Assignment 4: Decoding Algorithms

Instructor: Yejin Choi *CSE 517/447 Win 24*

Due at 11:59pm PT, March 8, 2024
(100 pt for both 447 and 517 with 5 pt extra credit, 14% towards the final grade)

In this assignment, you will implement and experiment with various decoding algorithms for language generation. First, we will focus on basic decoding techniques like greedy decoding, top-p/top-k sampling, and beam search. Next, we will delve into a lexically-constrained decoding algorithm (NeuroLogic).

Remember to **specify your collaborators** (including AI tools like ChatGPT) and **how they contribute** to the completion of your assignment at the beginning of your write-up. If you work on the assignment independently, please specify so, too. **NOT properly acknowledging your collaborators will result in -2 % of your overall score on this assignment.**

Required Deliverables

- **Code Notebook**: §1 and §2 share the same Python notebook (A4S12.ipynb); §3 has its own notebook (A4S3.ipynb). However, as you do not need to submit any code for §3, please only download the notebook for §1 and §2 as a Python file (`A4S12.py`) and submit it via Gradescope.
- **Write-up**:
  - For written answers, produce a single PDF for §1-3 and submit it in Gradescope. We recommend using Overleaf to typeset your answers in LaTeX, but other legible typed formats are acceptable. We do not accept hand-written solutions.

Acknowledgement

This assignment is designed by Ximing Lu (NeuroLogic decoding), Jaehun Jung (beam search), Gary Jiacheng Liu (other basic decoding), with invaluable feedback from Liwei Jiang, Alisa Liu, and Yejin Choi.

0 Set Up Evaluation Metrics

0.1 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.

0.2 Evaluation Metrics

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.

1 Basic Decoding Algorithms (40%)

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**.

In the notebook A4S12.ipynb, 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.**

1.1 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.

1.2 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,

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

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 $t \in (0, \infty)$.

1.3 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 $k \in [1, |V|]$.

1.4 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 $p \in [0, 1]$.

1.5 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 (20%):** Implement code blocks denoted by `TODO:` in Section 1 of **A4S12.ipynb**.
2. **Write-up (20%):** Answer the following questions in your write-up:
   - **Q1.1:** In greedy decoding, what do you observe when generating 10 times from the test prompt?
   - **Q1.2:** In vanilla sampling, what do you observe when generating 10 times from the test 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 has the best and worst perplexity? Fluency? Diversity?
   - **Q1.7:** Is there a method that wins over all others on all three metrics? If not, which method strikes the best balance in your opinion? (There is no single correct answer here, anything reasoning and faithful to your experimental results will receive full credit.)

2 Beam Search (40% + 2% Extra Credit)

You will implement `beam_search()` in Section 2 of the Python notebook A4S12.ipynb for this part of the assignment.

2.1 Beam Search Implementation

Conceptually, beam search is a straightforward extension to greedy decoding—where we retain the top-k hypotheses instead of the top-1 hypothesis in greedy decoding. However, implementing it is much more complex than the intuition. While there are many "arbitrary" decisions one needs to make to implement beam search, for the purpose of this assignment, we will do the following:

- Compared to greedy decoding, beam search introduces one additional hyper-parameter, `num_beams`, denoting the number of hypotheses we retain at each time step. For example, set `num_beams = 3`.
- At each time step, we have access to 3 hypotheses h_1, h_2, h_3 generated so far, along with the corresponding scores s_1, s_2, s_3 (typically defined as the average log probability of tokens in each hypothesis). To update the hypothesis, we compute next token distribution p_i for each hypothesis h_i, then take top-`num_beams` tokens (that makes the highest score when appended to the corresponding hypothesis) across all p_is.
- At each time step, in some of the hypotheses (say h_1), we may encounter the EOS token `<EOS>`. In greedy decoding, we can simply terminate the generation process, but in beam search, there still remain the 2 unfinished hypotheses. In this case, we move the *finished* hypothesis (h_1, which encountered EOS token) into a separate list. For the remaining *unfinished* hypotheses h_2 and h_3, we keep searching for the best next tokens, retrieving the top `num_beams = 3` tokens to create 3 new hypotheses (and their corresponding scores) for the next time step.
- After reaching the `max_length` step, we have two sets of hypotheses—a set of *unfinished* hypotheses that did not encounter `<EOS>` until the final step, and a set of *finished* hypotheses. Finally, we compare all hypotheses across the two sets, and return the one with the highest score.

Fill in the `TODO`s in `beam_search()`. Given the complexity of the implementation, we provide more structured skeleton for the algorithm, along with two helper classes `BeamHypothesisList` and `BeamManager`. Note that the two classes are already fully implemented, your job is to finish `beam_search()` using the two helper classes. You may find the beam search implementation in Huggingface transformers useful. In addition, we leave comments for the expected size of tensor variables for sanity check.

2.2 Evaluation

Run the evaluation cell. The code block will use `num_beams = 5`, `max_length = 100`, generating the best continuation for each prompt for the first 100 samples in the test set.

Note: The results from beam search evaluation are not directly comparable to the results from §1.5, even if we use exactly same evaluation data. This is because due to compute limitation, we set `num_beams = 5` and only output the best sequence we found from the 5 beams, while in §1.5, we sampled 10 sequences per each prompt and averaged the evaluation metrics across all 10 generations.

Deliverables:

1. **Code (25%):** Implement code blocks denoted by `TODO:` in Section 2 of **A4S12.ipynb**.
2. **Write-up (15% + 2% extra credit):** Answer the following questions in your write-up:

- **Q2.1:** Report the evaluation metrics for beam search you got after running the evaluation cell.
- **Q2.2:** During the implementation of beam search, we set `unfinished_beam_scores[:, 1:] = -1e9`, i.e., set all initial beam scores to a very small value except for the first beam. Briefly explain why we need this for the accurate initialization of beam scores.
- **Q2.3 (optional, 2% extra credit):** During the implementation of beam search, we actually retrieve top-`(2 * num_beams)` tokens at each step, instead of top-`num_beams` tokens. Briefly explain why we retrieve larger number of tokens than the number of hypotheses we retain. (Hint: consider a case where we retrieve top-`num_beams` tokens, and all top-`num_beams` tokens happend to be `<EOS>`—what would happen in `BeamManager.process`?)

3 Lexically Constrained Decoding: NeuroLogic (20% + 3% Extra Credit)

In this exercise, you will be provided a codebase of a simplified version of NeuroLogic Decoding to enable lexically constrained generation in `A4S3.ipynb`. In many scenarios, text generation applications need to incorporate lexical constraints, i.e., what words should or shouldn't appear in the outputs. Consider the task of generating a recipe from a set of ingredients, such as "garlic," "steak," and 'soy sauce." A generated recipe should cover all of those ingredients without hallucinating new ones (such as "pork" or "beans").

Here, we provide an implementation of NeuroLogic Decoding with two simplifications:

- We only consider **positive constraints** in this assignment (i.e., words to include), we will ignore negative constraints (i.e., words to exclude).
- All the constraint words to include would be **single token** in this assignment, we provide a function `convert_word_to_token` in the starting code.

Concretely, the task is to generate a continuation for a given prompt (e.g., *"It is sunny today."*) while including the given constraint word(s) (e.g., `dog`). While beam search aims to maximize the likelihood of the generated sequence, NeuroLogic Decoding leverages a variation of beam search to find optimal output sequences among the strings that also satisfy the given constraints. We can make two modifications to the beam search you implemented earlier to achieve the simplified NeuroLogic Decoding.

- Beam search expand the top `k` most probable next tokens at each branch, NeuroLogic Decoding do the same but also append the tokens that will *fulfill one more constraint*.
- Beam search selects the top `k` candidates with highest likelihood as hypothesises to expand for the next step, NeuroLogic Decoding selects hypotheses in consideration of both the likelihood and constraint satisfaction status. Concretely, we first group all the candidates based on the set of satisfied constraints and select the best ones from each group to fill the beam. We then proceed in rounds of filling the beam, visiting each group and taking the best scoring ones in rotation, until we reach k candidates.

3.1 NeuroLogic Implementation

To reduce your workload in this assignment, we provide all of the code of simplified NeuroLogic Decoding for you. **You do not need to implement anything, but make sure to read and understand the codebase.** You will be asked to answer conceptual questions about the codebase in your write-up.

Note that Neurologic decoding is a variation of beam search, so you should build your understanding on top of the beam search implementation in §2.

3.2 Evaluation

Run the evaluation cell. The code block will use `num_constraint = 5`, `num_beams = 5`, `max_length = 50`, generating the continuation that satisfied the given keyword constraints for each prompt *for the first 100 samples in the test set*. Same as before, it will output perplexity, fluency, and diversity evaluation metrics.

Deliverables:

1. **Write-up (20% + 3% extra credit):** Answer the following questions in your write-up:
   - **Q3.1:** Report the evaluation metrics for the NeuroLogic search you got after running the evaluation cell with our provided implementation. Show 3 example outputs decoded by NeuroLogic along with their corresponding prompts and constrained keywords; comment on their quality.

- **Q3.2:** Briefly explain what the following code blocks do in the provided implementation of the NeuroLogic Decoding:
  a. The `Word` class.
  b. The `ConstrainedHypothesis` class.
  c. The `ConstrainedCandidate` class.
  d. The `initialize_constraint()` function.
- **Q3.3:** What changes have been made to `BeamHypothesisList` and `BeamManager` for NeuroLogic compared to Beam search? Why they're necessary to enable constrained generation?
- **Q3.4:** Explain what the `rerank_beam()` does in a short paragraph or in pseudocode.
- **Q3.5 (optional, 3% extra credit):** In which case, the constraint satisfaction rate is guaranteed to be 100%? Please explain or write a short proof. Hint: consider the relationship between number of constraints and beam size.