Autocomplete Algorithm for Language Models

Kanghyeon Kim
KAIST
kaist19@kaist.ac.kr

Jiseon Kim
KAIST
jiseon_kim@kaist.ac.kr

Alice Oh
KAIST
alice.oh@kaist.edu

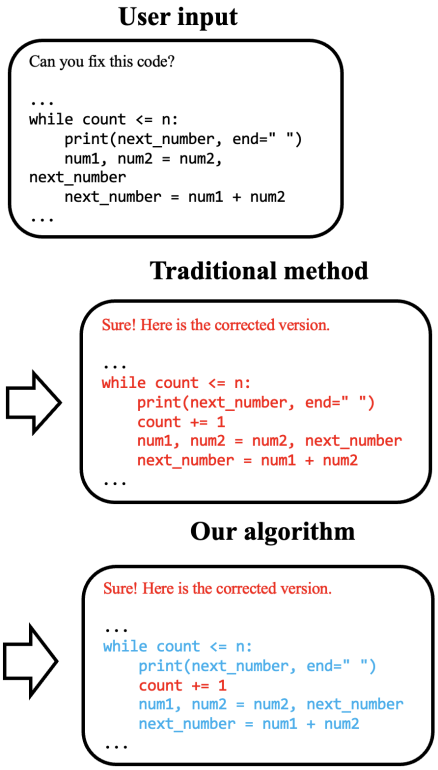

Figure 1: An example of response generation. The red tokens represent what had to be generated only by the large language model, while the blue tokens are those that could be autocompleted with the large language model and the N-gram model together.

Abstract

Generative language models often face challenges with computational inefficiency due to their autoregressive nature, where tokens are generated one at a time. This process can be resource-intensive, leading to high operational costs and environmental impact. In this paper, we propose an efficient algorithm that leverages an N-gram model to suggest tokens for text generation, thereby reducing the computational load on large language models (LLMs). Our approach reuses large segments of pre-existing text, significantly improving efficiency without compromising output quality. We demonstrate the effectiveness of our method through experiments on web browsing, summarization, and grammar correction tasks. Results show a notable improvement in text generation speed, making our algorithm a practical and scalable solution for enhancing LLM efficiency.

1 Introduction

Large language models (LLMs), trained on vast amounts of data, have revolutionized natural language processing tasks such as translation, summarization, and conversation simulation (Brown et al., 2020). Their integration into various applications has made sophisticated text generation accessible to a broad range of users.

Despite their impressive capabilities, LLMs often suffer from significant computational inefficiencies due to their autoregressive nature. In these models, tokens are generated sequentially, requiring numerous forward-passes, which lead to high computational costs and energy consumption. This inefficiency poses challenges in terms of operational expenses and environmental impact. Consequently, there is a pressing need for more efficient text generation methods that can produce high-quality outputs without the heavy computational burden of token-by-token generation.

Existing methods to improve efficiency include Blockwise Parallel Decoding (Stern et al., 2018), Speculative Sampling (Chen et al., 2023), and Assisted Generation (Joao Gante, 2023). These approaches typically involve using an additional, less powerful model to assist in the generation process, which can speed up the overall process but often requires additional training and resources.

In this paper, we introduce a novel algorithm that leverages an N-gram model to suggest tokens for text generation. Our method reuses large segments of pre-existing text, significantly improving efficiency without compromising the quality of the output. We demonstrate the effectiveness of our approach through experiments on various

tasks, including web browsing, summarization, and grammar correction. The results show a notable improvement in text generation speed, making our algorithm a practical and scalable solution for enhancing the efficiency of LLMs.

Our contributions are as follows: 1. We propose an efficient algorithm that leverages an N-gram model to suggest tokens for text generation, reducing the computational load on LLMs. 2. We demonstrate through experiments that our approach significantly improves the efficiency of text generation in specific use cases. 3. We provide a detailed analysis of the performance of our algorithm across different models and tasks, highlighting its strengths and limitations.

We find that our algorithm offers a practical and scalable solution to improving text generation efficiency, especially in scenarios where large segments of text can be reused from existing content.

2 Background and Related Work

As of 2024, an enourmous number of users use chatbot services (e.g. ChatGPT, Perplexity, and Claude) on a daily bases for a variety of tasks, which include tasks in which most part of the response can be copied from the prompt, such as grammar correction and summarization. Also, those services collect web browsing results before a response generation, and in this case, a large portion of responses are from the collected data as well. Our algorithm aims to deal with such inefficiencies caused by not just copying these repetitive parts but generating tokens one at a time.

Some previous works have focused on faster text generation for large language models. Blockwise Parallel Decoding (Stern et al., 2018) is a decoding scheme that makes predictions for multiple time steps in parallel. These predictions are validated with a scoring model. Speculative Sampling (Chen et al., 2023) uses a less powerful draft model for suggestion and validates it with a target model. Assisted Generation (Joao Gante, 2023) accelerates text generation with two large language models.

These approaches uses an additional model in common, which is weaker but faster than the original model. While it may be helpful for faster text generation, it requires an additional language model which has to be trained for such a use in some cases.

Our approach differs in that it does not rely on an additional large language model or require further training of such a model. Instead, we leverage a simple and efficient N-gram model to generate a set of suggested tokens based on the prompt and any provided context. This approach is computationally less intensive and bypasses the need for resource-heavy model training or the integration of a secondary language model. By doing so, our algorithm provides a practical and scalable solution to improving text generation efficiency, especially in scenarios where large segments of text can be reused from existing content.

3 Algorithm

Given a prompt, our algorithm first trains an N-gram model with the given prompt. When a prompt is provided with a context (web browsing result that was collected from the prompt), we use the context for training as well.

After training the N-gram model, the LLM begins to generate a response. Instead of passing the pre-generated response to LLM to sample the next token one at a time, we first pass the pre-generated response to the N-gram model to obtain a suggested set of tokens. For example, if the prompt from a user was "Who was the director of Harry Potter and the Deathly Hallows?" and the pre-generated response was "The director of," the N-gram model would suggest "Harry", "Potter", "and", "the", "the", "Deathly", "Hollows", and "?".

We pass the LLM the suggested tokens, added to the pre-generated response, all at once. In this way, we can obtain the distribution of the next tokens for the suggested tokens (the distribution of the next token after "Harry", the distribution of the next token after "Potter", and so on) all at once. With this information, our algorithm "validates" the suggested tokens. Starting from the first suggested token, it samples a token from the distribution and sees if the sampled token matches the suggested token. If it does, the suggested token becomes part of the pre-generated response. If it does not, that's where the LLM wants to deviate from the suggested tokens. The suggested token and the rest are dismissed, and instead, the algorithm adds the token that was sampled from the LLM to the pre-generated response. (For example, in the example above, the LLM would start deviating from the token "?" because it wants to say "was David Yates." instead.) With this new pre-generated response, the N-gram model suggests a set of tokens again, and this process is repeated until the last token is

generated. Using this algorithm, the LLM and the N-gram model can generate responses without having to generate tokens for repetitive parts one at a time.

The detailed steps of the algorithm are outlined in Algorithm 1.

3.1 Function Definitions

- **`LargeLanguageModel(input, cache)`**: This function represents the core language model's decoding mechanism. It takes the current input sequence and returns:
  - `distributions`: the distributions of the possible next tokens.
  - `cache`: cache data that allows the LLM to calculate only for newly added tokens when called next time.
- **`SampleToken(distribution)`**: This function takes a distribution of tokens and returns a sample token from the distribution.
- **`TrainNGramModel(text)`**: This function trains the N-gram model using the provided input sequence and returns the trained N-gram model.
- **`SuggestTokens(NGramModel, text, probability)`**: This function uses the trained N-gram model to suggest a set of tokens based on the input text and a specified probability threshold. It returns:
  - `suggested_tokens`: a set of suggested tokens with a cumulative probability over the specified threshold.

The algorithm iteratively generates the next token by either sampling directly from the model or by copying from existing text. This hybrid approach ensures that text generation is both efficient and contextually appropriate, significantly reducing the computational load compared to traditional token-by-token generation.

4 Experiments

4.1 Model Selection

We evaluated our algorithm using the following language models, fine-tuned for chatbot applications:

- **Llama-2-7b** (`Llama-2-7b-chat-hf`)
- **Llama-2-13b** (`Llama-2-13b-chat-hf`)
- **Meta-Llama-3-8B** (`Meta-Llama-3-8B-Instruct`)
- **Mixtral-8x-7B** (`Mixtral-8x-7B-Instruct-v0.1`)
- **Mistral-7B** (`Mistral-7B-Instruct-v0.3`)

4.2 Task Selection

We tested the models on four distinct tasks to measure performance and efficiency:

1. **Web Browsing** (Nakano et al., 2021): This dataset includes questions, web browsing results, and composed answers based on those results. Models were tested by providing questions and corresponding browsing results.
2. **Summarization** (See et al., 2017), (Hermann et al., 2015): This dataset contains CNN articles and their highlights. Models were tasked with summarizing these articles.
3. **Grammar Check**[1]: This dataset features pairs of sentences with grammatical errors and their corrected versions. Models were given erroneous sentences for correction.
4. **ChatGPT Prompts**[2]: This dataset contains pairs of prompts and responses from ChatGPT. Models were given prompts to generate responses.

4.3 Experimental Design

Two experiments were conducted for each model and dataset combination:

- **Experiment 1:** The models were provided with the prompts along with an instruction (e.g., "Summarize this article.").
- **Experiment 2:** The prompts from Experiment 1 were modified to include an additional instruction (e.g., "Answer in one sentence.") to limit the response length and evaluate the impact on efficiency.

To test the efficiency of our algorithm, we measured the average time taken to generate one token under two conditions: 1) using our algorithm, and 2) without using our algorithm. For each task, 100

[1] https://huggingface.co/datasets/Owishiboo/grammar-correction

[2] https://huggingface.co/datasets/MohamedRashad/ChatGPT-prompts

Algorithm 1 Efficient Text Generation Algorithm

Input: prompt
Input (optional): context

Require:
LargeLanguageModel(input, cache): Function that returns distributions and cache data.
SampleToken(distribution): Function that returns a sample token from a given distribution.
TrainNGramModel(text): Function that trains an N-gram model and returns the model.
SuggestTokens(NGramModel, text, probability): Function that returns a set of suggested tokens.

if context **exists then**
    pre_generated ← `concat`(prompt, context)
else
    pre_generated ← prompt
end if
NGramModel ← `TrainNGramModel`(pre_generated)
_, cache ← `LargeLanguageModel`(pre_generated, `None`)
while next_token ≠ **[end of sequence] do**
    suggested_tokens ← `SuggestTokens`(NGramModel, pre_generated, p)
    distributions, cache ← `LargeLanguageModel`(pre_generated, cache)
    for i **in** 1 **to** `length`(suggested_tokens) **do**
        sampled_token ← `SampleToken`(distributions[i])
        if sampled_token == suggested_tokens[i] **then**
            pre_generated ← `concat`(pre_generated, suggested_tokens[i])
        else
            pre_generated ← `concat`(pre_generated, sampled_token)
            break
        end if
    end for
end while
Output: pre_generated

prompts were used. The results, as detailed in Tables 1 and 2, represent the performance of our algorithm, with the baseline set to 100 (without our algorithm).

5 Analysis

In this section, we analyze the performance of the N-gram Parrot algorithm by comparing it against several state-of-the-art language models across different text generation tasks. The results are summarized in Tables 1 and 2.

5.1 Efficiency Gains

One of the primary objectives of our proposed algorithm is to enhance efficiency in text generation tasks. Our algorithm demonstrates a significant reduction in computational load for web browsing, summarization, and grammar check tasks for certain models. For instance, the Llama-2-7b-chat-hf model achieves an efficiency gain of approximately 15% in the web browsing task compared to traditional token-by-token generation methods. Similarly, notable improvements are observed in summarization and grammar check tasks.

The efficiency gains can be attributed to the algorithm's ability to reuse large segments of pre-existing text, reducing the need for generating each token from scratch. This is particularly evident in scenarios involving repetitive or formulaic text, where substantial portions can be copied directly, thereby minimizing computational overhead.

5.2 Performance Across Different Models

Table 1 and 2 provide a comparative analysis of the algorithm's performance across various models, including Llama-2-7b, Llama-2-13b, Meta-Llama-3-8B, Mixtral-8x7B, and Mistral-7B.

While LLMs such as Llama-2-7b can benefit from our algorithm for many tasks, other LLMs such as Meta-Llama-3-8B does not, which implies that the efficiency of the algorithm is highly dependent on how the language model has been trained. The LLMs that are trained not only to give a short answer to questions but also to provide explanations appear to benefit less or not benefit at all, as the capabilities of the N-gram model is limited to suggesting what's already written in the given prompt.

5.3 Task-Specific Observations

Although the efficiencies that the LLMs can gain from our algorithm differ depending on the task, for the ChatGPT-prompts task, all of the LLMs became rather slower. This suggests that our algorithm works better when a prompt contains enough sets of tokens that can be used for the N-gram model to suggest new tokens, and otherwise, it rather gets slower.

5.3.1 Limitations

Despite its advantages, the algorithm has limitations. For tasks where the input does not contain sufficient reusable token sequences, the algorithm's benefit diminishes, and it may even lead to inefficiencies. This is particularly evident in tasks like the ChatGPT-prompts, where models tend to generate more explanatory text, making less use of the N-gram suggestions.

Future work could focus on optimizing the algorithm to better handle such scenarios. One potential direction is to improve the algorithm's ability to discern when to switch from N-gram based suggestions to the model's generative capabilities, thereby minimizing the occurrence of "misses" that lead to inefficiencies. Additionally, further research could explore the integration of more sophisticated token suggestion mechanisms that adapt dynamically to the context and content of the input.

Our analysis suggests that while the algorithm shows promise in enhancing text generation efficiency, its performance is highly dependent on the specific characteristics of the language model and the nature of the task. Continued refinement and adaptation of the algorithm are essential to maximize its applicability and benefits across a broader range of use cases.

6 Conclusion

In this paper, we introduced the algorithm, a novel approach to improving the efficiency of text generation in large language models. Our experimental results demonstrate that this algorithm can achieve substantial computational savings while maintaining high-quality text output. By reusing existing text segments, the N-gram Parrot algorithm offers a promising direction for making sophisticated language models more accessible and environmentally friendly.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda

LLM	Web browsing	Summarization	Grammar check	ChatGPT-prompts
Llama-2-7b	**94.53%**	**94.04%**	**98.12%**	107.49%
Llama-2-13b	**98.79%**	**96.32%**	**95.83%**	104.90 %
Meta-Llama-3-8B	111.08%	107.09%	114.87%	116.00%
Mixtral-8x7B	109.74%	**99.79%**	110.90%	120.45%
Mistral-7B	101.84%	**98.41%**	105.74%	104.19%

Table 1: Comparison of model performance across different datasets

LLM	Web browsing	Grammar check
Llama-2-7b	**93.31%**	**87.86%**
Llama-2-13b	**98.36%**	**94.94%**
Meta-Llama-3-8B	102.49%	118.42%
Mixtral-8x7B	106.13%	104.99%

Table 2: Comparison of model performance across different datasets

Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling.

Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *NIPS*, pages 1693–1701.

Joao Gante. 2023. Assisted generation: a new direction toward low-latency text generation.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. Webgpt: Browser-assisted question-answering with human feedback. In *arXiv*.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models.