From Standalone to Combined: Optimizing PEFT Methods for Fine-Tuning CodeBERT on Code Seach

Junyin Zhao

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

A paradigm in the field of deep learning is the pre-training and subsequent fine-tuning of Large Language Models (LLMs). Fine-tuning refers to updating parts of a pre-trained model to perform better on a new task. However, fine-tuning LLMs, like Transformer-based CodeBERT, can be computationally expensive. This has motivated the development of parameter-efficient fine-tuning (PEFT) methods that merely update a small number of extra parameters during fine-tuning. While PEFT shows promise for natural language processing tasks, it is less studied for code intelligence tasks.

In this project, we systematically assess the efficacy of representative PEFT methods, namely LoRA, Prefix-tuning, and Adapter-tuning with its variants, when fine-tuning CodeBERT on the code search task. The results show most PEFT methods perform poorly, except LoRA. However, our proposed method, "loFF", combining LoRA and prioritized Adapter-tuning, outperforms full fine-tuning performance while only tuning 7.52% of parameters. Our further analysis indicates loFF strengthens CodeBERT's semantic modelling and handling of complex functions (code). These results indicate the efficacy of PEFT methods for a code intelligence task, encouraging a strategy of combining methods when the standalone PEFT methods fail. However, it also suggests that the effectiveness of PEFT approaches in other code intelligence tasks or other domains beyond natural language should be examined.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Junyin Zhao*)

Acknowledgements

First and foremost, I would like to convey my gratitude to my parents and friends for their unwavering support throughout the year. Especially for Shi and Feng, who, despite "behaving" like "netizens" :) , never ceased to bolster my confidence. Furthermore, thank you, my 3 teammates, your feedback is really helpful!

My sincere thanks go out to my project supervisors at Amazon, Prarit and Camille. Their guidance and support went beyond just the finish of the project. They enriched me with invaluable life lessons and wisdom, especially the ethos of taking the first step to truly gauge a challenge, and the importance of not denying myself before others. These lessons have profoundly reshaped how I face challenges and are meaningful in building my confidence, particularly in light of personal trauma from my childhood.

Furthermore, I am grateful to my school project supervisor, Iain, whose keen advice and insights were critical in refining this project.

I consider myself truly fortunate to have such insightful supervisors.

Table of Contents

Chapter 1

Introduction

Recently, the strategy of pre-training and then fine-tuning has become a paradigm in the deep learning field. Pre-training large models on generalized domain data and then fine-tuning them on task-specific data can significantly improve their performance compared to training them directly on task-specific data [8, 38, 4]. The domain of natural language processing (NLP) has witnessed significant advancements, particularly with the success of pre-training Transformer-based large language models (LLMs) and fine-tuning them for downstream tasks. These LLMs, such as GPT-4 [35], have demonstrated remarkable capabilities in natural language understanding and generation.

Considering the similarity between natural language and programming languages [19, 5], researchers have demonstrated the effectiveness of employing LLMs in the field of code intelligence [47, 10, 11]. Code intelligence refers to the development of AI techniques for programming languages. Typical tasks in this domain include code completion (predicting code based on previous segments), code summarization (providing brief descriptions of code snippets), and code search (finding relevant code for natural language queries).

Despite their effectiveness, LLMs face several challenges. Their capabilities rely on hundreds of millions of trainable parameters that are pre-trained with abundant data [50]. Although the pre-training step can be undertaken by large companies with substantial resources and the pre-trained models are typically open access online, fine-tuning these models on small, task-specific datasets remains computationally expensive. This is because the fine-tuning process re-trains all the parameters of the pre-trained model for each downstream task, necessitating a significant amount of labelled data for each task. Additionally, the large number of parameters in these pre-trained models makes them cumbersome to deploy and store on hardware during training. This limits their

accessibility to researchers with constrained resources.

1.1 Motivation

Parameter-Efficient Fine-Tuning (PEFT) is a technique designed to address these challenges. Unlike full fine-tuning, which updates all pre-trained weights and requires storing the entire model for every downstream task, PEFT retains and stores only the additional parameters introduced by modules appended to the model. This approach updates only a small fraction of the model's parameters and keeps the pre-trained weights unchanged. Despite this, some PEFT methods could obtain performance on par with full fine-tuning [21, 22, 28]. This strategy not only optimizes computational resources but also reduces the challenges of deployment and storage, making these powerful models more accessible.

While the efficacy of PEFT on natural language tasks is well-documented [16, 9, 22], its application in the field of code intelligence is still relatively underexplored. Successfully deploying PEFT for code intelligence could substantially cut training costs for organizations that are developing code intelligence tools. This, in turn, would broaden access to powerful AI coding assistants that leverage LLMs.

Several studies [41, 46] have employed Adapter-tuning [21], a prevalent PEFT approach, for specific code intelligence tasks, yielding favourable results. But they lack comparison across different PEFT methods. Research has shown [16, 34, 18] deploying different PEFT methods on the same downstream task can obtain diverse performance. Some methods do not even demonstrate the expected parameter efficiency on certain tasks. Given this background, our project aims to evaluate and compare the performance of various PEFT methods when deployed with LLMs for code intelligence tasks.

1.2 Objective And Hypothesis

The primary goal of this project is to explore and compare the **effectiveness** and **parameter efficiency** of different parameter-efficient fine-tuning (PEFT) methods on code intelligence tasks. Parameter efficiency refers to achieving strong performance with as few tuned parameters as possible. We aim to determine and compare the minimum parameter budget (number of parameters tuned) needed for the deployed PEFT methods to match the performance of full fine-tuning.

Specifically, we focus on applying three representative PEFT methods, *i.e.,* LoRA (Low-Rank Adaption) [22], Prefix-tuning [28], and Adapter-tuning [21], to fine-tune the CodeBERT model on the code search task. To better understand the similarities and differences between these methods, we divide these methods, including variants of Adapter-tuning, into groups based on the specific sub-layers they modify (details in Section 3.1) in the model. We will also evaluate the feasibility of combining the top-performing methods from each group, as prior work has shown that hybrid PEFT approaches can further improve performance on some natural language tasks [2, 34].

Our objectives can be summarized as:

- Evaluate the efficacy of standalone PEFT methods including LoRA, Prefix-tuning and Adapter-tuning with its variants;
- Develop and assess a combined method that integrates multiple PEFT methods;
- Conduct analysis to probe how these methods impact the model's performance.

Our hypotheses are:

- **Diverse Performance across PEFT Methods**: We anticipate that while certain methods might approach or even surpass the performance of full fine-tuning when given more trainable parameters, others may lag behind. Differences might also manifest between and within our categorized groups;
- **Synergistic Potential of PEFT Combinations:** Combining PEFT methods may lead to a further performance gain by harnessing their respective strengths, potentially eclipsing the performance of both full fine-tuning and isolated PEFT methods deployment.

1.3 Results Achieved

Standalone PEFT Methods Evaluations: LoRA, Prefix-tuning, and Adapter-tuning, along with their variants, were evaluated for fine-tuning CodeBERT on code search. Among them, only LoRA's performance approached that of full fine-tuning. This indicates the need to examine the efficacy of PEFT approaches other than natural language tasks.

Combined Method Evaluations: Our proposed approach "loFF", combining LoRA with prioritized Adapter-FFN, showed promising performance. With just 7.52% of

parameters tuned, it outperformed any standalone methods and full fine-tuning. This indicates the PEFT methods' efficacy in the code intelligence field.

Performance Analysis: Our attention analysis, which examines the model's attention weights, suggests loFF strengthens captured semantic features compared to standalone methods. Moreover, our qualitative analysis indicates loFF is better at handling long code sequences with repetitions than single methods.

1.4 Overview

The remainder of this report is structured as follows:

Chapter 2 provides background on pre-trained large language models (LLMs) for code intelligence tasks, fine-tuning, and parameter-efficient fine-tuning (PEFT) methods deployed.

Chapter 3 explains how we categorize PEFT methods into groups and brings in our research questions based on this classification.

Chapter 4 details the experiments to address the research questions from Chapter 3, including our experimental setup, evaluation metrics, results, and evaluations.

Chapter 5 presents a qualitative analysis examining the models' performance based on our experiments. We also discuss the limitations of our project here.

Finally, Chapter 6 summarizes our key findings with a discussion about their significance for PEFT methods and code intelligence, and suggests future work.

The Appendix (Chapter 7) provides our attention analysis and more experimental details.

Chapter 2

Background

In this chapter, we introduce some representatives of the Transformer-based LLMs used for code intelligence tasks with a brief comparison of their architectures and pre-training processes. Then, we introduce the fine-tuning process and the parameter-efficient fine-tuning (PEFT) methods we will use for our experiments.

2.1 Transformer-based, Pre-trained Code LLMs

The great generalizability demonstrated by Transformer-based [44] large pre-trained language models (LLMs), like BERT [8] and GPT series [38, 4], have motivated exploring their potential beyond Natural language processing (NLP). Code intelligence has emerged as such an area with massive publicly available programming data. Studies have shown that jointly pre-training on natural language and code enables models to learn associations between two modalities, achieving strong results on downstream tasks [2, 47]. Transformer-based code LLMs generally have three architectures: encoder-decoder, encoder-only, and decoder-only.

Encoder-decoder models have an encoder and decoder like the original Transformer [44]. Fig. 2.1 shows a basic structure of this type of model. The encoder encodes an input sequence into a feature representation, and then the decoder generates tokens in the output sequence conditioned on this encoding and the preceding generated tokens. Having both an encoder and decoder enables these models to complete both understanding and generation tasks, especially for tasks where input-output pairs are highly related, like translation. But they do not always surpass specialized single-component architectures [2, 48]. Prominent examples include CodeT5 [48], which predicts masked code tokens during pre-training, and AlphaCode [29], which generates

codes from encoder outputs containing bimodal sequences.

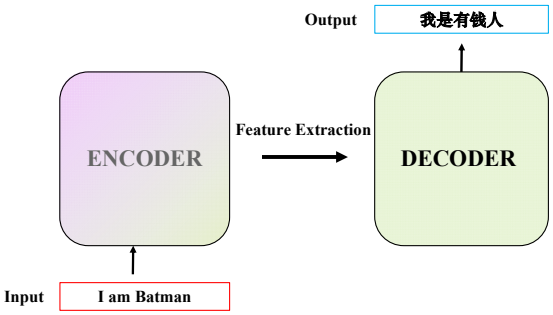


Figure 2.1: Encoder-Decoder architecture.

Encoder-only models utilize the bidirectional Transformer encoder, allowing tokens to attend to all the tokens in the input sequence, *i.e.,* those in the preceding as well as in the future timesteps. Fig. 2.2 shows an example of this type of model. After pre-training on code, they generate contextualized code embeddings significant for tasks like code search, which relies on semantic similarity of natural language query and its function sequence [23]. Representative examples include CodeBERT [11] and GraphCodeBERT [15], both built on RoBERTa [31] to predict masked code tokens during pre-training. GraphCodeBERT additionally incorporates code structure predictions. We use CodeBERT as our pre-trained code model because of its compatibility with the PEFT Python modules we deployed and relatively small model size.

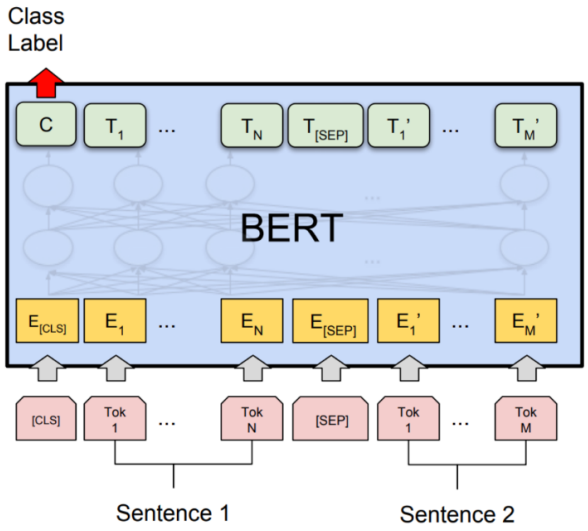


Figure 2.2: BERT, a typical example of encode-only architecture. Its notable feature is that the special *[CLS]* token (left bottom) learns to represent the whole sequence. The figure above is a reproduction of the original image taken from [8].

Decoder-only models use only the unidirectional Transformer decoder, which means each token can only attend to its preceding tokens. This design suits generation tasks that sequentially predict tokens based on prior context, without access to future tokens. Most decoder-only code models adopt GPT-2 [38] (Fig. 2.3) as their backbone and are pre-trained to predict future code tokens, known as Causal Language modelling. CodeGPT is a representative example [33].

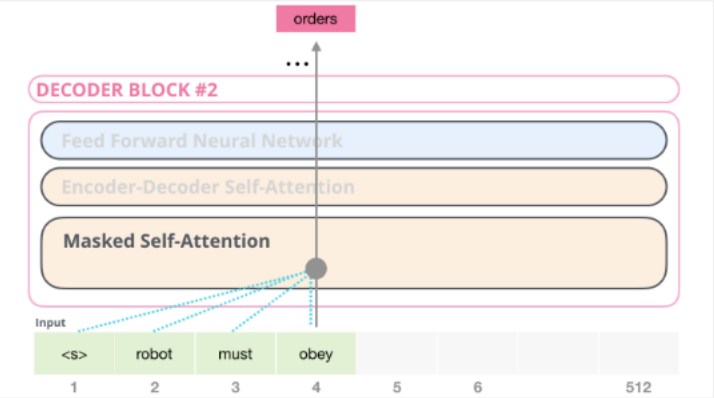


Figure 2.3: A simple illustration of GPT-2.[1] The model makes predictions based only on the previous tokens while future tokens are unavailable.

Many other code models like TreeBERT [24] fall under these architectures, but we do not list all of them. After pre-training, these models can process both natural and programming languages. They are then fine-tuned on downstream tasks requiring these capabilities.

2.2 Full Fine-Tuning and Its Alternatives

The fine-tuning process aims to leverage the general knowledge acquired during the pre-training step: by initializing with pre-trained weights, models can adapt quickly to new tasks without training from scratch. For example, Codex [6] employs the GPT-3 [4] model and fine-tunes it with programming languages for code intelligence tasks.

Full fine-tuning means updating all the model's pre-trained parameters. This can be computationally expensive and inefficient, especially with LLMs like GPT-3, which has 175 billion trainable parameters and requires approximately 350GB of storage.

To alleviate cost concerns, several alternatives have emerged. **Feature Extraction**:

[1] http://jalammar.github.io/illustrated-gpt2/

This approach uses the pre-trained model as a feature extractor to take hidden states from certain layers. A lightweight network is then trained on top of these extracted features, leveraging the knowledge from pre-training without the overhead of full fine-tuning [8, 31]. Fig. 2.2 shows a typical example of this technique, where the "C" token on the left top corner can be used as input to train neural networks for downstream tasks.
Partial Fine-Tuning: Only specific layers of the model are fine-tuned, while the rest remain unchanged. This allows extracting both low and high-level representations[2]and considerably reduces computational costs. However, these two methods are not always suitable across tasks or obtain comparable performance to full fine-tuning [42].

There are also other methods that can mitigate these issues and achieve great performance, such as **Knowledge Distillation** [20].[3] and using **Prompts** [4, 25].[4] While promising, these techniques fall outside the primary focus of our project. Next, we will introduce an emerging alternative to full fine-tuning called Parameter-Efficient Fine-Tuning (PEFT).

2.3 Parameter-Efficient Fine-Tuning (PEFT)

PEFT methods flexibly introduce small trainable neural networks (modules) into the sub-layers of Transformer-based models. During fine-tuning, only these modules' unique parameters are trained to learn task-specific features while pre-trained weights are unchanged. This acts like a "partial fine-tuning" approach, only updating parts of the model in fine-tuning. Moreover, the flexible quantity and inserting position of modules allow tuning both low and high-level representations, incorporating the benefits of the feature extraction approach. By only training lightweight modules with a bottleneck dimension[5] for each task, PEFT methods hold the following advantages:

- **Parameter efficiency**: PEFT tunes far fewer parameters than full fine-tuning, yet obtains similar or better performance. Our project focuses on this attribute.

[2]First (bottom) layers in a Transformer-based model normally learn low-level (pairwise) features while last (top) layers learn high-level features like semantics.

[3]This technique trains a smaller model (student) to mimic the behaviours of a larger pre-trained model (teacher). This allows the student to reproduce the teacher's outputs with considerably less computational overhead.

[4]Some phrases to guide generative LLMs like GPT-4 to produce phrases-related outputs without fine-tuning. They act as cues, allowing the model to invoke its pre-trained knowledge for specific tasks.

[5]Researchers found that pre-trained LLMs have a considerably lower "intrinsic dimension/rank" than their model size [1, 27], like 10e3 out of a 10e7 model, and their weights update can be done in this low dimension that is as effective as tuning them in the original model size [22].

- Scalability: Modules introduced by PEFT methods can be inserted at different positions of the models' sub-layers with various bottleneck dimensions, allowing flexibly changing the number of trainable parameters [37];
- Robustness: PEFT methods help mitigate overfitting and catastrophic forgetting[6] [12], which are two common problems faced by fine-tuning LLMs.

Next, we introduce PEFT methods used in this project and summarize them in Fig. 2.4.

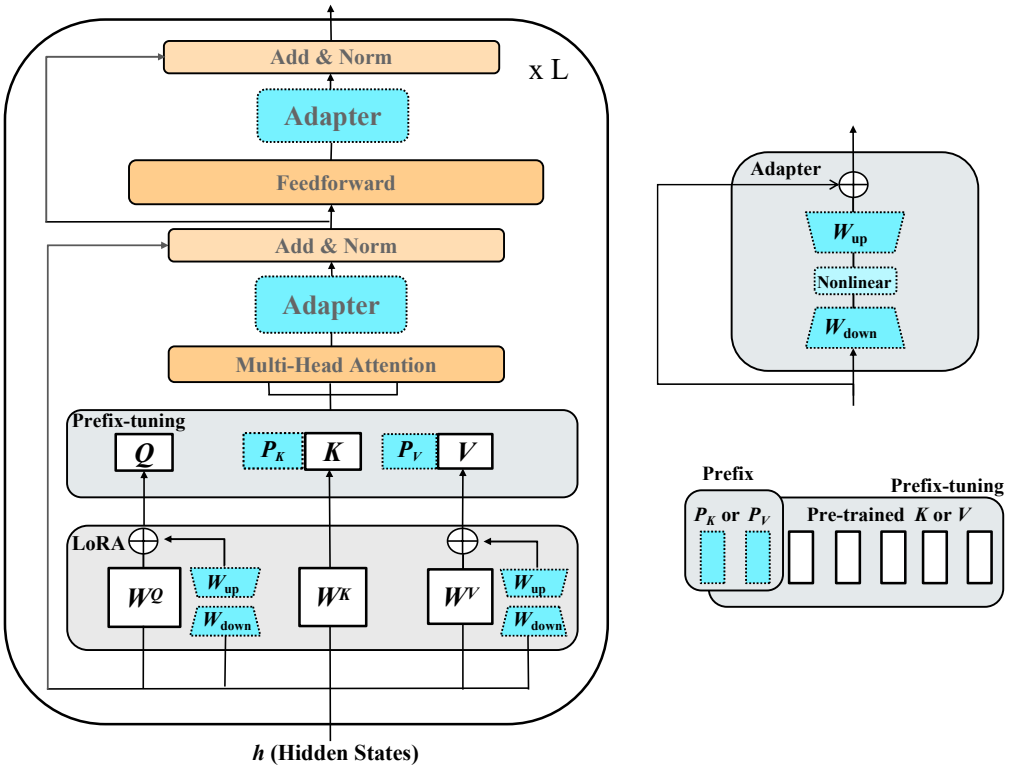


Figure 2.4: Illustration of the encoder architecture of Transformer-based models and the PEFT methods employed in this project. Blocks with dashed borders denote the additional modules introduced by these methods, *i.e.*, where the trainable parameters are located. We use h to collectively represent the hidden states of all sub-layers for simplicity. L denotes the number of encoder layers. Note that we emit the scale factor of LoRA in this figure, which is applied after the $\boldsymbol{W}_{up}$ projection.

2.3.1 Adapter-tuning

Adapter-tuning denotes a category of methods that introduce trainable Adapter modules (right top in Fig. 2.4) into the sub-layers of Transformer-based models. These mod-

[6] A situation where pre-trained LLMs forget previously learned information when learning new data.

ules are typically composed of two projection matrices, achieving parameter-efficient fine-tuning by operating in a bottleneck dimension. The first matrix, $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d \times r}$, compresses the input, $\boldsymbol{h} \in \mathbb{R}^{d}$, to a smaller bottleneck dimension r. r is a tunable hyperparameter and normally $r<d$. After a nonlinear activation function, the second matrix, $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{r \times d}$, projects the dimension back to d. A residual connection then combines this output with the original input, yielding $\boldsymbol{h}_{out}$.

$$\boldsymbol{h}_{out} \leftarrow \boldsymbol{h} + f(\boldsymbol{h}\boldsymbol{W}_{\text{down}})\boldsymbol{W}_{\text{up}}, \tag{2.1}$$

where f denotes the nonlinear activation function. When r grows to d, Adapter layers converge to a Multilayer Perceptron (MLP).

While variants of Adapter-tuning differ in module placement and quantity [43, 3, 40, 37], their architecture remains consistent. Typically, "Adapter-tuning" refers to the method proposed by Houlsby *et al.* [21], in which Adapter modules are added after the multi-head self-attention (ATTN) and feed-forward network (FFN) sub-layers within the model's encoder. However, sequentially adding them to the original model leads to a drawback: increased inference time.[7] In our study, we implement this canonical approach and its two variants: placing modules either only after the ATTN [43] or only after the FFN [37] sub-layers.

2.3.2 Prefix-tuning

Prefix-tuning [28] incorporates prefix vectors (length l) with the input of the multi-head attention (ATTN) sub-layers, treating them as additional tokens to learn task-specific information. Specifically, two matrices, $\boldsymbol{P}_k \in \mathbb{R}^{l \times d}$ and $\boldsymbol{P}_v \in \mathbb{R}^{l \times d}$, are concatenated with the original **keys** and **values** in each encoder layer. ATTN then operates with these extended keys and values:

$$\text{ATTN}\left(\mathbf{Q}, \text{concat}\left(\boldsymbol{P}_k, \boldsymbol{K}\right), \text{concat}\left(\boldsymbol{P}_v, \boldsymbol{V}\right)\right), \tag{2.2}$$

where $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ denote the original Query, Key, and Value, respectively. Each of them is obtained by projecting the hidden states, $\boldsymbol{h} \in \mathbb{R}^{d}$, through their respective linear projections. For instance, the Query, $\boldsymbol{Q}$, is formulated as $\boldsymbol{h}\boldsymbol{W}^{Q}$ (see Fig. 2.4).

Note that Prefix-tuning has a shortcoming as well: as the length l of the prefix vector increases, the model becomes constrained in processing extended input sequences,

[7] This occurs because the model becomes deeper.

given the inherent maximum sequence length limitation of models.[8] Similar works include P-tuning [30] and Prompt-tuning [26]. Prompt-tuning adapts Prefix-tuning, adding prefix vectors only to the first layer of the Transformer. However, this method merely works well with LLMs with billions of parameters. Thus, we do not utilize it in this project.

2.3.3 LoRA

LoRA [22] adds trainable low-rank matrices in the multi-head attention (ATTN) sub-layers of the Transformer to approximate weight adaptations in the "intrinsic rank" (see note 5 above).

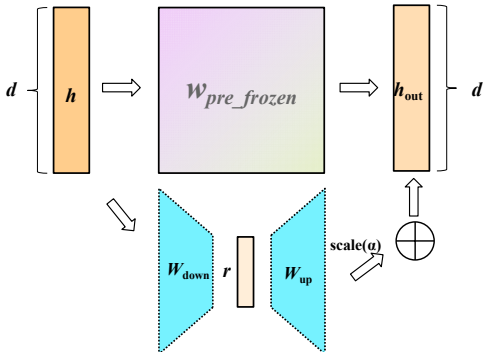


Figure 2.5: The figure above illustrates the structure and working of LoRA. LoRA employs two low-rank matrices, as shown in the bottom part of the figure, to approximate the weight-updating process during fine-tuning. The upper part of the illustration depicts outputs derived from pre-trained weights. In the figure, only the weights represented with dashed lines are updated during the fine-tuning phase.

Specifically, when fine-tuning the pre-trained weights $\boldsymbol{W}_{\text{pre}}$ to $\boldsymbol{W}_{\text{tuned}}$, instead of directly updating them, LoRA approximates this update with two matrices $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d \times r}$ and $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{r \times d}$, where $r << d$ (low rank). Thus, LoRA changes the full fine-tuning process from $\boldsymbol{W}_{\text{tuned}} \leftarrow \boldsymbol{W}_{\text{pre}}$ to $\boldsymbol{W}_{\text{tuned}} \leftarrow \boldsymbol{W}_{\text{pre\_frozen}} + \boldsymbol{W}_{\text{down}}\boldsymbol{W}_{\text{up}}$, where $\boldsymbol{W}_{\text{pre\_frozen}}$ indicates that the pre-trained weights are unchanged. These matrices are generally added to the **query** and **value** projections in the ATTN sub-layers.[9] As displayed in Fig.

[8] Generally, neural network models have a maximum accepted sequence length. Here it equals the prefix length plus the input sequence length.

[9] The authors found this is the best setting balancing simplicity and parameter-efficiency, compared with implying LoRA matrices to other combinations of elements in $\{\boldsymbol{W}^{\text{Query}}, \boldsymbol{W}^{\text{Key}}, \boldsymbol{W}^{\text{Value}}, \boldsymbol{W}^{\text{Output}}\}$.

2.5, LoRA passes the hidden states, $\boldsymbol{h} \in \mathbb{R}^d$, to the added matrices as input, modifying the projection output $\boldsymbol{h}_{out}$:

$$\boldsymbol{h}_{out} \leftarrow \boldsymbol{h}_{out} + \alpha \cdot \boldsymbol{h}\boldsymbol{W}_{\text{down}}\boldsymbol{W}_{\text{up}}, \tag{2.3}$$

where α is a fixed hyperparameter that can be tuned.[10] LoRA does not use a nonlinear activation function as Adapter-tuning does, so LoRA will converge to fully fine-tuning the pre-trained model if r equals d.

The primary advantage of LoRA is that it does not lead to increased inference time. This is because the formula "pre_frozen + down*up" can be summed to obtain one updated set of weights that retains the same size as the original.

2.3.4 Others

There are also other PEFT methods not employed in this project. For instance, BitFit [39] is a method that re-trains only the bias terms of the pre-trained model during fine-tuning. We do not list all of them in this report.

Considering that Adapter-tuning, Prefix-tuning, and LoRA typically yield strong results across natural language tasks [9], we focus on employing them for our experiments.

[10] The authors changed the value of α for different tasks, based on the model's performance on the validation set. Subsequently, it will not change during the fine-tuning process.

Chapter 3

Research Methodology

This section illustrates how we categorize the previously introduced PEFT methods. Based on this classification, we present our research questions.

3.1 A Simplified And Unified View Of PEFT Methods

We introduce a simplified classification of our deployed PEFT methods based on which core Transformer sub-layers they directly modify.[1] This offers a systematic view to better understand the similarities and differences between the methods.

The Transformer encoder consists of two main sub-layers (Fig. 3.1): multi-head self-attention (ATTN) and feed-forward network (FFN). Meanwhile, our studied PEFT methods modify the outputs of these sub-layers. Thus, we categorize the methods into three groups:

- Directly modifying ATTN sub-layer outputs;
- Directly modifying FFN sub-layer outputs;
- Directly modifying both ATTN and FFN outputs.

Fig. 3.1 and Table 3.1 shows a visual and tabulated summary of these categories, respectively. Additionally, to compare the parameter efficiency of these methods, we introduce how to count the number of tunable parameters of them in Section 4.1.5. The three categories can be elaborated as follows:

[1] In our project, "directly modifying" means adapting the outputs of sub-layers at the same encoder layer, rather than indirectly influencing the results in subsequent encoder layers as these encoder layers are stacked together.

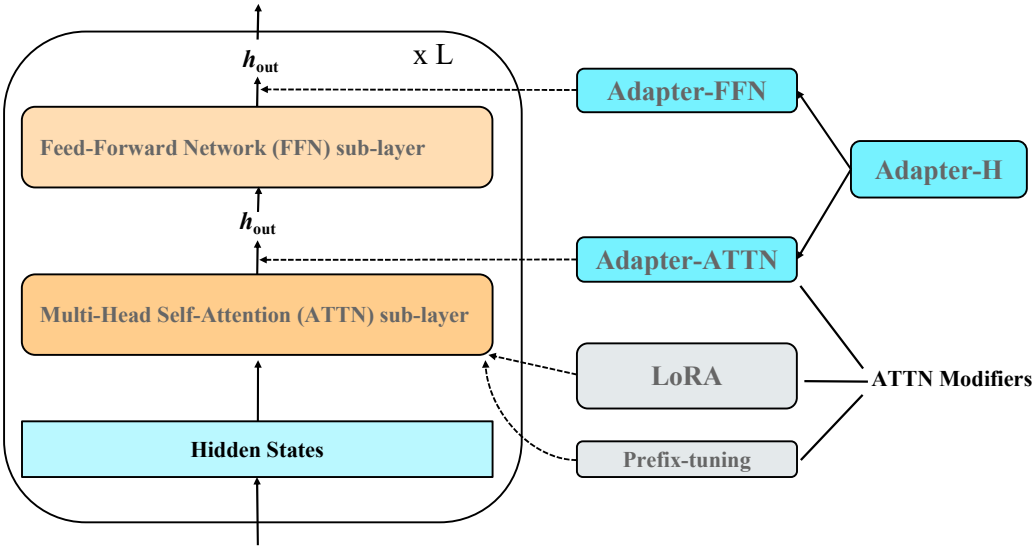


Figure 3.1: Illustration of an abstract architecture of the Transformer encoder (left part). The dashed lines point to the component that is directly altered by the PEFT methods (right part). $\boldsymbol{h}_{\mathbf{out}}$ represents the modified outputs (or hidden states) that are input into subsequent sub-layers. While we use the same sign to represent the modified output from both ATTN and FFN sub-layers for simplicity, they are typically different in general.

- ATTN Modifiers: This group directly modifies the outputs of the ATTN sub-layers. It includes methods such as Prefix-tuning and LoRA;
- Both ATTN and FFN Modifiers: We denote the configuration of Adapter-tuning proposed by Houlsby *et al.* [21] as Adapter-H, which uniquely modifies the outputs of both ATTN and FFN sub-layers, placing it in a distinct group;
- Position-Specific Adapter-tuning: As highlighted in Section 2.3.1, implementing Adapter-tuning offers us flexibility, allowing us to insert Adapter modules at various positions within the Transformer architecture. We consider two variants in our project. **Adapter-ATTN**: This configuration inserts Adapter modules immediately after the ATTN sub-layers [43]. It specifically modifies the ATTN sub-layers outputs, categorizing it under the ATTN Modifiers group; **Adapter-FFN**: This method introduces Adapter modules after the FFN sub-layers [37]. It exclusively alters the outputs of the FFN sub-layers, placing it in a distinct group.

Groups	Directly Modified Outputs	Methods
Group 1	multi-head self-attention (ATTN)	Prefix-tuning LoRA Adapter-ATTN
Group 2	feed-forward network (FFN)	Adapter-FFN
Group 3	both ATTN and FFN	Adapter-H

Table 3.1: A unified view of the PEFT methods we deploy in our project. "Modified Outputs" means that the PEFT methods we use directly modify the outputs of the specified sub-layer in each encoder layer.

Note that the PEFT methods we apply introduce modules in **each** encoder layer. This means that, while a PEFT method like Adapter-FFN targets a specific sub-layer (in this case, FFN), it has an **indirect** effect[2] on the outputs of both ATTN and FFN sub-layers in the subsequent encoder layers. Consequently, our classification is a **simplified** approach based on **direct** modifications. Based on this unified view, some questions arise naturally.

- **Effectiveness**: How do these PEFT methods perform under a consistent hyperparameter setting?
- **Intra-Group Efficiency**: Within the first group, which method offers superior parameter efficiency (Section 4.1.5)?
- **Inter-Group Comparison**: Across different groups, which type of direct modification (ATTN or FFN sub-layer outputs) is more effective for our task?
- **Optimal Efficiency with Adapter-H**: Considering the uniqueness[3] of Adapter-H, does it stand out as the most parameter-efficient method?
- **Combining Strengths**: Since methods in Group 1 can be deployed with Adapter-FFN,[4] could we combine the most efficient method from Group 1 with Adapter-FFN to achieve a performance gain?

To address these research questions, we introduce how we design each of them in the following section.

[2]Because the encoder layers are stacked with each other serially, the output of one layer is the input of the following layer, affecting the results of all subsequent layers. We denote this influence as the "indirect" modification in our project.

[3]Adapter-H is the only method which directly modifies the outputs of both ATTN and FFN sub-layers.

[4]Because they insert modules at different sub-layers in an encoder layer.

3.2 Research Questions (RQ)

To answer the above questions for the code search task (Section 4.1.2), we explore the following research questions and introduce our motivation and study design.

3.2.1 RQ1: Evaluating Adapter-H Performance (Group 3)

Group 3 denotes the group in Table 3.1.

Motivation: Previous research has demonstrated that the original Adapter configuration proposed by Houlsby *et al.* [21] can achieve comparable performance to full fine-tuning across natural language tasks with less than 10% of the total model parameters. In this research question, we adopt this original configuration, denoted as Adapter-H (see Fig. 3.1), to explore its effectiveness when applied to CodeBERT, a pre-trained code model, on the code search task.

Design: We will apply Adapter-H to fine-tune CodeBERT on the CodeXGLUE benchmark [33]. This benchmark includes the Python programming language, a language present in CodeBERT's pre-training dataset.

The bottleneck dimension (r) of Adapter modules plays a significant role in its performance, since it determines the number of trainable parameters. We will deploy Adapter-H with r in $\{64, 128, 256, 512\}$ based on preliminary experiments [21]. The upper limit of r is set to 512 to ensure it remains a "bottleneck" dimension, given the hidden size of CodeBERT is 768. We aim to evaluate the performance of Adapter-H with an increase in the bottleneck dimension. This will also reveal Adapter-H's effectiveness and parameter efficiency for this task. We will compare the results, in terms of the Mean Reciprocal Rank (MRR) metric (Section 4.1.4), and parameter efficiency, of Adapter-H against a fully fine-tuned CodeBERT, which serves as our baseline. We will run experiments 2 times with different seeds to obtain more robust results, reporting their mean values with standard deviation.

Hypothesis: As the only method that modifies the results of both the ATTN and FFN sub-layers, Adapter-H might offer an upper bound in terms of performance and efficiency among PEFT methods for this task. Additionally, a larger r may enhance Adapter-H's performance, but this would come at the expense of parameter efficiency due to the increase in trainable parameters.

3.2.2 RQ2: Comparing ATTN Modifiers Efficiency (Group 1)

We term the PEFT methods that directly modify the outputs of ATTN sub-layers as "ATTN Modifiers". These constitute Group 1, as represented in Table 3.1 and Fig. 3.1.

Motivation: Our first aim is to evaluate the effectiveness and parameter efficiency of the ATTN Modifiers group, which includes Prefix-tuning, LoRA, and Adapter-ATTN. Within this group, we expect to identify the top-performing method under the same parameter budget. This method will serve as a representative of the group's efficiency. By identifying the top performer, we will be able to make comparisons with other groups in RQ3 and use the representative in a combined approach for RQ4.

Design: We fine-tune CodeBERT on the CodeXGLUE benchmark, for the code search task, using each ATTN modifier, and evaluating them based on their MRR value and the number of tunable parameters. We use the fully fine-tuned CodeBERT as our baseline. To maintain a fair comparison, the hyperparameter settings are consistent (details in Section 4.1.6), except for the bottleneck dimensions of each method.

For Prefix-tuning, the length l of the prepended vectors is set to $\{5, 10, 20, 30, 50, 100, 256\}$, constrained by CodeBERT's maximum input length and the test set's maximum input sequence length.[5]

For LoRA, an initial step is to determine the scale factor α (Eq. 2.3.3) for optimal LoRA performance.[6] Experiments are conducted with r set to 64 and α in $\{1, 2, 3, 4\}$.[7] We select the α value that yields the best average MRR on the validation set over 2 runs. Subsequently, this value is used for LoRA with different bottleneck dimensions [34, 22].

Bottleneck dimensions (r) values are varied as $\{32, 64, 128, 256\}$ for LoRA and $\{64, 128, 256, 512$[8]$\}$ for Adapter-ATTN. When using the same r values, LoRA's tunable parameters are roughly double those of Adapter-ATTN and Prefix-tuning. Therefore, LoRA's r is set at half of that for Adapter-ATTN to compare their efficiency under an equivalent budget. More details can be found in Section 4.1.5. The mean performance, accompanied by the standard deviation over two runs, will be reported.

Hypothesis: A method may hold the highest parameter efficiency within Group 1. For these methods, increasing the bottleneck dimension (either l or r) may lead to

[5]The maximum acceptable input length for CodeBERT is 512, which means the sum of the input sequence length and the prefix vector length should not exceed 512. The maximum length of the prefix vectors we use here is 256, since the maximum length of the input sequence in the test set is also 256.

[6]The performance of LoRA is sensitive to this hyperparameter, and its optimal value varies across tasks [22].

[7]These are commonly used values from previous studies [2, 34].

[8]Same as Adapter-H, to satisfy a bottleneck or "low rank" dimension.

performance gains.

3.2.3 RQ3: Relative Efficiency Across Groups

Motivation: ATTN and FFN sub-layers of the Transformer encoder are believed to capture different feature patterns in data [13]. Based on this, we aim to identify which sub-layer modification type (Table 3.1)—namely, directly modifying ATTN only, FFN only, or both—exhibits superior efficiency for the code search task. This indicates which sub-layers are probably more influential for this task, providing us with insight into optimizing parameter allocation when combining PEFT methods in RQ4.

Design: To evaluate Adapter-FFN, we will fine-tune CodeBERT on CodeXGLUE using the same range of bottleneck dimensions as Adapter-ATTN in RQ2 (64 to 512). Based on the top performer identified for the ATTN Modifiers group in RQ2, we will compare it against Adapter-FFN (Group 2) and Adapter-H (Group 3). The comparison will use the mean MRR on the test set under the same parameter budget, contrasting against full fine-tuning as a baseline. This approach allows us to compare the performance of the three groups with shared hyperparameters and metrics.

Hypothesis: Considering that Adapter-H is the sole method directly modifying both ATTN and FFN sub-layers, it may emerge as the most parameter-efficient method for the code search task.

3.2.4 RQ4: Enhancing Efficiency Through Cross-Group Methods Combination

Motivation: Based on the results of previous research questions, we will identify the best-performing ATTN Modifier from Group 1 (Table 3.1) and evaluate Adapter-FFN's efficacy. This exploration leads us to examine the potential synergy between these two methods. The goal is to create a new method that falls under Group 3, which modifies the outputs of both ATTN and FFN sub-layers directly. Specifically, drawing insights from RQ3, we are interested in determining if a strategic combination of the top-performing method from Group 1 with Adapter-FFN can outperform the deployment of any standalone method on CodeBERT for the code search task.

Since the design of RQ4 relies on results from previous research questions, we will introduce its design and findings in Section 4.2.4.

Chapter 4

Experiments

In this chapter, we detail our experimental setup, including the baseline model, dataset, and the specific task on which we focus. We also introduce the implementation details of our experiments, such as hyperparameter settings.

4.1 General Setup

4.1.1 Baseline

Due to resource constraints and the compatibility of the CodeBERT [11] model with the PEFT libraries we use, we employ the publicly available CodeBERT-base model[1] for our experiments. Fully fine-tuning this model serves as our baseline. This allows us to compare the performance of the baseline with different PEFT methods, evaluated based on task-specific metrics and parameter efficiency.

Model Overview:

- **RoBERTa**: RoBERTa (Robustly optimized BERT approach) is a variant of BERT[2] [8] that exhibits great performance on several natural language understanding tasks. Key enhancements of RoBERTa include training on a larger dataset, using larger batch sizes, and extending the training duration. As a result, RoBERTa can handle longer input sequences compared to BERT. Several pre-trained code models, including CodeBERT, adopt RoBERTa as their backbone. RoBERTa is available in multiple size variants. In our experiments, we employ the RoBERTa-base version, which consists of 125 million parameters: 12 encoder

[1] https://huggingface.co/microsoft/codebert-base

[2] A pre-trained model with an encoder-only Transformer architecture. It achieved state-of-the-art performance on several natural language understanding tasks.

layers, 768-dimensional embeddings, and 12 attention heads for each attention sub-layer.

- **CodeBERT:** CodeBERT-base employs the RoBERTa-base architecture as its model structure. It is pre-trained on the CodeSearchNet [23] corpus, containing paired natural language (NL) and programming language (PL) examples. Its pre-training objectives include Masked Language Modelling [8] and Replaced Token Detection [7], which can be summarized as predicting the masked NL or PL tokens in an input sequence. Thus, CodeBERT can effectively encode bidirectional contexts for bimodal input sequences. It tokenizes the NL and PL using byte-level Byte Pair Encoding [38], which splits words and code segments into sub-word units (see App. 7.6 for an example). This tokenization approach allows CodeBERT to manage rare and out-of-vocabulary[3] terms. In summary, CodeBERT's ability to contextually encode both NL and PL sequences makes it a suitable choice for bimodal understanding tasks like code search, our focus in this project.

4.1.2 Task

Code Search: In code search, a natural language query is given as input, and the goal is to identify the most semantically similar code snippets (functions in our experiments) from a collection of candidate functions, including distractors (unrelated candidates). This can be achieved using **joint embeddings**. The objective is to project the queries and functions into a shared embedding space where semantically aligned queries and functions are positioned close to each other.

To represent an entire sequence as an embedding, a prevalent approach is to leverage the ***[CLS]*** token of BERT-based models. For models like CodeBERT, the final hidden state (last encoder layer) of the *[CLS]* token could be used as a sequence-level representation [8], encapsulating the contextual information of the input sequence. Given CodeBERT's capability to encode both natural language and function sequences with its *[CLS]* token, it is suited for the code search task achieved with joint embeddings.

The similarity between query and function embeddings can be quantified using their inner product. Therefore, the training objective for code search is contrastive learning:

[3]When the model encounters an unseen word, *i.e.,* a rare or out-of-vocabulary word, during inference, it can break it down with sub-words that might have been seen during training and represent this unfamiliar word with them.

maximizing the inner product for aligned query-function pairs, while minimizing it for mismatched distractor pairs. This can be formulated as:

$$-\frac{1}{N}\sum_{i}\log\left(\frac{\exp(CLS(c_i)^T \cdot CLS(q_i))}{\sum_j \exp(CLS(c_j)^T \cdot CLS(q_i))}\right), \tag{4.1}$$

where $CLS(c_i)$ is the embedding of the correct function, represented by the final hidden state of the *[CLS]* token encoded by CodeBERT. $CLS(c_j)$ (where $i \neq j$) denotes the embeddings of the distractors, *i.e.*, incorrect functions. $CLS(q_i)$ is the embedding of the associated query, which corresponds to the docstring of the functions (further details in Section 4.1.3).

The essence of this training objective can be understood in a way depicted in Fig. 4.1: positioning the query embedding, $CLS(q_i)$, close to its matching function, $CLS(c_i)$, while keeping it distant from distractor embeddings, $CLS(c_j)$, in the embedding space.

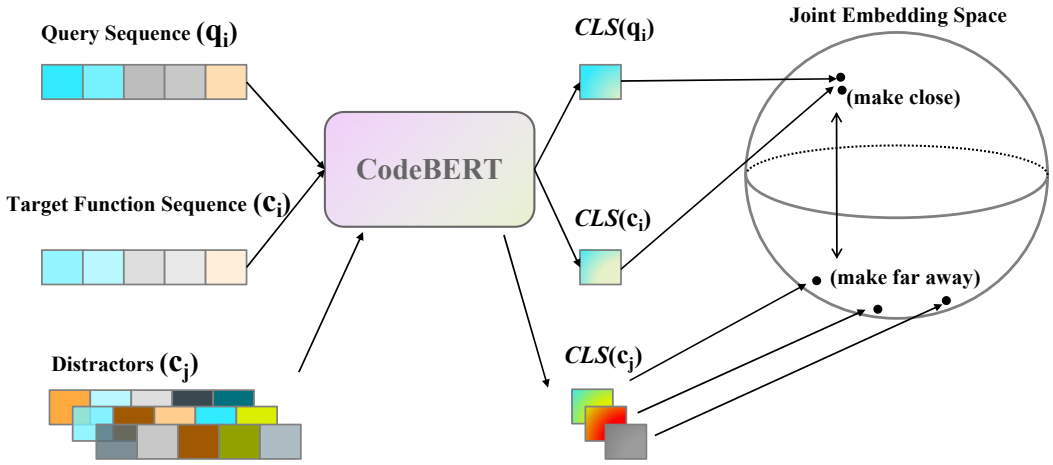


Figure 4.1: Illustration of the code search task objective: Embeddings of input sequences are encoded by CodeBERT and represented by the [*CLS*] tokens. The objective is to make the query embedding, $CLS(q_i)$, as close as possible to its semantically aligned function embedding, $CLS(c_i)$, while distancing it from distractors (unrelated functions) embeddings, $CLS(c_j)$.

4.1.3 Dataset

CodeXGLUE [33] is a widely-used benchmark for several code intelligence tasks, including code search. In our project, we employ the **CodeSearchNet AdvTest** dataset from CodeXGLUE for our code search task.

CodeSearchNet AdvTest is a cleaner version of the CodeSearchNet corpus [23]. It contains over 250,000 examples from the Python programming language. Each example pairs a function with its corresponding docstring. For code search, the **first paragraph** of a function's docstring is employed as the query for that function.

To better evaluate the generalizability of the trained model, each example's function name and variable names are **normalized** in both the validation and test set. This means that every function and the $i-th$ variable are represented as "Func" and "arg_i", respectively. A normalized example from this dataset is demonstrated in Fig. 4.2. Such normalization can be challenging for models, as original, semantically-rich names (normally used naming strategy) might be essential for discerning the function's intent or the role of variables. Besides, the maximum length of the function sequence is set to 256. Sequences longer than this limit will be truncated, while shorter sequences will be padded with an attention mask to avoid unnecessary computations.

All the models in our experiments are trained using the training set and then evaluated on the validation and test sets. The dataset's statistics are shown in Table 4.1.

Query:

Scans through a string for substrings matched some patterns.

Gold Code:

```
def matchall(text, patterns):
    ret = []
    for pattern in patterns:
        match = re.findall(pattern, text)
        ret += match
    return ret
```

Normalized Code:

```
def func(arg0, arg1):
    arg2 = []
    for arg3 in arg1:
        arg4 = re.findall(arg3, arg0)
        arg2 += arg4
    return arg2
```

Figure 4.2: An example from the **CodeSearchNet AdvTest**. Image taken from [33]

Language	Dataset	Count
Python	Training	251,820
	Validation	9,604
	Test	19,210

Table 4.1: Statistics of **CodeSearchNet AdvTest** used for the code search task.

4.1.4 Evaluation Metrics

Following prior work employing this dataset [11, 33], we evaluate our models' performance based on the Mean Reciprocal Rank (MRR). MRR is commonly used in ranking and retrieval tasks to assess a model's ability to highly rank relevant results for a query. This aligns with the goal of code search to semantically match queries to functions.

Specifically, MRR calculates the average reciprocal rank at which the most relevant result is retrieved across a set of queries Q (Eq. 4.2). The MRR score ranges from 0 to 1, with 1 being the ideal score. To illustrate, a MRR value of $\frac{1}{2}$ for a single query means the most relevant result is ranked second by the model.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}, \tag{4.2}$$

where $|Q|$ denotes the size of the query set, and $rank_i$ is the predicted rank of the correct answer for the i-th query.

4.1.5 Parameter Efficiency

Parameter efficiency measures the effectiveness of a method's performance in relation to its number of trainable parameters. A method is considered more parameter efficient if it can achieve comparable or superior experimental performance with fewer trainable parameters than another method.

In this project, we evaluate and compare the parameter efficiency of several PEFT methods. Specifically, we compute the percentage of trainable parameters used by each PEFT method relative to the full CodeBERT model size. Their performance is evaluated using the Mean Reciprocal Rank (MRR) metric. A PEFT method that achieves a higher MRR with fewer trainable parameters is considered more efficient. By comparing trainable parameters and MRR values, we can assess their relative parameter efficiency both amongst themselves and in relation to full fine-tuning.

We describe below the method for computing the number of tunable parameters for each PEFT method. For rounding efficiency and consistent comparison, we exclude the bias terms introduced by the Adapter modules, as their impact on the overall parameter efficiency is minimal. For instance, deploying Adapter-H with a bottleneck dimension of 512 adds only about 0.025% more parameters due to bias terms, especially when considering the CodeBERT model size of 125M.

CodeBERT is an encoder-only model (Section 4.1.1), and each of its encoder layers

contains a multi-head self-attention (ATTN) sub-layer and a feed-forward network (FFN) sub-layer. We denote the hidden state dimension as d and the bottleneck dimension introduced by PEFT methods as r (or prefix vector length l for Prefix-tuning). **For each encoder layer**, the number of additional trainable parameters brought by each method through their specialized modules (Fig. 2.4) is:

- LoRA: It introduces two trainable matrices, $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d\times r}$ and $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{r\times d}$, for both the query and value projections in the ATTN sub-layer. This brings in extra $4 \times r \times d$ parameters.
- Adapter-H: Similarly to LoRA, Adapter-H incorporates two matrices, $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d\times r}$ and $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{r\times d}$, to both ATTN and FFN sub-layers, adding a total of $4 \times r \times d$ parameters.
- Adapter-ATTN and Adapter-FFN: These two methods introduce Adapter modules only in the ATTN or FFN sub-layers, respectively. Thus, they add half of the parameters compared to Adapter-H, namely $2 \times r \times d$.
- Prefix-tuning: This method prepends prefix vectors with length l to both keys and values in the ATTN sub-layer, resulting in $2 \times l \times d$ parameters.

In summary, for a specific r, LoRA and Adapter-H update an equivalent number of parameters, while Adapter-ATTN and Adapter-FNN train half of that. When $l = r$, Prefix-tuning tunes the same number of parameters as Adapter-ATTN and Adapter-FNN. We summarize these computations in Table 4.2.

PEFT Methods	**#Trainable Parameters In Each Encoder Layer**
LoRA	$4 \times r \times d$
Adapter-H	$4 \times r \times d$
Adapter-ATTN, Adapter-FFN	$2 \times r \times d$
Prefix-tuning	$2 \times l \times d$

Table 4.2: The number of (#) trainable parameters in each encoder layer for the PEFT methods used in our experiments. r is the bottleneck dimension. d is the hidden state dimension of the model. l is the length of prefix vectors.

4.1.6 Implementation Details

Frameworks and Libraries: We implement our experiments using PyTorch [36] library and employ the HuggingFace Transformers library [51] to deploy the pre-trained CodeBERT-base model. We use OpenDelta [9] to implement the PEFT methods discussed in our project. Besides, we utilize the fine-tuning and evaluation pipelines provided by CodeXGLUE [33].

Hyperparameters: Our goal is to fine-tune CodeBERT for the code search task using PEFT techniques. We generally follow the fine-tuning hyperparameter setting provided by CodeXGLUE, while we make two modifications due to our hardware constraints: We decrease the batch size to 16 for both training and validation set; we set the learning rate to 1×10^{-5} based on the validation set performance of the fully fine-tuning CodeBERT, the results of which are presented in Table 7.3. The specific hyperparameters of PEFT methods, like the bottleneck dimension and scaling factor, are elaborated in the respective research question designs (Section 3.2).

Optimization: We employ the AdamW optimizer [32] to update the model's parameters and adopt a linear learning rate scheduler following the setup of CodeXGLUE. We train each model for 2 epochs. Within each epoch, we evaluate the model's performance on the validation set 10 times and save the best-performing model, which is used for test set evaluation. For robustness, most experiments are conducted twice with different random seeds, with their mean performance and standard deviation reported.[4] We run our experiments with 1 NVIDIA V100 card with 16GB of graphic memory.

[4]Due to limited resources, we ran full fine-tuning and our combined methods three times, while we ran the other methods only twice.

4.2 Experimental Results And Evaluation

4.2.1 RQ1: Evaluating Adapter-H Performance (Group 3)

Model	Note	#Parameters	Performance	
			Valid MRR (%)	Test MRR (%)
Baseline	Full Fine-Tuning	125M(100%)	**38.33**$_{\pm 0.09}$	**31.66**$_{\pm 0.12}$
Adapter-H-64	$r = 64$	235K(1.88%)	28.41$_{\pm 0.66}$	22.71$_{\pm 0.43}$
Adapter-H-128	$r = 128$	472K(3.76%)	29.30$_{\pm 0.26}$	23.52$_{\pm 0.25}$
Adapter-H-256	$r = 256$	943K(7.52%)	31.40$_{\pm 0.14}$	25.46$_{\pm 0.06}$
Adapter-H-512	$r = 512$	18M(15.04%)	33.62$_{\pm 0.35}$	27.45$_{\pm 0.46}$

Table 4.3: Results of fine-tuning CodeBERT on the code search task using Adapter-H are presented, where r denotes the bottleneck dimension of Adapter modules. We ran each experiment two times with different random seeds, and the mean performance (MRR) and standard deviation are shown. The model with the best validation MRR is selected and evaluated on the test set. Parameter counts are in millions (M) or thousands (K).

Results: As shown in Table 4.3, Adapter-H underperformed the baseline across all dimensions. However, its performance kept increasing with larger bottleneck dimensions (r), especially when r reached 256.

Evaluation: Our initial expectation was that Adapter-H, due to its unique ability to directly modify both ATTN and FFN sub-layers outputs, might show the upper bound among PEFT methods and obtain performance comparable to Full Fine-Tuning (baseline). However, even with the largest r set to 512 (which tunes 15.04% of the model size parameters), Adapter-H's test MRR was still about 0.4% lower than that of the baseline. This suggests that our hypothesis was only partially correct, indicating potential limitations of PEFT methods for this specific task.

A previous study [34] observed a similar trend: Adapter-H's performance in fine-tuning the RoBERTa model for the CoLA task[49] improved with an increasing bottleneck dimension, especially when r reached 256.[5] This suggests a potentially shared pattern in natural language understanding and programming language understanding tasks when deploying Adapter-H with the leverage of the *[CLS]* token.

[5]CoLA, a natural language understanding task assessing the linguistic acceptability of sentences, also uses the *[CLS]* token from BERT-based models to capture sentence-level information.

The poor performance of Adapter-H in prior research [34] for the CoLA task was attributed to hyperparameter choices. We concur with this perspective. Our default hyperparameter settings (Section 4.1.6), based on optimal full fine-tuning results, might not be the most suitable ones for fine-tuning models with a reduced parameter count. We leave this for future work, as we aim to evaluate multiple PEFT methods under the same hyperparameter settings.

Summary: Although increasing the bottleneck dimension improved Adapter-H's performance, it remained underperforming Full Fine-Tuning when fine-tuning CodeBERT on the code search task given our hyperparameter settings.

4.2.2 RQ2: Comparing ATTN Modifiers Efficiency (Group 1)

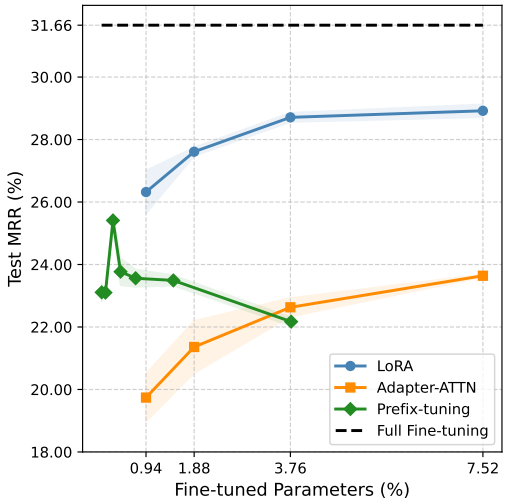


Figure 4.3: Performance variation when the fine-tuned parameters (bottleneck dimensions or length for Prefix-tuning) of three PEFT methods in Group 1 (ATTN Modifiers) are increased. The dashed line denotes full fine-tuning. Points indicate mean test MRR over two runs, with shaded areas as standard deviation.

Results: Fig. 4.3 and Table 4.4 show the performance of deploying different ATTN Modifiers (Fig. 3.1) for fine-tuning CodeBERT to code search. LoRA demonstrated the strongest performance among ATTN Modifiers, with a mean test MRR of 28.92% when the parameter budget increased to 7.52%. However, its performance almost plateaued upon reaching this parameter quota, and it still underperformed full fine-tuning (31.66%). Conversely, Adapter-ATTN showed poorer performance, only reaching 23.64% average

test MRR with the same quota. Meanwhile, Prefix-tuning's performance declined despite more tunable parameters, defying our hypothesis.

Model	Note	#Parameters	Performance	
			Valid MRR (%)	Test MRR (%)
Baseline	Full Fine-Tuning	125M(100%)	**38.33**$_{\pm 0.09}$	**31.66**$_{\pm 0.12}$
LoRA-32	$\alpha = 4$, $r = 32$	117K(0.94%)	32.44$_{\pm 0.77}$	26.32$_{\pm 0.69}$
LoRA-64	$\alpha = 4$, $r = 64$	235K(1.88%)	33.85$_{\pm 0.09}$	27.61$_{\pm 0.14}$
LoRA-128	$\alpha = 4$, $r = 128$	472K(3.76%)	35.32$_{\pm 0.11}$	28.71$_{\pm 0.15}$
LoRA-256	$\alpha = 4$, $r = 256$	943K(7.52%)	35.95$_{\pm 0.16}$	28.92$_{\pm 0.21}$
Adapter-ATTN-64	$r = 64$	117K(0.94%)	25.15$_{\pm 0.88}$	19.74$_{\pm 0.77}$
Adapter-ATTN-128	$r = 128$	235K(1.88%)	26.90$_{\pm 0.87}$	21.36$_{\pm 0.84}$
Adapter-ATTN-256	$r = 256$	472K(3.76%)	28.30$_{\pm 0.31}$	22.63$_{\pm 0.30}$
Adapter-ATTN-512	$r = 512$	943K(7.52%)	29.59$_{\pm 0.08}$	23.64$_{\pm 0.05}$
Prefix-tuning-10	l as 10	18K(0.14%)	28.50$_{\pm 0.31}$	23.10$_{\pm 0.08}$
Prefix-tuning-50	l as 50	92K(0.74%)	29.43$_{\pm 0.30}$	23.56$_{\pm 0.28}$
Prefix-tuning-100	l as 100	184K(1.48%)	29.31$_{\pm 0.14}$	23.49$_{\pm 0.17}$
Prefix-tuning-256	l as 256	472K(3.76%)	27.48$_{\pm 0.24}$	22.17$_{\pm 0.14}$

Table 4.4: Results on the code search task when varying bottleneck dimension (r) for LoRA and Adapter-ATTN, and prefix length (l) for Prefix-tuning. Two runs per setting with different random seeds. Mean results are shown with standard deviation. Best validation model evaluated on the test set. Parameter counts in thousands (K).

Evaluation: LoRA's stronger performance might be attributed to its parallel modification of attention inputs, enabling it to operate in a multi-headed manner alongside original keys/values (see Fig. 2.4). This allows it to capture more relationships, offering greater expressiveness than Adapter-ATTN's sequential output projection (single-headed). Prior work [37, 16] has also found that parallel modifications can outperform sequential tuning across NLP tasks, suggesting this pattern generalizes. Additionally, much like the residual connection [17], LoRA's additive integration (Eq. 2.3.3) might avoid overwriting pre-trained patterns (hidden states), while Adapter-ATTN is more likely to modify them during projection.

Contrary to our hypothesis, Prefix-tuning's performance declined (Fig. 4.3) as the length of prefix vectors increased, despite growing parameters. Fig. 7.9 shows the training curves of Prefix-256, which did not show a sign of overfitting as the validation

MRR did not demonstrate a notable decline. Thus, one possible reason is that the additional tokens disturbed learned relationships in the original input sequences. Moreover, initializing prefix vectors with random words [28] may have introduced variability in sequence semantics and *[CLS]* tokens' representations. This alignment disruption and semantic variation may override the benefits of increased tunable parameters.

Summary: For PEFT methods in Group 1, fine-tuning CodeBERT with LoRA achieved the highest parameter efficiency on the code search task. It obtained performance that was slightly inferior to Full Fine-Tuning. In contrast, both Adapter-ATTN and Prefix-tuning performed poorly. Interestingly, Prefix-tuning showed worse performance even with increased tunable parameters, suggesting increasing trainable parameters may not be a panacea for poor-performing PEFT methods on all tasks.

4.2.3 RQ3: Relative Efficiency Across Groups

From the results of RQ2, we observed that LoRA emerged as the most parameter-efficient method within Group 1. Given this, we expected to compare LoRA's relative efficiency with Adapter-FFN from Group 2 and Adapter-H from Group 3 under consistent hyperparameter settings.

Model	Note	#Parameters	Performance	
			Valid MRR (%)	Test MRR (%)
Baseline	Full Fine-Tuning	125M(100%)	**38.33**$_{\pm 0.09}$	**31.66**$_{\pm 0.12}$
LoRA-256	$\alpha = 4$, $r = 256$	943K(7.52%)	35.95$_{\pm 0.16}$	28.92$_{\pm 0.21}$
Adapter-H-256	$r = 256$	943K(7.52%)	31.40$_{\pm 0.14}$	25.46$_{\pm 0.06}$
Adapter-FFN-64	$r = 64$	117K(0.94%)	27.56$_{\pm 0.24}$	22.51$_{\pm 0.57}$
Adapter-FFN-128	$r = 128$	235K(1.88%)	29.82$_{\pm 0.26}$	23.91$_{\pm 0.09}$
Adapter-FFN-256	$r = 256$	472K(3.76%)	31.97$_{\pm 0.91}$	25.96$_{\pm 0.79}$
Adapter-FFN-512	$r = 512$	943K(7.52%)	32.45$_{\pm 0.21}$	26.35$_{\pm 0.23}$

Table 4.5: The results of the code search task when changing the bottleneck dimension r of Adapter-FFN. We conducted each experiment twice using different seeds and reported the mean values along with their standard deviation. The model that performs best on the validation set is selected for evaluation on the test set.

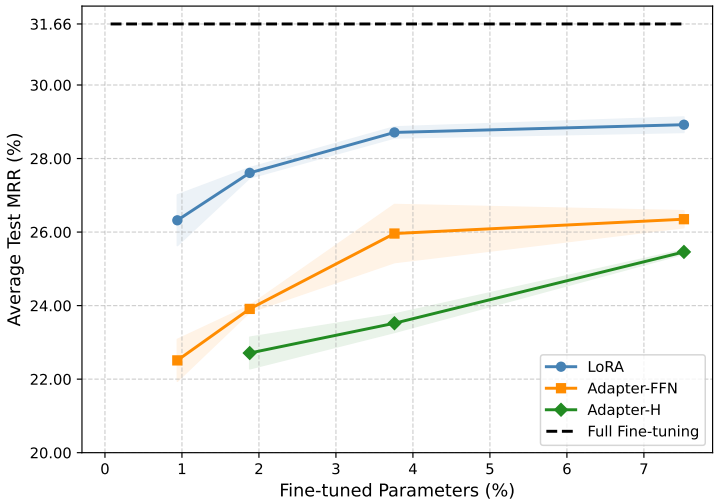


Figure 4.4: Performance variance when the number of fine-tuned parameters of PEFT methods from three groups (Table 3.1) are increased. Each point signifies the mean MRR value on the test set, and the shaded areas denote the standard deviation.

Results: Fig. 4.4 and Table 4.5 show the performance variation of Adapter-FNN, LoRA, and Adapter-H as the parameter budget increases. We observed that: **LoRA** outperformed other methods consistently with the same parameter budget, emerging as the most effective method across groups; **Adapter-FFN** surpassed Adapter-H, aligning results on several natural language tasks [34]; Adapter-FFN's performance improves with an increase in the bottleneck dimension, approximately plateauing when the fine-tuned parameters reach 7.52% ($r = 512$).

Evaluation: In contrast to our hypothesis, the uniqueness of directly modifying both multi-head attention (ATTN) and feed-forward network (FFN) sub-layers outputs did not enable Adapter-H to become the most effective method for this task. However, this uniqueness may lead to Adapter-H's considerable performance boost when increasing the parameter budget from 3.76% to 7.52% (see Fig. 4.4), where both LoRA and Adapter-FFN showed a sign of saturation.

Summary: LoRA demonstrated the best parameter efficiency among all the PEFT methods on the code search task, followed by Adapter-FFN and Adapter-H. Although this indicates that directly modifying the ATTN sub-layers might optimize CodeBERT more effectively than other modification approaches, a performance saturation of LoRA was observed, signalling potential limitations.

4.2.4 RQ4: Enhancing Efficiency Through Cross-Group Methods Combination

Based on the previous experimental results, we highlight the following three findings:

- **LoRA's Superiority**: LoRA demonstrated the highest parameter efficiency compared to other PEFT methods;
- **Performance Saturation**: The performance of both LoRA and Adapter-FFN seemed to plateau when the proportion of fine-tuned parameters increased from 3.76% to 7.52% (Fig. 4.4);[6]
- **Adapter-H's Significant Boost**: Adapter-H showed a significant performance gain when the fine-tuned parameters were increased from 3.76% to 7.52% (Fig. 4.4).

Hypothesis: These observations lead us to test whether combining LoRA with Adapter-FFN might overcome the limitations when they were applied individually (Performance Saturation). We posit that the performance boost observed in Adapter-H might be attributed to its capability to directly modify the outputs of both multi-head self-attention (ATTN) and feed-forward network (FFN) sub-layers, a feature absent in sole implementations of LoRA and Adapter-FFN.

Design: To validate this hypothesis, we propose "loff", a combination of LoRA and Adapter-FFN (LoRA + Adapter-FFN). We aim to examine its efficacy within a 7.52% parameter budget and investigate whether modifying both ATTN and FFN sub-layers outputs through LoRA and Adapter-FNN can emulate Adapter-H's performance surge, potentially breaking LoRA's performance stagnation at the 7.52% parameter budget. Based on prior experimental results, we deploy "loff" with the following configurations. Overview architectures of them are available in App. 7.3:

- **LOFF (Fig. 7.10)**: Equal budget allocation (3.76% each) for LoRA and Adapter-FFN.
- **loFF (Fig. 7.11)**: A smaller 1.88% budget for LoRA, due to its robust performance with a limited parameter quota,[7] with the remaining budget allocated to Adapter-FFN.

[6]The 7.52% parameter budget stems from the largest bottleneck dimension we utilized for Adapter modules (512), constrained by CodeBERT's 768 hidden sizes. With this bottleneck size, Adapter-FFN/ATTN methods fine-tune 7.52% of the total parameters.

[7]LoRA achieved comparable results across the 1.88% (27.61 Test MRR) to 7.52% (28.92 Test MRR) tunable parameters range, motivating us to test if a minimal allocation to LoRA suffices.

- **LOff (Fig. 7.12)**: In contrast, a 1.88% allocation to Adapter-FFN with the remainder designated to LoRA,[8] representing a more LoRA-centric approach.

Model	Note	#Parameters	Performance	
			Valid MRR (%)	Test MRR (%)
Baseline	Full Fine-Tuning	125M(100%)	**38.33**$_{\pm 0.09}$	**31.66**$_{\pm 0.12}$
loFF (1.88% + 5.64%)	$r_{lo} = 64$, $r_{ff} = 384$	943K(7.52%)	**39.19**$_{\pm 0.55}$	**32.59**$_{\pm 0.53}$
LOFF (3.76% + 3.76%)	$r_{lo} = 128$, $r_{ff} = 256$	943K(7.52%)	37.98$_{\pm 0.29}$	31.39$_{\pm 0.24}$
LOff (5.64% + 1.88%)	$r_{lo} = 192$, $r_{ff} = 128$	943K(7.52%)	38.47$_{\pm 0.20}$	32.15$_{\pm 0.24}$
Average	Average of "loff"	943K(7.52%)	38.55	32.04

Table 4.6: The results for code search when combining LoRA (denoted lo/LO) and Adapter-FFN (ff/FF). r_{lo}, r_{ff} indicate the bottleneck dimensions of LoRA and Adapter-FFN respectively. Models named "loff" are our proposed combinations. The uppercase letters indicate which method has a larger parameter allocation, with the parameter quota shown. We conducted experiments **three** times with different seeds for all the above fine-tuning methods.

Results: Table 4.6 shows the performance for the different configurations of the combined method "loff". When compared to the baseline, all the configurations of "loff" demonstrated promising results. Notably, the loFF configuration achieved the highest performance, successfully overcoming the plateau of LoRA and even surpassing the full fine-tuning model while fine-tuning only 7.52% of model size parameters.

Evaluation: Observations suggest LoRA's performance plateau arises from its sole focus on adjusting the ATTN outputs. As attention mechanisms primarily capture low-level pairwise relationships between input tokens (largely learned during pre-training), over-tuning them may offer diminishing returns. This aligns with Hu *et al.*'s findings [22], emphasizing that LoRA only emphasizes task-specific pre-trained dimensions. Our attention analysis further supports this, showing similar attention results across LoRA models with varying bottleneck dimensions (detailed in App. 7.1). Therefore, instead of maximizing attention tuning, Adapter-FFN may help further refine LoRA's adapted patterns, as discussed next.

Adapter-FFN may help by aggregating the low-level patterns LoRA amplifies in the feed-forward network (FFN) sub-layers. These layers aggregate and refine LoRA's

[8] Previous results (RQ3) imply that directly modifying ATTN sub-layers outputs with LoRA might be more effective for this task than modifying FFN sub-layers.

amplified patterns into high-level semantic features [13], essential for tasks like code search that require understanding the semantics of inputs. Without tuning FFN sub-layers, models might fail to optimally transform the patterns into the joint embedding space (Fig. 4.1). This is supported by our experiments directly using pre-trained CodeBERT for code search, which yielded a 0.30% validation MRR.

However, relying solely on Adapter-FFN has drawbacks as well. The *[CLS]* token, which should represent the entire sequence, was not explicitly designed for this role during CodeBERT's pre-training.[9] Thus, while Adapter-FFN aggregates semantic patterns, the *[CLS]* token might fail to encompass sentence-level information without the fine-tuning of sentence-level tasks. This synergy between LoRA's low-level pattern extraction and Adapter-FFN's high-level refinement could lead to the superior performance of the combined method, loff.

Among the configurations, loFF performed best for the code search task, probably because minimal LoRA tuning adequately captures task-specific patterns with others learned from pre-training. Meanwhile, allocating more parameter budget to Adapter-FFN enables aggregating patterns into task-specific semantic representations that are especially meaningful for this task. Interestingly, this observation stands in contrast to our initiation for LOff drawn from LoRA's superiority, indicating the priority of adapting FFN sub-layers over ATTN sub-layers for this task. This is also indicated by our attention analysis, which showed that tuning FFN sub-layers has a slightly greater impact on capturing semantic patterns than changing the bottleneck dimensions of LoRA (detailed in App. 7.1).

However, we have to say that while our interpretations draw from previous studies and our experiments, they remain speculative. The intricate interplay between attention and FFN sub-layers and their true influence requires deeper investigation.

Summary: Combining LoRA and Adapter-FFN did further enhance parameter efficiency compared to the single PEFT methods when fine-tuning CodeBERT on the code search task. Among the configurations tested under a 7.52% parameter budget, loFF emerged as the most efficient approach, suggesting that tuning the feed-forward network (FFN) sub-layers may be a priority for this task.

[9] As we introduced in Section 4.1.1, CodeBERT's pre-training objectives aim to predict tokens. This means *[CLS]* knows how to capture contextual information from other tokens but may not know how to specifically represent an entire input sentence, which needs to be "invoked" with fine-tuning.

Chapter 5

Discussion

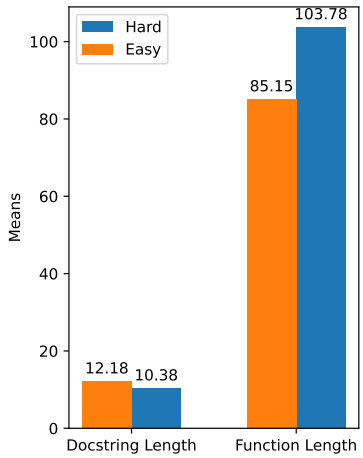
5.1 Qualitative Analysis

We conducted a qualitative analysis to examine CodeBERT's performance when fine-tuned with different methods on the code search task. We studied "easy" examples correctly predicted by Prefix-tuning-256,[1] LoRA-256,[2] full fine-tuning,[3] and loFF.[4] We further explored "hard" examples incorrectly predicted by all methods. Additionally, we inspected examples where loFF succeeded but LoRA-256 failed, aiming to understand loFF's performance gains. More statistics of these examples are available in App. 7.5.

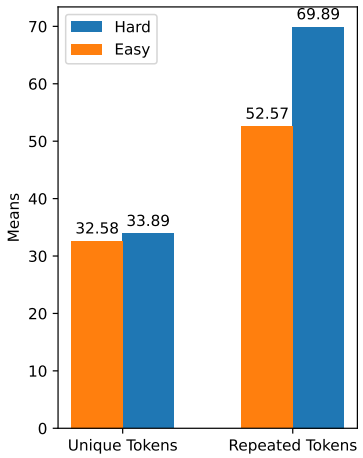
5.1.1 Easy and Hard Examples

In the code search task, the objective is to evaluate the semantic similarity between a given query docstring[5] and a target code snippet, which is represented as a sequence of function tokens. We first calculated the statistical attributes of these inputs. This involved measuring their average length and complexity—the number of repeated and unique tokens within the functions. These characteristics are illustrated in Fig. 5.1.

[1]The model with the worst performance on this task.
[2]The model faces a performance bottleneck that is resolved by loFF.
[3]Our baseline model.
[4]The model with the best performance on this task.
[5]In the dataset we used, it is the first paragraph of docstrings of a function.



(a) Mean sequence length of queries and code snippets.



(b) Mean complexity of code snippets.

Figure 5.1: Comparison of the statistical attributes of queries (docstring) sequences and target code snippets (function) sequences of easy and hard examples.

Docstring Length: The hard examples have shorter docstring lengths on average. This suggests that function complexity might not solely depend on docstring length but also on other factors, such as the specificity of the language used. The average query length for hard examples is around 9. This implies that extra tokens introduced by Prefix-tuning, especially with a length of 256, could significantly disrupt the semantic meaning of the input query. This disruption might be further compounded if these additional tokens are initialized with word tokens that are uncorrelated to the function's meaning, probably leading to their poor performance (Fig. 4.3).

Moreover, shorter docstrings might lack the sufficient description needed for the model to fully grasp the function's objective, such as 261446 (idx) in Table 5.1. This can make matching these terse docstrings with their respective functions more challenging. A promising future direction would be to rewrite such docstrings more descriptively. This could potentially improve the ranking of functions with vague descriptions.

Function Length: Hard examples have significantly longer function lengths on average. This aligns with the common intuition that functions with more tokens (code segments) are harder to understand and predict.

Unique Tokens And Repeated Tokens: Hard examples contain more unique tokens and a significantly higher number of repeated tokens on average, which indicates a higher level of code complexity and a tendency toward code repetition or redundancy.

Recall that function and variable names are normalized to placeholders like "Func"

and "arg_0" in the test set. Given this and our previous observations, it is reasonable to understand why hard examples are challenging: (1) Long function sequences make it difficult for the fixed-dimension *[CLS]* token to fully capture long-term dependencies[6] and the overall function semantics; (2) Unique tokens might employ libraries or operations that are not covered in the training data, complicating predictions; (3) Repeated placeholders with minimal semantic information can dilute sequence semantics, as the *[CLS]* token might pay undue[7] attention to these repeated tokens.

Examples. Next, we examine specific examples to better understand the impacts of function length, function complexity, and token repetition. Table 5.1 presents the docstrings (untokenized queries) of some samples from both easy and hard examples.

Example Type	Docstring
Easy (idx = 262144)	Returns the jQuery DataTables CSS file to version number.
Easy (idx = 262165)	Check if the given information is a URL.
Easy (idx = 262185)	Count the number of BEL relations generated.
Hard (idx = 261425)	This rolls up the feature functions above and returns a single dict.
Hard (idx = 261432)	r"""Levinson-Durbin recursion.
Hard (idx = 261446)	Handle the 'files = !ls' syntax.

Table 5.1: Docstrings (untokenized query sequence) of some easy and hard examples. "idx" is the specific id of this function in the dataset.

It is evident from the provided examples that the docstrings of easy functions are specific, employing detailed and clear language to describe the function's intent. Conversely, while the docstrings of hard examples are concise as well, they may require additional context for comprehension. For instance, the ambiguous docstring for idx 261425 lacks clarity about what the "functions above" refers to. Another challenging case is idx 261432, which carries the docstring "r"""Levinson-Durbin recursion.". While the *[CLS]* token might recognize this theoretical concept from its pre-training data,[8] its respective function includes around 60 lines of codes with 9 normalized variables. Matching a succinct 3-word concept with nearly 60 lines of abstract codes

[6]The diminishing statistical dependence of two tokens in a sequence as their distance increases is a common issue when processing long input sequences with neural networks.

[7]The attention mechanism tends to assign more weights to repeated tokens, as it performs a weighted sum of all the tokens in the sequence.

[8]CodeBERT uses the RoBERTa as its base model, which is pre-trained on sources like Wikipedia that probably includes this theorem.

(available in App. 7.7), though truncated,[9] is inherently challenging. This reinforces the earlier point: vague docstrings lack enough contexts, and even when the *[CLS]* token grasps the docstring's content, the lengthy and normalized code might lead to a semantic discrepancy between the query and its implementation.

5.1.2 Fixed Examples

In this section, we inspected some examples predicted incorrectly by LoRA-256 but fixed by our proposed method, loFF. We aim to understand how the combined method helps LoRA-256 overcome its performance plateau. We conjecture that our proposed method, loFF, may help capture long-term dependencies that might be missed by solely modifying the ATTN (multi-head self-attention) sub-layers. This might be especially beneficial for longer functions, like the ones found in hard examples. Additionally, we believe it could assist the model in recognizing functions with a large number of unique and repeated tokens.

Results:

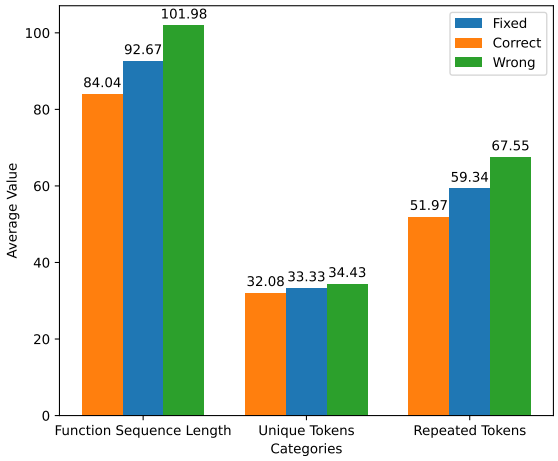


Figure 5.2: Statistical attributes of function examples: those predicted correctly or incorrectly by both LoRA-256 and loFF, and those predicted incorrectly by LoRA-256 but fixed by loFF.

Fig. 5.2 shows that the fixed examples exhibit moderately longer function lengths and more repetitions compared to those predicted correctly by both models. One **sample** of fixed examples is available in App. 7.8. This function contains 256 tokens in total,

[9] As highlighted in Section 2.3.2, transformer-based models have a maximum input sequence length, set to 256 in our experiments.

with 8 repeated normalized variable names, underlining the pattern of repetitions in long functions. Interestingly, the number of unique tokens in the fixed examples is similar to that in the correct predictions of both models. This observation suggests that the performance gain of loFF is primarily associated with its ability to handle longer sequences and repetitions, rather than recognizing unique tokens. This contradicts our initial hypothesis. In summary, this pattern indicates that loFF may excel at capturing long-term dependencies and effectively balancing the attention weights of repetitions, rather than enhancing the recognition of unique tokens.

Summary: loFF handles long functions with multiple repeated tokens more effectively than standalone PEFT methods do, whereas unique tokens in the functions may have less impact on their predictive difficulty. By combining Adapter-FFN with LoRA, the model might be better at capturing long-term dependencies and minimizing undue attention to irrelevant repetitions.

5.2 Threats To Validity

5.2.1 External Validity

We conducted experiments exclusively on the CodeXGLUE benchmark, which contains only the Python programming language for the code search task. This limitation raises questions about the effectiveness of the PEFT methods we deployed, as well as our proposed method, loFF, when applied to datasets that include other programming languages or other code intelligence tasks. Furthermore, due to our limited computational resources, our experiments implemented only the CodeBERT model. Other pre-trained code models, such as GraphCodeBERT [15] and UniXcoder [14], are also competent in the code search task but were not studied in this work. Testing loFF and the PEFT methods we employed on other datasets, programming languages, tasks, and models will provide a more comprehensive evaluation of their generalizability.

5.2.2 Internal Validity

Parameter Efficiency Calculation. Most prior studies omit the bias terms[10] introduced by the Adapter module's projection layers when comparing the parameter efficiency of

[10]Each Adapter module has two bias terms for down projection and up projection, adding r (bottleneck dimension) and d (hidden states dimension) parameters.

different PEFT methods. For simplicity, we followed this common approach, so the percentage of trainable parameters reported differs slightly from precise values.

Hyperparameter Setting. (1) Default Hyperparameter Setting: To ensure a consistent comparison across all the fine-tuning methods we implemented, we adopted a default hyperparameter setting. This configuration was optimized specifically for the full fine-tuning (FFT) of CodeBERT on the validation set. However, this setting might not be ideal for every PEFT method we employed. For example, the learning rate we selected, while optimal for FFT, may be less suitable for PEFT methods since they update fewer parameters during training. Consequently, some experimental results might be influenced by this unified hyperparameter setting. Furthermore, some previous studies [18, 2] fine-tuned the hyperparameters separately for each PEFT method. With such an approach, each method could potentially reach its full performance potential, which might lead to different performance rankings than what we have obtained. (2) Scaling Factor Of LoRA: We examined the influence of the scaling factor, α, on LoRA by evaluating its performance across different α values on the validation set (results depicted in Fig. 5.3). The result shows that larger α values improve LoRA's performance, However, an α value larger than 1 may unfairly benefit LoRA over methods without a scaling hyperparameter, as it acts similarly to amplifying the learning rate. While this proves the benefit of adjusting α for LoRA, it also suggests our comparisons may be biased in LoRA's favour under our shared hyperparameter setting.

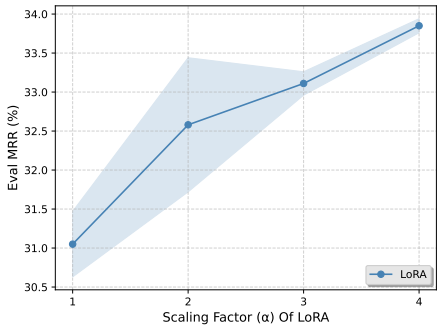


Figure 5.3: Performance variance of different scaling factors for LoRA on the code search task. Each point signifies the mean MRR value on the validation set, and the shaded areas denote the standard deviation.

Other Limitations. Other threats include small training and test batch sizes used, the randomness in model initialization and small running times (**ideally** at least 3 runs for all experiments). All of them could potentially affect our experimental findings.

Chapter 6

Conclusions

In this project, we conducted an exploration of parameter-efficient fine-tuning (PEFT) methods, specifically Prefix-tuning, LoRA, and Adapter-tuning, applied to the CodeBERT model for the code search task. We aimed to examine the efficacy and parameter efficiency of these methods when fine-tuning large language models (LLMs) pre-trained with programming languages for code intelligence tasks.

Our experiments indicated that except for LoRA, single PEFT methods were ineffective for this task, suggesting the necessity of evaluating them outside of natural language tasks. However, our "loFF" approach, combining LoRA with Adapter-FFN prioritized, outperformed full fine-tuning with just 7.52% of the model's parameters tuned. This not only highlights PEFT methods' potential in code intelligence but also indicates a salient takeaway: strategically combining PEFT methods as a potential remedy when standalone approaches perform poorly.

Our attention analysis (App. 7.1) revealed that "loFF" strengthens the model's ability to capture semantic features. Moreover, our qualitative analysis highlighted loFF's enhanced ability in processing lengthy and repetitive function sequences.

Future work includes evaluating the generalizability of the PEFT methods we deployed for more models and tasks. Additionally, investigating hyperparameter optimizations specific to each PEFT method, exploring refined PEFT approaches like the parallel/scaling Adapter [2], and designing dynamic parameter allocation strategies for "loff" tailored to different tasks are interesting directions.

In summary, this project offers encouraging indications that, with selective combination, LLMs can be parameter-efficiently fine-tuned on code intelligence tasks, potentially paving the way for more resource-efficient applications in the field of code intelligence.

Bibliography

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[3] Ankur Bapna, Naveen Arivazhagan, and Orhan Firat. Simple, scalable adaptation for neural machine translation. *arXiv preprint arXiv:1909.08478*, 2019.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[9] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.

[10] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing, 2021.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[13] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. *arXiv preprint arXiv:2012.14913*, 2020.

[14] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unix-coder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[16] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*, 2021.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[18] Ruidan He, Linlin Liu, Hai Ye, Qingyu Tan, Bosheng Ding, Liying Cheng, Jia-Wei Low, Lidong Bing, and Luo Si. On the effectiveness of adapter-based tuning for pretrained language model adaptation. *arXiv preprint arXiv:2106.03164*, 2021.

[19] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[20] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[21] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

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

[23] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[24] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR, 2021.

[25] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[26] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[27] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018.

[28] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[29] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[30] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *arXiv preprint arXiv:2103.10385*, 2021.

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[32] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[33] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[34] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen tau Yih, and Madian Khabsa. Unipelt: A unified framework for parameter-efficient language model tuning, 2022.

[35] OpenAI. Gpt-4 technical report, 2023.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[37] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*, 2020.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[39] Shauli Ravfogel, Elad Ben-Zaken, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked languagemodels. *arXiv preprint arXiv:2106.10199*, 8, 2021.

[40] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. Adapterdrop: On the efficiency of adapters in transformers. *arXiv preprint arXiv:2010.11918*, 2020.

[41] Iman Saberi, Fatemeh Fard, and Fuxiang Chen. Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering. *arXiv preprint arXiv:2307.08540*, 2023.

[42] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. *arXiv preprint arXiv:2304.05216*, 2023.

[43] Asa Cooper Stickland and Iain Murray. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. In *International Conference on Machine Learning*, pages 5986–5995. PMLR, 2019.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[45] Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics.

[46] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. One adapter for all programming languages? adapter tuning for code search and summarization. *arXiv preprint arXiv:2303.15822*, 2023.

[47] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*, 2023.

[48] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[49] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *arXiv preprint arXiv:1805.12471*, 2018.

[50] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[51] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

Chapter 7

First appendix

7.1 Attention Analysis

Motivation: In Section 4.2.4, we observed that loFF can overcome the plateau faced by LoRA-256, which is the most effective PEFT method when applied individually for the code search task. Moreover, loFF even outperformed the full fine-tuning model's performance. Based on our results and analysis, we argued that:

- (1) The plateau faced by LoRA may be attributed to the fact that the attention results do not vary considerably when increasing the tunable parameters (*i.e.*, bottleneck dimension) for LoRA.
- (2) loFF's superior performance compared to LOff[1] suggests that modifying the feed-forward network (FFN) sub-layers might be more crucial for this task. This is likely because the FFN helps capture high-level (semantic) patterns in the last encoder layers.

Design: To scrutinize our hypothesis, we conducted an attention analysis. We selected an example from the test set that posed challenges for both LoRA-64 and LoRA-256 but is correctly predicted by loFF. The example chosen had the query: **"Return relative path if path is local."**, paired with the following input function sequence:

```
def Func(arg_0=None, arg_1=None):
    if path_is_remote(arg_0) or not os.path.isabs(arg_0):\n
```

[1]LOff predominantly modifies attention, whereas loFF leans more towards modifying the feed-forward network (FFN). Refer to App. 7.3 for a visual comparison.

```
        return arg_0\n
    else:\n
        return os.path.relpath(arg_0, arg_1)
```

Then, we carried out the following analysis for each of our arguments, respectively.

7.1.1 Attention Weights Variation Of Different LoRA

Experiments and Results: To understand the attention behaviour across different LoRA configurations, we analysed the multi-head self-attention (ATTN) outputs for LoRA-64 and LoRA-256 using the example introduced earlier. Specifically, we focused on the attention weights of the *[CLS]* token, which represents the entire input sequence in our task, across different attention heads in the last encoder layer. We chose this layer because it captures richer semantic information, a fact we will validate in subsequent analysis.

Fig. 7.1 provides a comparison of the attention results between LoRA-64 and LoRA-256 for the query sequence. Note that here the attention results mean the attention scores (weights) after taking softmax[2] of the scaled dot-product of input queries and keys (query token sequence here).

[2]To make the obtained weights a distribution. The results are also called logits.

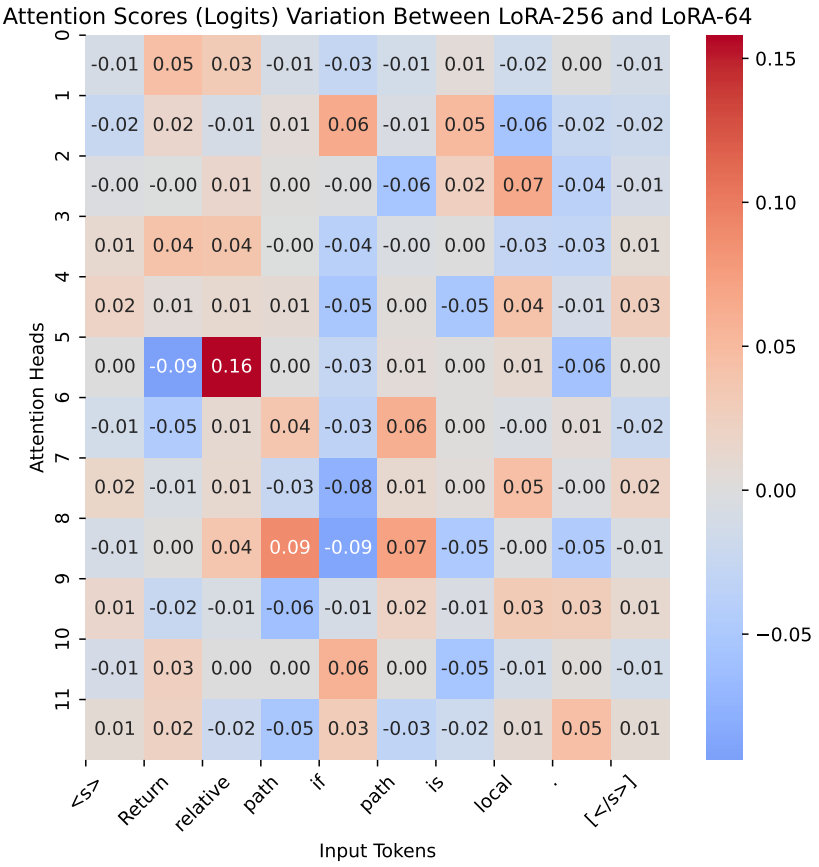


Figure 7.1: Attention score variations between LoRA-64 and LoRA-256 are depicted. The X-axis represents tokens in the input query sequence, while the Y-axis denotes the 12 different attention heads. Each cell in the matrix illustrates the difference in attention weights between the two LoRA configurations for a specific token-head combination. $< s >$ and $< /s >$ symbolize the *[CLS]* and *[SEP]* tokens, respectively. The (i, j) value in this matrix represents the attention weight of the j-th attention head from the *[CLS]* token to the i-th token in the input query. For instance, the value in the right bottom corner of this matrix shows a difference in attention scores of 0.01 between the two LoRA models, specifically from the *[CLS]* ($< s >$) token to the *[SEP]* ($< /s >$) token in the last attention head of the last encoder layer.

From the figure, we observe that the attention weights exhibit minor variations between the two configurations. The computed maximum absolute difference is 0.15, and the mean absolute difference is 9.31e-10. These findings suggest that there may be limited improvement in pairwise pattern capture as the bottleneck dimension increases. This provides a potential explanation for the performance plateau observed in LoRA.

Similarly, Fig. 7.2 displays the differences in attention weights for the function sequence, with an even smaller mean absolute difference of 3.72e-10.

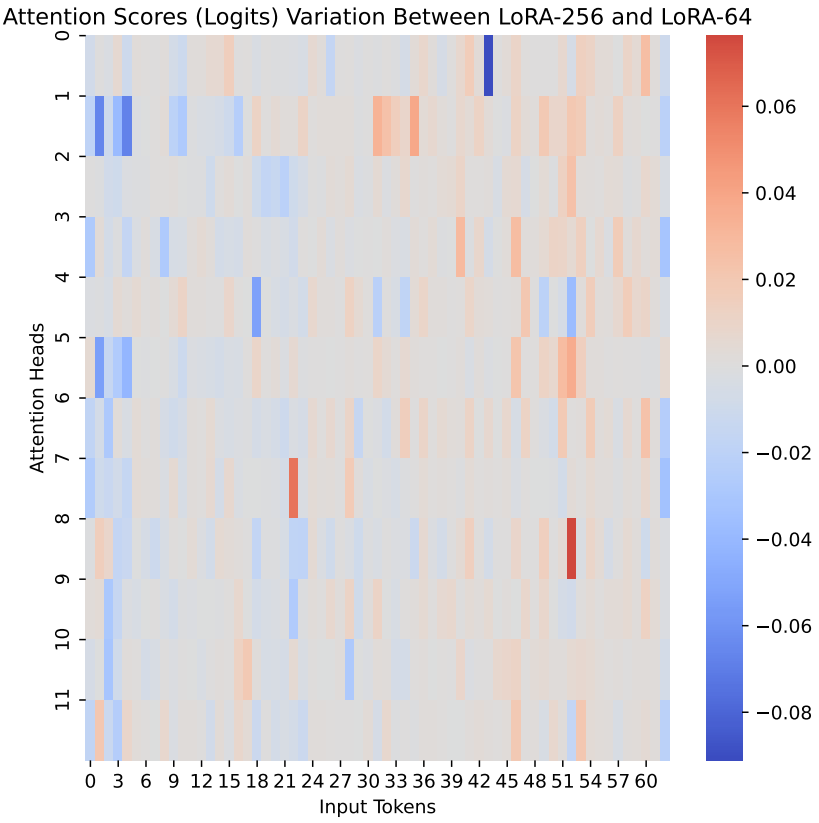


Figure 7.2: Variations in attention scores for the selected example's input function sequence between LoRA-64 and LoRA-256. The X-axis represents tokens in the input (function) sequence, while the Y-axis denotes different attention heads. The value at (i, j) in this matrix represents the attention weight from the j-th attention head, specifically from the *[CLS]* token to the i-th token in the input function. For instance, the value in the right bottom corner of this matrix indicates the difference in attention scores between the two models from the *[CLS]* token to the *[SEP]* token (the last token) in the final attention head of the last encoder layer.

Summary: The minimal variations in attention weights between LoRA-64 and LoRA-256 indicate that increasing the bottleneck dimension offers limited benefits in terms of capturing pairwise patterns, which may lead to LoRA's performance saturation.

7.1.2 The Ability to Capture Semantics

Design: For the query **"Return relative path if path is local"**, we identified certain tokens—specifically "**path**" and "**remote**"—that seem to carry significant semantic information. This observation is based on their relevance to the query's intent, as deduced from our subjective analysis. Our examination focused on the pairwise relationships highlighted by the attention sub-layers in both LoRA-64 and loFF (a hybrid of LoRA-64 and Adapter-FNN-384, as illustrated in Fig. 7.11). We aimed to investigate if the modifications in the feed-forward network (FFN) sub-layers, introduced by Adapter-FFN, could potentially improve the model's ability to identify more complex semantic patterns.

Experiments and Results. *Semantics Captured Lastly:* We employed Bertviz [45] to visualize attention scores, providing insights into the pairwise patterns recognized by loFF's attention layers. For visual clarity, we present only segments of the function sequence. Fig. 7.3 shows the attention scores from the 10th encoder layer (counting from 0). From this figure, we found that tokens such as "remote" and "return", which contain significant semantic information, are not emphasized. However, in the final encoder layer (Fig. 7.4), these tokens are prominently highlighted, which are indicated by lines connecting the *[CLS]* token ($< s >$) to them or by the **depth of token colours**. This observation aligns with previous research, suggesting that high-level semantic patterns are mainly captured in the last encoder layers [42, 13].

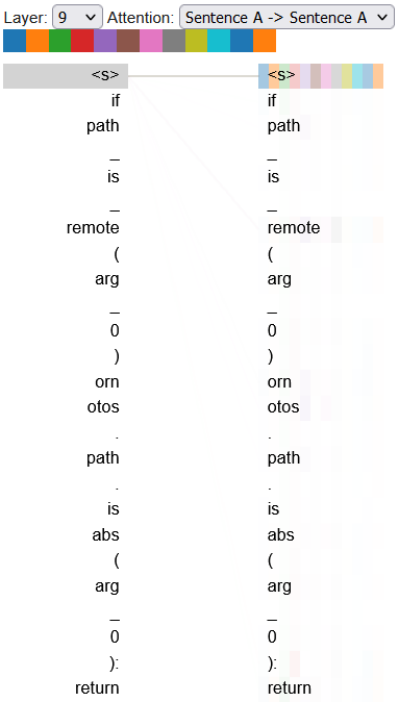


Figure 7.3: Attention scores from the 10th Encoder Layer.

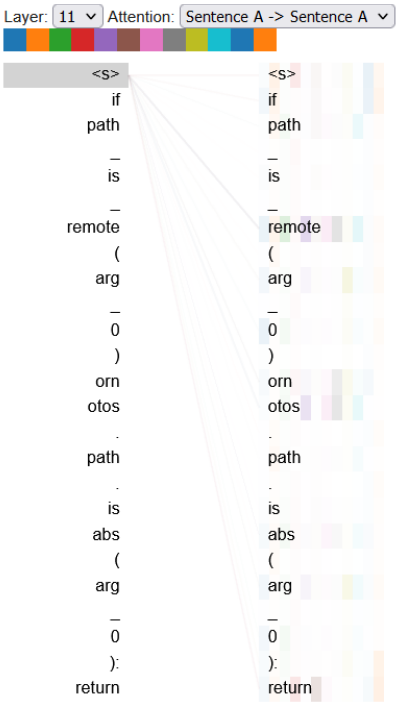


Figure 7.4: Attention scores from the last Encoder Layer.

Ablation Study For loFF-Combining LoRA-64 And Adapter-FFN: Next, we examined the attention scores between LoRA-64 and loFF. Fig. 7.5 shows the attention weights from the last encoder layer's 1st head of LoRA-64, whereas Fig. 7.6 does the same for loFF. Surprisingly, although LoRA-64's prediction was inaccurate for this instance, it recognized the semantic importance of tokens such as "remote" and "return". However, in comparison to loFF, LoRA-64 allocated less attention to these tokens. This is indicated by the thinner lines (**or lighter token colours**) connecting the *[CLS]* ($< s >$) token to them. This observation implies that modifying FFN sub-layer outputs by incorporating Adapter-FFN might guide the *[CLS]* token's attention towards tokens containing semantic information.

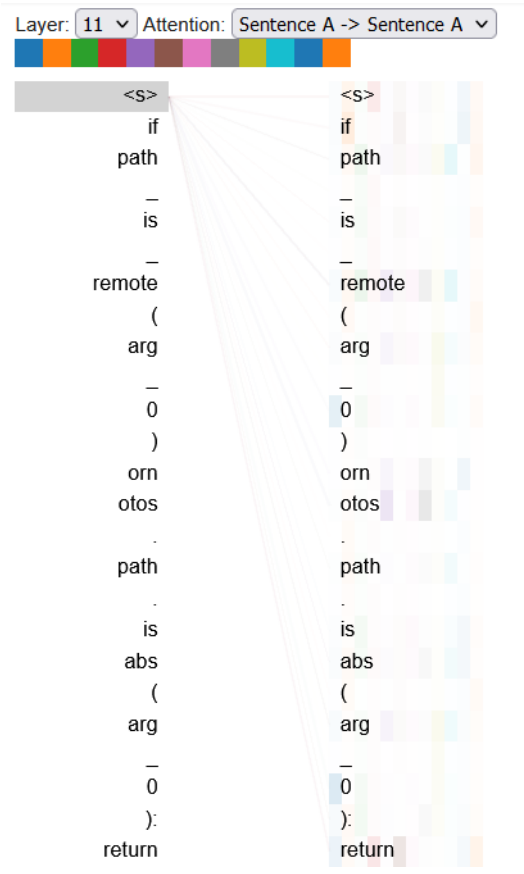


Figure 7.5: Lora-64

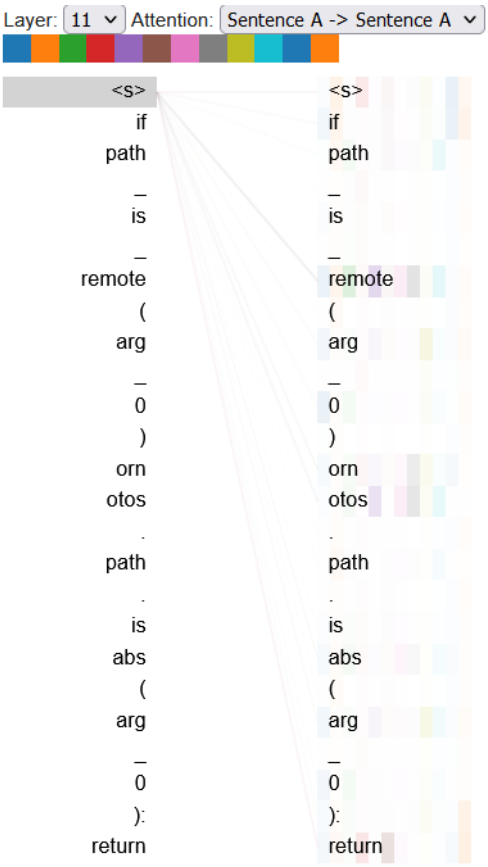


Figure 7.6: loFF

Compare loFF with LoRA-256: To ensure the observed distinctions were not a consequence of loFF having more trainable parameters than LoRA-64, we drew a comparison between the attention scores of loFF and LoRA-256. Thus, both methods fine-tuned an equal 7.52% of the model's parameters. Fig. 7.7 illustrated the comparison, which corroborates our earlier observations: the integration of Adapter-FFN seemingly enhances the *[CLS]* token's ability to capture sequence semantics.

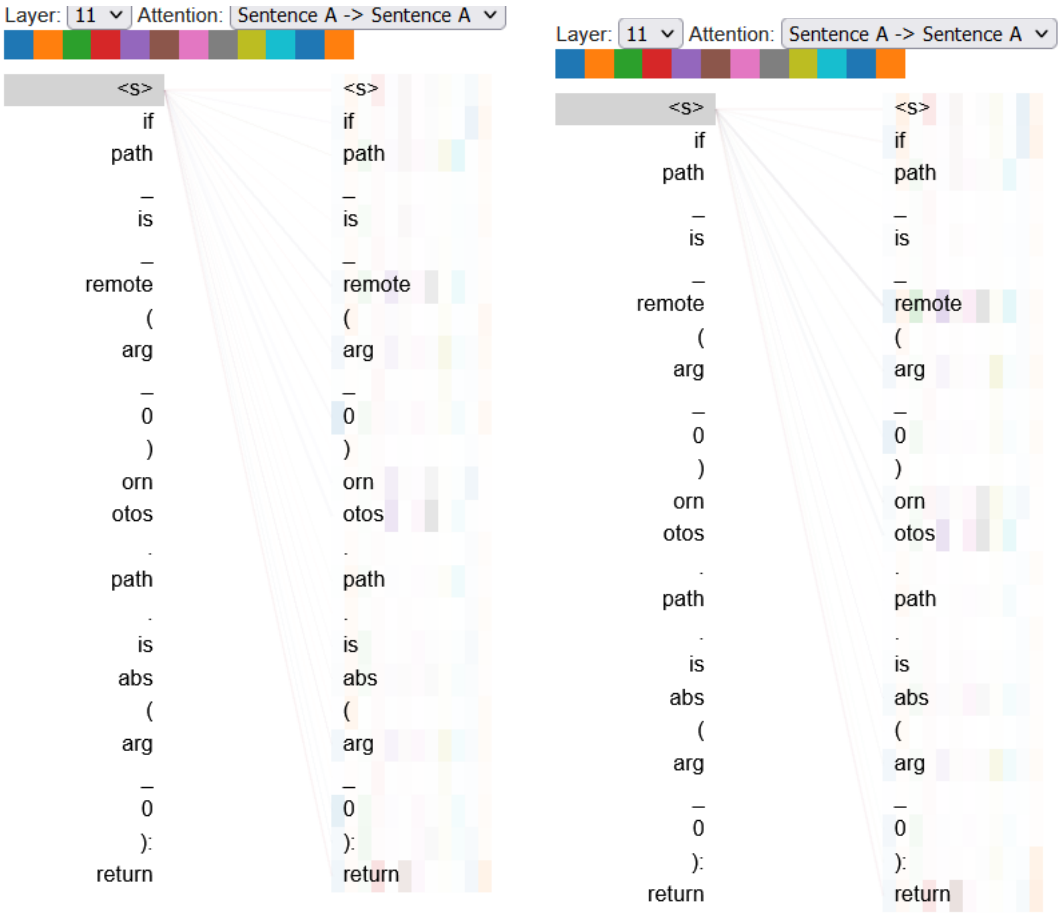


Figure 7.7: Attention score comparison from the 1st head of the last encoder layer between Lora-256 and loFF. Line thickness denotes attention score values. LoRA-256 results are displayed on the left, and loFF's on the right. Although sharing the same 7.52% tunable parameter budget, loFF shows slightly larger attention weights (thicker lines or **deeper token colours**) on key semantic tokens like "remote" and "return".

However, this attention weight enhancement is subtle. The question then becomes: did this subtle increase in attention weights cause the final correct prediction, or was it the subsequent FFN sub-layers contribution? This conundrum even deepens when considering the interplay between ATTN and FFN sub-layers, given the encoder layers' stacked architecture: did this interplay actually make correct predictions? We suppose more rigorous and deeper exploration are needed to answer this question.

Summary: loFF pays more attention to semantic patterns captured in the last encoder layers over standalone PEFT methods. This enhancement may be attributed to the incorporation of Adapter-FFN to modify FFN sub-layers outputs.

7.1.3 A Further Step

Based on the results from the last section, we tried to explore the performance of adding modules only to the last encoder layers, as these are the places where semantic features are captured. Specifically, we deployed loFF's configuration, but only in layers 9-12th of CodeBERT (Fig. 7.8). The result was great compared to the standalone methods. With about 2.52% of the parameters fine-tuned, this setting achieved a test MRR of 29.75%. This was comparable to the highest test MRR we got from LoRA-256 (29.83%), while LoRA-256 updates 7.52% of the model's parameters. Table 7.1 summarizes the results.

Model	Parameter Budget(%)	Highest test MRR (%)
LoRA-256	7.52%	29.83%
loFF_9-12	**2.52%**	29.75%

Table 7.1: Results for using the loFF configuration on the last four encoder layers (9-12th), compared with LoRA-256. loFF_9-12 denotes that we only added modules to the 9-12 encoder layers and updated them during fine-tuning

This result may indicate that, for semantically-related tasks, updating the top layers might be more efficient. However, the results probably would not be comparable to those obtained from modifying all layers. There is a trade-off to consider, just like "partial fine-tuning" (Section 2.2).

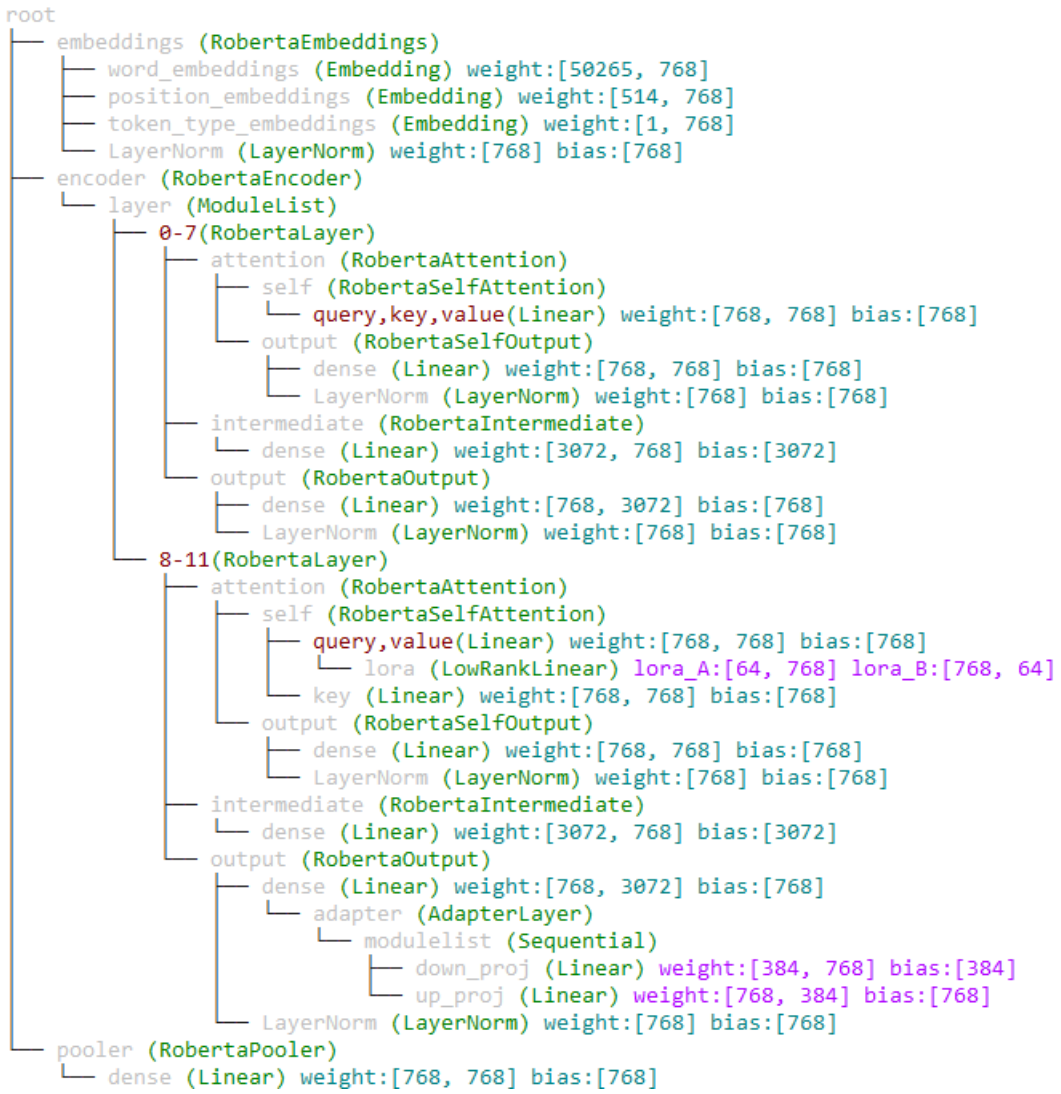


Figure 7.8: An overview of the model architecture of loFF_9-12. Compared with loFF (Fig. 7.11), we only add modules to the 9-12th encoder layers.

7.2 Training Curves of Prefix-256

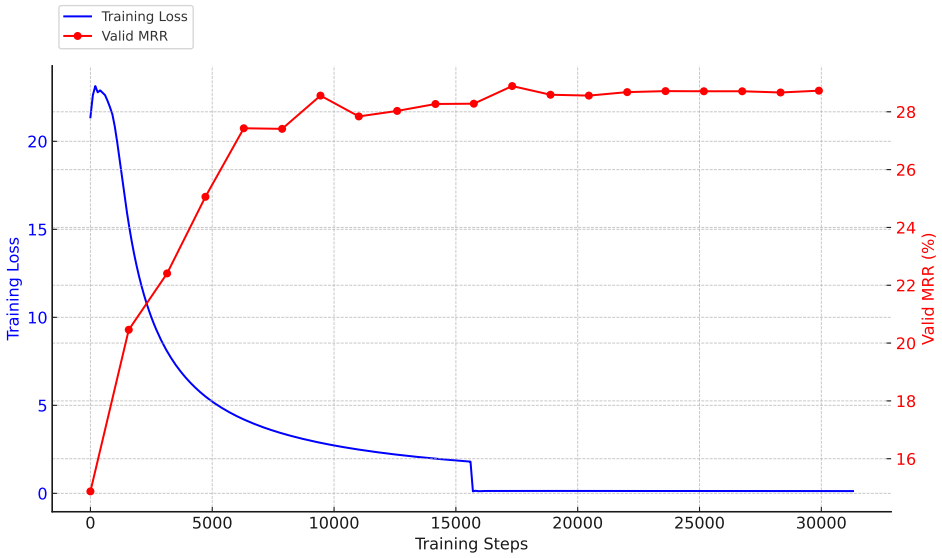


Figure 7.9: The training curves of deploying Prefix-256 for fine-tuning CodeBERT on the code search task. We trained CodeBERT for 2 epochs. Overall, the training loss consistently diminished during the first training epoch, which ended at around the 15,000 training step. As the model progressed into the second epoch, this reduction plateaued, indicating convergence. As for the performance on the validation set, we evaluated the model's performance 10 times for each training epoch. As can be seen, the validation MRR value gradually grew, eventually becoming saturated without showing a notable decline. This indicates that the observed poor performance of Prefix-256 is not a consequence of overfitting, a scenario typically signified by both a drop in training loss and validation performance. Through the qualitative analysis we conducted in Section 5.1.1, we argued that the excessive tokens brought by prefix vectors disturb the model and cause Prefix-256's poor performance.

7.3 “loff” Architecture Overview

7.3.1 LOFF

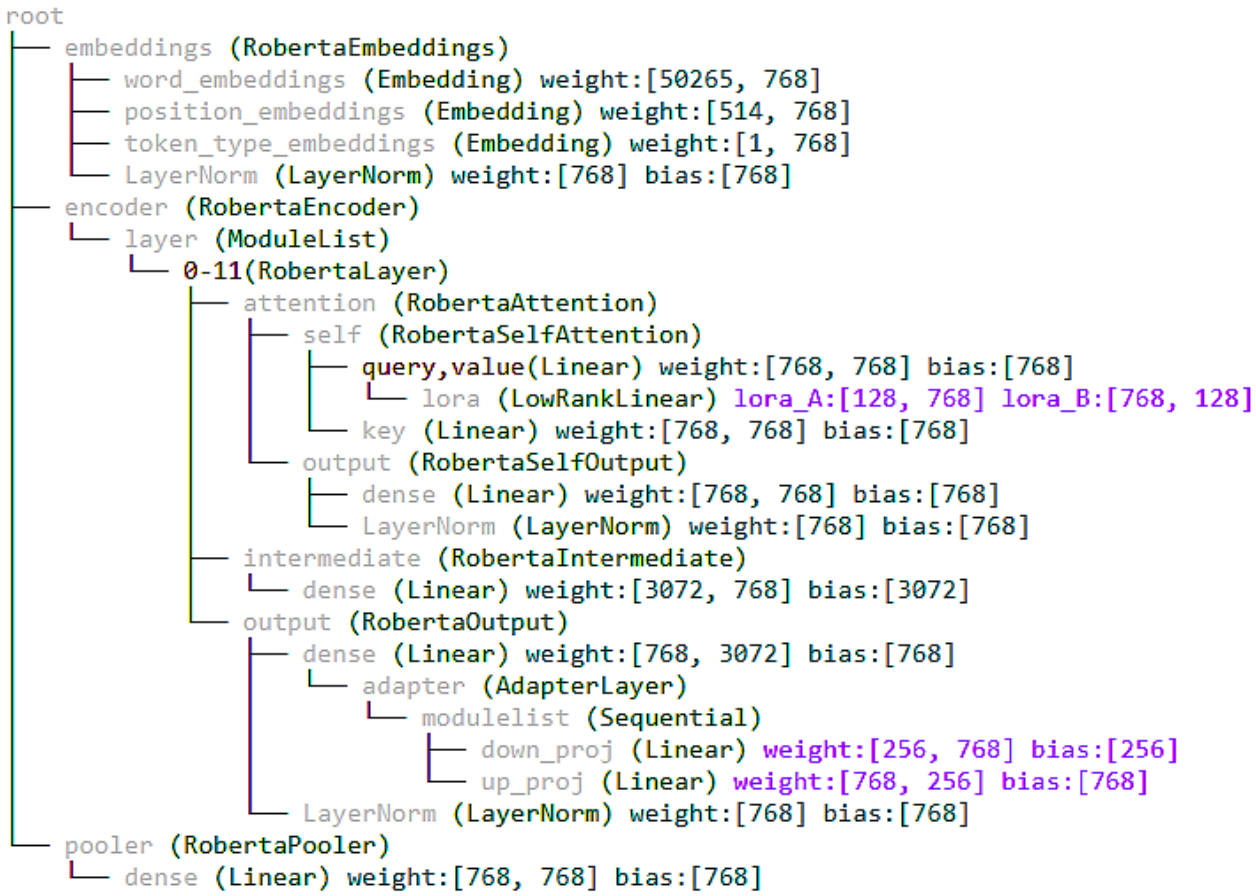


Figure 7.10: An overview of the model architecture of LOFF: LoRA introduces four matrices (to the query and value) while the Adapter introduces only two. Thus, the bottleneck dimension of LoRA (128) is half that of the Adapter modules (256) when they are allocated the same parameter budget (3.76%). The weights highlighted in purple are introduced by the PEFT methods and represent the only trainable parameters; all other weights are frozen during fine-tuning.

7.3.2 IoFF

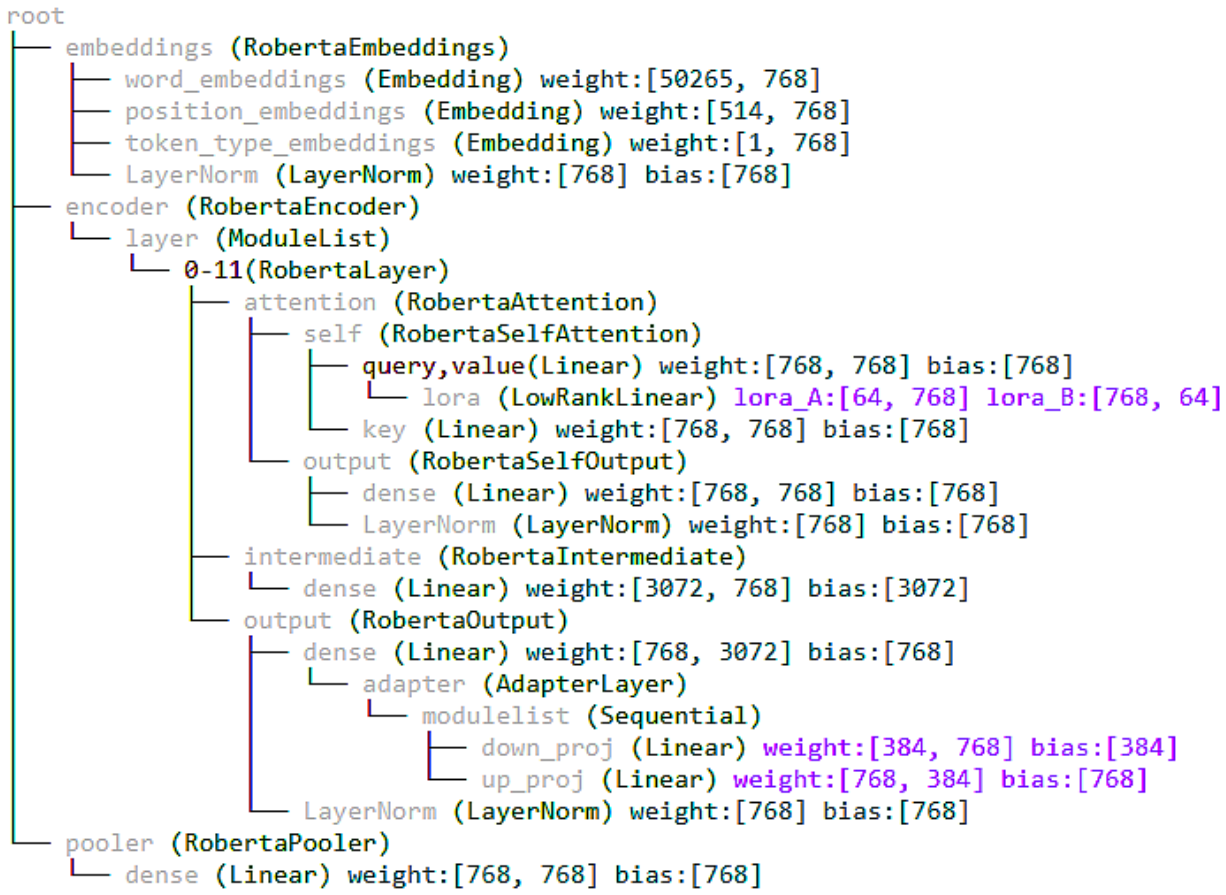


Figure 7.11: An overview of the model architecture of IoFF is presented. LoRA contributes an additional 1.88% to the model's parameters, while the Adapter modules account for 5.64% of the model's parameters. The weights highlighted in purple come from the PEFT methods. These are the only trainable parameters, with all other weights remaining frozen during fine-tuning.

7.3.3 LOff

Figure 7.12: An overview of the LOff model architecture is provided. LoRA contributes an additional 5.64% of trainable parameters to the model size, whereas the Adapter modules account for 1.88%. The weights highlighted in purple are introduced by the PEFT methods and represent the trainable parameters; all other weights remain frozen during fine-tuning.

7.4 Hyperparameters

7.4.1 General Setting of Fine-tuning Hyperparameters

Hyperparameter	Description	Value
max_token_length	Max input sequence length after tokenization	256
train_batch_size	Batch size per GPU for training	16
eval_batch_size	Batch size per GPU for evaluation	16
learning_rate	The initial learning rate for Adam	1e-5
adam_epsilon	Epsilon for Adam optimizer	1e-8
max_grad_norm	Max gradient norm	1.0
max_steps	Total number of training steps to perform	31478
save_steps	Save model for evaluation every X updating steps	3100
seed	Random seed for initialization	123456, 654321, 123
epoch	The number of training epoch	2
warmup_steps	Linear warmup over warmup_steps	3147

Table 7.2: Hyperparameters and their descriptions. Most of them follow the setting from CodeXGLUE [33] except for training batch size and learning rate due to limited resources.

7.4.2 Learning Rate

Learning Rate	Valid MRR (%)
1e-5	**38.33**$_{\pm 0.09}$
3e-5	35.06$_{\pm 0.07}$
4e-5	34.55$_{\pm 0.06}$

Table 7.3: Results for different learning rates on the validation set. We run three experiments with different seeds, and the mean values of them are reported with standard deviation. A higher MRR value indicates better performance.

7.5 Statistical Attributes of Examples

Here we show some statistical attributes of the examples analyzed.

Category	Correct Predictions	Incorrect Predictions
Prefix-256	2878	**16332**
LoRA-256	3893	15317
Full Fine-Tuning	4453	14757
loFF	**4684**	14526

Table 7.4: Performance of different fine-tuning methods on the test set. (Total: 19210 examples).

Category	Count
Easy Examples	1974
Hard Examples	4072
Both Correct	3221
Both Incorrect	13854
Fixed Examples	1463

Table 7.5: Easy/Hard examples denote those correctly/incorrectly predicted by all the models in Table 7.4. Both correct/incorrect denote the examples predicted correctly/incorrectly by both LoRA-256 and loFF. Fixed examples denote those predicted incorrectly by LoRA-256 but fixed by loFF.

We also counted the examples that were **correctly predicted by LoRA-256 but incorrectly predicted by loFF**. The statistics show that there are 672 examples in this category in total. As for the other statistical attributes, they almost all lie between the "Both Correct" and the "Fixed Examples" categories. This indicates that LoRA-256 demonstrates "improved" performance compared to the relatively easier examples, namely the "Both Correct" ones.

7.6 Tokenized Example

Here, we present a tokenized example from the CodeXGLUE dataset using CodeBERT. As illustrated in Fig. 7.13, both the query and function sequences are tokenized using byte-level Byte Pair Encoding [38], which breaks words into sub-words. Additionally, the function and variable names in the code snippets from the CodeXGLUE benchmark

are further normalized to serve as placeholders, such as “arg_1”. “_” usually denotes blank. “$< s >$” is the *[CLS]* token and “$< /s >$” is the [SEP] token to separate sentences in a BERT-type model.

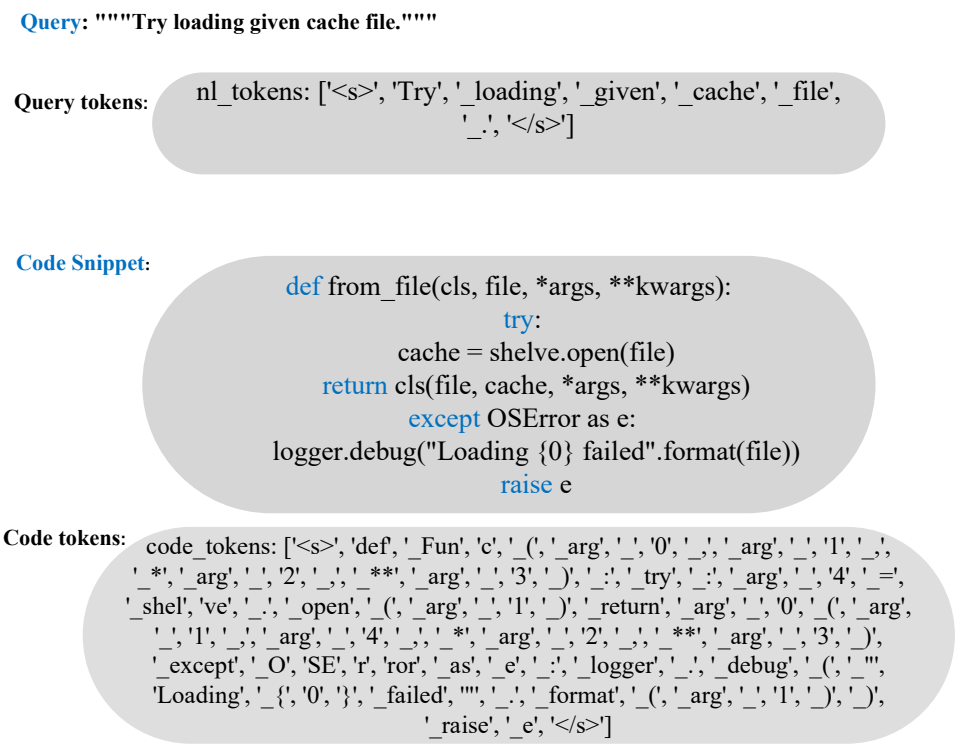


Figure 7.13: A tokenized example from the dataset we use.

7.7 A Hard Example

A normalized function example that is not predicted correctly by all the models deployed in this project. The query for this example is: “**r“““Levinson-Durbin recursion.**”.

```
def Func(arg0, arg1=None, arg2=False):
    if arg1 is None:
        arg1 = arg0_len
    else:
        assert arg1 <= arg0_len,
        'arg1 must be less than size of the input data'
        arg1 = arg1
```

```
    arg0_realdata = numpy.isrealobj(arg0)
    if arg0_realdata is True:
        arg3 = numpy.zeros(arg1, dtype=float)
        arg4 = numpy.zeros(arg1, dtype=float)
    else:
        arg3 = numpy.zeros(arg1, dtype=complex)
        arg4 = numpy.zeros(arg1, dtype=complex)

    arg5 = arg0_0

    for arg6 in range(0, arg1):
        arg7 = arg0_1[arg6]
        if arg6 == 0:
            arg8 = -arg7 / arg5
        else:
            for arg9 in range(0, arg6):
                arg7 = arg7 + arg3[arg9] * arg0_1[arg6-arg9-1]
            arg8 = -arg7 / arg5
        if arg0_realdata:
            arg5 = arg5 * (1. - arg8**2.)
        else:
            arg5 = arg5 * (1. - (arg8.real**2+arg8.imag**2))
        if arg5 <= 0 and arg2==False:
            raise ValueError("singular matrix")
        arg3[arg6] = arg8
        arg4[arg6] = arg8 # save reflection coeff at each step
        if arg6 == 0:
            continue

        arg6_half = (arg6+1)//2
        if arg0_realdata is True:
            for arg9 in range(0, arg6_half):
                arg9_reverse = arg6-arg9-1
                arg7 = arg3[arg9]
                arg3[arg9] = arg7 + arg8 * arg3[arg9_reverse]
```

```
                if arg9 != arg9_reverse:
                    arg3[arg9_reverse] += arg8*arg7
        else:
            for arg9 in range(0, arg6_half):
                arg9_reverse = arg6-arg9-1
                arg7 = arg3[arg9]
                arg3[arg9] = arg7 + arg8 * arg3[arg9_reverse].conjugate()
                if arg9 != arg9_reverse:
                    arg3[arg9_reverse] = arg3[arg9_reverse] + \
                    arg8 * arg7.conjugate()

    return arg3, arg5, arg4
```

7.8 A fixed Example

An example incorrectly predicted by LoRA-256 but corrected by loFF is presented below. The query for this function is "**Intraday strategies will often not hold positions at the day's end.**" Even though the query is abstract and challenging to comprehend without context, loFF managed to predict it accurately. It's possible that words like "date", "24H", and "period_close" offer significant semantic information for the query. Notably, the term "end" in the query has a strong semantic relation to "close" in "period_close". This is especially relevant since "close" appears toward the end of the function, distant from the *[CLS]* token. Such an observation might imply that loFF has a heightened ability to discern long-term relationships within the function sequence.

Interestingly, this could also suggest that these repeated, normalized variables might be more informative than we initially assumed. This challenges our hypothesis that they are almost semantically meaningless, as discussed in "Unique Tokens And Repeated Tokens" in Section 5.1.1.

```
def Func(arg_0, arg_1, arg_2, arg_3=23):
    arg_4 = arg_2.copy()
    arg_4.index.names = ['date']
    arg_4['value'] = arg_4.amount * arg_4.price
    arg_4 = arg_4.reset_index().pivot_table(arg_5='date', values='value', \
    arg_13='symbol').replace(np.nan, 0)
```

```
    arg_4['date'] = arg_4.index.date
    arg_4 = arg_4.groupby('date').cumsum()
    arg_4['exposure'] = arg_4.abs().sum(axis=1)
    arg_7 = (arg_4['exposure'] == \
    arg_4.groupby(pd.TimeGrouper('24H'))['exposure'].transform(max))
    arg_4 = arg_4[arg_7].drop('exposure', axis=1)
    arg_4['cash'] = -arg_4.sum(axis=1)
    arg_8 = arg_1.copy().shift(1).fillna(0)
    arg_9 = arg_1.iloc[0].sum() / (1 + arg_0[0])
    arg_8.cash[0] = arg_9
    arg_4.index = arg_4.index.normalize()
    arg_11 = arg_8.add(arg_4, fill_value=0)
    arg_11.index.name = 'period_close'
    arg_11.columns.name = 'sid'
    return arg_11
```