HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION

_____o0o_____

PROJECT REPORT

PYTHON CODE GENERATION

Course : Project 2

Course ID : IT3930E

Supervisor : Prof. Tong Van Van

Members :

Name	Student ID	Email
Phan Dinh Truong	20214937	truong.pd214937@sis.hust.edu.vn
Pham Quang Trung	20214935	trung.pq214935@sis.hust.edu.vn
Dang Kieu Trinh	20214933	trinh.dk214933@sis.hust.edu.vn

Hanoi, July 2024

ACKNOWLEDGEMENT

This project would not have been possible without the invaluable support and guidance of Professor Tong Van Van. His extensive knowledge and expertise in the field of Natural Language Processing and Large Language Models have been instrumental in shaping the direction and success of this project. We are deeply grateful for his mentorship and insightful feedback throughout the development process.

We would also like to express our sincere gratitude to our peers for their collaborative efforts and constructive feedback during the course of this project. Their contributions and shared perspectives have enriched the quality of this work and fostered a positive and productive research environment.

ABSTRACT

This project investigates the potential of CodeT5+, a transformer-based model, for text-to-code generation in Python. We explore the model's ability to generate syntactically and semantically correct code from natural language descriptions, aiming to improve its accuracy and efficiency through fine-tuning and domain-specific knowledge integration. Our experiments involve training CodeT5+ models with different sizes using both full fine-tuning and the Low-Rank Adaptation (LoRA) technique. We evaluate the models using CodeBLEU, BLEU, pass@1, and pass@10 metrics on MBPP and HumanEval benchmarks. We demonstrate that fine-tuning with full fine-tuning on the larger CodeT5p-770m model yields the best performance. Furthermore, LoRA proves to be a viable option for accelerating the training process without significant performance loss. Our findings highlight that learning from human-written natural language feedback is a promising avenue for enhancing the code generation capabilities of CodeT5+.

TABLE OF CONTENTS

1. INTRODUCTION

In the rapidly evolving field of artificial intelligence, code generation stands out as a critical area of research with significant implications for software development. Code generation involves the automatic creation of executable source code based on high-level descriptions or natural language inputs. This capability can dramatically streamline the development process, reduce the potential for human error, and enable even those with limited programming knowledge to contribute effectively to software projects.

This project focuses on leveraging advanced machine learning models, specifically CodeT5+, to enhance the text-to-code generation process.CodeT5+ is a transformer - based model tailored for programming language tasks, building on the success of the T5 (Text-To-Text Transfer Transformer) framework. By fine-tuning CodeT5+, we aim to improve its accuracy and efficiency in generating Python code from natural language descriptions.

The primary objectives of this project are:

1. To explore the capabilities of CodeT5 in generating syntactically and semantically correct Python code.

2. To identify the challenges and limitations inherent in current text-to-code generation models.

3. To develop methods for improving the performance of CodeT5 through fine-tuning and incorporating domain-specific knowledge.

This report details the methodologies employed, the experimental setup, the results obtained, and the insights gained from this investigation. Through this work, we aspire to contribute to the ongoing advancements in code generation technologies and to pave the way for more sophisticated and user-friendly programming tools.

2. RELATED WORK

2.1 Transformer-based models

The field of code generation has seen significant advancements with the advent of deep learning and transformer-based models. Numerous studies have explored the application of these models to improve the accuracy and efficiency of automatic code generation from natural language descriptions.

Transformer-based models have revolutionized natural language processing (NLP) in recent years, and their impact has extended to the domain of programming languages and code generation.

The Transformer architecture, introduced in the seminal paper "Attention Is All You Need", consists of an encoder-decoder structure. The encoder processes the input sequence, while the decoder generates the output sequence. The key innovation of Transformers is the multi-head attention mechanism, which allows the model to attend to different parts of the input sequence simultaneously.

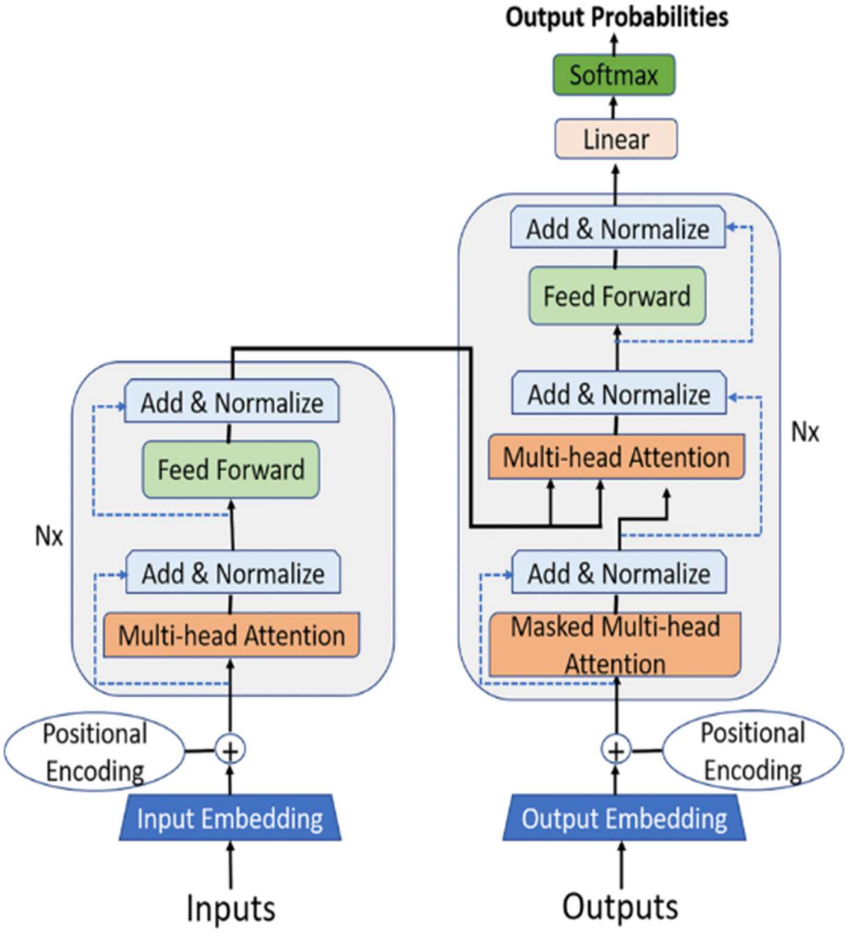

Figure 1. Architecture of Transformer base model

The success of Transformers in NLP tasks has inspired researchers to adapt them for code-related tasks. Transformer-based models have shown remarkable performance on various code-related tasks, such as code completion, code

summarization, and code translation. By learning from large-scale code corpora, these models can generate syntactically correct and semantically meaningful code snippets

The encoder-decoder structure of Transformers aligns naturally with the task of generating code from natural language descriptions or other forms of input. The encoder can process the input sequence, such as a natural language query or a partial code snippet, while the decoder generates the corresponding code output.

Transformers have several advantages over traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) for sequence-to-sequence tasks like code generation. They can handle long-range dependencies more effectively, are more parallelizable, and require less time to train.

2.2. T5 and CodeT5, CodeT5+

Building upon the success of transformer-based models in natural language processing, researchers have developed specialized models for code understanding and generation. Three notable models in this domain are T5, CodeT5, and CodeT5P.

2.2.1. T5: Text-to-Text Transfer Transformer

T5, or Text-to-Text Transfer Transformer, is a versatile model that frames all NLP tasks as a sequence-to-sequence problem.

T5 has been pre-trained on a massive amount of text data, allowing it to capture rich linguistic knowledge. When applied to code generation, T5 can take natural language descriptions as input and generate corresponding code snippets as output.

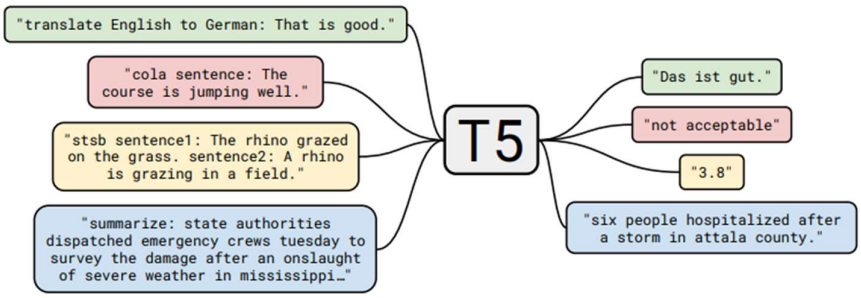

Figure 2. Illustration of T5 for text understanding and text generation tasks

By treating both the input and output as text sequences, T5 can be fine-tuned for various tasks, including code generation.

2.2.2. CodeT5: Code-Specific Pre-training

CodeT5 is a model specifically designed for code understanding and generation tasks. It builds upon the T5 architecture but incorporates code-specific pre-training objectives to better capture the semantics and structure of programming languages.

One key innovation in CodeT5 is the introduction of identifier-aware pre-training tasks. These tasks focus on understanding the meaning and relationships of code identifiers, which are crucial for generating meaningful and coherent code.

CodeT5 has demonstrated strong performance on various code-related tasks, such as code completion, code summarization, and code translation. Its ability to understand and generate code makes it a valuable tool for developers and researchers in the field of AI-assisted programming.

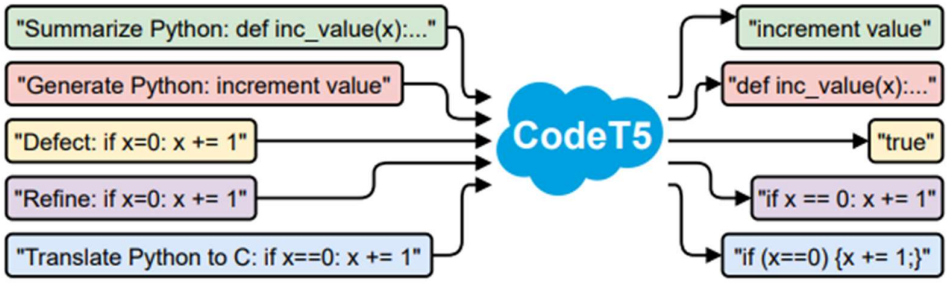

Figure 3. Illustration of CodeT5 for Code understanding and generation tasks

2.2.3. CodeT5P: Enhanced Code Generation

CodeT5P is an extension of CodeT5 that further enhances its code generation capabilities. It incorporates additional pre-training techniques and architectural modifications to improve the quality and coherence of generated code.

One notable feature of CodeT5P is its ability to handle multiple programming languages seamlessly. By leveraging the shared knowledge across different languages, CodeT5P can generate code in various languages with high accuracy and fluency.

CodeT5P has achieved state-of-the-art results on several code generation benchmarks, showcasing its effectiveness in generating syntactically correct and semantically meaningful code. Its performance highlights the potential of transformer-based models in revolutionizing the way developers write and interact with code.

2.3. Fine-Tuning Techniques

Fine-tuning large language models has become an essential technique for adapting pre-trained models to specific downstream tasks. However, the process of fine-tuning can be computationally expensive and memory-intensive, especially when dealing with large models. To address these challenges, several parameter-efficient fine-tuning techniques have been proposed, including Low-Rank Adaptation (LoRA), Quantized LoRA (QLoRA)

2.3.1. Low-Rank Adaptation (LoRA)

LoRA is a fine-tuning technique that reduces the number of trainable parameters by decomposing the weight updates into low-rank matrices. Instead of updating the entire weight matrix, LoRA introduces two smaller matrices, A and B, which are learned during fine-tuning. The original weight matrix remains frozen, and the updates are approximated by the product of A and B.

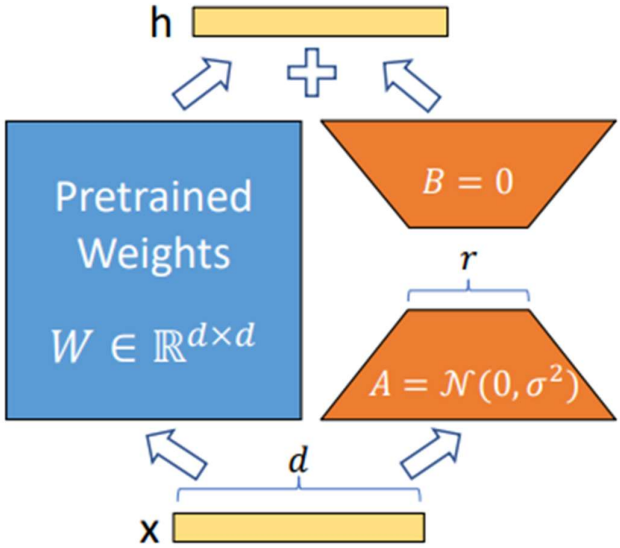

Figure 4. Illustration of LoRA reparametrization

LoRA offers several advantages, including reduced memory footprint, faster training and adaptation, and the ability to scale to larger models. By keeping the original weights frozen, LoRA allows for the creation of multiple lightweight and portable models for various downstream tasks

2.3.2. Quantized LoRA (QLoRA)

QLoRA is an extension of LoRA that combines quantization techniques with low-rank adaptation. It aims to further reduce the memory requirements and computational cost of fine-tuning. QLoRA uses 4-bit quantization for the pre-trained model weights while keeping the LoRA matrices in higher precision.

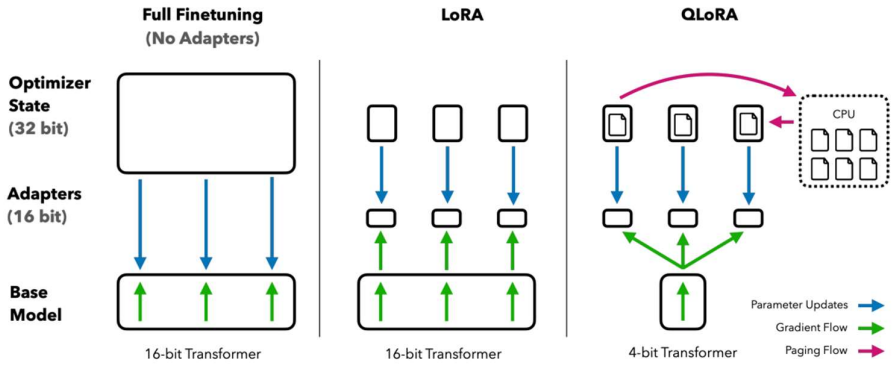

Figure 5. The different between full fine-tuning, LoRA, QLoRA

LoRA achieves performance comparable to full fine-tuning while significantly reducing memory usage and training time. It enables the fine-tuning of large models on resource-constrained devices and facilitates the deployment of adapted models in production environments

2.4. Domain-Specific Fine-Tuning

Domain-specific fine-tuning is a technique that adapts a pre-trained language model to a particular domain or task by further training it on a dataset specific to that domain. This process allows the model to acquire specialized knowledge and improve its performance in the target domain.

In the context of code generation, domain-specific fine-tuning involves training a pre-trained model on a dataset of code snippets and their corresponding natural language descriptions or comments. By exposing the model to domain-specific code examples, it learns the patterns, syntax, and semantics specific to that programming language.

For our project, we focus on fine-tuning for Python code generation. Python is a widely used programming language known for its simplicity, versatility, and extensive ecosystem of libraries and frameworks. By fine-tuning a pre-trained model specifically for Python code generation, we aim to leverage the model's general language understanding capabilities and adapt them to the intricacies of Python programming.

3. METHOD

3.1. Datasets and Task Design

3.1.1. CodeAlpacaPython Dataset

We generate the current Python version from the CodeAlpaca dataset by sifting through Python data samples. These sample codes that cannot be statically analyzed are omitted to ensure that only valid Python code is included. CodeAlpacaPython contains longer and more complex examples than CoNaLa, allowing for a more comprehensive evaluation of PEFT for code generation work. The dataset has 8.48k/912 sample data for train/test trainers.

3.1.2. Conala Dataset

We use an edited version of the CoNaLa dataset. This dataset is collected from StackOverflow and contains manually annotated automatic language sense and encoding pairs. This edited version ensures each sample in the file and test run contains at least one Python function that does not appear in the train set, and the examples from the StackOverflow post are parsed into other files. The dataset has 2,135/201/543 sample data for test/test/test trainers and 594k samples mined.

3.1.3. MBPP Dataset

MBPP (Most Basic Python Programming) is a specialized dataset for Python code generation. This data set is designed to test the capabilities of the models while solving the mechanism that sets the problem to mean. It includes problem builders with increasing levels, allowing the performance of models to be evaluated on many different levels. The MBPP dataset includes many bivariate classified and tested data samples, creating a rich database for training and evaluating biological models.

3.1.4. Task Design

We use the following specific design task and pass into model to generate appropriate response to complete the request. *### Instruction: Create a Python script for this problem: {prompt_text} ### Response:* This design helps the model understand a clear distinction between requests (instructions) and responses (responses), which has proven to be an effective outcome in the past and the output is target Python code.

3.2. Preprocessing

To take advantage of multitask learning, we use different output lengths to train the model. This allows the model to handle code segments of varying complexity and length, improving the model's efficiency and generalization ability.

We do not use the entire data for training but only the training data of CodeAlpacaPython and MBPP. As for CoNaLa, we only randomly take 500 samples

for training. This selection helps ensure the diversity and quality of the training data.

For the CodeAlpacaPython dataset, we only use outputs containing “def” and “return” in the code for training. This reduces the number of samples from 8.48k to 4.67k, helping the model focus on code segments with clear and complete function structures.

Due to differences in tab usage for Python code in each dataset, we replace spaces (“ ” – 4 spaces) with the tab character “\t” to avoid formatting errors. This helps ensure consistency in the training code and makes it easier to process.

Another important step in preprocessing is to remove unnecessary comments in the source code. This helps avoid the model misunderstanding between code and comments, and reduces the maximum length of the target, thereby helping the model learn more effectively.

The total number of samples in the training data set includes 374 samples from MBPP, 4673 samples from CodeAlpaca, and 500 samples from CoNaLa. The specific sample numbers and distribution are illustrated in the figure below.

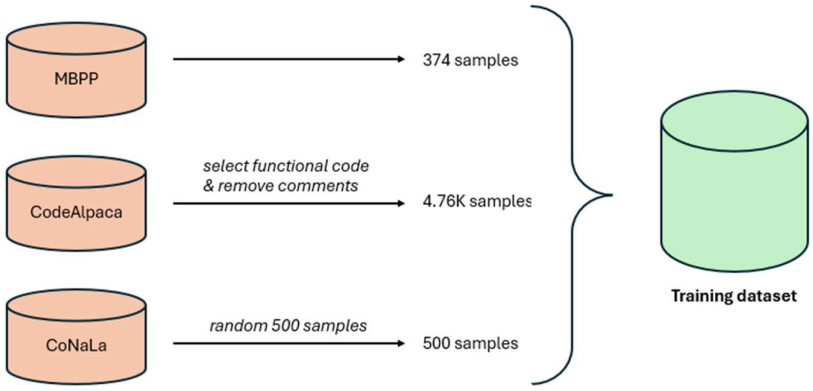

Figure 6. Diagram of preprocessing dataset from multiple sources

3.3. Training

The problem is to train two CodeT5p models with different sizes: CodeT5-220m (small) and CodeT5p-770m (medium). Training is performed with two methods: full fine-tuning and using LoRA (Low-Rank Adaptation).

For the CodeT5-220m model, a full fine-tuning method is applied. This means that all the model parameters are retrained. To optimize the training process, the configuration parameters are set as follows: batch size is 12, gradient accumulation steps is 2, learning rate is 1e-5 and number of training epochs is 15. This method allows Optimize the entire model, exploiting the model's maximum capabilities.

For the CodeT5p-770m model, due to limited computational resources (16GB VRAM of the P100 GPU on Kaggle), we use both full fine-tuning and the LoRA technique. Full fine-tuning is performed with a batch size of 3, gradient accumulation steps of 4, achieving a batch size equivalent to 12. The number of training epochs is reduced to 5.

Additionally, to reduce the number of parameters need training, we fix all parameters of the decoder except cross attention (connecting encoder and decoder). This retains the decoder's powerful text generation capabilities while still allowing for fine-tuning of parameters related to the interaction between encoder and decoder.

With the CodeT5p-770m model, the LoRA technique is additionally applied to reduce the number of parameters that need to be trained. LoRA is set up with the parameters r = 8, lora_alpha = 16 and target_modules = ["q", "k", "v"]. After applying LoRA, the number of parameters that need to be trained is only 3.54M, accounting for 0.47% of the total parameters of the model. This allows for a significant reduction in the computational resources required to train the model while still maintaining high performance.

4. RESULTS

To evaluate the effectiveness of the model, we use popular evaluation metrics in previous code generation studies. One of the metrics used is CodeBLEU, an extended version of BLEU (Bilingual Evaluation Understudy) adapted for code. CodeBLEU compares the generated code string with the reference code string, evaluating the degree of grammatical and structural similarity.

In this study, we will evaluate the model on the MBPP-test data set, a data set that evaluates the effectiveness of the text-to-code conversion model. MBPP-test is designed to evaluate the ability to generate accurate and efficient code from text descriptions.

Besides CodeBLEU, we also use the pass@k index to evaluate the ability to create correct and executable code. It estimates the rate for which ≥1 of k model samples passes all the unit tests. We use the empirical estimate of this quantity from Chen et al. (2021), the formula to calculate pass@k is as follows:

$$pass@k := E_{task}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

for n total programs (where n ≥ k) and c correct programs for the given task.

In this study, we will evaluate the model on the HumanEval data set, a data set that evaluates the effectiveness of the code generation model on simple programming problems. We will set N = 20 (number of code programs created) and k = 1 (select the best code program) and k = 10 (select 10 best code programs) to compare the effectiveness of the model.

Our main results are represent in table:

Table 1. Results of CodeBLEU and BLEU on MBPP and pass@k(%) on HumanEval

	CodeBLEU	BLEU	pass@1	pass@10
Codet5p-220m (Base)			12.0	20.7
CodeT5p-220m SFT	12.14	2.08	5.3	12.0
CodeT5p-770m (Base)	9.69	2.31	15.5	27.2
CodeT5p-770m SFT	**15.35**	4.0	**16.1**	**27.7**
CodeT5p-770m LoRA	13.4	**4.73**	15.5	27.3

The experiments involved fine-tuning Salesforce's CodeT5p for code generation tasks, with a specific focus on comparing the performance of traditional training methods and the use of Low-Rank Adaptation (LoRA) for faster training. The

evaluation metrics included CodeBLEU, BLEU, pass@1, and pass@10, measured on the MBPP and HumanEval benchmarks.

1. CodeT5p-770m SFT: This model achieved the highest CodeBLEU score of 15.35, indicating superior code generation quality compared to other models. It also performed best in pass@1 (16.1) and pass@10 (27.7).

2. CodeT5p-770m LoRA: Utilizing LoRA for faster training resulted in competitive performance with a CodeBLEU score of 13.4 and the highest BLEU score of 4.73. The pass@1 (15.5) and pass@10 (27.3) metrics were also comparable to the best results.

3. CodeT5p-220m SFT: The smaller model fine-tuned with SFT showed moderate performance improvements with a CodeBLEU score of 12.14, but lower pass@1 (5.3) and pass@10 (12.0) scores compared to larger models.

4. Baseline Models: The base versions of CodeT5+ 220M and 770M models demonstrated lower performance across all metrics, underscoring the benefits of fine-tuning and adapting the models.

The experiments highlight the effectiveness of fine-tuning Salesforce's CodeT5p models for code generation tasks. The use of LoRA for faster training shows promise, achieving competitive results while potentially reducing training time and resource requirements. The traditional full fine-tuning approach (SFT) yielded the best overall performance, particularly with the larger 770M model.

5. DISCUSSION

By training CodeT5 models with two different sizes, we see some limitations and areas for improvement:

- Training efficiency: Using conventional training methods, the model's performance does not reach an outstanding level. This shows the need to research and apply more advanced training techniques to optimize model performance.

- Data quality: Training data, especially functional code data sets, are limited in quantity and quality. This results in the model being able to only handle simple and common tasks for users. For complex problems like LeetCode or CodeForce, the model often makes mistakes and gives incorrect results.

- Trainability: Training models larger than 1B in size (such as CodeT5p-2B and CodeT5p-6B) is currently not feasible due to access restrictions. This is because training large models requires huge computational resources and can cause copyright issues.

- Application areas: The model is currently only suitable for common software development tasks. Expanding the application to specialized fields such as computer science, mathematics, or fields requiring deeper expertise will require additional research and appropriate training data.

6. FUTURE WORK

- *Collect more data:* In the future, we will collect more data to improve the accuracy and diversity of the models. Some possible data sources include GitHub, StackOverflow, and other open source repositories. Having more data helps the model learn better and produce higher quality code.

- *Expand domain specific:* We will also expand the scope of model specializations, such as in the areas of Machine Learning and Deep Learning. This will give the model the ability to generate more specialized and relevant code for these specific areas, meeting the increasing needs of users.

- *Generate solution plan before generating complete Python code:* Additionally, we will experiment with creating solution plans before creating complete Python code for complex problems. This method will be applied to problems on CodeForce or LeetCode, helping the model solve complex problems more effectively.

- *Reinforcement Learning from Human Feedback:* Finally, we will apply training with human feedback to improve the quality and accuracy of the generated code. Feedback from users will help the model learn from mistakes and improve continuously, producing high-quality code that matches real-world requirements.

7. CONCLUSION

This report presents a comprehensive study on fine-tuning Salesforce's CodeT5p for code generation tasks. The primary focus is on evaluating the performance of traditional fine-tuning methods versus the Low-Rank Adaptation (LoRA) approach, which aims to accelerate the training process. The models are assessed using CodeBLEU, BLEU, pass@1, and pass@10 metrics on the MBPP and HumanEval benchmarks.

Key findings indicate that the CodeT5p-770m model fine-tuned with SFT (Supervised Fine-Tuning) achieves the highest performance across most metrics, with notable improvements in CodeBLEU, pass@1, and pass@10 scores. The use of LoRA also demonstrates competitive results, especially in terms of BLEU scores, suggesting its viability for efficient training without significant loss of performance.

The results underline the effectiveness of advanced fine-tuning techniques in enhancing code generation capabilities, offering valuable insights for future research and practical applications in automated code synthesis

REFERENCES

[1] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoderdecoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 (2021).

[2] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023).

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

[4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. arXiv preprint arXiv:2305.14314 (2023).

[5] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. arXiv preprint arXiv:2308.01240 (2023).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph el. at. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/pdf/2107.03374. (2021)

[7] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Xiaofei Ma, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, et al. 2023. Exploring Continual Learning for Code Generation Models. arXiv preprint arXiv:2307.02435 (2023).

[8] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. arXiv preprint arXiv:2212.10481 (2022).

[9] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. arXiv preprint arXiv:2303.15822 (2023).

[10] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020).

[11] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, , et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021).

[12] Angelica Chen, Jer´ emy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, Ethan Perez. 2024. Improving Code Generation by Training with Natural Language Feedback. arxiv.org/pdf/2303.16749 (2024).

[13] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. Advances in Neural Information Processing Systems 35 (2022), 1950–1965.