Small Language Models for specific objectives on resource constrained devices

Rajat Kumar Thakur
202211070@diu.iiitvadodara.ac.in
Indian Institute of Information Technology Vadodara
Gujarat, India

Anirban Dasgupta
anirbandg@iitgn.ac.in
Indian Institute of Technology Gandhinagar
Gujarat, India

Abstract

The exponential growth of IoT devices and edge computing has made efficient AI models that can be run on low-resource hardware more sought after. This research delves into the design of Small Language Models for the generation of Python code, with real-time use cases like running and evaluating machine logs on edge devices. Two lightweight models, TinyLlama and StarCoder, were trained on a combination of varied datasets like SciDocs, Astronomy Stack Text, Wikipedia Field of Science, and The Stack for code generation for user's natural language queries. Pruning, weight sharing, and early exit strategies were employed to achieve maximum performance. Results indicate that pruning had a drastic impact on latency for TinyLlama, making it more viable for real-time applications on the cost of code quality. StarCoder was less accurate in producing correct executable code for scientific data even after optimization, indicating potential avenues for improvement. This research demonstrates the viability of small language models for domain specific use cases on low-resource hardware while highlighting difficulties in achieving the equilibrium between efficiency and functionality, paving the way for future research in lightweight deployment of AI.

Keywords: Small Language Models, Code Generation, Model Optimization, Resource Constrained Devices, Edge Computing

1 Introduction

The fast pace of edge computing combined with the proliferation of Internet of Things (IoT) devices has brought about an immediate need for artificial intelligence (AI) models that are capable of working optimally under the constraints of limited hardware resources. Platforms like IoT sensors, smartphones, and embedded systems tend to be constrained by memory space, computational power, and power supply, making it difficult to deploy large-scale, computationally expensive models. While Large Language Models (LLMs) are extremely capable, they tend to be too resource-intensive for such environments, causing increased latency, excessive energy usage, and unrealistic memory consumption. This has prompted the creation of Small Language Models, which aim to strike a balance between performance and efficiency, thereby facilitating real-time applications on the edge device.

One of the main uses in this framework is machine logs and datasets analysis and inspection. Machine logs play a crucial part in system debugging, monitoring, and preventative maintenance, yet real-time processing in edge devices is not always possible due to limited resources. On the other hand, dataset parsing and interpretation on-site can be critical to operations such as data preprocessing, feature extraction, and report generation. Cloud computing has traditionally been the method for these operations, which causes latency, bandwidth limitations, and concerns over privacy, particularly in environments with intermittent connectivity or where data security levels are high.

To overcome these limitations, this paper targets the creation of SLMs that are capable of data set analysis, writing and running Python code, and presenting results to users directly on resource-limited devices. By activating these features on-device, we seek to provide real-time, offline data processing and code creation independent of cloud resources. In addition to removing latency and consumption of bandwidth, this also provides greater privacy and security by keeping sensitive data in local possession. For this study, we chose two advanced state-of-the-art SLMs: TinyLlama [11], with 1.1 billion parameters, is optimized for efficiency and flexibility and is best suited for a broad variety of tasks on the edge device and, StarCoder [6], which is fine-tuned specifically for code generation, is particularly strong at generating Python code for the processing of dataset and log files. To improve their edge deployability, we used optimization methods such as pruning, weight sharing, and early exit [2]. These methods are intended to minimize model size, latency, and memory usage but preserve the capability to generate functional and executable code.

The organization of this report is as follows: Section 2 provides information on related work in SLMs and optimization methods of edge computing. Section 3 describes our methodology, from dataset choice to model training and optimization methods. Section 4 contains experimental results, comparing the code generation, execution, and resource usage of the baseline and optimized models. Section 5 provides the implications of our results, the trade-offs we faced, and possible future directions. Lastly, Section 6 concludes the report with an overview of key insights and contributions.

SRIP, IIT Gandhinagar
2025.

2 Related Work

The rapid expansion of edge computing and Internet of Things (IoT) devices has spurred significant interest in developing artificial intelligence (AI) models that can operate efficiently on resource-constrained hardware. Large language models (LLMs) have demonstrated remarkable capabilities in tasks such as log analysis and code generation, but their computational intensity often makes them impractical for deployment on devices with limited memory and processing power. Small Language Models (SLMs), designed to balance performance and efficiency, are emerging as a viable solution for enabling real-time applications like log processing and code generation on edge devices. This section reviews recent advancements in LLMs and their applications, focusing on three key works: LogLLM for log-based anomaly detection, data-efficient fine-tuning for code generation, and LLaVaOLMoBitnet1B for multimodal processing. We discuss their contributions and limitations, particularly in the context of our goal to develop SLMs for generating and executing Python code on resource-constrained devices.

2.1 LogLLM: Log-based Anomaly Detection Using Large Language Models

Guan et al. [4] proposed LogLLM, a framework that leverages LLMs for log-based anomaly detection, a critical task for maintaining software system reliability. LogLLM employs BERT to extract semantic vectors from log messages and Llama, a transformer decoder-based model, to classify log sequences as normal or anomalous. A novel projector aligns the vector spaces of BERT and Llama, ensuring a cohesive understanding of log semantics. Unlike traditional methods that rely on complex log parsers to extract templates, LogLLM uses regular expressions for preprocessing, simplifying the pipeline. The framework is trained through a three-stage procedure, enhancing its adaptability to diverse log formats. Experimental results on four public datasets—HDFS, BGL, Liberty, and Thunderbird—demonstrate that LogLLM outperforms state-of-the-art methods, particularly in handling unstable logs with varying formats [4].

Despite its strengths, LogLLM has notable limitations for our use case. The reliance on large models like BERT and Llama results in high computational demands, making it challenging to deploy on resource-constrained devices such as IoT sensors or embedded systems. Additionally, LogLLM is designed specifically for anomaly detection, not for generating executable Python code, which limits its applicability to our objective of developing SLMs for code generation and log processing. The framework's focus on semantic analysis, while powerful, does not address the need for lightweight models capable of real-time code execution on edge devices.

2.2 Data-efficient LLM Fine-tuning for Code Generation

Lv et al. [7] addressed the challenge of fine-tuning LLMs for code generation, aiming to bridge the performance gap between open-source and closed-source models. Their approach introduces a data selection strategy that prioritizes high-quality, complex data samples while ensuring alignment with the original dataset's distribution. They also propose a "dynamic pack" technique for tokenization, which minimizes padding tokens to reduce computational resource consumption. Experimental results show that by using only 40% of the OSS-Instruct dataset, their fine-tuned DeepSeek-Coder-Base-6.7B model achieves an average performance of 66.9%, surpassing the 66.1% performance obtained with the full dataset [7].

While this work significantly improves the efficiency of fine-tuning for code generation, it has limitations for our project. The approach still relies on relatively large models, such as the 6.7-billion-parameter DeepSeek-Coder, which may require substantial computational resources for training and inference, making it less suitable for resource-constrained environments. Furthermore, the focus is on fine-tuning existing LLMs rather than developing SLMs specifically optimized for edge devices. The computational overhead of fine-tuning, even with data-efficient methods, may still pose challenges for deployment on devices with limited memory and processing power.

2.3 LLaVaOLMoBitnet1B: Ternary LLM goes Multimodal!

Sundaram and Iyer [10] introduced LLaVaOLMoBitnet1B, the first ternary multimodal LLM capable of processing both image and text inputs to produce coherent textual responses. By using ternary weights (3-bit weights instead of standard 32-bit or 16-bit weights), the model significantly reduces its size and computational requirements, making it more accessible for devices with limited resources. The model is fully open-sourced, along with training scripts, to encourage further research. The authors highlight a two-step training pipeline, including a pre-training phase for feature alignment and an end-to-end instruction fine-tuning phase, and discuss challenges associated with ternary models, such as potential precision trade-offs [10].

While LLaVaOLMoBitnet1B represents a significant step toward efficient LLMs, its limitations are evident in the context of our project. The model's primary focus is on multimodal tasks, combining image and text processing, rather than on code generation or log analysis. The use of ternary weights, while reducing resource demands, may introduce accuracy trade-offs that could affect the precision required for generating executable Python code. Additionally, its multimodal nature does not align directly with our goal of developing SLMs for code generation and log processing on edge devices, where text-based tasks are the primary focus.

2.4 Additional Context: Log Analysis and Code Generation

Beyond these specific works, the broader field of log analysis and code generation provides additional context. For instance, traditional deep learning methods for log analysis, such as DeepLog and LogAnomaly, rely on LSTM networks to capture sequential patterns but often struggle with semantic understanding [3, 8]. These methods require extensive preprocessing and feature engineering, which can be impractical for real-time applications on edge devices. Similarly, code generation models like Codex and Code Llama have shown impressive performance but are typically too large for resource-constrained environments [1, 9]. Recent efforts to make LLMs more efficient, such as quantization and model distillation, have shown promise but often focus on general language tasks rather than specific applications like log processing or code generation.

2.5 Summary and Research Gap

The reviewed works highlight significant advancements in leveraging LLMs for log analysis and code generation. LogLLM demonstrates the power of LLMs in capturing semantic information for anomaly detection, but its computational intensity limits its use on edge devices. The data-efficient fine-tuning approach by Lv et al. improves the efficiency of code generation but still relies on large models, which may not be suitable for resource-constrained environments. LLaVaOLMoBitnet1B offers a promising approach to reducing model size through ternary weights, but its multimodal focus is not directly applicable to our task.

Our research addresses these gaps by developing SLMs specifically optimized for generating and executing Python code on resource-constrained devices. By focusing on lightweight models like TinyLlama [11] and StarCoder[6] and applying optimization techniques such as pruning, weight sharing, and early exit, we aim to create a solution that is both efficient and effective for real-time applications, such as parsing datasets, generating code, and displaying results to users on edge devices. This work builds on the strengths of prior research while addressing their limitations in terms of computational efficiency and task specificity.

3 Methodology

In this work, we focus on developing Small Language Models (SLMs) for specific objectives, such as reading machine logs, that can be trained and perform inference on resource-constrained devices. Our approach combines dataset curation, parameter-efficient fine-tuning (PEFT), and model optimization techniques to enable scientific code generation on such devices. The methodology consists of three core phases: dataset curation and preparation, parameter-efficient fine-tuning, and model optimization techniques, followed by an evaluation framework.

3.1 Dataset Curation and Preparation

We integrate multi-domain scientific sources into a unified training corpus to ensure a diverse and comprehensive dataset for training our SLMs. The dataset is defined as:

$$\mathcal{D} = \mathcal{D}_{\text{scidocs}} \cup \mathcal{D}_{\text{astronomy}} \cup \mathcal{D}_{\text{science}} \cup \mathcal{D}_{\text{code}} \tag{1}$$

Where:

- $\mathcal{D}_{\text{scidocs}}$: 25k paper abstracts from AllenAI's SciDocs
- $\mathcal{D}_{\text{astronomy}}$: 15k astronomy problems (DPO-text)
- $\mathcal{D}_{\text{science}}$: 15k Wikipedia science entries
- $\mathcal{D}_{\text{code}}$: 20k Python files from The Stack (filtered for scientific libraries)

To prepare the data for training, we transform non-code entries using prompt templates to convert them into a format suitable for code generation. Specifically:

$$\text{prompt} = \begin{cases} f_{\text{title}}(\text{title}) + f_{\text{abstract}}(\text{abstract}) & \text{for papers} \\ f_{\text{problem}}(\text{problem\_text}) & \text{for astronomy} \\ f_{\text{task}}(\text{text}) & \text{for science topics} \end{cases} \tag{2}$$

Here, f_{title}, f_{abstract}, f_{problem}, and f_{task} are functions that extract and format the relevant parts of the input data into a standardized prompt. This ensures consistency across different data types. The overall dataset preparation process is illustrated in Figure 1, which shows the integration of multi-source data and the application of prompt engineering.

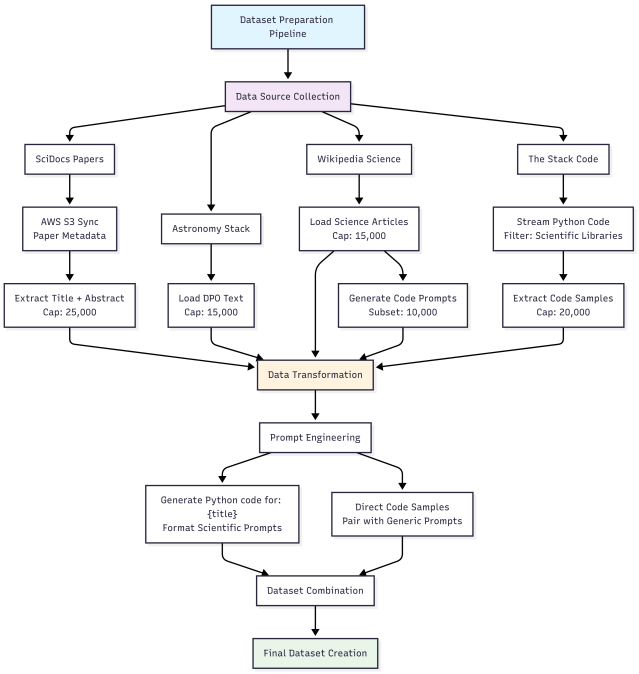


Figure 1. Dataset preparation workflow showing multi-source integration and prompt engineering

3.2 Parameter-Efficient Fine-Tuning

We employ the QLoRA fine-tuning method, which integrates 4-bit quantization with Low-Rank Adaptation (LoRA)[5] to efficiently fine-tune pre-trained small language models. The procedure is detailed in Algorithm 1. It begins by initializing the base model parameters θ_{base}, followed by applying 4-bit quantization to produce θ_{4bit}. Subsequently, LoRA adapters, represented as $\Delta W = B \cdot A$, are injected into the model. The adapters are then optimized to minimize the loss function $\mathcal{L}(\theta_{\text{4bit}} + \Delta W; \mathcal{D})$ with respect to a given dataset $\mathcal{D}$.

Algorithm 1 QLoRA Fine-Tuning

1: Initialize θ_{base} (pre-trained model)
2: Apply 4-bit quantization: $\theta_{\text{4bit}} \leftarrow Q_{\text{NF4}}(\theta_{\text{base}})$
3: Inject LoRA adapters: $\Delta W = B \cdot A$, where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$
4: Optimize: $\min_{\Delta W} \mathcal{L}(\theta_{\text{4bit}} + \Delta W; \mathcal{D})$

3.3 Model Optimization Techniques

To make the models suitable for resource-constrained devices, we apply several optimization techniques that reduce the model size and inference time without significantly degrading performance. Specifically, we implement three compression strategies:

3.3.1 Structured Pruning: We remove the least important weights based on their L1 or L2 norms. The pruning loss is defined as:

$$\mathcal{L}_{\text{prune}} = \sum_{l=1}^{L} \|\mathbf{W}^{(l)} \odot \mathbf{M}^{(l)}\|_F^2 \tag{3}$$

where $\mathbf{M}^{(l)}$ is a binary mask that indicates which weights are kept, determined by thresholds based on weight magnitudes.

3.3.2 Weight Sharing: We share parameters between the embedding layer and the language model head:

$$\mathbf{W}_{\text{embed}} = \mathbf{W}_{\text{lm\_head}}^{\top} \tag{4}$$

This reduces the total number of parameters and can improve generalization.

3.3.3 Early Exit: We implement an early exit mechanism that terminates inference when the model's confidence exceeds a certain threshold:

$$\text{Exit if } \max(\mathbf{p}^{(l)}) > \tau \quad \forall l \in \{l_{\text{exit}_1}, \ldots, l_{\text{exit}_k}\} \tag{5}$$

This allows for faster inference by avoiding unnecessary computations in deeper layers. Figure 2 provides a visual representation of optimization techniques [2].

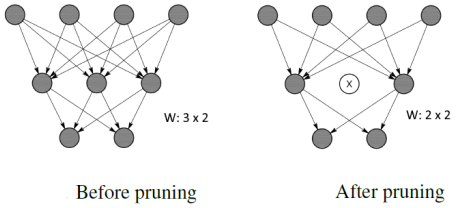


Figure 2. Pruning applied to a neural network

3.4 Evaluation Framework

We assess the performance of our models using a dual set of metrics that capture both inference efficiency and code quality.

3.4.1 Inference Efficiency We measure:

$$\text{Latency} = \frac{1}{N} \sum_{i=1}^{N} t_{\text{gen}}^{(i)}$$

$$\text{Memory} = \max(\text{GPU\_mem})$$

$$\text{Throughput} = N_{\text{success}} / t_{\text{total}}$$

where $t_{\text{gen}}^{(i)}$ is the generation time for the i-th sample, GPU_mem is the peak GPU memory usage, and N_{success} is the number of successfully generated codes within the total time t_{total}.

3.4.2 Code Quality For code quality, we define:

$$\text{Validity} = \mathbb{I}[\text{Code compiles}]$$

$$\text{Accuracy} = \mathbb{I}[\text{Execution produces correct output}]$$

$$\text{Q-Score} = \frac{1}{5} \sum_{k=1}^{5} \mathbb{I}[\text{Criterion}_k \text{ satisfied}]$$

The Q-Score is based on five criteria: appropriate library usage, correct result printing, inclusion of visualizations, data generation capabilities, and syntactic correctness. The entire evaluation process, including code generation, repair, and execution validation, is illustrated in Figure 3.

3.5 Datasets

To support our methodology, we utilize the following datasets:

3.5.1 Datasets

- Wikipedia Field of Science: https://huggingface.co/datasets/millawell/wikipedia_field_of_science
- Astronomy Stack DPO Text: https://huggingface.co/datasets/David-Xu/astronomy-stack-dpo-text
- SciDocs: https://github.com/allenai/scidocs
- The Stack: https://huggingface.co/datasets/bigcode/the-stack

These components form the foundation of our approach, ensuring that the SLMs are both effective and efficient for resource-constrained devices.

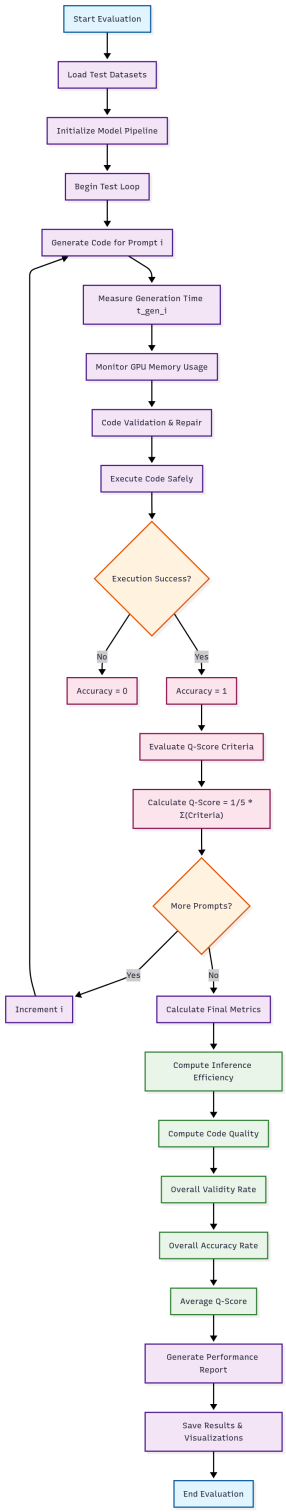


Figure 3. Evaluation pipeline showing code generation, repair, and execution validation

4 Results

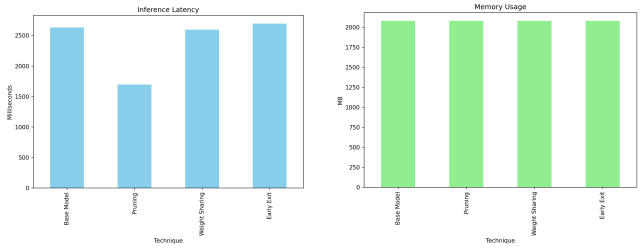
In this section, we present the performance evaluation of various optimization techniques applied to two small language models: StarCoder and TinyLLaMA. The techniques evaluated include the Base Model, Pruning, Weight Sharing, and Early Exit (for StarCoder), and Base Model and Pruning (for TinyLLaMA). We assess the models based on metrics such as inference latency, throughput, memory usage, inference success rate, syntax errors, valid code rate, execution success rate, and code quality score.

4.1 Results for StarCoder

Table 1 presents the performance metrics for the StarCoder model across different optimization techniques. Figure 4a visualizes the inference latency, while Figure 4b shows the memory usage.

Technique	Latency (ms)	Throughput (samples/s)	Memory (MB)	Inference Success (%)	Syntax Errors
Base Model	2664.09	0.342	2080.48	100	5
Pruning	2737.22	0.333	2080.61	100	0
Weight Sharing	2711.61	0.339	2080.61	100	1
Early Exit	2571.31	0.356	2081.48	100	3

Table 1. Performance Metrics for StarCoder Optimization Techniques



(a) Inference Latency for StarCoder Optimization Techniques

(b) Memory Usage for StarCoder Optimization Techniques

Figure 4. Comparison of Optimization Techniques applied on StarCoder

From Table 1, we observe that the Early Exit technique achieves the lowest inference latency (2571.31 ms) and the highest throughput (0.356 samples/s), while maintaining a high inference success rate of 100%. However, it results in 3 syntax errors, which is higher than Pruning (0 errors) and Weight Sharing (1 error). The Base Model has the highest number of syntax errors (5). Memory usage remains relatively consistent across all techniques, with a slight increase for Early Exit (2081.48 MB).

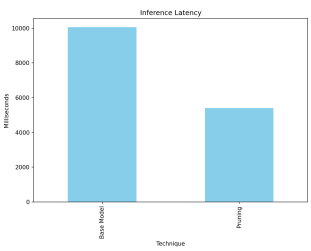
4.2 Results for TinyLLaMA

Table 2 presents the performance metrics for the TinyLLaMA model, comparing the Base Model and Pruning techniques. Figures 5a, 5b, 6a, and 6b visualize the inference latency, valid code rate, execution success rate, and code quality score, respectively.

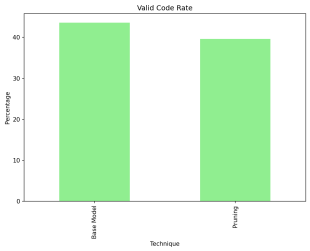
For TinyLLaMA, the Pruning technique significantly reduces inference latency from 10057.68 ms to 5402.01 ms, nearly halving the time required for inference. However, this comes at the cost of reduced code quality, with the valid code

Table 2. Performance Metrics for TinyLLaMA Optimization Techniques

Technique	Latency (ms)	Throughput (samples/s)	Memory (MB)	Inference Success (%)	Valid Code (%)	Execution Success (%)	Quality Score
Base Model	10057.68	0.10	4208.8	100.0	43.6	56.9	65.6
Pruning	5402.01	0.19	4209.3	100.0	39.6	47.5	62.0

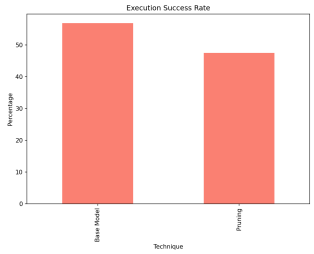


(a) Inference Latency for TinyLLaMA Optimization Techniques

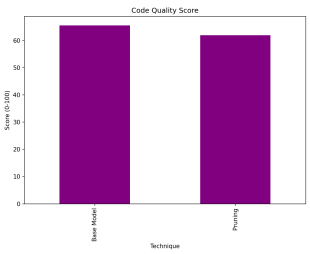


(b) Valid Code Rate for TinyLLaMA Optimization Techniques

Figure 5. Performance Metrics for TinyLLaMA Optimization Techniques



(a) Execution Success Rate for TinyLLaMA Optimization Techniques



(b) Code Quality Score for TinyLLaMA Optimization Techniques

Figure 6. Execution and Code Quality Metrics for TinyLLaMA Optimization Techniques

rate decreasing from 43.6% to 39.6%, execution success rate from 56.9% to 47.5%, and the quality score from 65.6 to 62.0. Memory usage remains almost unchanged, with a negligible increase from 4208.8 MB to 4209.3 MB.

4.3 Comparative Analysis

Comparing the two models, StarCoder generally exhibits lower inference latency and higher throughput than TinyLLaMA across all techniques. For instance, the Base Model for StarCoder has a latency of 2664.09 ms, compared to 10057.68 ms for TinyLLaMA. However, TinyLLaMA's Pruning technique still results in higher latency (5402.01 ms) than any technique applied to StarCoder. This suggests that StarCoder may be more efficient in terms of speed, while TinyLLaMA might require further optimization to achieve comparable performance.

In terms of code quality, TinyLLaMA's Base Model achieves a higher quality score (65.6) compared to the metrics available for StarCoder, which do not include direct quality scores but rather syntax errors. The Pruning technique for TinyLLaMA reduces the quality score to 62.0, indicating a trade-off between speed and code quality.

Overall, the results demonstrate that optimization techniques such as Pruning and Early Exit can significantly improve inference speed, but may impact other performance metrics such as code quality and syntax errors. The choice of technique should be guided by the specific requirements of the application, balancing speed, efficiency, and output quality.

5 Discussion

6 Conclusion

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Oliveira Pinto, Jared Kaplan, ..., and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374

[2] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv preprint* abs/1710.09282 (2017). https://arxiv.org/abs/1710.09282

[3] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1285–1298. doi:10.1145/3133956.3134015

[4] Wei Guan, Jian Cao, Shiyou Qian, Jianqi Gao, and Chun Ouyang. 2024. LogLLM: Log-based Anomaly Detection Using Large Language Models. *CoRR* abs/2411.08561 (2024). doi:10.48550/ARXIV.2411.08561

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. https://openreview.net/forum?id=nZeVKeeFYf9 Also available as arXiv:2106.09685.

[6] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and Nicolas Chapados. 2024. StarCoder2 and The Stackv2: The Next Generation. *CoRR* abs/2402.19173 (2024). doi:10.48550/ARXIV.2402.19173

[7] Weijie Lv, Xuan Xia, and Sheng-Jun Huang. 2025. Data-efficient LLM Fine-tuning for Code Generation. *CoRR* abs/2504.12687 (2025). doi:10.48550/ARXIV.2504.12687

[8] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*. 4739–4745. doi:10.24963/ijcai.2019/658

[9] Benoît Rozière, Jonas Gehring, Florian Gloeckle, ..., and Thomas Scialom. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint* abs/2308.12950 (2023). https://arxiv.org/abs/2308.12950

[10] Jainaveen Sundaram and Ravi Iyer. 2024. LLaVaOLMoBitnet1B: Ternary LLM goes Multimodal! *CoRR* abs/2408.13402 (2024). doi:10.48550/ARXIV.2408.13402

[11] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. *CoRR* abs/2401.02385 (2024). doi:10.48550/ARXIV.2401.02385