.3.3 Evaluate Model Qualitatively (2 points)

- Generate summaries using the PEFT model
- Compare outputs across all three approaches:
  Original zero-shot model Fully fine-tuned model PEFT model
- Analyze the quality of summaries considering different aspects

1.3.4 Evaluate Model Quantitatively (3 points)

- Generate summaries using the PEFT model for the test dataset
- Calculate ROUGE metrics for all three approaches
- Compare performance improvements:
  Analyze PEFT vs. original model metrics Compare PEFT vs. full fine-tuning results
- Evaluate the trade-off between performance and computational efficiency

Notes:

- Students should thoroughly document their analysis of the trade-offs between approaches
- Pay particular attention to the ROUGE metric improvements across different models
- Consider and document the computational resources required for each approach
- Note any challenges or limitations encountered during the fine-tuning process

Homework 4

Problem 2: LLM Inference Benchmarking (25 points)

2.1 Overview

This assignment will introduce you to benchmarking Large Language Model (LLM) inference using vLLM, a high-performance inference engine. You will use a small model (OPT-125M) to understand the fundamentals of throughput measurement and analyze how different parameters affect inference speed. Notebook

2.2 Required Libraries

- PyTorch with CUDA support
- vLLM
- Transformers
- Matplotlib, Pandas, Seaborn (for visualization)

2.3 Tasks and Scoring

2.3.1 Environment Setup (2 points)

- Successfully install all required packages
- Verify CUDA availability
- Import necessary libraries

2.3.2 Implementation (13 points)

- Create Synthetic Requests (5 points)
  - Implement the *create_synthetic_requests* function
  - Handle token length requirements correctly
  - Create appropriate prompts for benchmarking
  - Validate input/output token lengths
- Implement Benchmarking Function (8 points)
  - Create the *run_benchmark* function
  - Configure LLM parameters correctly
  - Implement proper timing measurements
  - Handle batch processing effectively
  - Calculate and return appropriate metrics

2.3.3 Experiments and Analysis (10 points)

- Parameter Studies (6 points)
  - Batch size experiments (2 points)
    - Test at least 2-3 different batch sizes
    - Document impact on throughput
  - Input/output length experiments (2 points)
    - Test at least 2-3 different sequence lengths
    - Analyze effect on processing speed
  - Request count experiments (2 points)
    - Test varying numbers of concurrent requests
    - Evaluate scaling behavior
- Analysis and Visualization (4 points)
  - Create clear visualizations of results
  - Provide thorough analysis of findings
  - Compare different parameter configurations
  - Draw meaningful conclusions

2.4 Implementation Notes

- Model Selection
  - Use the OPT-125M model for consistency
  - Ensure proper model loading and configuration
- Experiment Guidelines
  - Document all experimental parameters
  - Maintain consistent testing conditions
  - Consider resource limitations
  - Multiple runs may be needed for stability
- Performance Metrics
  - Focus on tokens/second throughput
  - Consider both input and output processing
  - Account for batch processing effects
- Resource Considerations
  - Be mindful of GPU memory usage
  - Consider Google Colab notebook environment limitations
  - Adjust batch sizes and sequence lengths accordingly

2.5 Submission Requirements

- Complete, working code implementation
- Clear experimental results and visualizations
- Thorough analysis of findings
- Documentation of methods and parameters used

2.6 Grading Criteria

- Correctness of implementation
- Quality of experimental design
- Depth of analysis
- Clarity of documentation and results

Question 3: Implement ReAct Agent with Multiple Tools 25 points

Implement a ReAct (Reasoning and Acting) agent as described by Yao et al. [2022], incorporating three main tools: search, compare, and analyze. This agent should be able to handle complex queries by reasoning about which tool to use and when. Notebook

1. (4 points) Implement the search tool using the SerpAPI integration from previous questions. Ensure it can be easily used by the ReAct agent.
   - Proper integration with SerpAPI
   - Formatting the search results for use by the ReAct agent
2. (5 points) Create a custom comparison tool using LangChain's `Tool` class. The tool should accept multiple items and a category as input and return a comparison result.
   - Implementing the comparison logic
   - Creating an appropriate prompt template for the comparison
   - Proper error handling for invalid inputs
3. (5 points) Implement an analysis tool that can summarize and extract key information from search results or comparisons. This tool should use the OpenAI model to generate insightful analyses.
   - Implementing the analysis logic
   - Creating an appropriate prompt template for the analysis
   - Ensuring the analysis output is concise and relevant
4. (6 points) Integrate these tools with a ReAct agent using LangChain. Your implementation should:
   - Use LangChain's `initialize_agent` function with the `AgentType.ZERO_SHOT_REACT_DESCRIPTION` agent type
   - Include all three tools (search, compare, analyze) as available actions for the agent
   - Implement proper error handling and fallback strategies

- Ensure smooth transitions between tools in the agent's reasoning process

5. (5 points) Implement a simple Streamlit user interface for your ReAct agent. Your implementation should include:
   - A text input field for users to enter their queries
   - A button to submit the query and trigger the ReAct agent
   - A display area for showing the final results
   - (Optional) A section to display the step-by-step reasoning process of the ReAct agent

Submission Requirements

Please submit the following items as part of your solution:

1. Your complete code implementation for the ReAct agent and its tools.
2. A sample question that you used to test your tool (make it complex enough to demonstrate the use of multiple tools).
3. The final answer provided by your ReAct agent for the sample question.
4. The complete history traces of the ReAct agent for your sample question, showing its thought process, actions, and observations. Your traces should follow a format similar to this example:

```
Thought: I need to find information about top smartphones first
Action: Search[top smartphones 2023]
Observation: [Search results about top smartphones]
Thought: Now I should compare the top two options
Action: Compare[iPhone 14 Pro, Samsung Galaxy S23 Ultra, smartphones]
Observation: [Comparison result]
Thought: I should analyze this comparison for the user
Action: Analyze[comparison result]
Observation: [Analysis of the comparison]
Final Answer: [Your agent's final response to the user's query]
```

Ensure that your submission clearly demonstrates the agent's ability to reason about which tool to use and how to interpret the results from each tool. Your history traces should show a logical flow of thoughts, actions, and observations, culminating in a final answer that addresses the initial query.

Note: Ensure that your ReAct agent can seamlessly switch between these tools based on the task at hand. The agent should be able to reason about which tool to use next and how to interpret the results from each tool.

Problem 4: Time Series Forecasting (25 points)

In this problem, we will use a pre-trained `TinyTimeMixer` model for time-series forecasting. The specific model we will use is the TTM-1024-96 model. That means the TTM model can take an input of 1024 time points (`context_length`), and can forecast upto 96 time points (`forecast_length`) in the future. We will use the pre-trained TTM in two settings:

Homework 4

1. **Zero-shot**: The pre-trained TTM will be directly used to evaluate on the test split of the target data. Note that the TTM was NOT pre-trained on the target data.
2. **Few-shot**: The pre-trained TTM will be quickly fine-tuned on only 5% of the train split of the target data, and subsequently, evaluated on the test part of the target data.

You will work with the `etth1` dataset available here. Notebook

4.1 Environment Setup (2 points)

- Install the `tsfm` library and all other required packages
- Setup the important arguments

4.2 Data processing pipeline (3 points)

- Load the dataset and display the first 20 rows
- Plot date vs "MUFL"
- Use the `TimeSeriesPreprocessor` (TSP) module for data preparation and get the train, valid, and test dataset splits. What is the length of your train, valid, and test splits?

4.3 Zero-shot evaluation method (4 points)

- Perform zero-shot evaluation using the pre-trained model on the test dataset and calculate the evaluation error.
- Plot the zero-shot predictions on the test dataset

4.4 Zero-shot evaluation by truncating the length (4 points)

Let's say your application needs to forecast 24 hours in the future. You can still use the TTM-1024-96 model and set `prediction_filter_length`=24 argument during model loading. Try it on etth1, and note the evaluation error (on all channels).

4.5 Few-shot finetune and evaluation method (4 points)

Try few-shot with 5% and 10% forecasting on `etth1`. Freeze the backbone and fine-tune for only 1 epoch. What is the evaluation error you get for the two cases?

4.6 Few-shot evaluation by changing the loss-function (4 points)

Try few-shot 5% forecasting on etth1 by changing the loss to mae (mean absolute error). Freeze the backbone and fine-tune for only 1 epoch. What is the evaluation error you get?

4.7 Zero-shot on channel 0 and 2 (4 points)

In your notebook, add `prediction_channel_indices`=[0,2] during model loading to forecast only 0th and 2nd channels. In this case, execute the following code and note the output shape.

References

Shunyu Yao, Jian Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/pdf/2210.03629.