COMP3361: Assignment 3 Tutorial

Natural Language Processing - Spring 2025

Yiheng Xu

University of Hong Kong

April 22, 2025

Outline

1. **Decoding Algorithms**
2. **Few-Shot Math Reasoning**
3. **Building LLM Agents**

Section 1: Decoding Algorithms

- Understanding the methods to generate text from language models
- Implementing different decoding strategies:
  - Greedy decoding
  - Vanilla sampling
  - Temperature sampling
  - Top-k sampling
  - Top-p (nucleus) sampling
- Evaluating generation quality: perplexity, fluency, diversity

What Are Decoding Algorithms?

- Language models output probability distributions over vocabulary
- Decoding algorithms: Methods to convert these probabilities into actual token sequences
- **Key question**: Given logits (pre-softmax values), how do we choose the next token?
- Trade-off between:
  - **Quality/Fluency**: How natural/coherent is the text?
  - **Diversity**: How varied/creative are the outputs?

The Decoding Function Framework

- You'll implement core functions for each decoding method
- The assignment provides a wrapper function:

```
def decode(prompts, max_len, method, **kwargs):
    # Handles batching, max length, EOS token, etc.
    # Calls the specific decoding method function
    # You only need to implement the core method
    # ...

```

- Each core function takes pre-softmax logits as input and outputs next token indices

Greedy Decoding

- Simplest approach: always select token with highest probability
- Implementation concept:

```
def greedy(next_token_logits):
    # argmax over last dimension
    return next_token_logits.argmax(dim
    =-1)
```

- **Note**: No softmax needed—`argmax` is enough

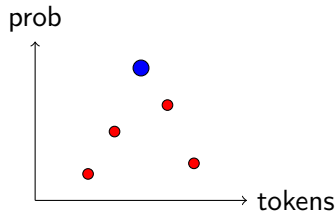


Figure: Always pick highest probability

Vanilla Sampling

- Randomly sample from the probability distribution
- Implementation concept:

```
def sample(next_token_logits):
    # Convert logits to probabilities
    # using softmax
    # Sample a token from this
    # probability distribution
    pass

```

- Helps with diversity but can be erratic

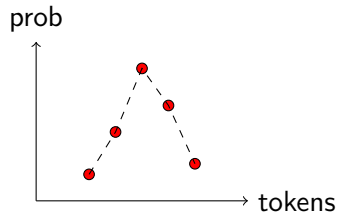


Figure: Sample from full distribution

Temperature Sampling

- Controls "randomness" of generation:

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

- t is the temperature parameter:
  - $t \to 0$: More deterministic (greedy)
  - $t = 1$: Standard softmax (vanilla)
  - $t > 1$: More random (flatter distribution)

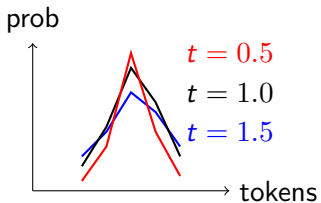


Figure: Effect of temperature on distribution

Top-k Sampling

- Only sample from top k most probable tokens
- Discards unlikely tokens from distribution
- Common values: $k = 20, 40, 100$
- Implementation steps:
  - Find top-k tokens by probability
  - Mask out other tokens by setting to $-\infty$
  - Normalize probabilities for sampling

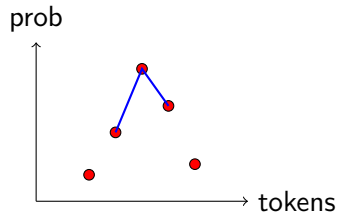


Figure: Sample from top-k tokens (k=3)

Top-p (Nucleus) Sampling

- Sample from the smallest set of tokens whose cumulative probability exceeds p
- Adaptive: number of tokens considered varies based on distribution
- Common values: $p = 0.9, 0.95$
- Implementation steps:
  - Sort tokens by probability
  - Compute cumulative probabilities
  - Take tokens until reaching threshold p
  - Normalize and sample

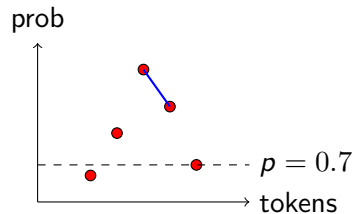


Figure: Top tokens until cumulative prob $\geq p$

Evaluation Metrics

- **Perplexity**: Measures how "surprised" a model is by the text
  - Lower = more natural language
  - PPL $= \exp(-\frac{1}{N}\sum_{i=1}^{N}\log p(w_i|w_{<i}))$
- **Fluency**: Rating of grammatical correctness
  - Uses a classifier trained on CoLA corpus
  - Higher = more grammatically correct
- **Diversity**: Measured as ratio of unique n-grams to total n-grams
  - Higher = more diverse text
  - Calculated for unigrams, bigrams, and trigrams

Expected Trade-offs

- Greedy decoding:
  - ✅ High fluency, low perplexity
  - ❌ Low diversity, repetitive text
- Vanilla sampling:
  - ✅ High diversity
  - ❌ Higher perplexity, lower fluency
- Temperature / Top-k / Top-p:
  - ⚖️ Balance between quality & diversity
  - ⚙️ Tunable parameters for different applications

Implementation Tips

- For greedy: Consider `torch.argmax()`
- For sampling: Use `torch.multinomial()` to sample from probability distribution
- For temperature: Scale logits before applying softmax
- For top-k: Use `torch.topk()` to find k highest values
- For top-p: `torch.sort()` can order probabilities for cumulative sum
- Remember to handle batched inputs (first dimension is batch size)
- Watch out for numerical stability (using `log_softmax` can help)

Section 2: Few-Shot Math Reasoning with LLMs

- Using LLMs for mathematical problem-solving
- Exploring in-context learning techniques:
  - Few-shot learning
  - Chain-of-thought (CoT) reasoning
- GSM8K dataset for evaluation (grade school math problems)

In-Context Learning

- Method for guiding LLMs without fine-tuning
- Provide examples within the prompt
- **Few-shot learning**: Show model a few examples before asking it to solve a new problem
- Leverages model's ability to:
  - Recognize patterns
  - Follow demonstrated formats
  - Apply similar reasoning strategies

GSM8K Dataset

- Grade School Math 8K: Collection of 8,000 grade school math word problems
- Examples:
  - "There are 15 trees in the grove. Grove workers will plant trees in the grove today. After they are done, there will be 21 trees. How many trees did the grove workers plant today?"
  - "Leah had 32 chocolates and her sister had 42. If they ate 35, how many pieces do they have left in total?"
- Tests multi-step reasoning
- Requires numerical computation

Basic Few-Shot Prompting

- Implementation for `FewShotReasoner`:
  - Format examples as Question-Answer pairs
  - Build the prompt with examples followed by the new question
  - Submit to LLM and extract the answer
- Template structure:

```
Answer the following questions.

Question: {Question1}
Answer: {Answer1}

Question: {Question2}
Answer: {Answer2}
...

Question: {New question}
Answer:

```

Chain-of-Thought Prompting

- Extension of few-shot learning that shows step-by-step reasoning
- Prompts the model to "think aloud" before answering
- Example CoT answer:

```
Question: There are 15 trees in the grove.
Grove workers will plant trees in the grove today.
After they are done, there will be 21 trees.
How many trees did the grove workers plant today?

Answer: There are 15 trees originally.
Then there were 21 trees after some more were planted.
So there must have been 21 - 15 = 6.
So the answer is 6.

```

Implementing Few-Shot CoT

- Similar to basic few-shot, but include reasoning steps
- Need function to extract final answer after generation

```
def extract_answer(cot_text):
    # Look for final answer in the reasoning
    # Common patterns:
    # "So the answer is X" or
    # "Therefore, X" or "X is the answer"
    # Use regex to find numerical answers
    pass

```

- Important to handle different formats correctly

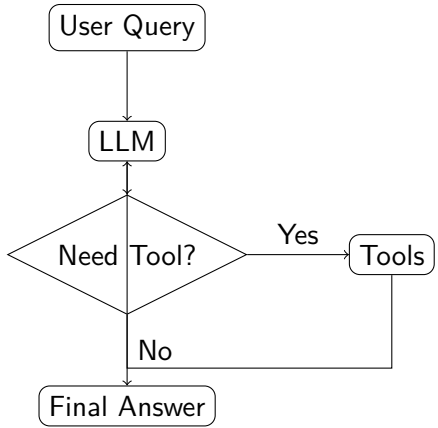
Key Considerations for CoT

- Consistent formatting of examples
- Selecting diverse and representative examples
- Extracting the final numerical answer from verbose responses
- Regular expression patterns that can capture various answer formats
- Handling edge cases (e.g., decimal answers, negative numbers)
- Evaluating answer correctness beyond exact string matching

Section 3: Building and Evaluating LLM Agents

- Moving beyond standalone LLM capabilities
- Augmenting LLMs with tools for:
  - Web search
  - Webpage content extraction
  - Python code execution
- Creating an agent loop for multi-step reasoning
- Evaluating performance across different task types

Agent Architecture



- **Agent Loop**: Repeatedly consult LLM for next steps
- **Tool Use**: Call external functions when needed
- **Multi-step Reasoning**: Break complex tasks into manageable pieces

Tools Overview

- `FinalAnswerTool`:
  - Standardizes output format
  - Signals end of agent execution
- `GoogleSearchTool`:
  - Retrieves information from the web
  - Enables access to factual knowledge
- `VisitWebpageTool`:
  - Extracts content from specific URLs
  - Converts HTML to markdown for LLM processing
- `PythonInterpreterTool`:
  - Executes Python code
  - Enables calculations and data processing

Tool Implementation - Web Search

```python
class GoogleSearchTool(Tool):
    name = "web_search"
    description = "Performs a google web search..."
    inputs = {
        "query": {"type": "string", "description":
                  "The search query to perform."},
    }
    output_type = "string"

    def forward(self, query: str) -> str:
        # Use Serper API to perform web search
        # Convert results to formatted text
        # Return top search results
        pass
```

Tool Implementation - Visit Webpage

```python
class VisitWebpageTool(Tool):
    name = "visit_webpage"
    description = "Visits a webpage at the given url..."
    inputs = {
        "url": {"type": "string", "description":
                "The url of the webpage to visit."},
    }
    output_type = "string"

    def forward(self, url: str) -> str:
        # Fetch HTML content
        # Convert to markdown for LLM processing
        # Handle various error cases
        pass
```

ToolCallAgent Implementation

Key components to implement:

- `_execute_tool`: Execute a tool with given arguments
- `_parse_tool_arguments`: Extract tool name and args from LLM response
- `run`: Main agent loop that:
  1. Consults the LLM for next action
  2. Parses the output to identify tool call
  3. Executes the tool and captures result
  4. Feeds result back to LLM for next step
  5. Repeats until final answer or max steps reached

Agent Prompt Engineering

Effective system prompts for agents should:

- Clearly describe available tools and their capabilities
- Explain the expected format for tool calls
- Encourage the agent to:
  - Break down complex problems
  - Use tools judiciously (only when needed)
  - Think step-by-step
  - Provide reasoning before taking actions
  - Return final answer when task is complete
- Include examples of proper tool usage (few-shot examples)

Evaluation Datasets

- **SimpleQA**:
  - Factual questions requiring web search
  - Tests basic knowledge retrieval
  - Example: "What is the durability of the Istarelle spear from Demon's Souls (2009)?"
- **MATH**:
  - Mathematical problems requiring calculation
  - Tests computational reasoning
  - Example: "How many zeroes are at the end of $42!$ (42 factorial)"
- **GAIA**:
  - Complex research-oriented questions
  - Requires multiple capabilities (search, calculation, synthesis)
  - Example: "How many studio albums were published by Mercedes Sosa between 2000 and 2009 (included)?"

Comparative Analysis

Expect different performance patterns:

- **Vanilla Agent** (no tools):
  - ✅ Simple problems within model knowledge
  - ❌ Struggles with factual accuracy, current events
- **Tool-calling Agent**:
  - ✅ Factual questions (via search tools)
  - ✅ Complex calculations (via Python interpreter)
  - ✅ Research questions requiring multiple steps
  - ❌ May struggle with tool selection, overuse
- Key metrics: accuracy, relevance, computational correctness

Common Challenges

- **Tool Selection**: When to use which tool
- **Parsing Tool Calls**: Extracting structured data from LLM outputs
- **Hallucinations**: LLM may fabricate tool responses
- **Context Management**: Keeping track of information across steps
- **Error Handling**: Recovering from tool failures
- **Stopping Criteria**: Determining when task is complete
- **Tool Overuse**: Unnecessary tool calls when LLM could answer directly

Open Problem: Multi-Agent Systems

For the extra credit challenge:

- Design multiple specialized agents working together
- Potential agent roles:
  - Planning Agent: Breaks task into subtasks
  - Coding Agent: Performs computations
  - Knowledge Retrieval Agent: Gathers information

Implementation Tips and Codebase Updates

- Updated the `decode` function to be compatible with the latest transformers.
- Renamed the `evaluate` helper file to `eval_utils` to prevent conflicts with S1.
- The Kaggle notebook platform offers better GPUs (two T4s) and a more stable quota as an alternative.

Q&A

Questions?

Good luck with Assignment 3!