Breaking the Attention Trap in Code LLMs: A Rejection Sampling Approach to Enhance Code Execution Prediction

Anonymous ACL submission

Abstract

Code-specific Large Language Models (Code LLMs) have greatly improved performance across code-related tasks, offering substantial benefits in practical applications. However, existing research reveals significant performance bottlenecks in Code Execution tasks, which requires models to predict the execution results of given code snippets. This study identifies that, the *Attention Trap* phenomenon in training data constitutes a key constraint on model performance. To address this phenomenon, we propose the Attention Cracking with Rejection Sampling (AC-RS) method. The method first applies structural optimization to training data to eliminate attention traps. Then, it conducts secondary training on the outputs generated by the fine-tuned model to mitigate potential negative impacts from manual data intervention. Experimental results show that AC-RS significantly enhances the accuracy of Code Execution while preserving models' original capabilities. Notably, the optimized 7B model achieves Code Execution accuracy comparable to 32B model and GPT-4o.

1 Introduction

With the rapid advancement of large language models (LLMs) (OpenAI, 2022; Ouyang et al., 2022; OpenAI et al., 2024; Touvron et al., 2023a,b; Grattafiori et al., 2024; Bai et al., 2023; Yang et al., 2024), Code LLMs have attracted substantial academic and industrial attention due to their applicability and broad potential. From early models like StarCoder (Li et al., 2023) and CodeLlama (Rozière et al., 2024) to recent advancements including Deepseek Coder (Guo et al., 2024; DeepSeek-AI et al., 2024) and Qwen Coder (Qwen-Team, 2024; Hui et al., 2024), Code LLMs have shown remarkable performance across code-related tasks.

However, studies (Austin et al., 2021; Nye et al., 2021; Gu et al., 2024) indicate that current Code LLMs underperform in Code Execution tasks. Austin et al. (2021) reveals that even 137B model struggles to predict execution results of basic Python code, and fine-tuning only provides minimal performance gains. Nye et al. (2021) attributes this to the lack of explicit step-by-step reasoning before giving the predicted results. While previous work focuses on reasoning deficiencies, our work reveals that attention traps in widely-used LeetCode[1] training data fundamentally constrain execution prediction capabilities.

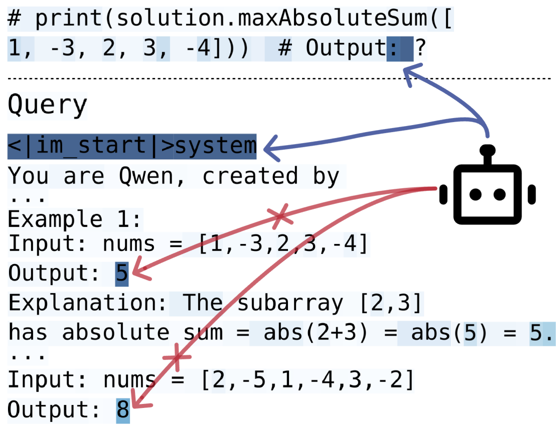

Figure 1: *Attention Trap* in Leetcode data.

When models process training data with lexical similarities, their attention mechanisms become overly focused on surface-level token correlations while neglecting deeper abstract relationships between data components. Lexical similarity-induced cognitive bias exhibits universality in deep learning systems (Gururangan et al., 2018; Liusie et al., 2022; Chew et al., 2024). For instance, models tend to misclassify samples containing categorical lexical cues (e.g., texts with "cinema" being erroneously categorized as "film"). We term this phenomenon "*Attention Trap*" in code execution training scenarios utilizing LeetCode data, and investigate how the *Attention Trap* affects the training process. Figure 1 demonstrates how trained

[1] https://leetcode.com/

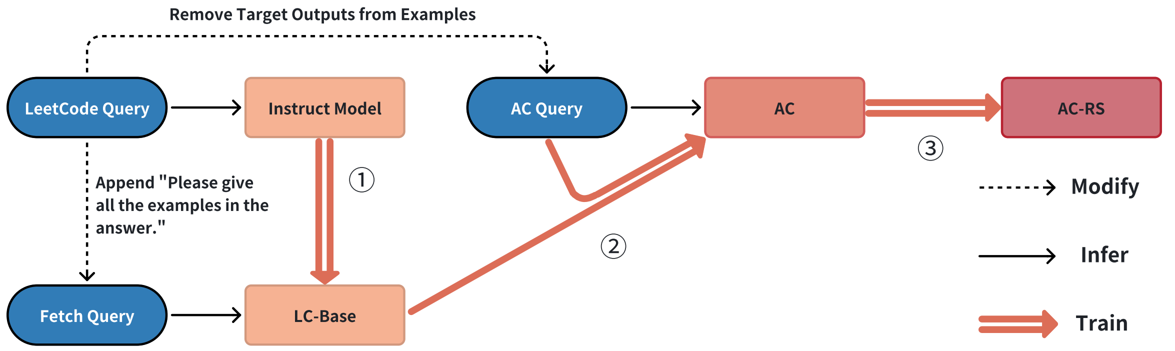

Figure 2: Pipeline of Attention Cracking with Rejection Sampling (AC-RS). The two edges of the orange arrow corresponding to "Train" represent the Query and Response used for training, respectively. For instance, Arrow ② denotes the use of AC Query as the Query and the result obtained through LC-Base model inference on Fetch Query as the Response, which fine-tunes the Instruct Model to derive the AC model. The full pseudocode is provided in Appendix C.

model distributes attention during predicting Code Execution result token. The target outputs in example sections (e.g., the values 5 and 8 in "Output: 5" and "Output: 8") exactly match the current token, which attracts high attention weights. During learning, models excessively attend to these target outputs in the input queries, preventing proper modeling of the multi-step reasoning chain connecting problem descriptions, program code, and execution results. Full example are provided in Appendix A.

To eliminate attention traps and analyze its effects on training process, we propose Attention Cracking with Rejection Sampling. Our method contains two stages: (1) Attention Cracking (AC) modifies training data to eliminate attention traps; (2) Rejection Sampling (RS) (Liu et al., 2024b) employs self-generated model outputs for secondary training, preventing performance degradation from manual data modifications. Experimental results demonstrate that AC-RS significantly improves performance with minimal data requirements. Using only 1,000 LeetCode samples, our method achieves 13.57% improvements on the Code Execution tasks of LiveCodeBench (Jain et al., 2024). It also shows 10.96% gains on Test Output Prediction tasks, which require predicting results from problem descriptions rather than code, while maintaining code generation capabilities.

2 Related Works

The field of Code LLMs originated from data-centric methodologies and has gradually developed into a thriving research area (Jiang et al., 2024). Early studies in code-related domains adopted data construction methods from general-purpose domains. For instance, Chaudhary (2023) employed the Self-Instruct (Wang et al., 2023) approach to automatically generate code instruction dataset CodeAlpaca. Luo et al. (2023) further enhanced this dataset through Evol-Instruct (Xu et al., 2023), training the WizardCoder model. Additionally, Magicoder (Wei et al., 2024) attempted to generate high-quality instruction tuning data using open-source code. As data-related challenges were progressively addressed, multiple high-performance open-source code models emerged. Representative examples include the Qwen Coder series and DeepSeek Coder series. Concurrently, researchers achieved notable progress in other dimensions of code-related tasks. Frameworks like MFTCoder (Liu et al., 2024a) and models like Phi (Abdin et al., 2024) advanced the field through multi-task training strategies and parameter efficiency improvements, respectively.

3 Method

This section details the implementation of AC-RS method. To ensure fair comparison and better prepare for subsequent training data generation, we first train LC-Base model using Leetcode data through Rejection Sampling (① in Figure 2). AC-RS method then introduces two formal stages: Attention Cracking and Rejection Sampling. The full pseudocode of AC-RS is provided in Appendix C.

3.1 Attention Cracking

The AC stage aims to eliminate attention traps. Concretely, by removing target outputs from

Table 1: Accuracy(%) on LiveCodeBench. Qwen2.5-Coder, DeepSeek-Coder, CodeLlama are all Instruct models.

Model	Size	Code Gen	Self Repair	Test Output Prediction	Code Execution	Avg
GPT-4o-0806	-	49.35%	59.75%	76.02%	58.04%	60.79%
Qwen2.5-Coder	32B	52.61%	62.25%	70.81%	57.41%	60.77%
CodeLlama	7B	10.29%	10.50%	25.11%	20.46%	16.59%
DeepSeek-Coder	6.7B	19.44%	24.25%	26.02%	39.67%	27.35%
Qwen2.5-Coder	7B	36.44%	45.75%	49.55%	44.68%	44.11%
LC-Base	7B	38.07%	48.50%	54.52%	48.23%	47.33%
AC	7B	37.42%	46.75%	57.92%	58.04%	50.03%
AC-RS	7B	39.54%	47.75%	60.41%	58.25%	51.49%

queries, we prevent models from relying on superficial pattern matching. This forces models to allocate attention to problem descriptions and generated code for output reasoning, effectively eliminating attention traps. The AC stage modifies LeetCode queries through two operations: (1) **AC Queries**: Remove target outputs from examples in original queries for training. (i.e., Remove "Output: 5" and "Output: 8" in Figure 1) (2) **Fetch Queries**: Append "Please give all the examples in the answer." at query endings, increasing the likelihood of including examples in retrieved results. Fetch Queries collect generation results from the LC-Base model. Generated results are processed by selecting responses with the same examples as queries, prioritizing those passing tests. Finally, we fine-tune the Instruct model using AC Queries and processed results to drive the AC model.

3.2 Rejection Sampling

To prevent performance degradation from AC stage data modifications, we introduce a RS stage. This mechanism directly uses AC queries to obtain outputs from AC-trained models, eliminating attention distortion caused by query-output mismatches. Through quality filtering of model-generated responses, RS substantially reduces training difficulty while mantaining data quality. Notably, we pre-applied Rejection Sampling in both LC-Base model and AC model training stages.

In implementation, we encountered output formatting issues when applying RS with LeetCode data(Appendix B). To resolve this, we developed specialized Helper models by combining CodeAlpaca samples (Chaudhary, 2023) with LeetCode/Fetch/AC queries. These Helper models effectively replace direct model generations for training purposes.

Table 2: Accuracy(%) on HE and MBPP. Qwen* represents Qwen2.5-Coder-7B-Instruct.

Model	HE	HE+	MBPP	MBPP+
Qwen*	87.19%	82.20%	83.33%	71.67%
LC-Base	86.10%	80.30%	84.92%	74.07%
AC	85.24%	79.63%	77.25%	67.20%
ACnE	84.63%	79.09%	75.66%	65.87%
AC-RS	86.46%	80.67%	83.07%	72.75%

4 Experiments

4.1 Datasets & Models

During training, we validate the AC method using LeetCode data from Shen and Zhang (2024). To build helper models, we randomly select 10,000 samples from CodeAlpaca (Chaudhary, 2023) and obtain corresponding outputs via GPT-4o-20240806 (OpenAI, 2024). For evaluation, we employ LiveCodeBench (Jain et al., 2024), HumanEval (HE) (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks with the EvalPlus framework (Liu et al., 2023) to assess AC-RS effectiveness. We additionally use the HumanEval (HE) (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks to evaluate the generalization code generation capabilities.

For model selection, Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) serves as baseline model. Comparative experiments include CodeLlama-7B-Instruct (Rozière et al., 2024), DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024), GPT-4o-20240806, and Qwen2.5-Coder-32B-Instruct (Hui et al., 2024). All models are trained using LLaMA Factory (Zheng et al., 2024) and deployed via vLLM (Kwon et al., 2023). Detailed experimental configurations are elaborated in Appendix D.

4.2 Results

Table 1 presents performance comparisons between AC-RS and other models on LiveCodeBench. Experimental results demonstrate that AC-RS outperforms Qwen2.5-Coder-7B-Instruct across all evaluation tasks. On the tasks central to our research objectives, Test Output Prediction and Code Execution, the method improves accuracy from 49.55% to 60.41% and 44.68% to 58.25%. Remarkably, AC-RS slightly outperforms larger models like Qwen2.5-Coder-32B-Instruct and GPT-4o-20240806 in Code Execution results.

We analyze contributions from both AC and RS stages. During AC implementation, models show significant gains in Test Output Prediction and Code Execution by avoiding attention traps in LeetCode data. However, this comes with a 0.65% decrease in general Code Generation ability comparing to LC-Base. This trade-off stems from using Fetch Query outputs as training data, which introduces misalignment issues between queries and outputs, causing distortion in data probability distributions. The Rejection Sampling (RS) stage addresses two critical challenges: It resolves query-output alignment issues through self-generated training data from AC-trained models, while simultaneously reducing model adaptation complexity. This stage further improves performance in Test Output Prediction and Code Execution, while maintaining Code Generation performance without degradation.

Experimental results from HumanEval (HE) and MBPP benchmarks (Table 2) further validate method robustness. During the AC stage alone, we observe performance declines of 0.86% on HE and 7.67% on MBPP, confirming the risks of data distribution disruption from single-stage optimization. However, the RS stage successfully mitigates these declines, with AC-RS ultimately matching LC-Base performance on both benchmarks.

4.3 Ablations & Discussions

This section systematically analyzes two core questions: (1) the necessity of introducing Fetch Queries, and (2) different implementations of the AC method.

ACnR vs. AC: How Output Refetch Amplifies Attention Shifting Figure 3 presents experimental results for ACnR (**A**ttention **C**racking with no **R**efetch). This approach modifies queries while retaining original outputs. Results show that ACnR improves Test Output Prediction and Code Execution performance, but achieves weaker gains (2.72% and 0.62% improvements over LC-Base) compared to the Refetch-enhanced AC method.

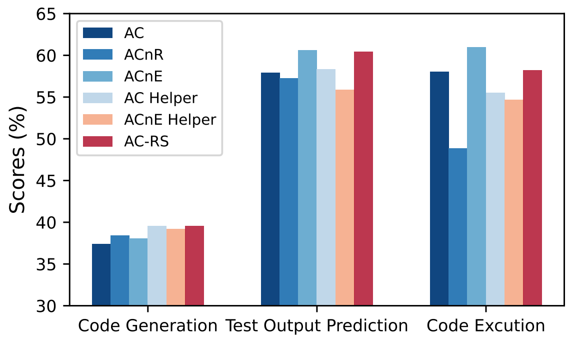

Figure 3: Model performance differences on LiveCodeBench in the ablation study. (ACnR refers to AC with no Refetch, ACnE refers to AC with no Example).

The limited improvement stems from insufficient example coverage in original outputs. Statistical analysis reveals that only 43.1% of LC-Base outputs contain examples. By introducing specially designed Fetch Queries, we increase the example-containing output ratio to 99.9%, significantly improving data collection efficiency.

ACnE vs. AC: Trade-offs Between Difficulty and Generalization A comparable approach to AC, termed ACnE (**A**ttention **C**racking with **n**o **E**xamples), eliminates entire example sections from LeetCode queries. ACnE shows comparable performence to AC-RS in Figure 3 but with increased learning demands: Models must not only predict execution results but also autonomously generate test cases. High leaning demands incurs two substantial costs: (1) Reduced generalization capability: Table 2 shows ACnE underperforms AC on both HE and MBPP benchmarks. (2) Limited multi-dataset compatibility: When trained with 10,000 CodeAlpaca samples (ACnE_Helper), performance declines significantly due to gradient signal dilution from other training data.

5 Conclusion

Our study proposes AC-RS method. The AC stage eliminates attention traps in training data through data restructuring. The RS stage addresses performance degradation by training models with self-generated outputs. Experimental results demonstrate that our 7B model trained with AC-RS achieves superior performance on LiveCodeBench. Notably, it matches the Code Execution accuracy of 32B parameter model and performs comparably to GPT-4o.

6 Limitations

While AC-RS effectively eliminate attention traps in Code Execution training data, two limitations persist: First, our validation remains constrained by the scarcity of high-quality open-source code instruction data and computational resource limitations. Second, AC-RS specifically targets Code Execution tasks. Systematically identifying diverse attention traps across massive training data and developing universal solutions remains an unresolved research challenge.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, and 110 others. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, and 29 others. 2023. Qwen technical report. *Preprint*, arXiv:2309.16609.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Oscar Chew, Hsuan-Tien Lin, Kai-Wei Chang, and Kuan-Hao Huang. 2024. Understanding and mitigating spurious correlations in text classification with neighborhood analysis. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 1013–1025, St. Julian's, Malta. Association for Computational Linguistics.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 540 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *Preprint*, arXiv:2401.03065.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel Bowman, and Noah A. Smith. 2018. Annotation artifacts in natural language inference data. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 107–112, New Orleans, Louisiana. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Wei Jiang, Hang Yu, and Jianguo Li. 2024a. Mftcoder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 5430–5441, New York, NY, USA. Association for Computing Machinery.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. 2024b. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*.

Adian Liusie, Vatsal Raina, Vyas Raina, and Mark Gales. 2022. Analyzing biases to spurious correlations in text classification tasks. In *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 78–84, Online only. Association for Computational Linguistics.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *Preprint*, arXiv:2112.00114.

OpenAI. 2022. Introducing chatgpt. `https://openai.com/index/chatgpt/`.

OpenAI. 2024. Hello gpt-4o. `https://openai.com/index/hello-gpt-4o/`.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

Qwen-Team. 2024. Code with codeqwen1.5. `https://qwenlm.github.io/blog/codeqwen1.5/`.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Wei Shen and Chuheng Zhang. 2024. Policy filtration in rlhf to fine-tune llm for code generation. *Preprint*, arXiv:2409.06957.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023b. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 52632–52657. PMLR.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *Preprint*, arXiv:2304.12244.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, and 43 others. 2024. Qwen2 technical report. *Preprint*, arXiv:2407.10671.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *Preprint*, arXiv:2403.13372.

A Examples for attention trap

Figure 4 and Figure 5 compare attention weight distributions between the LC-Base model and AC-RS model. The LC-Base model demonstrates clear attention trap patterns when processing LeetCode training data. During output learning, the model disproportionately focuses on reference answers in input queries rather than problem descriptions or code logic. Visual analysis reveals two dominant attention patterns in the LC-Base model: (1) Strong focus on sequence-initial tokens (2) Heightened attention to recent colon symbols (:) and space character. Beyond these common high-attention elements, tokens corresponding to example answers in queries receive maximum attention weights, followed by other example answers and their explanatory components. The AC method eliminates this trap through targeted training data modifications. These modifications enable proper attention allocation to critical elements like problem statements, input parameters, and program code. This improvement allows effective capture of abstract reasoning patterns in Code Execution tasks. The approach better utilizes the performance potential inherent in training data.

~~~~
<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.<|im_end|>
<|im_start|>user
You are given an integer array nums. The absolute sum of a subarray [numsl, numsl+1, ..., numsr-1, numsr] is abs(numsl + numsl+1 + ...
+ numsr-1 + numsr).
Return the maximum absolute sum of any (possibly empty) subarray of nums.
Note that abs(x) is defined as follows:

If x is a negative integer, then abs(x) = -x.
If x is a non-negative integer, then abs(x) = x.

Example 1:

Input: nums = [1,-3,2,3,-4]
Output: 5
Explanation: The subarray [2,3] has absolute sum = abs(2+3) = abs(5) = 5.

Example 2:

Input: nums = [2,-5,1,-4,3,-2]
Output: 8
Explanation: The subarray [-5,1,-4] has absolute sum = abs(-5+1-4) = abs(-8) = 8.


Constraints:

1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4


Please complete the following Python code:
```python
class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
```<|im_end|>
<|im_start|>assistant
To solve the problem of finding the maximum absolute sum of any subarray, we can utilize a strategy that involves tracking two running
computations as we iterate through the array:

1. **Maximum Subarray Sum (Kadane's Algorithm)**: This helps us find the maximum sum of any contiguous subarray. We'll use this to find
the maximum positive sum a subarray can achieve.

2. **Minimum Subarray Sum**: Similarly, this helps us find the minimum sum of any contiguous subarray. By negating the array elements
and applying Kadane's again, we can find the minimum sum, which helps in calculating the maximum absolute sum when considering negative
subarray sums.

The maximum absolute sum of a subarray can be either the maximum subarray sum or the maximum negative of the minimum subarray sum.
Additionally, any subarray that happens to sum to zero will also contribute to the maximum absolute sum as zero's absolute value is
zero, which might be the maximum if other max sums are negative.

Here's the implementation of the above logic:

```python
from typing import List

class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
        max_sum = 0
        min_sum = 0
        current_max = 0
        current_min = 0

        for num in nums:
            current_max += num
            current_min += num

            # Update global max and min sums
            max_sum = max(max_sum, current_max)
            min_sum = min(min_sum, current_min)

            # Reset current max and min if they drop below zero
            if current_max < 0:
                current_max = 0
            if current_min > 0:
                current_min = 0

        # The result is the maximum of the maximum positive sum
        # and the absolute value of the minimum negative sum
        return max(max_sum, abs(min_sum))

# Example usage:
# solution = Solution()
# print(solution.maxAbsoluteSum([1, -3, 2, 3, -4]))  # Output: 5
# print(solution.maxAbsoluteSum([2, -5, 1, -4, 3, -2]))  # Output: 8
```

### Explanation:

- **current_max** and **current_min** are used to track the sum of the current subarray.
- As we iterate through the array, we update these values by adding the current number.
- We update the global maximum and minimum sums (`max_sum` and `min_sum`) whenever the current values exceed or drop below them,
respectively.
- If at any point `current_max` or `current_min` becomes negative or positive, we reset them to zero because starting a new subarray
from the next element might yield a higher sum.
- Finally, the result is the maximum of `max_sum` and the absolute value of `min_sum`, ensuring we account for both positive and
negative subarray contributions to the maximum absolute sum.
~~~~

0.000	0.030	0.060	0.090	0.120	0.150	0.180	0.210	0.240	0.270	0.30+

0.000	0.030	0.060	0.090	0.120	0.150	0.180	0.210	0.240	0.270	0.30+

Current Step

Figure 4: Attention weights of the LC-Base model trapped in attention traps.

````
<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.<|im_end|>
<|im_start|>user
You are given an integer array nums. The absolute sum of a subarray [numsl, numsl+1, ..., numsr-1, numsr] is abs(numsl + numsl+1 + ...
+ numsr-1 + numsr).
Return the maximum absolute sum of any (possibly empty) subarray of nums.
Note that abs(x) is defined as follows:

If x is a negative integer, then abs(x) = -x.
If x is a non-negative integer, then abs(x) = x.


Example 1:

Input: nums = [1,-3,2,3,-4]

Example 2:

Input: nums = [2,-5,1,-4,3,-2]


Constraints:

1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4


Please complete the following Python code:
```python
class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
```<|im_end|>
<|im_start|>assistant
To solve the problem of finding the maximum absolute sum of any subarray, we can utilize a strategy that involves tracking two running
computations as we iterate through the array:

1. **Maximum Subarray Sum (Kadane's Algorithm)**: This helps us find the maximum sum of any contiguous subarray. We'll use this to find
the maximum positive sum a subarray can achieve.

2. **Minimum Subarray Sum**: Similarly, this helps us find the minimum sum of any contiguous subarray. By negating the array elements
and applying Kadane's again, we can find the minimum sum, which helps in calculating the maximum absolute sum when considering negative
subarray sums.

The maximum absolute sum of a subarray can be either the maximum subarray sum or the maximum negative of the minimum subarray sum.
Additionally, any subarray that happens to sum to zero will also contribute to the maximum absolute sum as zero's absolute value is
zero, which might be the maximum if other max sums are negative.

Here's the implementation of the above logic:

```python
from typing import List

class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
        max_sum = 0
        min_sum = 0
        current_max = 0
        current_min = 0

        for num in nums:
            current_max += num
            current_min += num

            # Update global max and min sums
            max_sum = max(max_sum, current_max)
            min_sum = min(min_sum, current_min)

            # Reset current max and min if they drop below zero
            if current_max < 0:
                current_max = 0
            if current_min > 0:
                current_min = 0

        # The result is the maximum of the maximum positive sum
        # and the absolute value of the minimum negative sum
        return max(max_sum, abs(min_sum))

# Example usage:
# solution = Solution()
# print(solution.maxAbsoluteSum([1, -3, 2, 3, -4]))  # Output: 5
# print(solution.maxAbsoluteSum([2, -5, 1, -4, 3, -2]))  # Output: 8
```

### Explanation:

- **current_max** and **current_min** are used to track the sum of the current subarray.
- As we iterate through the array, we update these values by adding the current number.
- We update the global maximum and minimum sums (`max_sum` and `min_sum`) whenever the current values exceed or drop below them,
respectively.
- If at any point `current_max` or `current_min` becomes negative or positive, we reset them to zero because starting a new subarray
from the next element might yield a higher sum.
- Finally, the result is the maximum of `max_sum` and the absolute value of `min_sum`, ensuring we account for both positive and
negative subarray contributions to the maximum absolute sum.
````

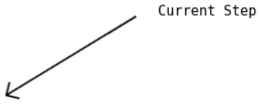

Figure 5: Attention weights of AC-RS on training data.

B Qwen2.5-Coder-7B-Instruct Prediction Example

````
```python
class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
        max_sum = 0
        current_sum = 0
        for num in nums:
            current_sum = max(0, current_sum + num)
            max_sum = max(max_sum, current_sum)
        current_sum = 0
        for num in nums:
            current_sum = max(0, -current_sum - num)
            max_sum = max(max_sum, current_sum)
        return max_sum
```
````

Figure 6: Generation result of Qwen2.5-Coder-7B-Instruct on the example illustrated in Figure 4.

C AC-RS Method Pseudocode

Algorithm 1 AC-RS Algorithm

Require: LeetCode queries $\mathbf{Q_L}$, Pre-trained instruct model $\mathbf{\Theta}$
Ensure: Prediction model $\mathbf{\Theta_{AC\text{-}RS}}$

// Training LC-Base
$\mathbf{R_{Inst}} \leftarrow \textsc{GenerateResponses}(\mathbf{\Theta}, \mathbf{Q_L})$
$\mathbf{\tilde{R}_{Inst}} \leftarrow \textsc{FilterByExecution}(\mathbf{R_{Inst}})$
$\mathbf{\Theta_{LC\text{-}Base}} \leftarrow \textsc{FineTune}(\mathbf{\Theta}, \mathbf{Q_L}, \mathbf{\tilde{R}_{Inst}})$

// AC (Attention Cracking) Stage
$\mathbf{Q_F} \leftarrow \textsc{AppendText}(\mathbf{Q_L}$, "Please give all the examples in the answer.")
$\mathbf{R_{LC\text{-}Base}} \leftarrow \textsc{GenerateResponses}$ $(\mathbf{\Theta_{LC\text{-}Base}}, \mathbf{Q_F})$
$\mathbf{\tilde{R}_{LC\text{-}Base}} \leftarrow \textsc{FilterByExampleNum}$ $\&\textsc{Execution}(\mathbf{R_{LC\text{-}Base}}, \mathbf{Q_F})$
$\mathbf{Q_{AC}} \leftarrow$ Remove target outputs from examples in $\mathbf{Q_L}$ (i.e., "Output: 5" and "Output: 8" in Figure 1)
$\mathbf{\Theta_{AC}} \leftarrow \textsc{FineTune}(\mathbf{\Theta}, \mathbf{Q_{AC}}, \mathbf{\tilde{R}_{LC\text{-}Base}})$

// RS (Rejection Sampling) Stage
$\mathbf{R_{AC}} \leftarrow \textsc{GenerateResponses}(\mathbf{\Theta_{AC}}, \mathbf{Q_{AC}})$
$\mathbf{\tilde{R}_{AC}} \leftarrow \textsc{FilterByExampleNum}$ $\&\textsc{Execution}(\mathbf{R_{AC}}, \mathbf{Q_{AC}})$
$\mathbf{\Theta_{AC\text{-}RS}} \leftarrow \textsc{FineTune}(\mathbf{\Theta}, \mathbf{Q_{AC}}, \mathbf{\tilde{R}_{AC}})$
return $\mathbf{\Theta_{AC\text{-}RS}}$

function $\textsc{FilterByExampleNum}$ $\&\textsc{Execution}(\mathbf{R}, \mathbf{Q})$
return Filtered responses with same examples as $\mathbf{Q}$, prioritizing test-passing ones
end function

D Experiment Settings

LiveCodeBench Version LiveCodeBench serves as a continuously updated benchmark where each category of tasks contains multiple versions. To ensure clear experimental variables and reproducibility, we specify the exact versions and sample information used. For Code Generation tasks, we employ the latest version v3 available at experiment initiation, containing 612 test samples from May 1, 2023 to September 1, 2024. Self Repair tasks rely on error outputs from Code Generation tasks, but their test sets differ in this study. This occurs because the Self Repair test set only updated to version v1 during our experiments, containing 400 test samples from May 1, 2023 to April 1, 2024. Test Output Prediction uses version v1 with 442 samples from May 1, 2023 to April 1, 2024. Code Execution employs version v2 containing 479 test samples from May 1, 2023 to December 1, 2023. We note that LiveCodeBench leaderboard data changes cause slight sample count mismatches within identical time ranges. For example, the Self Repair tasks actually contain 439 samples (May 1, 2023 to April 1, 2024) on the leaderboard, exceeding our reported 400 samples. This difference stems from subsequent updates adding 39 new samples from March 1, 2024 to April 1, 2024.

Hyperparamter Settings We maintain consistent parameter configurations for both model training and inference. The training process uses full-parameter bf16 precision mode with sequence length 4096 and batch size 32. To optimize memory usage, we enable Deepspeed framework's O2 optimization level. This configuration allows complete training on a server with 4 NVIDIA A800 80G GPUs. Models undergo 5 full training epochs with initial learning rate $1 * 10^{-5}$ using a cosine learning rate scheduler. During inference, we follow LiveCodeBench's standard test script configuration: topp=0.95 and temperature=0.2. For cost control, we request single outputs from GPT-4o during data collection. For local models performing Rejection Sampling, we consistently execute 20 output predictions with topp=0.8 and temperature=0.95 to ensure sampled data quality. In addition, we incorporated 30 extra samples with the same format as test data to ensure instruction following.