CodeAnubad at BLP-2025 Task 2: Efficient Bangla-to-Python Code Generation via Iterative LoRA Fine-Tuning of Gemma-2

Soumyajit Roy
roysoumyajit@icloud.com

Abstract

This paper presents our submission for Task 2 of the Bangla Language Processing (BLP) Workshop, which focuses on generating Python code from Bangla programming prompts in a low-resource setting. We address this challenge by fine-tuning the gemma-2-9b instruction-tuned model using parameter-efficient fine-tuning (PEFT) with QLoRA. We propose an iterative self-improvement strategy that augments the extremely limited training data (74 examples) by reusing verified correct predictions from the development set, alongside LoRA rank experiments (8, 16, 32), observing a clear correlation between rank and accuracy, with rank 32 delivering the best results. Compared to translation-based and retrieval-augmented baselines, our approach achieves significantly higher accuracy, with a pass rate of 47% on the development set and 37% on the hidden test set. These results highlight the effectiveness of combining iterative data augmentation with rank optimisation for specialised, low-resource code generation tasks.

1 Introduction

The ability of large language models (LLMs) to generate code has significantly advanced software development and computer science education. However, most progress has been concentrated on high-resource languages like English. The BLP Task 2 (Raihan et al., 2025c) presents a valuable challenge: extending these capabilities to Bangla, a language spoken by more than 230 million people yet underrepresented in mainstream NLP research. The task requires a system to take a programming prompt written in Bangla and generate a Python script that passes a set of hidden unit tests.

This task is particularly challenging due to two factors: the need for models to comprehend nuanced, procedural instructions in a non-English language, and the extremely limited size of the initial training dataset (74 examples). Our work aims to tackle this by adapting a powerful, pre-trained LLM and introducing a novel training strategy to overcome data scarcity.

Our primary contributions are as follows:

- We introduce an iterative self-improvement pipeline that uses the model's own correct generations on the development set to augment the training data, progressively boosting performance without external data.
- We provide a comparative analysis of different base models (Gemma-2, Code Llama, StarCoder) and LoRA configurations, identifying Gemma-2-9b-it model with a rank of 32 as the most effective combination.
- Our approach achieves a final pass rate of 37% on the official test set and 47% on the development set, showcasing a viable method for low-resource code generation.

2 Related Work

2.1 Large Language Models for Code Generation

Large Language Models (LLMs) have transformed automated code generation. Codex (Chen et al., 2021), powering GitHub Copilot, showcased the potential of training on large-scale code corpora, followed by open-source models like StarCoder (Li et al., 2023) and Code Llama (Touvron et al., 2023). While highly effective in languages such as Python and JavaScript, their performance on low-resource natural languages like Bangla remains underexplored.

Recent efforts target multilingual code generation. Benchmarks such as HumanEval-XL (Peng et al., 2024) and CRUXEval-X (Xu et al., 2025) evaluate cross-lingual generalization and reasoning, while works like Li et al. (2025) and Liu et al. (2025) study zero-shot transfer and retrieval-augmented generation. However, most approaches

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 556–560

rely on translation or parallel resources, leaving open challenges for direct adaptation in severely low-resource settings. Our work addresses this through iterative fine-tuning on Bangla-specific prompts.

2.2 Parameter-Efficient Fine-Tuning (PEFT)

Fully fine-tuning multi-billion parameter models is computationally prohibitive for most researchers. PEFT methods have emerged as a solution. Low-Rank Adaptation, or LoRA, was introduced by Hu et al. (2021), who proposed freezing the pre-trained model weights and injecting small, trainable low-rank matrices into the transformer layers. This reduces the number of trainable parameters by orders of magnitude. Dettmers et al. (2023) later proposed QLoRA, which applies LoRA on top of a quantized base model, such as 4-bit, making it possible to fine-tune massive models on a single GPU.

2.3 Low-Resource NLP

NLP for low-resource languages like Bangla faces persistent data scarcity, which is even more acute for specialised tasks such as code generation. We address this with an iterative self-improvement approach related to pseudo-labeling, where a model's own predictions are reused as training data. Traditional pseudo-labeling assigns high-confidence predictions to unlabeled data and retrains with thresholds to limit errors (Lee, 2013), while self-training incorporates pseudo-labels without verification (Yarowsky, 1995). In contrast, our method augments training data only with verified predictions from the development set—confirmed via pass rates against hidden unit tests—thereby reducing error propagation in procedural tasks like Bangla-to-Python generation. Unlike general self-training, which risks amplifying noisy labels, our approach establishes a targeted feedback loop that curates high-quality, in-domain pairs without external data, emphasizing execution-based validation over probabilistic confidence for syntactic and semantic accuracy.

3 Dataset and Data Augmentation Strategy

The dataset provided for the task consists of a small training split with 74 samples and a development split with 400 samples (Raihan et al., 2025a), whereas the test dataset consisted of 500 samples (Raihan et al., 2025b). Each sample is a JSON object that contains a Bangla instruction (instruction), a corresponding Python solution (response), and other metadata.

Given the extremely small size of the initial training set, we designed an **iterative self-improvement strategy** to augment our data using the model's own predictions on the development set. This process was executed in the following stages:

- **Initial Training:** The base model was first fine-tuned on the original 74 training samples. This initial model achieved a pass rate of 38% in the 400-sample development set.
- **Data Augmentation:** We identified the 152 correct predictions (38% of 400) from the development set. These high-quality, model-verified instruction-response pairs were then added to the original training data, creating an augmented set of 226 samples.
- **Iterative Re-training:** The model was re-trained from its original checkpoint using this new, larger dataset. This step, combined with training at a higher LoRA rank, improved the development set pass rate from 38% to 42%. A final re-training with an optimised LoRA rank further boosted this score to 47%.

This iterative data curation strategy was central to our ability to improve performance despite the initial data scarcity.

4 Methodology

Our approach is centered around the supervised fine-tuning of a pre-trained LLM using an iterative data augmentation strategy. All training was conducted on a single NVIDIA A6000 GPU, with each run completing in under five minutes due to the small dataset and an early stopping callback monitoring validation loss.

4.1 Base Model Selection

We conducted preliminary experiments with several open-source models to select the best foundation for our task. Using the development set for evaluation, we found that StarCoder (Li et al., 2023) achieved a pass rate of only 13%, while CodeLlama-3.1-8B-it (Touvron et al., 2023) reached 32%. The Gemma-2-9b-it model demonstrated a superior baseline performance, reaching an accuracy of 38%, justifying its selection.

Model Configuration	Dev Set Pass Rate
Translation-based (mBART + Gemma-2)	6%
RAG (Retrieval-Augmented)	11%
StarCoder (Base)	13%
CodeLlama-3.1-8B-it	32%
Gemma-2-9B-it (r=8)	38%
Gemma-2-9B-it (r=16)	42%
Gemma-2-9B-it (r=32)	47%

Table 1: Performance comparison between different models.

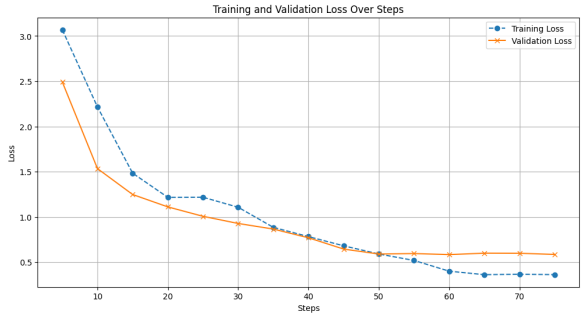

Figure 1: Convergence Graph

We hypothesise that its strong performance stems from a robust instruction-following capability derived from its pre-training, providing a better foundation for interpreting Bangla prompts. We also attempted to use larger 30B+ parameter models, but they overfit rapidly on the small dataset.

In addition to evaluating specialised code generation models, we explored simpler baseline approaches to establish the necessity of our iterative fine-tuning strategy in this low-resource setting. First, we implemented a translation-based method, where Bangla prompts were translated to English using a pre-trained translation model like mBART (Tang et al., 2020) before feeding them into the base gemma-2-9b-it model for code generation. This approach yielded a pass rate of only 6% on the development set, highlighting the challenges of cross-lingual transfer without targeted adaptation. Second, we tested a RAG pipeline using SentenceTransformer ('paraphrase-multilingual-MiniLM-L12-v2') and a FAISS index. Using CodeLlama-7b as the generator, this k=3 few-shot approach achieved only an 11% pass rate, limited by the scarcity of relevant Bangla-aligned retrieval data. These results underscore the inadequacy of off-the-shelf methods for Bangla-specific code generation, justifying our parameter-efficient fine-tuning method with iterative self-improvement, which significantly outperforms these baselines by achieving up to 47% on the development set.

4.2 Fine-Tuning with QLoRA

We applied QLoRA (Dettmers et al., 2023) to enable efficient fine-tuning of the selected base model.

Quantization. The base model was loaded in 4-bit precision using NF4 quantization from bitsandbytes, with computation performed in bfloat16.

LoRA Configuration. Following the LoRA framework (Hu et al., 2021), we experimented with ranks of 8, 16, and 32, observing pass rates of 38%, 42%, and 47% respectively. Our final model used a rank (r) of 32 and a lora_alpha value of 64, applied to all attention and feed-forward layers.

4.3 Training

We used the SFTTrainer for finetuning the model. Key parameters included:

- **Optimizer:** Adam (paged_adamw_8bit)
- **Learning Rate:** 5×10^{-5} with a cosine scheduler
- **Effective Batch Size:** 8 (2 per device $\times$ 4 gradient accumulation steps)
- **Precision:** bfloat16 with flash attention for faster training

Training was configured with early stopping on validation loss (patience=3), which consistently triggered after a short period, indicating rapid convergence on the small dataset.

5 Results

Our final model achieved a pass rate of 47% on the development set and 37% on the final hidden test set. The iterative data augmentation strategy and experimenting with LoRA rank was critical to our performance, improving the development set pass rate from an initial 38% to 47%—a relative improvement of 24%.

The training and validation loss curve for our final training run in Figure 1 shows that the model began to overfit after approximately 60 steps, and our early stopping mechanism correctly identified the optimal checkpoint, preventing performance degradation.

5.1 Qualitative Analysis

The success of our final model over other configurations can be attributed to two synergistic factors:

- **Superior Base Model Foundation:** The gemma-2-9b-it model's superior performance compared to code-specific models like Code Llama and StarCoder suggests that its robust general instruction-following capabilities, honed during pre-training, provided a better foundation for interpreting the nuanced Bangla prompts. It was more adaptable to the cross-lingual, instructional nature of the task.
- **Targeted Data Strategy:** The iterative self-improvement method was highly effective because it directly addressed the core problem: a lack of training data. Instead of relying on synthetic data from another model, we used our own model's evolving capabilities to curate a high-quality, in-domain dataset. This created a positive feedback loop where each training iteration made the model a better data curator for the next, leading to significant performance gains that would have been impossible with the original 74 samples alone.

5.2 LoRA Rank Experimentation

A key part of our experimentation was finding the optimal LoRA rank (r). We observed a clear correlation between the rank and performance on this dataset:

- A rank of 8 yielded a 38% pass rate.
- A rank of 16 combined with the augmented dataset yielded a 42% pass rate.
- A rank of 32 on this same augmented dataset yielded a 47% pass rate.

Our final and best-performing model used the iterative data augmentation strategy combined with a LoRA rank of 32. This submission achieved a **47% pass rate** on the development set and a **37% pass rate** on the final hidden test set.

5.3 Successful Generations

Analysis of our model's 400 predictions on the development set reveals distinct patterns of success and failure.

The model demonstrated a strong capability for generating complex, multi-part algorithms that require more than just pattern matching. For instance, in problem ID 35, the model was tasked with implementing heap sort. It correctly generated the full algorithm, including a logically sound nested *heapify* helper function.

```
def heap_sort(lst):
    def heapify(arr, n, i):
        # ... (correct heapify logic)

    # ... (correct main sort logic)
    heap_sort_util(lst, len(lst))
    return lst
```

This suggests that the fine-tuning process successfully elicited the base model's underlying algorithmic reasoning capabilities.

5.4 Error Analysis

The 213 failed test cases on the development set can be categorized into several groups. The most common failure mode was the omission of necessary dependencies. For example, in problem ID 26 (calculating a triangle's area), the model generated the mathematically correct formula *(n*n*sqrt(3))/4* but failed with a *NameError* because it did not include the required *from math import sqrt* statement. Similar *NameError* failures occurred for other libraries like *itertools* (ID 91) and re (ID 222). Other frequent errors included *TypeError* (e.g., trying to perform tuple arithmetic), *IndexError*, and logical flaws leading to *AssertionError*. These errors indicate that while the model is adept at generating the core logic, it struggles to consistently produce complete, self-contained, and executable scripts.

6 Conclusion

In this paper, we presented an efficient approach to Bangla-to-Python code generation by fine-tuning Gemma-2-9B with QLoRA and an iterative self-improvement strategy that augments scarce training data using the model's verified outputs. LoRA rank experiments revealed higher ranks improve performance, outperforming translation-based and retrieval-augmented baselines. Our findings highlight iterative fine-tuning and hyperparameter optimisation as practical for low-resource code generation tasks.

For future work, we propose two main directions. First, generating a larger, higher-quality augmented dataset could provide a stronger foundation for training. Second, exploring alternative model architectures, such as dedicated encoder-decoder models, may yield better results for this cross-lingual translation-like task.

Limitations

While our approach demonstrates a viable method for this low-resource task, we acknowledge several

key limitations that constrained performance and highlight directions for future work.

Our iterative strategy is fundamentally dependent on the quality and diversity of the small initial dataset (74 samples), which risks amplifying initial data biases. This data scarcity also constrained our model choice to a 9B model, as larger 30B+ parameter models quickly overfit. Furthermore, our study is limited in its baseline comparisons. We did not compare our approach to methods using synthetic data distilled from stronger models, nor did we perform an ablation study to isolate the performance impact of QLoRA quantization against full-precision fine-tuning.

A final limitation, evident from error analysis, is the model's tendency to generate logically correct but syntactically incomplete code, most often causing *NameError* from missing imports (e.g., failing to include *import math*). This highlights a gap between learning core algorithms and the scaffolding (imports, class definitions, etc.) needed for executable scripts.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Lilian Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 38 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Dong-Hyun Lee. 2013. Pseudo-label : The simple and efficient semi-supervised learning method for deep neural networks.

Mingda Li, Abhijit Mishra, and Utkarsh Mujumdar. 2025. Bridging the language gap: Enhancing multilingual prompt-based code generation in llms via zero-shot cross-lingual transfer. *Preprint*, arXiv:2408.09701.

Raymond Li, Loubna Ben Allal, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Ferrandis, Rohan Ghosh, Niklas Muennighoff, Pratik Mishra, Manan Dey, Rachel Bawden, Ankur Dey, Yacine Jernite, Nouamane Tazi, Julien Launay, Margaret Mitchell, Thomas Wolf, Harm de Vries, Alexander M Rush, and Teven Le Scao. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Wei Liu, Sony Trenous, Leonardo F. R. Ribeiro, Bill Byrne, and Felix Hieber. 2025. Xrag: Cross-lingual retrieval-augmented generation. *Preprint*, arXiv:2505.10089.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *Preprint*, arXiv:2402.16694.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025a. mHumanEval - a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11432–11461, Albuquerque, New Mexico. Association for Computational Linguistics.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025b. Tigercoder: A novel suite of llms for code generation in bangla. *arXiv preprint arXiv:2509.09101*.

Nishat Raihan, Mohammad Anas Jawad, Md Mezbaur Rahman, Noshin Ulfat, Pranav Gupta, Mehrab Mustafy Rahman, Shubhra Kanti Karmakar, and Marcos Zampieri. 2025c. Overview of BLP-2025 task 2: Code generation in bangla. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Association for Computational Linguistics (ACL).

Yuqing Tang, Chau Tran, Xian Li, Peng-Jen Chen, Naman Goyal, Vishrav Chaudhary, Jiatao Gu, and Angela Fan. 2020. Multilingual translation with extensible multilingual pretraining and finetuning. *Preprint*, arXiv:2008.00401.

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. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2025. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. *Preprint*, arXiv:2408.13001.

David Yarowsky. 1995. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics*, ACL '95, page 189–196, USA. Association for Computational Linguistics.