Applying Curriculum Learning to Bug Fixing: Finetuning CodeT5

Md Zahidul Haque
Departament of Computer Science
William & Mary
Williamsburg, USA
mhaque@wm.edu

Abstract—**Deep neural networks, particularly Transformer-based models, have shown significant promise in automating software engineering tasks. Among these, bug-fixing remains a critical yet time-consuming activity often performed manually by developers. Leveraging pre-trained models like CodeT5 for automated bug-fixing has demonstrated potential, but performance can be hindered by the complexity and variability of the data. In this work, we introduce a curriculum learning approach to finetune CodeT5 for bug-fixing. Curriculum learning organizes training data into batches based on complexity, enabling the model to learn progressively from simpler to more complex examples. By fine-tuning CodeT5 on incrementally complex datasets, our approach aims to enhance generalization and performance. Experimental results demonstrate notable improvements in BLEU scores, highlighting the effectiveness of curriculum learning in improving code-based deep learning models for bug-fixing tasks.**

I. Introduction

Curriculum learning (CL) has gained attention in deep learning as a strategy inspired by the natural process of human and animal learning. It involves training a model progressively, starting with simpler tasks or data and gradually moving toward more complex ones. In this work, we apply curriculum learning to fine-tune CodeT5, a Transformer-based model, for automated bug-fixing tasks. By introducing training data in increasing levels of complexity, we aim to improve the model's generalization ability and performance.

Bug-fixing is a critical but time-consuming activity in software engineering, directly impacting software quality and delivery timelines. Automating this task reduces the developer's burden while ensuring quicker resolution of errors. Transformer-based models like CodeT5 have shown promise in automating code-related tasks such as bug-fixing, code summarization, and code generation by processing tokenized input and learning contextual features.

Our dataset comprises over 123,804 supervised instances of buggy and fixed Java code pairs, mined from GitHub repositories. Instead of training CodeT5 on the entire dataset at once, as done in traditional fine-tuning, we hypothesize that a progressive learning strategy would lead to better generalization and improved performance. Training data is sorted by a predefined complexity metric and divided into multiple batches, allowing the model to learn from simpler examples before progressing to more complex ones.

Figure 1 illustrates our Progressive Curriculum Fine-tuning Approach, where the pre-trained CodeT5 model is fine-tuned on increasingly challenging batches of data. In contrast, Figure 2 shows the classical fine-tuning approach, where the dataset is used without complexity-based organization. By adopting this strategy, we aim to enhance the model's ability to handle unseen and complex bug-fixing scenarios.

Experimental results demonstrate that curriculum learning improves bug-fixing performance, as reflected in BLEU scores obtained on test datasets. This approach enables the model to learn progressively, mimicking human cognitive processes, and ultimately enhances its ability to fix bugs effectively.

II. Related Work

Automated bug-fixing has become a crucial area of research in software engineering, with the goal of improving code correctness through automated modifications. Traditional approaches, such as the generate-and-validate paradigm, relied on generating candidate fixes using methods like code mutations, templates, or program synthesis, followed by validation against test cases. While these techniques demonstrated utility in specific contexts, they often depended heavily on expert knowledge and static code analysis, which limited their scalability and adaptability to unseen scenarios [1].

The emergence of deep learning (DL) has significantly advanced automated bug-fixing by enabling models to learn patterns from large datasets of buggy and fixed code pairs. Facebook's Getafix system, for example, used a learning-based approach to generate fixes from a repository of human-written patches [2]. Similarly, Tufano et al. employed neural machine translation (NMT) techniques to learn bug-fixing patches from the wild, demonstrating the capability of DL models to generalize to previously unseen code errors [1].

Transformer-based architectures have further revolutionized the field by introducing powerful attention mechanisms that capture long-range dependencies in code. Pre-trained models

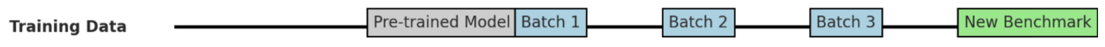

Fig. 1. Curriculum learning fine-tuning: The model is fine-tuned progressively on batches of increasing complexity.

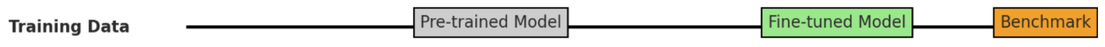

Fig. 2. Classical fine-tuning: The model is fine-tuned on the entire dataset without complexity-based organization.

such as CodeBERT [3] and CodeT5 [4] have shown state-of-the-art performance on tasks like bug-fixing, code completion, and summarization. CodeT5, in particular, is specifically designed for code-related tasks and leverages the sequence-to-sequence framework to translate buggy code into corrected versions. These models have established a robust foundation for automated bug-fixing by effectively modeling complex relationships within code.

Curriculum learning (CL) has emerged as a complementary methodology to improve deep learning performance, inspired by the gradual learning process in humans. Bengio et al. introduced the concept of curriculum learning, demonstrating its effectiveness in various domains by organizing training data in increasing order of complexity [5]. By presenting simpler examples first and gradually progressing to more challenging ones, CL helps models avoid the pitfalls of overwhelming complexity early in training, enabling better generalization and more robust performance.

In the domain of software engineering, Kant et al. applied curriculum learning to program synthesis, showing that structured data progression enhances model performance in generating accurate code snippets [6]. However, the application of curriculum learning to automated bug-fixing tasks remains underexplored. Our work aims to bridge this gap by employing progressive curriculum learning to fine-tune CodeT5 for bug-fixing. Unlike traditional fine-tuning approaches, which train on the entire dataset at once, our method sorts buggy and fixed code pairs based on predefined complexity metrics. These pairs are then divided into batches of increasing difficulty, enabling the model to learn simpler patterns before tackling more complex code structures.

This progressive approach aligns with prior research on curriculum learning in software engineering but focuses explicitly on enhancing bug-fixing performance. Through this study, we aim to establish a new benchmark for automated bug-fixing by integrating the strengths of pre-trained Transformer models with the structured learning benefits of curriculum learning.

III. Methodology

This section outlines the methodology employed to fine-tune a pre-trained model using curriculum learning for automated bug-fixing. By progressively organizing training data based on complexity, we aimed to enhance the model's ability to generalize effectively, thereby improving its performance.

A. Curriculum Learning for Bug-Fixing

Curriculum learning (CL) was implemented to fine-tune a pre-trained transformer model for the bug-fixing task. This involved two key steps: defining a complexity function to quantify the difficulty of training instances and classifying instances into complexity-based batches for progressive fine-tuning.

1) Complexity Function: To measure the difficulty of each code instance, we developed a complexity function based on the Levenshtein distance. This metric calculates the minimum number of token insertions, deletions, and substitutions required to transform a buggy code snippet into its corresponding fixed version. The key steps are as follows:

- Treat buggy and fixed Java methods as tokenized sequences of characters (e.g., separated by spaces or tabs).
- Compute the number of token changes needed to transform the buggy sequence into the fixed sequence.
- Assign a numerical complexity score to each code pair based on the computed distance.

This function provided a consistent measure of difficulty across all supervised instances, enabling systematic organization of training data.

2) Classification of Instances: Once the complexity scores were calculated, the dataset was divided into four cumulative complexity classes: **Low Complexity, Low-Medium Complexity, Medium-High Complexity, and High Complexity**. Instances were classified using the following thresholds:

- Low Complexity: Scores below the first quartile.
- Low-Medium Complexity: Scores in the interquartile range (below the median).
- Medium-High Complexity: Scores in the interquartile range (above the median).
- High Complexity: Scores above the third quartile.

Training batches were cumulative, with each subsequent batch including all instances from previous batches. For example, the second batch combined Low and Low-Medium Complexity instances, while the final batch contained all instances from the dataset. This progressive training approach allowed the model to first focus on simpler patterns before tackling more challenging examples.

B. Fine-Tuning for Bug-Fixing

1) Dataset: For fine-tuning, we used the "code-refinement" section from the CodeXGLUE project [1], which contains buggy and fixed Java method pairs. The dataset was mined from public GitHub repositories, spanning events from March 2011 to October 2017. Commits with messages containing fix-related terms (*fix*, *solve*) and bug-related terms (*bug*, *issue*, *problem*, *error*) were selected to ensure relevance.

Additionally, we introduced a combined dataset (small+medium) to evaluate curriculum learning on a broader range of complexity levels. The data splits are summarized in Table I:

TABLE I
DATASET SIZE FOR BUG-FIXING.

Block	Small	Medium	Small+Medium
Train	46,680	52,364	99,044
Validation	5,835	6,545	12,380
Test	5,835	6,545	12,380

2) Metrics: To evaluate the quality of fine-tuning, we used the BLEU-4 score, a widely used metric for text generation tasks. The BLEU-4 score is calculated as:

$$\text{BLEU-4} = BP \times \exp\left(\frac{1}{4}\sum_{n=1}^{4}\log(\text{precision}_n)\right) \tag{1}$$

where:

- BP is the brevity penalty:
$$BP = \min\left(1, \frac{\text{output length}}{\text{reference length}}\right) \tag{2}$$
- precision_n represents the proportion of n-gram matches between the candidate and the reference.

BLEU-4 evaluates similarity across multiple granular levels, making it particularly suitable for code generation tasks. Scores were computed for the test sets of all datasets (small, medium, and small+medium).

3) Hyperparameter Tuning and Validation: The fine-tuning process relied on the pre-trained Salesforce/codet5-base model, which is specialized for code-related tasks. Hyperparameters were selected through extensive experimentation to balance computational efficiency and performance. Table II presents the final configuration:

To optimize computational efficiency, mixed-precision training (`fp16`) was employed, which reduced memory usage and accelerated computations. The model was trained for a fixed number of epochs without early stopping, as the training process relied on predefined hyperparameters and batch scheduling rather than dynamic termination criteria. These hyperparameters were selected based on experimentation and yielded optimal results for progressive curriculum learning.

[1]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-refinement/data

TABLE II
FINE-TUNING HYPERPARAMETERS FOR BUG-FIXING

Hyperparameter	Value
Pretrained Model	Salesforce/codet5-base
Language	Java
Source Length	256
Target Length	256
Train Batch Size	4
Eval Batch Size	4
Gradient Accumulation Steps	8
Learning Rate	5×10^{-5}
Number of Epochs	3
Evaluation Strategy	Epoch
Save Strategy	Epoch
Weight Decay	0.01
Mixed Precision	Enabled (fp16)

IV. Study Design

This section outlines the hypothesis and methodological approach used to evaluate the impact of curriculum learning on the task of bug-fixing in Java methods. Building on prior research, we aim to demonstrate how progressive curriculum learning influences BLEU scores, a widely used metric for evaluating model performance in code-related tasks.

A. Research Question

This study investigates whether progressively introducing data in batches, sorted by complexity, during fine-tuning improves the performance of deep learning models for bug-fixing tasks. Specifically, we aim to evaluate how curriculum learning impacts model generalization and BLEU scores compared to traditional fine-tuning methods.

B. Data Analysis & Metrics

The data used in this study consists of buggy and fixed Java methods, divided into three subsets: small, medium, and small+medium datasets. Curriculum learning was applied by organizing training data into batches based on increasing complexity, sequentially fine-tuning the pre-trained CodeT5 model on these batches. The process started with the *small* dataset, moved to the *medium* dataset, and finally combined both datasets into the *small+medium* dataset.

BLEU scores were chosen as the primary evaluation metric to assess the similarity between predicted fixes and reference fixes. During training, BLEU scores were computed and logged after processing each set of batches to evaluate the model's performance at different stages of progressive fine-tuning.

C. BLEU Score Trends & Observations

The BLEU scores for the small, medium, and combined datasets during curriculum learning are visualized in Figure 3. Key observations include:

- **Small Dataset:** After training on the 4 batches of the small dataset, the BLEU score reached 69%, reflecting the limited diversity and complexity in this dataset. While sufficient for simpler bug-fixing tasks, the model struggled to generalize to more challenging scenarios.

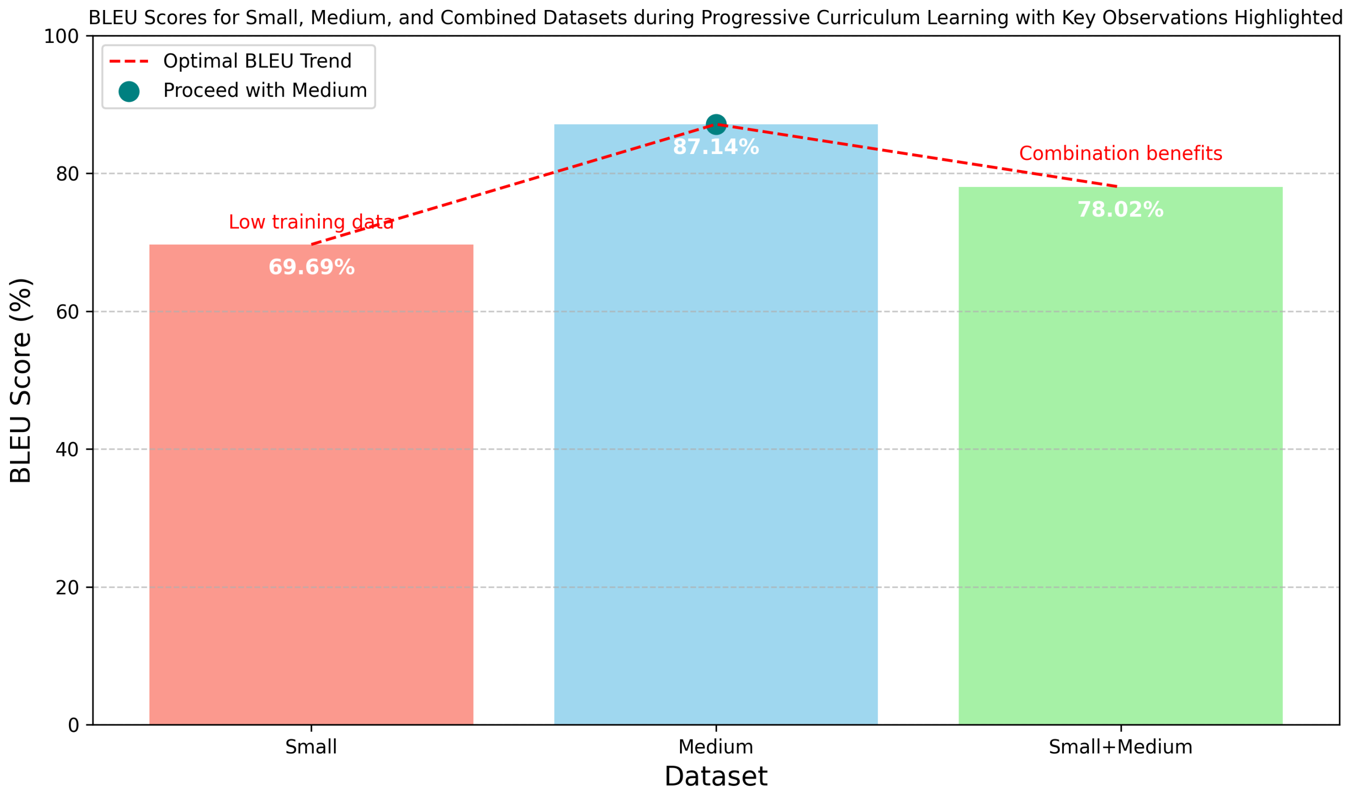

Fig. 3. BLEU Scores during Progressive Curriculum Learning: The model was fine-tuned sequentially on batches from the small, medium, and combined datasets. BLEU scores increased after training on the medium dataset but declined slightly after including the combined dataset, reflecting the trade-offs between data diversity and complexity.

- **Medium Dataset:** With the addition of 4 medium dataset batches, the BLEU score improved to 87%, suggesting that the medium dataset introduced a balance of complexity and size that significantly enhanced the model's generalization ability.
- **Small+Medium Dataset:** Incorporating 4 more batches from the combined dataset led to a BLEU score of 78%. This decrease highlights the trade-off between introducing more diverse and complex data and the challenges of handling increased complexity.

D. *Implications for Model Performance*

The results demonstrate that curriculum learning can enhance model performance by gradually introducing data complexity. The medium dataset showed the best results, indicating that a balanced dataset provides optimal conditions for effective learning. However, the decline in BLEU score when transitioning to the small+medium dataset suggests that over-diversification or excessive complexity may hinder performance.

These findings align with the principles of curriculum learning, emphasizing the importance of a structured and progressive training approach. Further research is needed to refine batching strategies and explore additional complexity metrics to optimize curriculum learning for software engineering tasks.

V. Results Discussion

The results of the bug-fixing task were evaluated using BLEU scores obtained during the fine-tuning and inference processes. As outlined in Sections 3 and 4, we monitored model performance throughout the progressive curriculum learning process by logging BLEU scores at each training stage. These scores served as a quantitative measure of how well the model adapted to progressively complex data batches.

Following fine-tuning, inference was conducted to generate predictions, which were compared against the supervised test instances. The BLEU scores, computed using a modified evaluation script from the CodeXGLUE project, provided insights into the model's ability to generalize and handle varying complexities. These results highlight the potential of curriculum learning in enhancing bug-fixing tasks.

A. *BLEU Score Analysis*

The BLEU scores for the three datasets—small, medium, and small+medium—are visualized in Figure 3. Key observations from the analysis are as follows:

- **Small Dataset:** The BLEU score for the small dataset was relatively low, likely due to the limited diversity and complexity of the data, which hindered the model's ability to generalize effectively.

- **Medium Dataset:** This dataset achieved the highest BLEU score, indicating that its balanced size and complexity provided optimal conditions for effective learning.
- **Small+Medium Dataset:** The combined dataset demonstrated moderate performance, reflecting the trade-offs between increased data variety and the challenges of handling higher complexity.

For the small+medium dataset, we observed a notable difference in performance when comparing traditional fine-tuning and progressive curriculum learning (CL). In the traditional approach, the model was trained on the entire small+medium dataset at once, achieving a BLEU-4 score of 0.7952. In contrast, the progressive curriculum learning approach, where the model was fine-tuned in 8 batches (4 batches of small followed by 4 batches of medium data), achieved a significantly higher BLEU-4 score of 0.8714.

This result highlights the effectiveness of curriculum learning when applied incrementally to datasets like small and medium. By progressively introducing complexity, the model demonstrated improved generalization and learning capabilities. Importantly, this comparison is fair because both methods use the same dataset split but differ in training strategies.

To provide additional context, we also evaluated the performance of the entire CL pipeline, where the model was fine-tuned sequentially across 12 batches (4 batches each for small, medium, and combined small+medium data). This approach achieved a BLEU-4 score of 0.7802, slightly lower than the traditional fine-tuning score of 0.7952. The decline may be attributed to the challenges of managing sequential learning across combined datasets, including overfitting and forgetting.Table III summarizes the BLEU-4 scores across the datasets and methodologies:

TABLE III
COMPARISON OF BLEU-4 SCORES FOR TRADITIONAL FINE-TUNING AND PROGRESSIVE CURRICULUM LEARNING

Approach	BLEU-4 Score
Traditional Fine-Tuning (Small+Medium)	0.7952
Curriculum Learning (Small $\rightarrow$ Medium)	0.8714
Curriculum Learning (Full Pipeline)	0.7802

The results suggest the following:

- Incremental CL (Small $\rightarrow$ Medium) is superior, with a significant BLEU-4 improvement of 8% compared to traditional fine-tuning on the same dataset.
- The full CL pipeline (Small $\rightarrow$ Medium $\rightarrow$ Small+Medium), though marginally less effective than traditional fine-tuning for small+medium, highlights the trade-offs in handling combined datasets with progressive strategies.
- Future research should focus on improving the sequential learning process to retain earlier gains while leveraging the complexity of larger datasets.

Figure 3 illustrates these trends, emphasizing the superior performance of the medium dataset and the gradual learning improvements achieved through progressive curriculum learning. However, the results highlight the need for careful evaluation of curriculum learning's effectiveness, particularly when dealing with larger or more complex datasets like small+medium.

B. Qualitative Evaluation of Predictions

To complement the quantitative BLEU score analysis, we tested the model on a set of buggy Java methods. The predicted fixes and their correctness are analyzed below.

Examples:

- **Buggy Code:**
```
public int divide(int a, int b) {
    if (b == 0) {
        return -1;
    }
    return a / b;
}
```
  Predicted Fixed Code:
```
public int divide(int a, int b) {
    return a / b;
}
```
  Verdict: *Incorrect* (Missing division-by-zero check).
- **Buggy Code:**
```
public int add(int a, int b) {
    return a - b; // Incorrect logic
}
```
  Predicted Fixed Code:
```
public int add(int a, int b) {
    return a + b;
}
```
  Verdict: *Correct* (Logic corrected to addition).
- **Buggy Code:**
```
public int max(int a, int b) {
    if (a > b) {
        return b; // Incorrect return
    }
    return a;
}
```
  Predicted Fixed Code:
```
public int max(int a, int b) {
    return a > b ? a : b;
}
```
  Verdict: *Correct* (Logic converted to ternary operator).

Summary:

- **Correct Predictions:** The model successfully fixed the logical errors in `add` and `max` functions.
- **Incorrect Prediction:** For `divide`, the model failed to handle the edge case of division by zero.

Implications: These examples illustrate that while the model is adept at fixing standard logical bugs, it struggles with handling safety-critical checks and corner cases. Future training strategies could include more examples with such edge cases to improve robustness.

C. Discussion

The results confirm that curriculum learning can enhance model performance by leveraging a progressive approach to training. By starting with simpler data and gradually increasing complexity, the model was able to generalize effectively and handle challenging bug-fixing scenarios.

The medium dataset's performance suggests that a balance between size and complexity is crucial for optimal training. In contrast, the small dataset lacked sufficient diversity, while the combined dataset introduced complexities that may have impacted learning efficiency. These findings validate the hypothesis that curriculum learning can provide meaningful improvements for code-based tasks like bug-fixing, particularly when datasets are structured to balance diversity and complexity.

VI. Conclusions & Future Work

This study investigated the application of curriculum learning to fine-tune the CodeT5 model for automated bug-fixing. By progressively training on batches of increasing complexity, the model demonstrated improved performance, as evidenced by BLEU scores. The medium dataset yielded the best results, highlighting the importance of balanced dataset composition for effective learning.

While the results are promising, further refinements to the complexity function and batching strategy could enhance the impact of curriculum learning. Future work could explore alternative complexity metrics and extend curriculum learning to other code-related tasks, such as code summarization or clone detection. Additionally, integrating dynamic curriculum learning approaches, where the training order adapts based on real-time performance, may unlock further improvements.

Incorporating additional evaluation metrics beyond BLEU, such as exact match or edit distance, could provide a more comprehensive understanding of the model's performance. These directions offer opportunities to establish curriculum learning as a robust strategy for improving deep learning systems in software engineering tasks.

References

[1] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3340544

[2] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[4] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[5] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *International Conference on Machine Learning*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:873046

[6] N. Kant, "Recent advances in neural program synthesis," *CoRR*, vol. abs/1802.02353, 2018. [Online]. Available: http://arxiv.org/abs/1802.02353