Stutter Makes Smarter: Learning Self-Improvement for Large Language Models

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) excel in generating coherent text but are limited by their large parameters and high memory requirements. Recent studies suggest that dynamically adjusting inference operations can enhance performance without significantly increasing model size. We introduce the stutter mechanism, which enables self-improvement by selectively applying additional layers to challenging tokens, mimicking a human stutter to allocate more computational effort where needed. Our experiments with Pythia models show that the stutter mechanism consistently improves performance across benchmarks. Notably, the Pythia-410M-stutter model outperforms the larger Pythia-1B model on WinoGrande and WSC. Additionally, our method is data-efficient, requiring less than 1% of the pretraining data for additional training. These results demonstrate the stutter mechanism's potential to enhance LLMs' efficiency and performance in real-world applications.

1 Introduction

Decoder-only transformers are the standard for large language models, excelling in generating coherent and contextually relevant text. However, efficiency and adaptability to varying input complexities remain areas for improvement. Typically, transformers process all inputs uniformly, ignoring varying difficulty levels. Inspired by recent upscaling studies, we aim to enhance language capabilities without significantly increasing model size.

In this paper, we propose the stutter mechanism, a minimally intrusive method to dynamically enhance a transformer's language ability through self-improvement. Similar to a human stuttering at key points in speech, this method selectively applies additional layers to challenging tokens, thereby improving performance without significant resource increase. This approach focuses on how to apply more layers once challenging tokens are identified, and it is compatible with any method that determines which tokens deserve more computational effort. The stutter mechanism requires only minor modifications to the existing transformer architecture, making it a minimally intrusive yet highly effective way to enhance the model's language capabilities through self-improvement.

We implemented our method on Pythia-160M, Pythia-410M, and Pythia-1B. Results show that the stutter mechanism effectively improves accuracies on various benchmarks. With this mechanism, smaller models can outperform larger ones. Our contributions are threefold:

- **Innovative Mechanism for Enhanced Language Capability**: We introduce the stutter mechanism, a novel and minimally intrusive method that dynamically allocates additional computational resources to more challenging tokens. This mechanism is compatible with existing methods for identifying tokens that require more computational effort, making it a versatile addition to current transformer architectures.
- **Performance Improvements on Various Benchmarks**: We demonstrate that the stutter mechanism consistently enhances the performance of transformer models on various bench-

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

marks. Specially, the Pythia-410M model, enhanced by the stutter mechanism, outperforms the larger Pythia-1B model on WinoGrande and WSC.

- **Data and Computational Efficiency**: We show that only one billion tokens (less than 1% of the pretraining data) are sufficient to train the stutter mechanism, reducing the computation time and cost significantly.

2 Methods

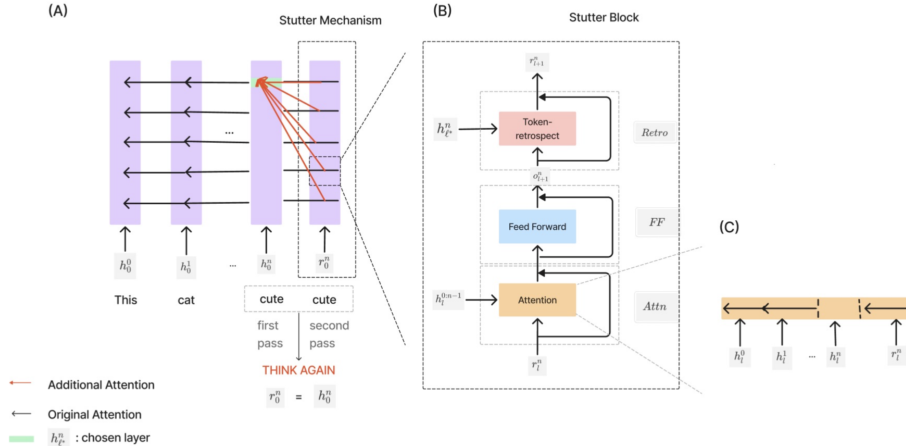


Figure 1: Overview of the proposed model architecture and stutter mechanism. (A) **Model Architecture**. Each purple column represents an inference step. Starting from the bottom, tokens are embedded as h_0^n and propagated through the transformer. When thinking upon the token (e.g., "*cute*") , the same token is fed into the model again for the second pass. During the second pass, the stutter mechanism is applied, using the hidden states of the chosen layer (highlighted). (B) **Stutter block.** In the second pass, each layer includes a stutter block with token-retrospect map applied after the pretrained feed-forward and attention mechanisms, along with a residual connection. (C) **Skipped attention**. During the second pass, the attention mechanism skips the hidden state from the first pass while attending to the previous tokens as usual.

In a prototypical transformer with L layers and a sequence of tokens $X = \{x_1, \dots, x_N\}$, the input representation of layer l and token n is denoted as h_l^n. The initial input h_0^n corresponds to the embedding of the previous output token. The transformation at layer l is given by $h_{l+1}^n = \text{FF}(\text{Attn}(h_l^{0:n-1}, h_l^n))$, where FF is the feed-forward network and Attn is the attention mechanism[1]. By the end of L layers, the output h_{L+1}^n is converted into logits by the language head $y^n = \text{Head}(h_{L+1}^n)$.

2.1 Stutter mechanism

The stutter mechanism enhances the model's ability to process a specific token n by performing inference twice. In the first pass, the model processes token n and stores the hidden state h_{l*}^n, capturing its semantic information. The hidden state before the last layer $h_{l*}^n = h_L^n$ is chosen as the semantic information from the first pass.

In the second pass, the stutter mechanism is applied, and each layer includes a stutter block, consisting of the original Attn and FF components, along with a token-retrospect map utilizing h_{l*}^n. The intermediate representation of layer l in the second pass is r_l^n, with $r_0^n = h_0^n$. The input r_l^n of the layer l first goes through the original architecture, producing an output $o_{l+1}^n = \text{FF}(\text{Attn}(h_l^{0:n-1}, r_l^n))$.

The o_{l+1}^n is then integrated with the hidden states from the first pass h_{l*}^n using the token-retrospect map. For layers l not higher than the chosen layer l^*, the transformation is described by:

$$r_{l+1}^n = \text{token-retrospect}(o_{l+1}^n, h_{l*}^n) + o_{l+1}^n,$$

[1]For simplicity, we have omitted the notation for positional embedding, normalization layers, and residual connections, although they are typically present in transformer architectures

where the token-retrospect map is the key component of the stutter mechanism. It is defined as:

$$\text{token-retrospect}(o_{l+1}^n, h_{l^*}^n) = \left(\frac{q_{o_{l+1}^n}^T k_{h_{l^*}^n}}{\sqrt{d_k}} \right) v_{h_{l^*}^n}, \qquad \forall l \leq l^*,$$

where $q_{o_{l+1}^n} = W_l^q o_{l+1}^n$, $k_{h_{l^*}^n} = W_l^k h_{l^*}^n$, $v_{h_{l^*}^n} = W_l^v h_{l^*}^n$ and W_l^q, W_l^k and W_l^v are additional attention parameters for training.

To enhance token generation with the help of its own insights, we apply attention to two hidden states linearly without using Softmax in the token-retrospect map. This allows the model to leverage stored hidden states for additional context. The stutter mechanism integrates the original model's result with hidden states from the chosen layer, improving contextual understanding and token generation.

2.2 Training and Loss

To train the proposed architecture, we start with an existing transformer and freeze all its weights except those in the token-retrospect map. Our primary objective is to demonstrate the effectiveness of the stutter mechanism, so the selection of specific tokens to stutter is beyond the scope of this paper. Therefore, during training, we stutter every token exactly once. Initially, we pass the training sequence X through the inherited transformer to capture $h_{l^*}^{0:N}$. Then, we train the stutter transformer by stuttering at every token, with each layer augmented by attending to the additional input. Only the additional attention parameters in the token-retrospect map are trained, which constitute only 10% of the entire model, requiring less data for training. Performance saturation was achieved with only 1 billion tokens, which is less than 1% of the pretraining data, showing competitive data efficiency.

We use the next token prediction loss as our primary loss term. This loss function is essential for language modeling tasks because it evaluates the model's ability to predict the next token in a sequence given the previous tokens.

3 Experiments

We used "The Pile" as our training dataset, a large-scale text corpus with about 210 million samples. We trained on 1 billion tokens for each model, using a parallel training setting similar to the Pythia model, combining hidden states, MLP outputs, attention outputs and token-retrospect outputs. We stored the input hidden states of the L-th layer for each token and initialized the token-retrospect map with Gaussian initialization. Checkpoints were saved every 5000 steps and evaluated on the LAMBADA dataset. Stuttering was enabled for all tokens during inference, with each token allowed to repeat once.

3.1 Evaluation

We evaluate the performance of different size of Pythia models on various benchmark datasets:

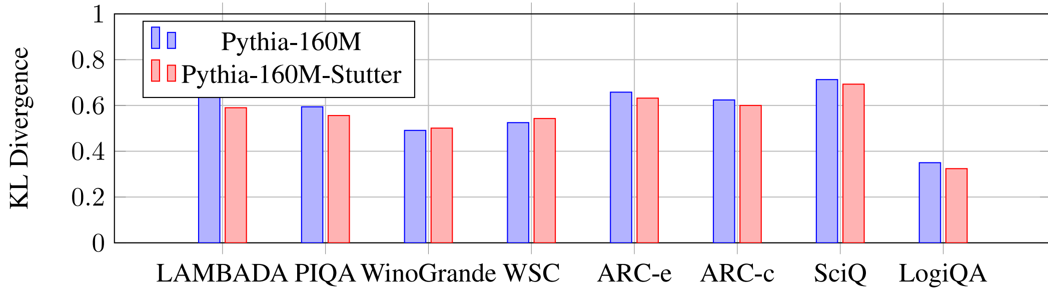
- **Pythia Model**: We used Pythia 160M, 410M, and 1B as base models to show the stutter mechanism's effectiveness across scales.
- **Benchmarks**: Evaluations were conducted on LAMBADA, PIQA, WinoGrande, WSC, ARC-e, ARC-c, SciQ, and LogiQA datasets, testing various aspects of language understanding and reasoning.

This section provides a comprehensive analysis of the performance improvements, distribution alignment, and layer effectiveness of the stutter mechanism in Pythia models. The analysis is divided into three main points:

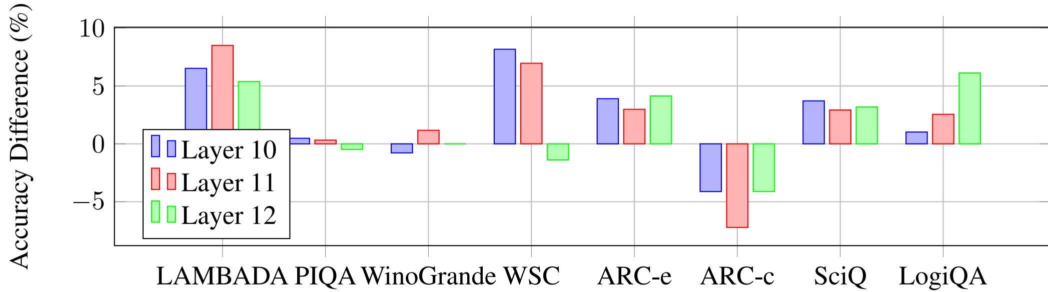
- **Performance Analysis of Pythia Models**: The stutter mechanism generally enhances performance across benchmarks. As shown in Table 1, Pythia-160M-Stutter improves LAMBADA 5-shot accuracy from 0.271 to 0.295 and 0-shot accuracy from 0.353 to 0.383. Similar improvements are seen in Pythia-410M and Pythia-1B models. Notably, Pythia-410M-Stutter achieves performance close to Pythia-1B, and even outperforms it in WSC and WinoGrande evaluation.

Table 1: Performance of Pythia-160M/410M/1B and Pythia-160M/410M/1B-Stutter on Various Benchmarks. Metrics are presented as 5-shot accuracy / 0-shot accuracy.

Benchmark	160M	160M-Stutter	410M	410M-Stutter	1B	1B-Stutter
LAMBADA	0.271 / 0.353	0.295 / 0.383	0.442 / 0.516	0.449 / 0.524	0.485 / 0.562	0.509 / 0.578
PIQA	0.625 / 0.623	0.631 / 0.625	0.680 / 0.667	0.688 / 0.682	0.714 / 0.707	0.716 / 0.700
WinoGrande	0.513 / 0.513	0.519 / 0.519	0.533 / 0.532	0.538 / 0.538	0.534 / 0.534	0.542 / 0.542
WSC	0.575 / 0.575	0.615 / 0.615	0.659 / 0.659	0.670 / 0.670	0.666 / 0.667	0.681 / 0.681
ARC-e	0.442 / 0.436	0.456 / 0.449	0.545 / 0.518	0.553 / 0.519	0.586 / 0.569	0.596 / 0.572
ARC-c	0.180 / 0.194	0.185 / 0.180	0.218 / 0.214	0.219 / 0.219	0.256 / 0.244	0.257 / 0.240
SciQ	0.780 / 0.754	0.789 / 0.776	0.892 / 0.815	0.894 / 0.829	0.917 / 0.839	0.927 / 0.853
LogiQA	0.235 / 0.196	0.225 / 0.201	0.230 / 0.216	0.215 / 0.213	0.238 / 0.225	0.216 / 0.224



(a) KL Divergence evaluation over 8 benchmarks.



(b) Pythia-160M-Stutter with Different Chosen Layers (0-shot) - Baseline Subtracted

Figure 2: (a) KL Divergence (b) Ablation study of different chosen layers

- **KL divergence Analysis**: We evaluated the KL divergence of Pythia-160M and Pythia-160M-Stutter with Pythia-1B as the target distribution. The stutter mechanism effectively aligns the output distribution of the smaller Pythia-160M model closer to that of the larger Pythia-1B model, as shown in Figure 2a.
- **Effectiveness of h_{l^*}**: We experimented with employing the stutter mechanism at different layers of the Pythia-160M model. Figure 2b shows that attending to layer 10 and layer 11 yields similar performance, while layer 12 generally results in lower improvements. This suggests that the last layer filters out some semantic information, making it less effective for the stutter mechanism.

4 Conclusion and Future Work

We propose the stutter mechanism to enhance LLM performance by facilitating an extended thinking process. This approach optimizes computational efficiency and improves performance across benchmark datasets. Future research could focus on optimizing the repeating mechanism, refining heuristics for the stutter mechanism, and interpreting the reasoning mechanism of LLMs to build trust and transparency in AI systems.

116 # References

117 [1] Takuya Akiba, Makoto Shing, Yujin Tang, Qi Sun, and David Ha. Evolutionary optimization of
118 model merging recipes. *arXiv preprint arXiv:2403.13187*, 2024.

119 [2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal,
120 Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel
121 Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler,
122 Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott
123 Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya
124 Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle,
125 M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information*
126 *Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

127 [3] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam
128 Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm:
129 Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–
130 113, 2023.

131 [4] Siqi Fan, Xin Jiang, Xiang Li, Xuying Meng, Peng Han, Shuo Shang, Aixin Sun, Yequan Wang,
132 and Zhongyuan Wang. Not all layers of llms are necessary during inference. *arXiv preprint*
133 *arXiv:2403.02181*, 2024.

134 [5] Danny Halawi, Jean-Stanislas Denain, and Jacob Steinhardt. Overthinking the truth: Under-
135 standing how language models process false demonstrations. *arXiv preprint arXiv:2307.09476*,
136 2023.

137 [6] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza
138 Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al.
139 Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

140 [7] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child,
141 Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language
142 models. *arXiv preprint arXiv:2001.08361*, 2020.

143 [8] Sanghoon Kim, Dahyun Kim, Chanjun Park, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeon-
144 woo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, Changbae Ahn, Seonghoon Yang, Sukyung
145 Lee, Hyunbyung Park, Gyoungjin Gim, Mikyoung Cha, Hwalsuk Lee, and Sunghun Kim.
146 SOLAR 10.7B: Scaling large language models with simple yet effective depth up-scaling. In
147 Yi Yang, Aida Davani, Avi Sil, and Anoop Kumar, editors, *Proceedings of the 2024 Conference*
148 *of the North American Chapter of the Association for Computational Linguistics: Human*
149 *Language Technologies (Volume 6: Industry Track)*, pages 23–35, Mexico City, Mexico, June
150 2024. Association for Computational Linguistics.

151 [9] Vedang Lad, Wes Gurnee, and Max Tegmark. The remarkable robustness of llms: Stages of
152 inference? *arXiv preprint arXiv:2406.19384*, 2024.

153 [10] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi,
154 Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. The lazy neuron phenomenon: On emergence
155 of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022.

156 [11] Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat,
157 Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open
158 sparse autoencoders everywhere all at once on gemma 2. *arXiv preprint arXiv:2408.05147*,
159 2024.

160 [12] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivas-
161 tava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient
162 llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176.
163 PMLR, 2023.

164 [13] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al.
165 Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[14] Qi Sun, Marc Pickett, Aakash Kumar Nain, and Llion Jones. Transformer layers as painters. *arXiv preprint arXiv:2407.09298*, 2024.

[15] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

A Appendix / supplemental material

A.1 Related work

In this section, an overview of key concepts and techniques relevant to the development of transformer models is provided. We discuss the architecture and scaling trends of decoder-only transformers, methods for upscaling and pruning, and approaches to improve computational efficiency. Additionally, we explore the loss functions used in training and the confidence levels of transformers in token prediction.

A.1.1 Decoder-only transformers

The Generative Pre-trained Transformer (GPT) series by OpenAI showcases the power of decoder-only transformer architectures [13, 2]. GPT-2, released in 2019 with 1.5 billion parameters, demonstrated impressive text generation capabilities. GPT-3, introduced in 2020, expanded to 175 billion parameters, significantly enhancing performance and enabling more complex and accurate text generation. This progression highlights the trend that increasing model parameters leads to substantial performance improvements [6].

As the number of parameters increases, the depth of the model also tends to increase. For example, GPT-2 has 48 layers, while GPT-3 scales up to 96 layers. This trend is also observed in various large language models where more layers are added to accommodate the growing number of parameters, thereby enhancing the model's capacity to learn complex patterns and dependencies in the data [15]. This scaling law is further supported by studies showing that larger models continue to improve performance with increased size [7].

A.1.2 Upscaling

While increasing the number of parameters and layers can enhance model performance, it also introduces significant computational challenges. To address these challenges, upscaling methods are employed to increase the parameter count and the depth of a transformer. These methods can be broadly categorized into training-free attempts and upscale-and-train attempts. Training-free upscaling involves techniques such as parameter sharing and repeating layers without additional training. Recently, merged LLMs have shown success in improving performance without re-training. An evolutionary algorithm is proposed in [1] to search for a better merge combination which is costly and limits the number of repetitions.

On the other hand, upscale-and-train methods involve increasing the model size and then training it on large datasets to achieve better performance. For instance, the SOLAR 10.7B model demonstrates effective depth upscaling techniques that significantly enhance model performance [8]. Additionally, the authors in [3] discuss how scaling pathways can be used to efficiently upscale models.

A.1.3 Layers skipping and pruning

Despite the benefits of upscaling, the increased model size can lead to inefficiencies during inference. To decrease the runtime computational requirements of a transformer, various methods such as layer skipping and pruning are employed. Layer skipping involves dynamically skipping certain layers during inference based on the input data, thereby reducing the computational load. Pruning, on the other hand, involves removing less important weights or neurons from the model, which can significantly reduce the model size and inference time while conceding some performance. The authors in [4] explore these techniques in detail, showing how selective layer usage can maintain performance while reducing computational costs. Another approach proposed in [12, 10] demonstrates that layer sparsity can be contextualized, suggesting that not all layers are necessary for processing simpler

input tokens. In addition, observations from [5] show that early-exiting in critical layers (around layer 28 in GPT2-XL) improves the model performance.

A.1.4 How confident is a transformer on a given token

Understanding the training and inference processes is essential [11], but it is equally important to evaluate the model's confidence in its predictions. The confidence of a transformer on a given token can be measured by the probability distribution it outputs for the next token prediction. Studies have shown that transformers can generate high-confidence predictions for certain tokens, which can be used to gauge the model's certainty in its predictions. While there are extensive studies on the overall performance of transformers in generating sequences, there is ongoing research to understand the confidence levels at the token level. For example, authors in [14, 9] discuss the confidence and interpretability of transformer layers in generating specific tokens. Additionally, the study delves into how models process and generate tokens with varying levels of confidence [5].