Assignment 3

COMP3361: Natural Language Processing - University of Hong Kong

Spring 2025

1 Decoding Algorithms (35%)

In this exercise, you will implement a few basic decoding algorithms in the notebook: **greedy decoding**, **vanilla sampling**, **temperature sampling**, **top-k sampling**, and **top-p sampling**.

1.1 Dataset

In this assignment, we focus on the open-ended story generation task (data available here). This dataset contains *prompts* for story generation, modified from the ROCStories dataset.

Fluency: The CoLA classifier is a RoBERTa-large classifier trained on the CoLA corpus (Warstadt et al., 2019), which contains sentences paired with grammatical acceptability judgments. We will use this model to evaluate fluency of generated sentences.

Diversity: The Count of Unique N-grams is used to measure the diversity of the generated sentences.

Naturalness: The Perplexity of generated sentences under the language model is used to measure the naturalness of language. You can directly use the perplexity function from HuggingFace evaluate-metric package for this assignment.

In the notebook, we have provided a wrapper function `decode()` that takes care of batching, controlling max length, and handling the EOS token. You will be asked to implement the core function of each method: **given the pre-softmax logits of the next token, decide what the next token is.**

1.2 Greedy Decoding

The idea of greedy decoding is simple: select the next token as the one that receives the highest probability. **Implement the `greedy()` function that processes tokens in batch.** Its input argument `next_token_logits` is a 2-D FloatTensor where the first dimension is batch size and the second dimension is the vocabulary size, and you should output `next_tokens` which is a 1-D LongTensor where the first dimension is the batch size.

The softmax function is monotonic—in the same vector of logits, if one logit is higher than the other, then the post-softmax probability corresponding to the former is higher than that corresponding to the latter. Therefore, for greedy decoding you won't need to actually compute the softmax.

1.3 Vanilla Sampling, Temperature Sampling

To get more diverse generations, you can randomly sample the next token from the distribution implied by the logits. This decoding is called sampling, or vanilla sampling (since we will see more variations of sampling). Formally, the probability of for each candidate token w is

$$p(w) = \frac{\exp z(w)}{\sum_{w' \in V} \exp z(w')}$$

where $z(w)$ is the logit for token w, and V is the vocabulary. This probability on all tokens can be derived at once by running the softmax function on vector $\mathbf{z}$.

Temperature sampling controls the randomness of generation by applying a temperature t when computing the probabilities. Formally,

$$p(w) = \frac{\exp(z(w)/t)}{\sum_{w' \in V} \exp(z(w')/t)}$$

where t is a hyper-parameter.

Implement the `sample()` and `temperature()` functions. When testing the code we will use $t = 0.8$, but your implementation should support arbitrary $t \in (0, \infty)$.

1.4 Top-k Sampling

Top-k sampling decides the next token by randomly sampling among the k candidate tokens that receive the highest probability in the vocabulary, where k is a hyper-parameter. The sampling probability among these k candidate tokens should be proportional to their original probability implied by the logits, while summing up to 1 to form a valid distribution.

Implement the `topk()` function that achieves this goal. When testing the code we will use $k = 20$, but your implementation should support arbitrary $k \in [1, |V|]$.

1.5 Top-p Sampling

Top-p sampling, or nucleus sampling, is a bit more complicated. It considers the smallest set of top candidate tokens such that their cumulative probability is greater than or equal to a threshold p, where $p \in [0, 1]$ is a hyper-parameter. In practice, you can keep picking candidate tokens in descending order of their probability, until the cumulative probability is greater than or equal to p (though there's more efficient implementations). You can view top-p sampling as a variation of top-k sampling, where the value of k varies case-by-case depending on what the distribution looks like. Similar to top-k sampling, the sampling probability among these picked candidate tokens should be proportional to their original probability implied by the logits, while summing up to 1 to form a valid distribution.

Implement the `topp()` function that achieves this goal. When testing the code we will use $p = 0.7$, but your implementation should support arbitrary $p \in [0, 1]$.

1.6 Evaluation

Run the evaluation cell. This will use the first 10 prompts of the test set, and generate 10 continuations for each prompt with each of the above decoding methods. Each decoding method will output its overall evaluation metrics: perplexity, fluency, and diversity.

Deliverables:

1. **Code (15%):** Implement code blocks.

2. **Write-up (10%):** Answer the following questions in your write-up:

    - **Q1.1:** In greedy decoding, what do you observe when generating 10 times from the dev prompt?

- **Q1.2:** In vanilla sampling, what do you observe when generating 10 times from the dev prompt?
- **Q1.3:** In temperature sampling, play around with the value of temperature t. Which value of t makes it equivalent to greedy decoding? Which value of t makes it equivalent to vanilla sampling?
- **Q1.4:** In top-k sampling, play around with the value of k. Which value of k makes it equivalent to greedy decoding? Which value of k makes it equivalent to vanilla sampling?
- **Q1.5:** In top-p sampling, play around with the value of p. Which value of p makes it equivalent to greedy decoding? Which value of p makes it equivalent to vanilla sampling?
- **Q1.6:** Report the evaluation metrics (perplexity, fluency, diversity) of all 5 decoding methods. Which methods have the best and worst perplexity? Fluency? Diversity?

2 Applying Large Language Models to Math Reasoning (25%)

In this section, you will work with pre-trained large language models (LLMs) to tackle tasks in math reasoning. The focus is on exploring different prompting strategies to enhance the model's performance on these tasks without additional training. This part of the assignment is designed to give you practical experience in leveraging LLMs for specific applications by using prompts effectively.

You will use the GSM8K [Cobbe et al., 2021] benchmarks for evaluating the model's performance on code generation and math reasoning tasks, respectively. These benchmarks are designed to test the model's ability to solve math problems that require logical reasoning and multi-step calculations.

2.1 Q2.1: Few-shot Math Reasoning

Few-shot learning in math reasoning involves providing the model with a small number of example problems and solutions to help it grasp the task's requirements. This technique is aimed at enhancing the model's ability to solve new math problems by learning from the structure and logic of the provided examples.

The GSM8K benchmark, which includes a diverse set of grade-school level math word problems, will be used for this task. In this part, we evaluate the first 50 examples of GSM8K's test set, employing 8-shot in-context examples provided in the ipython notebook.

`build_input` In this function, you should build 8-shot direct prompts with the short answer. The prompt template should be:

Answer the following questions.

Question: {$Q_{example1}$}

Answer: {$A_{example1}$}

Question: {$Q_{example2}$}

Answer: {$A_{example2}$}

...

Question: {$Q_{inference}$}

Answer:

2.2 Q2.2: Few-shot Chain-of-Thought Math Reasoning

Chain-of-thought prompting [Wei et al., 2022] encourages the model to generate intermediate steps or reasoning processes when solving problems, rather than jumping directly to the final answer. In few-shot chain-of-thought math reasoning, the model is given a few examples where the solution process is detailed step-by-step. This teaches the model to approach new problems by generating a reasoned sequence of steps leading to the solution, which can improve performance on complex reasoning tasks.

We provided the chain-of-thought demo answers `cot_answer` and you should implement the `FewShotCoTReasoner` to finish the CoT evaluation process.

3 Building and Evaluating LLM Agents (40%)

In the previous sections, we explored how Large Language Models (LLMs) generate text and solve reasoning problems through various decoding strategies and prompting techniques. While powerful, these approaches rely solely on the model's inherent capabilities with no access to external tools or information sources.

This section introduces **LLM Agents** - systems that leverage LLMs as their reasoning engine while extending their capabilities through interaction with external tools. An LLM Agent uses the language model to determine which actions to take, when to use specific tools, and how to interpret the results to accomplish complex tasks.

3.1 Agent Architecture Overview

The core architecture of an LLM Agent typically consists of:

1. **The LLM**: Acts as the "brain" that processes tasks, decides on actions, and interprets results
2. **Tools**: External functions that extend the LLM's capabilities (e.g., web search, calculators, APIs)
3. **Agent Loop**: The orchestration mechanism that manages the interaction between the LLM and tools

You will build this system incrementally, starting with basic LLM interaction through a vanilla Chat Agent, then implementing specialized tools, and finally constructing a complete tool-calling framework that allows the LLM to reason through multi-step tasks.

3.2 Datasets and Evaluation

To evaluate agent performance, we'll use three diverse datasets that highlight different capabilities:

1. **SimpleQA**: Factual questions requiring up-to-date knowledge, typically answerable via web search
2. **MATH**: Grade-school math problems requiring precise calculation and reasoning
3. **GAIA**: Complex research-oriented tasks potentially requiring multiple capabilities (knowledge retrieval, calculation, and synthesis)

Each dataset presents unique challenges that will reveal the strengths and limitations of different agent configurations.

3.3 Implementation Roadmap

Your implementation will follow these steps:

1. **Basic Chat Agent**: Implement a simple agent that interacts with the LLM directly with no tools, establishing a baseline
2. **Tool Implementation**: Build essential tools including:
   - `FinalAnswerTool`: For standardizing agent outputs
   - `GoogleSearchTool`: For retrieving real-time information from the web
   - `VisitWebpageTool`: For extracting content from specific URLs
   - `PythonInterpreterTool`: For executing calculations and code
3. **Tool-Calling Agent**: Develop the agent loop that enables:
   - Tool selection based on LLM reasoning
   - Tool execution and result processing
   - Multi-step reasoning with tool interaction
4. **Comparative Evaluation**: Analyze performance differences between vanilla approaches and tool-augmented agents across different task types

Through this process, you will gain practical experience in building and evaluating LLM agents while developing an understanding of how different tools and architectures impact performance on various tasks.

3.4 Open Problem (Extra 15%)

In this open problem, your goal is to enhance the capabilities of your tool-calling LLM agent to better handle complex, research-oriented tasks from the GAIA dataset by exploring advanced agent architectures, specifically a **multi-agent system**. Here, multiple specialized agents collaborate to solve a task, extending beyond the single-agent framework.

Your tool-calling agent, built with tools like `GoogleSearchTool`, `VisitWebpageTool`, and `PythonInterpreterTool`, has improved performance on SimpleQA, MATH, and GAIA datasets. However, GAIA's complexity—requiring planning, knowledge retrieval, computation, and synthesis—suggests room for further improvement. Multi-agent systems offer a promising approach by distributing tasks across specialized agents.

Design and implement a multi-agent system to improve performance on the GAIA dataset. Each agent should handle a specific role (e.g., planning, retrieval, calculation), collaborating to solve complex tasks.

3.4.1 Suggested Approaches

- **Planning Agent**: Breaks tasks into sub-tasks and assigns them.
- **Coding Agent**: Performs computations with the Python interpreter.
- **Knowledge Retrieval Agent**: Gathers and summarizes information using search tools.
- ...

3.4.2 Requirements

- **Implementation**: Extend your framework to support at least two agents, demonstrating collaboration on a GAIA task.
- **Evaluation**: Test on GAIA examples, comparing to the single tool-calling agent.
- **Documentation**: Explain your design, challenges, and reflections in ipynb.

Deliverables and Submission

Similar to assignment 1, please clear all output and rerun all cells to ensure they are executable and retain all logs. Complete questions in Section 1. Discuss the performance of FewShotReasoner and FewShotCoTReasoner in Section 2. Submit your ipynb file with the name `UNIVERSITYNUMBER.ipynb`.

Literatur

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021. URL https://api.semanticscholar.org/CorpusID:239998651.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022. URL https://api.semanticscholar.org/CorpusID:246411621.