

# An Empirical Study of Parameter-Efficient Fine-Tuning Methods for Pre-trained Code Models

Jiaxing Liu

School of Computer Science  
Fudan University  
Shanghai, China  
jiaxingliu21@m.fudan.edu.cn

Chaofeng Sha

School of Computer Science  
Fudan University  
Shanghai, China  
cfsha@fudan.edu.cn

Xin Peng

School of Computer Science  
Fudan University  
Shanghai, China  
pengxin@fudan.edu.cn

**Abstract**—Pre-trained code models (e.g. CodeBERT and CodeT5) have demonstrated their code intelligence in various software engineering tasks, such as code summarization. And full fine-tuning has become the typical approach to adapting these models to downstream tasks. However, full fine-tuning these large models can be computationally expensive and memory-intensive, particularly when training for multiple tasks. To alleviate this issue, several parameter-efficient fine-tuning methods (e.g. Adapter and LoRA) have been proposed to only train a small number of additional parameters, while keeping the original pre-trained parameters frozen. Although these methods claim superiority over the prior techniques, they seldom make a comprehensive and fair comparison on multiple software engineering tasks. Moreover, besides their potential in reducing fine-tuning costs and maintaining approximate performance, the effectiveness of these methods in low-resource, cross-language, and cross-project scenarios is inadequately studied.

To this end, we first conduct experiments by fine-tuning state-of-the-art code models with these methods on both code understanding tasks and code generation tasks. The results show that, by tuning only 0.5% additional parameters, these methods may achieve comparable or higher performance than full fine-tuning in code understanding tasks, but they may exhibit slightly weaker performance in code generation tasks. We also investigate the impact of these methods with varying numbers of training samples and find that, a considerable number of samples (e.g. 1000 for clone detection) may be required for them to approximate the performance of full fine-tuning. Our experimental results in cross-language and cross-project scenarios demonstrate that by freezing most pre-trained parameters and tuning only 0.5% additional parameters, these methods achieve consistent improvements in models' transfer learning ability in comparison to full fine-tuning. Our code and data are available at <https://github.com/anonymous-ase23/CodeModelParameterEfficientFinetuning>.

**Index Terms**—pre-trained code models, fine-tuning, parameter-efficient, transfer learning

## I. INTRODUCTION

With the rise of Transformer architecture and “pre-train then fine-tune” paradigm, large pre-trained code models such as CodeBERT [1] and CodeT5 [2] have shown promising results in various software engineering tasks [3] [4] [5] [6]. Traditionally, such models are adapted to downstream tasks through full fine-tuning, where all the model parameters are tuned. However, as the size of these models increases from hundreds of millions [7] to hundreds of billions [8], this

process can become computationally expensive and memory-intensive. Additionally, full fine-tuning can result in a separate copy of fine-tuned model parameters for each task, making it prohibitively expensive when serving models that perform numerous tasks. For example, Niu et al. [9] fully fine-tuned 24 pre-trained code models on 22 software engineering tasks with 3 different random seeds to conduct an empirical comparison of pre-trained models of source code, which is costly to train and maintain individually.

To alleviate this issue, various parameter-efficient fine-tuning (PEFT) methods have been applied in recent years [10] [11] [12]. For example, Xie et al. [10] utilized prefix tuning to better capture project-specific knowledge with limited training data, achieving promising results. Similarly, Wang et al. [12] introduced adapter tuning for code search and summarization tasks, which involves inserting small neural modules known as adapters to each layer and fine-tuning only these adapters. In a comparative study on four code-processing tasks, Ayupov et al. [13] evaluated two widely used PEFT methods, adapters and LoRA [14], and found that they can achieve comparable or even better performance than full fine-tuning for code understanding tasks. These methods aim to only tune a small number of additional parameters while keeping the pre-trained parameters frozen, thus reducing the computational cost and memory requirements of fine-tuning large models.

Despite recent advancements in PEFT methods [15] [16], there remains a lack of comprehensive and fair comparisons of their effectiveness across multiple software engineering tasks. Additionally, numerous researchers have investigated code models using large-scale datasets and found that the performance of fine-tuning strongly rely on the amount of downstream data [17] [18]. However, limited labeled data, i.e., low-resource scenarios, are common in practical software engineering tasks [10] [19]. Therefore, it is essential to evaluate the effectiveness of these methods in low-resource scenarios. Moreover, while previous studies in the natural language processing (NLP) field have shown that PEFT methods demonstrate superior robustness in out-of-distribution evaluation [20] [21], the transfer ability of these methods in software engineering tasks remains to be further studied.

To mitigate these gaps, we present an empirical study of PEFT methods in various code understanding and code

generation tasks across multiple scenarios. Our study covers state-of-the-art PEFT methods, including Adapter [22], LoRA [14], Prefix tuning [21], and MHM [20]. Specifically, we fine-tune four pre-trained code models on four source code tasks to evaluate the performance of PEFT methods in comparison with full fine-tuning. Additionally, to assess their effectiveness in low-resource scenarios, we conduct experiments with varying numbers of training samples on the defect detection and code translation tasks. Furthermore, to evaluate the transfer ability of these PEFT methods, we design cross-language and cross-project experiments on the code summarization task. Specifically, we fine-tune pre-trained models with these methods on one dataset and evaluate the models on other datasets.

The main contributions of our paper are as follows:

- We experimentally conduct a comprehensive and fair comparison of multiple PEFT methods on code understanding and code generation tasks. Our results demonstrate that methods such as Adapter, LoRA, and MHM can achieve approaching performance compared to full fine-tuning, while updating only 0.5% of the overall parameters in pre-trained models.
- To assess the effectiveness of these methods in low-resource scenarios, we conduct low-resource experiments. Our results suggest that, as the number of training examples increases, these methods are capable of approximating or even surpassing the performance achieved by full fine-tuning.
- We also investigate the impact of these methods on model transfer ability in cross-language and cross-project scenarios. Our results demonstrate that these methods can enhance the transfer ability of models without compromising their performance.

## II. PRELIMINARIES

In this section, we introduce some preliminary knowledge regarding our work. As PEFT methods often involve modifications to pre-trained model's layers, this section first provide a comprehensive recap of the Transformer architecture, followed by an overview of the PEFT methods utilized in this study.

### A. The Transformer Architecture

The Transformer [23] is a widely used deep neural network architecture, which can capture long-term dependencies between input tokens by self-attention mechanism. Specifically, a standard Transformer block includes two main sub-layers: multi-head attention (MHA) and fully connected feed-forward network (FFN). The MHA enables the model to encode each token by attending to the other tokens in the input sequence. The calculation of attention function depends on the three components of queries, keys and values, denoted as matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{d_k}$ , respectively. The Scaled Dot-Product Attention can be formulated as follows:

$$\text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}, \quad (1)$$

Given a sequence of  $m$  vectors  $\mathbf{x} \in \mathbb{R}^{m \times d}$  as input, where  $d$  denotes the hidden size of the Transformer model, MHA projects  $m$  to the query  $\mathbf{Q}$ , the key  $\mathbf{K}$  and the value  $\mathbf{V}$  with trainable parameterized weight  $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{d \times d_k}$ , respectively, to compute the output of each head:

$$\text{head} = \text{Attn}(\mathbf{x}\mathbf{W}_q, \mathbf{x}\mathbf{W}_k, \mathbf{x}\mathbf{W}_v) \quad (2)$$

Multi-head attention performs the attention function in parallel over  $N$  heads and concatenates the attention heads from different representation sub-spaces. The concatenated MHA value would be fed into the output projection, which is followed by the FFN. The FFN contains two linear transformations and one ReLU activation, formulated as follows:

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \quad (3)$$

where  $\mathbf{W}_1 \in \mathbb{R}^{d \times d_m}$ ,  $\mathbf{W}_2 \in \mathbb{R}^{d_m \times d}$  are trainable parameters, and  $d_m$  is typically set to  $4d$  and acts as a scaling factor. Finally, a residual connection is used followed by layer normalization.

### B. Full Fine-tuning and PEFT Methods

Pre-trained language models have proven to be highly effective in NLP field [24] [25] [26], and fine-tuning has become the dominant approach for adapting such models to new tasks. Fine-tuning involves utilizing the pre-trained model parameters as initialization and rapidly adapting them to new tasks without requiring training from scratch. The conventional way for fine-tuning is tuning all of the model parameters. For instance, when fine-tuning BERT for a downstream task, the original output layer is replaced with a task-specific layer, such as a linear classifier layer, and all initialized model parameters are trained, including token embedding parameters, Transformer blocks' parameters, and the task-specific layer parameters.

However, as previously mentioned, full fine-tuning on all pre-trained model parameters can be computationally expensive, particularly given the increasing size of pre-trained language models, which can range from hundreds of millions [7] to hundreds of billions [8]. To address this challenge, previous work has developed PEFT methods that involve adapting only a small number of the model parameters or learning external modules for new tasks while keeping most of the pre-trained language model parameters frozen. In this section, we provide a brief overview of several state-of-the-art PEFT methods employed in this study, including Adapter [22], Prefix tuning [21], LoRA [14], and MHM [20], as illustrated in Figure 1.

**Adapter:** The adapter technique [22] inserts small modules between transformer layers, which contains two projection layers and a nonlinear layer. Given a hidden input vector  $h$ , the output of the adapter is:

$$h \leftarrow h + f(\mathbf{h}\mathbf{W}_{\text{down}})\mathbf{W}_{\text{up}} \quad (4)$$

where  $f$  is a nonlinear activation function, such as ReLU,  $\mathbf{W}_{\text{down}} \in \mathbb{R}^{d \times r}$  and  $\mathbf{W}_{\text{up}} \in \mathbb{R}^{r \times d}$  are projection layer parameters. And  $r$  is the adapter's dimension, usually smaller than  $d$ , to act as a bottleneck and limit the number of additional

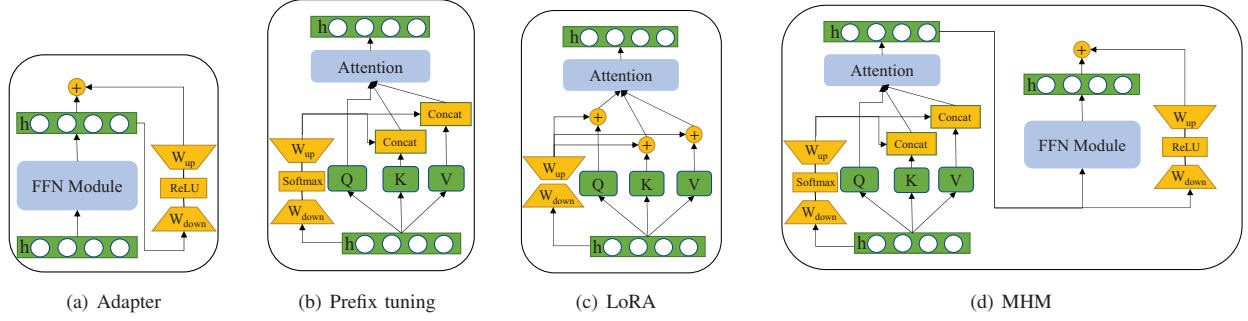


Fig. 1. Graphical illustration of several state-of-the-art PEFT methods. It is noted that these methods are of many variants and we use the typical design.

parameters. Following Houlsby et al. [22], we adopt two adapters sequentially for each transformer layer, one after the MHA sub-layer and one after the FFN sub-layer.

**Prefix Tuning:** Inspired by textual prompting methods [8], prefix tuning [21] introduces  $l$  trainable prefix vectors to the keys and values of the multi-head attention at every block (Figure 1(a)). These vectors are concatenated with the original key  $K$  and value  $V$ . This leads to the computation of each head in Eq(1) becoming:

$$head = Attn(\mathbf{xW}_q, [\mathbf{P}_k, \mathbf{xW}_k], [\mathbf{P}_v, \mathbf{xW}_v]), \quad (5)$$

where  $\mathbf{P}_k, \mathbf{P}_v \in \mathbb{R}^{l \times d_k}$  are prefix vectors. Following He et al. [20], we construct  $\mathbf{P}_k, \mathbf{P}_v$  with two trainable projection layers  $\mathbf{W}_{down} \in \mathbb{R}^{l \times m}$ ,  $\mathbf{W}_{up} \in \mathbb{R}^{m \times d_k}$  and a nonlinear activation function *softmax*, respectively. Then the prefix vectors can be formulated as *softmax*( $\mathbf{W}_{down}$ ) $\mathbf{W}_{up}$ .

**LoRA:** LoRA [14] injects trainable low-rank decomposition matrices into each layer of the Transformer architecture to reduce the number of trainable parameters for downstream tasks. For a pre-trained weight matrix  $\mathbf{W}_0 \in \mathbb{R}^{d \times k}$ , LoRA constrain its update by representing the latter with a low-rank decomposition  $\mathbf{W}_0 + \Delta W = \mathbf{W}_0 + \mathbf{W}_{down}\mathbf{W}_{up}$ , where  $\mathbf{W}_{down} \in \mathbb{R}^{d \times r}$ ,  $\mathbf{W}_{up} \in \mathbb{R}^{r \times k}$  are additional trainable parameters. In this work, we applies to the query, value and key projection matrices in the multi-head attention module, as shown in Figure 1(c).

**MHM:** He et al. [20] investigated the insertion form of adapters and found that parallel adapters (PA) outperformed sequential adapters in their experiments. Building upon this work, they deconstructed the design of the aforementioned PEFT methods and proposed a novel method called MHM (Figure 1(d)), which comprises prefix tuning and parallel adapters. In MHM, parallel adapters inserts adapters in parallel with the FFN modules, thus receiving the same input as the FFN modules. In contrast, standard adapters are inserted after the self-attention layers and FFN modules. For comparison, we also employ PA as one of the experimental PEFT methods.

### III. EXPERIMENTAL SETUP

#### A. Research Questions

This paper aims to investigate the performance of state-of-the-art PEFT methods in multiple scenarios. In this study, we conduct experiments to answer the following research questions:

**RQ1: What is the effectiveness of PEFT methods compared to full fine-tuning?** To examine the performance of PEFT methods in comparison to full fine-tuning, we conduct preliminary fine-tuning experiments on pre-trained code models using PEFT methods that update only 0.5% additional parameters. This setting follows the methodology of previous studies.

**RQ2: How does the performance of PEFT methods vary with the number of trainable parameters?** The number of trainable parameters is a crucial hyper-parameter in the context of PEFT methods. Intuitively, an increase in the number of trainable parameters corresponds to a larger learning capacity of the model. Therefore, we conduct an investigation to observe the performance variation of the PEFT method as the number of trainable parameters increases.

**RQ3: What is the effectiveness of PEFT methods in low-resource scenarios?** While the performance of fine-tuning depends strongly on the amount of downstream data available, limited labeled data, i.e., low-resource scenarios, are frequently encountered in practical software engineering tasks such as linguistic smell detection [27]. Therefore, it is essential to investigate the effectiveness of these PEFT methods in low-resource scenarios.

**RQ4: What is the effectiveness of PEFT methods in cross-language scenarios?** The cross-language scenario refers to the evaluation of the performance of a model trained on a particular programming language on other programming languages. Previous research in NLP has shown that PEFT methods, such as Adapter, enable rapid adaptation to new tasks without catastrophic forgetting [28], and often demonstrate superior robustness in out-of-distribution evaluation [21]. Additionally, Ahmed et al. [29] studied multilingual training for software engineering and found that fine-tuning with cross-language datas is promising to show high capacity of deep

learning models. These findings motivate us to fine-tune pre-trained code models on code summarization datasets in cross-language settings to assess the transfer ability of these PEFT methods.

**RQ5: What is the effectiveness of PEFT methods in cross-project scenarios?** Although previous models for neural code summarization have been trained and evaluated on extensive datasets containing independent pairs of code-summary from various software projects, they are seldom assessed for their efficacy on specific projects, which is crucial for developers in practical settings. This scenario is commonly known as project-specific code summarization [10]. The transfer ability in cross-project scenarios requires models trained on one project dataset to show promising results when tested on another project dataset. To assess the performance of PEFT methods in this scenario, we conduct fine-tuning experiments on various project-specific datasets.

### B. Tasks and Datasets

In this section, we introduce four downstream tasks and related datasets studied in our work. The statistics of the datasets are listed at Table I.

TABLE I  
STATISTICS OF DATASETS

Tasks	Datasets	#Train	#Validate	#Test	Language
Defect Detection	Devign	21,854	2,732	2,732	C
Code Clone Detection	BigCloneBench	90,102	4,000	4,000	Java
Code Summarization	CodeSearchNet	164,923	5,183	10,955	Java
		251,820	13,914	14,918	Python
		241,241	12,982	14,014	PHP
		167,288	7,325	8,122	Go
		24,927	1,400	1,261	Ruby
		58,025	3,885	3,291	JavaScript
Project-specific Code Summarization	Spring-Boot	1,576	226	450	Java
	Spring-Framework	7,714	1,102	2,204	Java
	Spring-Security	1,465	209	418	Java
	Apache-Kafka	2,415	345	690	Java
	Apache-Flink	8,086	1,155	2,310	Java
Code Translation	Java-C#	10,295	499	1,000	Java & C#
	C#-Java	10,295	499	1000	Java & C#

**Defect Detection:** Given a code snippet, the defect detection task aims to identify whether it may pose some risks to a software system, such as memory leakage and Denial of Service attacks. Following Zeng et al. [30], we utilize the Devign dataset [31] for our experiments. Devign consists of code functions extracted from two open-source C libraries, FFmpeg and Qemu. The dataset is manually labeled to indicate whether the function contains a security vulnerability.

**Code Clone Detection:** The objective of code clone detection is to determine whether a pair of code snippets are semantically similar to each other. We adopt a well-known benchmark for evaluating code clone detection, BigCloneBench [32], which is a collection of various Java methods that includes both cloned and non-cloned pairs.

**Code Summarization:** Code summarization aims to generate a concise and informative natural language summary of source code. For this task, we employ two datasets: CodeSearchNet [33] and project-specific code summarization dataset [10]. The former comprises code-text pairs from six programming languages. In this work, we focus on fine-tuning code models solely on the Java dataset and evaluate their performance on other datasets for cross-language experiments. The latter includes Java source code snippets collected from active open-source repositories of Apache and Spring, which are used to investigate the effectiveness of PEFT methods in cross-project scenarios.

**Code Translation:** Code translation, also known as code migration, involves the conversion of code from one programming language to another. We use the same dataset in CodeXGLUE benchmark [34], which is collected from several public open source repositories to migrate legacy software from Java to C#.

### C. Pre-trained Code Models

In this work, we aim to investigate the effectiveness of state-of-the-art PEFT methods on code understanding and generation tasks. Specifically, we conduct fine-tuning experiments on two pre-trained transformer-based encoder models, CodeBERT<sup>1</sup> [1] and GraphCodeBERT<sup>2</sup> [35], for defect detection and clone detection tasks, as they have shown promising performance on various code understanding tasks. Additionally, for code summarization and translation tasks, we utilize CodeT5<sup>3</sup> [36] and PLBART<sup>4</sup> [37] as our backbone models, which achieve the state-of-the-art performance in previous studies [9] [30].

### D. Metrics

In this study, we employ metrics consistent with prior studies [9] [30] to ensure a fair comparison of performance. Specifically, we adopt accuracy for detection, and F1, precision, and recall for clone detection. For code summarization, we employ BLEU-4, a widely used accuracy-based measure for natural language machine translation. For code translation, we adopt BLEU-4, accuracy (i.e., exact matching), and CodeBLEU [38]. CodeBLEU can capture specific semantic features of the code considering data flow matching and syntax matching between the generated code and the target code.

### E. Implementation Details

As suggested in prior studies, we set the learning rate to  $1e^{-4}$  for PEFT methods while retaining a learning rate of  $5e^{-5}$  for full fine-tuning. During all fine-tuning experiments, we employ the Adam optimizer, use a batch size of 16, and set a fixed Transformer input window size of 512. For defect detection and clone detection tasks, we fine-tune code models for 15 epochs. In the case of fine-tuning models

<sup>1</sup><https://huggingface.co/microsoft/codebert-base>

<sup>2</sup><https://huggingface.co/microsoft/graphcodebert-base>

<sup>3</sup><https://huggingface.co/Salesforce/codet5-base>

<sup>4</sup><https://huggingface.co/uclanlp/plbart-base>

TABLE II  
AVERAGE RESULTS ON THE TEST SET OF DEFECT DETECTION TASK AND CLONE DETECTION TASK. ALL THE PEFT METHODS ONLY TUNE 0.5% ADDITIONAL PARAMETERS. THE HYPER-PARAMETERS ARE DEFINED IN SECTION II-B, AND  $\Delta$  REPRESENTS THE ABSOLUTE DIFFERENCE BETWEEN THE BEST PEFT METHOD AND FULL FINE-TUNING.

Task	Defect Detection	Clone Detection		
Dataset	Devign	BigCloneBench		
Metrics	Accuracy	Precision	Recall	F1
CodeBERT <sub>Full(100%)</sub>	64.92	94.19	94.06	94.05
CodeBERT <sub>Adapter(0.5%,r=32)</sub>	<b>64.10</b>	94.80	<b>94.71</b>	<b>94.70</b>
CodeBERT <sub>LoRA(0.5%,r=16)</sub>	62.62	<b>95.20</b>	93.23	94.20
CodeBERT <sub>Prefix(0.5%,l=16,m=400)</sub>	56.98	89.20	89.02	89.00
CodeBERT <sub>PA(0.5%,r=32)</sub>	61.63	94.27	94.22	94.22
CodeBERT <sub>MHM(0.5%,r=16,l=8,m=200)</sub>	<b>64.10</b>	94.24	94.17	94.17
$\Delta$	-0.82	+1.01	+0.71	+0.65
GraphCodeBERT <sub>Full(100%)</sub>	65.10	94.45	94.40	94.40
GraphCodeBERT <sub>Adapter(0.5%,r=32)</sub>	62.49	<b>95.32</b>	<b>95.30</b>	<b>95.30</b>
GraphCodeBERT <sub>LoRA(0.5%,r=16)</sub>	62.71	94.70	93.52	94.10
GraphCodeBERT <sub>Prefix(0.5%,l=16,m=400)</sub>	56.39	90.31	90.05	90.04
GraphCodeBERT <sub>PA(0.5%,r=32)</sub>	61.10	94.22	94.11	94.11
GraphCodeBERT <sub>MHM(0.5%,r=16,l=8,m=200)</sub>	<b>63.79</b>	94.28	94.23	94.23
$\Delta$	-1.31	+0.87	+0.90	+0.90

TABLE III  
AVERAGE RESULTS ON THE TEST SET OF CODE SUMMARIZATION TASK AND CODE TRANSLATION TASK. THE HYPER-PARAMETERS ARE DEFINED IN SECTION II-B, AND  $\Delta$  REPRESENTS THE ABSOLUTE DIFFERENCE BETWEEN THE BEST PEFT METHOD AND FULL FINE-TUNING.

Task	Code Summarization	Code Translation					
Dataset	Java	Java to C#			C# to Java		
Metrics	BLEU-4	BLEU-4	Accuracy	CodeBLEU	BLEU-4	Accuracy	CodeBLEU
CodeT5 <sub>Full(100%)</sub>	21.04	83.96	65.90	87.66	80.04	67.00	85.60
CodeT5 <sub>Adapter(0.5%,r=30)</sub>	19.95	<b>79.28</b>	56.80	83.54	76.26	62.04	<b>82.53</b>
CodeT5 <sub>LoRA(0.5%,r=10)</sub>	20.37	78.31	55.95	82.98	75.22	60.00	81.85
CodeT5 <sub>Prefix(0.5%,l=32,m=700)</sub>	14.29	59.60	34.50	66.49	58.28	43.60	65.19
CodeT5 <sub>PA(0.5%,r=30)</sub>	20.42	77.38	54.50	82.03	75.56	60.00	82.24
CodeT5 <sub>MHM(0.5%,r=15,l=16,m=350)</sub>	<b>20.61</b>	79.13	<b>60.65</b>	<b>83.70</b>	<b>76.71</b>	<b>66.60</b>	82.22
$\Delta$	-0.43	-4.68	-5.25	-3.96	-3.33	-0.40	-3.07
PLBART <sub>Full(100%)</sub>	19.76	80.19	58.63	84.16	77.50	62.55	83.76
PLBART <sub>Adapter(0.5%,r=36)</sub>	<b>18.95</b>	<b>72.21</b>	<b>48.80</b>	<b>77.59</b>	<b>69.34</b>	<b>55.85</b>	<b>76.81</b>
PLBART <sub>LoRA(0.5%,r=12)</sub>	17.78	67.46	41.40	68.11	62.39	45.80	70.37
PLBART <sub>Prefix(0.5%,l=32,m=800)</sub>	13.32	46.32	28.47	50.00	47.94	29.46	55.78
PLBART <sub>PA(0.5%,r=36)</sub>	16.45	61.24	35.50	69.01	63.06	48.30	69.29
PLBART <sub>MHM(0.5%,r=18,l=16,m=400)</sub>	17.06	62.19	36.37	68.58	62.60	45.70	63.89
$\Delta$	-0.81	-7.98	-9.83	-6.57	-8.16	-7.70	-6.95

for code summarization and code translation tasks, we use early stopping for all models based on their performance on validation set, with a long training epoch of 30 and a default dropout rate of 0.1. We vary the random seed and repeat each experiment three times, averaging the results to evaluate the effectiveness of these methods. We execute all models on eight Linux machines, each of which is equipped with an Intel(R) Core(TM) i9-10980XE @ 3.00GHz CPU and a 24GB NVIDIA GeForce RTX 3090 GPU.

#### IV. RESULTS AND DISCUSSION

A. *RQ1: What is the effectiveness of PEFT methods compared to full fine-tuning?*

In this section, we perform preliminary fine-tuning of pre-trained code models using PEFT methods with only 0.5% additional parameter. The results are presented in Tables II

and III, where the hyper-parameters are defined in Section II-B, and  $\Delta$  represents the absolute difference between the best PEFT method and full fine-tuning. From Table II, we observe that when fine-tuning CodeBERT, the  $\Delta$  value for accuracy in defect detection is -0.82%, while the  $\Delta$  values for precision, recall, and F1 in clone detection are 1.01%, 0.71%, and 0.65%, respectively. And similar results can be observed when fine-tuning the GraphCodeBERT model. These results suggest that, in code understanding tasks, PEFT methods can achieve comparable or better performance than full fine-tuning. Moreover, MHM exhibits the best performance in defect detection, while Adapter performs the best in clone detection. However, prefix tuning shows a significant decrease in performance in both tasks.

From Table II, we find that when fine-tuning code models for code summarization, the  $\Delta$  values for CodeT5 and

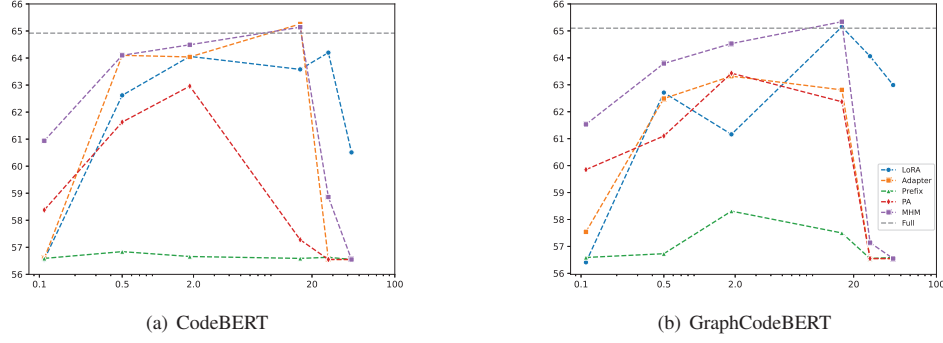


Fig. 2. Testing accuracy of fine-tuning CodeBERT and GraphCodeBERT with PEFT methods on defect detection task with varying number of trainable parameters. The horizontal and vertical axes represent the ratio of trainable parameters and the models' accuracy on the test set, respectively.

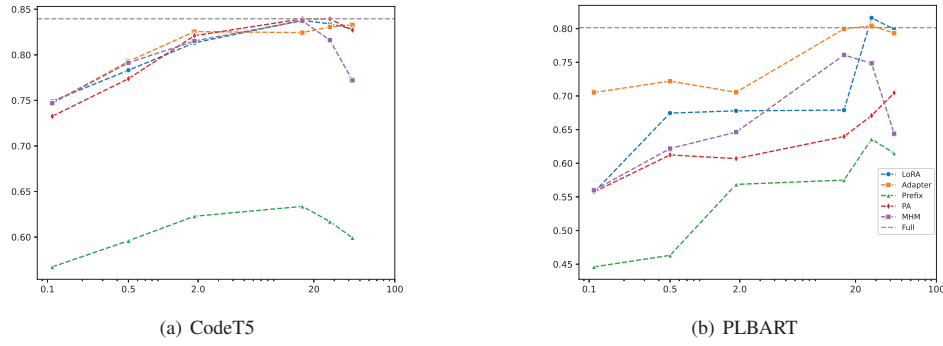


Fig. 3. Testing BLEU-4 scores of fine-tuning CodeT5 and PLBART with PEFT methods on code translation task (JAVA-C#) with varying number of trainable parameters. The horizontal and vertical axes represent the ratio of trainable parameters and the models' BLEU-4 scores on the test set, respectively.

PLBART are -0.43% and -0.81%, respectively. It indicates that PEFT methods consistently maintain comparable performance in code summarization. However, when fine-tuning pre-trained code models for code translation, the  $\Delta$  values indicate a significant degradation in performance. For example, when fine-tuning CodeT5 for Java to C# translation, the  $\Delta$  values for BLEU-4, accuracy and CodeBLEU are -4.68%, -5.25% and -3.96%, respectively. And when fine-tuning the PLBART model for Java to C# translation, the  $\Delta$  values for BLEU-4, accuracy and CodeBLEU are -7.98%, -9.83% and -6.57%, respectively. These findings indicate that, when updating only 0.5% additional parameters, PEFT methods may underperform full fine-tuning in code generation tasks. The observation is consistent with Ayupov et al. [13] in NLP.

In addition, we can observe a significant decrease of 6.75% for CodeT5 and 6.44% for PLBART in BLEU-4 scores when using prefix tuning compared to full fine-tuning in code summarization. Furthermore, when fine-tuning CodeT5 and PLBART with prefix tuning in code translation tasks, the performance decrease is even more substantial. These findings reveal that prefix tuning is not comparable to full fine-tuning and other PEFT methods in code generation tasks when updating only 0.5% additional parameters in code generation tasks. It is in agreement with the observations found by Hu

et al. [14] in the NLP field, who suspect the reason may be that prefix tuning might cause the input distribution to shift away from the pre-training data distribution. Moreover, in contrast to the findings of He et al. [20], our results indicate that parallel adapters (PA) do not consistently outperform sequential adapters (Adapter), and both forms of adapters demonstrate comparable performance.

**Finding 1.** When updating only 0.5% additional parameters, PEFT methods may achieve comparable or higher performance than standard full fine-tuning in code understanding tasks, but they may exhibit slightly weaker performance in code generation tasks.

*B. RQ2: How does the performance of PEFT methods vary with the number of trainable parameters?*

We conduct experiments to investigate the impact of the number of trainable parameters on PEFT methods. Specifically, we fine-tune models on defect detection and code translation (JAVA-C#) tasks, as they exhibit a more noticeable decline in performance with PEFT methods when updating only 0.5% additional parameters compared to other tasks in RQ1. To vary the number of trainable parameters, we vary the bottleneck dimension  $r$  of Adapters in the range  $\{8, 32, 64,$

640, 1280, 2560}, corresponding to {0.1%, 0.5%, 1.8%, 16%, 27%, 43%} of additional trainable parameters.

The results presented in Figure 2 demonstrate that, as the number of trainable parameters increases, PEFT methods initially improve the performance on the defect detection task. However, the performance eventually decreases after a certain point. One potential explanation for this phenomenon is that introducing a large number of additional parameters (e.g., 27%) can make it increasingly challenging to tune these parameters with limited samples to achieve promising performance because they are non-pre-trained. Additionally, when the proportion of trainable parameters in the model increases to 16%, LoRA, MHM, and Adapter methods exhibit higher accuracy compared to the Full fine-tuning approach.

Regarding the code translation task, the results presented in Figure 3 indicate that, as the number of trainable parameters increases, the model's performance rarely degrades significantly even when the number of trainable parameters reaches 43%. This difference in performance may be due to the fact that efficiently fine-tuning CodeT5 and PLBART on the code summarization task is more challenging than the defect detection task, requiring more trainable parameters to achieve promising results. Furthermore, we can observe that, with a certain number of trainable parameters (e.g., 27.5%), both Adapter and LoRA methods can outperform full fine-tuning (Figure 3(b)). These findings suggest that, while the results from RQ1 reveal that PEFT methods with updating only 0.5% additional parameters, may not perform as well as full fine-tuning in code generation tasks, PEFT can outperform full fine-tuning as the number of trainable parameters increases.

**Finding 2.** The PEFT methods can achieve better experimental results than full fine-tuning in both code understanding tasks and code generation tasks, as the number of trainable parameters increases. However, the performance of these methods tends to decrease as the number of parameters is further increased, especially in code understanding tasks.

C. RQ3: What is the effectiveness of PEFT methods in low-resource scenarios?

As outlined in Section III-A, low-resource scenarios are frequently encountered in practical software engineering tasks, and the effectiveness of fine-tuning is heavily contingent upon the availability of downstream data. Therefore, we evaluate the performance of PEFT methods in low-resource scenarios by conducting experiments on clone detection and code summarization tasks. These two tasks are selected since the results in RQ1 denote that, compared to other tasks, PEFT methods can attain a more approximate performance to full fine-tuning when applied in the two tasks. To simulate low-resource scenarios, following Wang et al. [12], we randomly sample datasets of 600, 1,200, 3,000, and 6,000 samples for code summarization task. Similarly, we randomly sample datasets of 100, 200, 500, and 1,000 samples for defect detection

TABLE IV  
AVERAGE RESULTS ON F1 SCORES OF CLONE DETECTION TASK IN LOW-RESOURCE SCENARIOS.  $\Delta$  REPRESENTS THE ABSOLUTE DIFFERENCE BETWEEN THE BEST PEFT METHOD AND FULL FINE-TUNING.

Task	Clone Detection			
#Training Samples	100	200	500	1,000
CodeBERT <sub>Full</sub> (100%)	77.42	83.37	88.92	91.15
CodeBERT <sub>Adapter</sub> (0.5%, <i>r</i> =32)	74.94	<b>81.95</b>	87.99	90.22
CodeBERT <sub>LoRA</sub> (0.5%, <i>r</i> =10)	74.06	81.23	86.99	89.58
CodeBERT <sub>PA</sub> (0.5%, <i>r</i> =32)	75.43	81.88	86.63	89.76
CodeBERT <sub>MHM</sub> (0.5%, <i>r</i> =16, <i>l</i> =8, <i>m</i> =200)	<b>76.70</b>	81.21	<b>88.49</b>	<b>90.48</b>
$\Delta$	-0.72	-1.42	-0.43	-0.67
GraphCodeBERT <sub>Full</sub> (100%)	79.10	84.93	89.05	91.04
GraphCodeBERT <sub>Adapter</sub> (0.5%, <i>r</i> =32)	77.15	83.41	87.91	89.41
GraphCodeBERT <sub>LoRA</sub> (0.5%, <i>r</i> =16)	76.27	82.92	<b>88.15</b>	89.91
GraphCodeBERT <sub>PA</sub> (0.5%, <i>r</i> =32)	77.55	<b>83.85</b>	87.40	89.99
GraphCodeBERT <sub>MHM</sub> (0.5%, <i>r</i> =16, <i>l</i> =8, <i>m</i> =200)	<b>77.86</b>	81.95	87.67	<b>91.05</b>
$\Delta$	-1.14	-1.08	-0.90	+0.01

TABLE V  
AVERAGE RESULTS ON BLEU-4 SCORES OF CODE SUMMARIZATION TASK IN LOW-RESOURCE SCENARIOS.  $\Delta$  REPRESENTS THE ABSOLUTE DIFFERENCE BETWEEN THE BEST PEFT METHOD AND FULL FINE-TUNING.

Task	Code Summarization			
#Training Samples	600	1,200	3,000	6,000
CodeT5 <sub>Full</sub> (100%)	16.09	18.87	18.65	19.36
CodeT5 <sub>Adapter</sub> (0.5%,r=30)	<b>15.54</b>	18.45	18.20	<b>19.33</b>
CodeT5 <sub>LoRA</sub> (0.5%,r=10)	15.17	<b>18.68</b>	<b>19.05</b>	19.25
CodeT5 <sub>PA</sub> (0.5%,r=30)	15.02	18.18	18.58	18.66
CodeT5 <sub>MHM</sub> (0.5%,r=15,l=16,m=350)	15.12	17.98	18.66	18.86
$\Delta$	-0.55	-0.19	+0.40	-0.03
PLBART <sub>Full</sub> (100%)	14.33	15.65	15.71	16.15
PLBART <sub>Adapter</sub> (0.5%,r=36)	12.59	<b>15.54</b>	<b>16.26</b>	16.01
PLBART <sub>LoRA</sub> (0.5%,r=12)	12.39	15.35	15.03	<b>16.03</b>
PLBART <sub>PA</sub> (0.5%,r=36)	<b>12.88</b>	15.32	15.21	15.92
PLBART <sub>MHM</sub> (0.5%,r=18,l=16,m=400)	12.78	15.36	15.33	16.00
$\Delta$	-1.45	-0.11	+0.55	-0.12

task, based on the size of the original datasets. We apply PEFT methods to these sampled datasets, except for prefix tuning, which exhibits a significant reduction in performance compared to other methods in RQ1 and RQ2.

Our results are presented in Tables IV and Table V. Firstly, we can find that, comparing with RQ1, the  $\Delta$  value decrease from +0.65% to -0.72% when fine-tuning CodeBERT in clone detection with 100 training samples. And similar results can be observed in code summarization. These finds indicate that, in low-resource scenarios, the performance of PEFT methods would show a obvious degradation. Secondly, we can find that, in order for PEFT methods to achieve comparable performance to full fine-tuning, 1,000 training samples are needed for defect detection and 1,200 training samples are needed for code summarization. This could be attributed to the fact that fine-tuning models with PEFT methods starts from non-pre-trained parameters, and thus a sufficient number of samples are required to achieve promising results.

Thirdly, we can observe that, when fine-tuning GraphCodeBERT in clone detection with the number of training samples varying from 100 to 1,000, the  $\Delta$  values are -

TABLE VI

AVERAGE RESULTS ON BLEU-4 SCORES OF CODE SUMMARIZATION TASK IN CROSS-LANGUAGE SCENARIOS. THESE MODELS ARE FINE-TUNED ON JAVA DATASET AND ARE TESTED ON OTHER PROGRAMMING LANGUAGE DATASETS.  $\Delta$  REPRESENTS THE ABSOLUTE DIFFERENCE BETWEEN THE BEST PEFT METHOD AND FULL FINE-TUNING.

Task Dataset	Code Summarization				
	Ruby	JavaScript	Go	PHP	Python
CodeT5 <sub>Full</sub> (100%)	15.01	15.36	15.11	23.43	18.26
CodeT5 <sub>Adapter</sub> (0.5%, $r=30$ )	15.33	15.63	<b>15.99</b>	24.14	18.96
CodeT5 <sub>LoRA</sub> (0.5%, $r=10$ )	<b>15.64</b>	<b>15.79</b>	15.78	<b>24.61</b>	<b>19.06</b>
CodeT5 <sub>PA</sub> (0.5%, $r=30$ )	15.42	15.48	15.95	24.29	18.92
CodeT5 <sub>MHM</sub> (0.5%, $r=15$ , $l=16$ , $m=350$ )	15.50	15.57	15.96	24.41	18.94
$\Delta$	+0.63	+0.43	+0.88	+1.18	+0.80
PLBART <sub>Full</sub> (100%)	11.13	11.18	11.48	21.76	16.16
PLBART <sub>Adapter</sub> (0.5%, $r=36$ )	<b>13.25</b>	<b>12.86</b>	<b>11.87</b>	<b>21.95</b>	17.31
PLBART <sub>LoRA</sub> (0.5%, $r=12$ )	11.59	11.89	11.33	21.24	<b>17.49</b>
PLBART <sub>PA</sub> (0.5%, $r=36$ )	10.31	10.58	10.26	20.15	15.13
PLBART <sub>MHM</sub> (0.5%, $r=18$ , $l=16$ , $m=400$ )	10.44	11.00	11.12	20.13	16.07
$\Delta$	+2.12	+1.68	+0.39	+0.19	+1.33

1.14%, -1.08%, -0.90% and +0.01%. And when fine-tuning CodeT5 in clone summarization with the number of training samples varying from 600 to 6,000, the  $\Delta$  values are -0.55%, -0.19%, +0.40% and -0.03%. These findings suggest that, as the number of training samples increases, PEFT methods tend to achieve better performance, even outperform that of full fine-tuning, despite updating only 0.5% additional parameters. Furthermore, we note that, in low-resource scenarios, Adapter and LoRA tend to produce superior results when fine-tuning pre-trained code models for code summarization task, whereas MHM frequently leads to superior performance when applied to defect detection task. It suggests that different PEFT methods can achieve promising results for different tasks in low-resource scenarios.

**Finding 3.** In low-resource scenarios, as the number of training samples increases, the performance of PEFT methods shows a remarkable improvement, leading to a reduction in the gap between these methods and full fine-tuning. Furthermore, in some cases, PEFT methods can even outperform full fine-tuning, despite only updating 0.5% additional parameters.

*D. RQ4: What is the effectiveness of PEFT methods in cross-language scenarios?*

In this section, we conduct experiments to assess the performance of PEFT methods in cross-language scenarios. Specifically, We fine-tune CodeT5 and PLBART models with PEFT methods on Java code summarization task and evaluate these models using these methods on datasets in other programming languages, including Ruby, JavaScript, Go, PHP, and Python.

The results are presented in Table VI. Our results reveal that, when evaluating fine-tuned CodeT5 models in cross-language scenarios, the  $\Delta$  values are 0.63%, 0.43%, 0.88%, 1.18%, and 0.80% for testing on Ruby, JavaScript, Go, PHP, and Python datasets, respectively. Similarly, the  $\Delta$  values are 2.12%, 1.68%, 0.39%, 0.19%, and 1.33% for evaluating

fine-tuned PLBART models on the corresponding datasets. These findings suggest that PEFT methods exhibit a significant improvement over full fine-tuning in cross-language scenarios, indicating that they can significantly enhance the transfer ability of models in such scenarios. Furthermore, Adapter and LoRA usually demonstrate best experimental results for all cross-language evaluation.

**Finding 4.** When updating only 0.5% additional parameters, the PEFT methods demonstrate better transfer ability in cross-language scenarios, although they show only comparable performance on code summarization tasks compared to full fine-tuning.

*E. RQ5: What is the effectiveness of PEFT methods in cross-project scenarios?*

In this section, we conduct experiments to assess the effectiveness of PEFT methods in cross-project scenarios. Specifically, we train CodeT5 models using both full fine-tuning and PEFT methods on one project dataset and evaluate their performance on another project dataset. For comparison, we firstly conduct experiments for project-specific code summarization task and report the average BLEU-4 scores in Table VII, where models are fine-tuned and evaluated on the same dataset. Furthermore, we illustrate the relative improvement in BLEU-4 scores of PEFT methods over full fine-tuning in Figure 4. In these heatmap figures, the horizontal and the vertical axes represent the different training sets and test sets, respectively, and each row corresponds to a model fine-tuned on one training sets and evaluated on different testing sets.

As shown in Table VII, we can find that full fine-tuning often outperform PEFT methods and achieve the best performance. And from the diagonal values in Figure 4(b)(c)(d), we can observe that the relative BLEU-4 improvement is generally negative. For instance, when fine-tuning CodeT5 with Adapter, the diagonal values are -0.56%, -0.77%, 0.23%, -1.00%, and -0.39%. These findings suggest that PEFT methods

TABLE VII  
AVERAGE RESULTS ON BLEU-4 SCORES OF PROJECT-SPECIFIC CODE SUMMARIZATION TASK. THESE MODELS ARE FINE-TUNED AND TESTED ON THE SAME DATASET.

Task Dataset	Project-specific Code Summarization				
	Spring-boot	Spring-framework	Spring-security	Apache Flink	Apache Kafka
CodeT5 <sub>w/o-FT</sub>	1.60	1.57	0.76	0.95	1.80
CodeT5 <sub>Full(100%)</sub>	<b>38.45</b>	32.67	36.61	<b>31.38</b>	<b>26.26</b>
CodeT5 <sub>Adapter(0.5%,r=30)</sub>	37.89	31.90	35.84	28.36	25.87
CodeT5 <sub>LoRA(0.5%,r=10)</sub>	37.29	31.13	<b>36.71</b>	30.84	26.18
CodeT5 <sub>PA(0.5%,r=30)</sub>	36.83	31.07	34.57	29.62	25.76
CodeT5 <sub>MHM(0.5%,r=15,l=16,m=350)</sub>	38.15	<b>33.28</b>	35.37	31.11	26.02

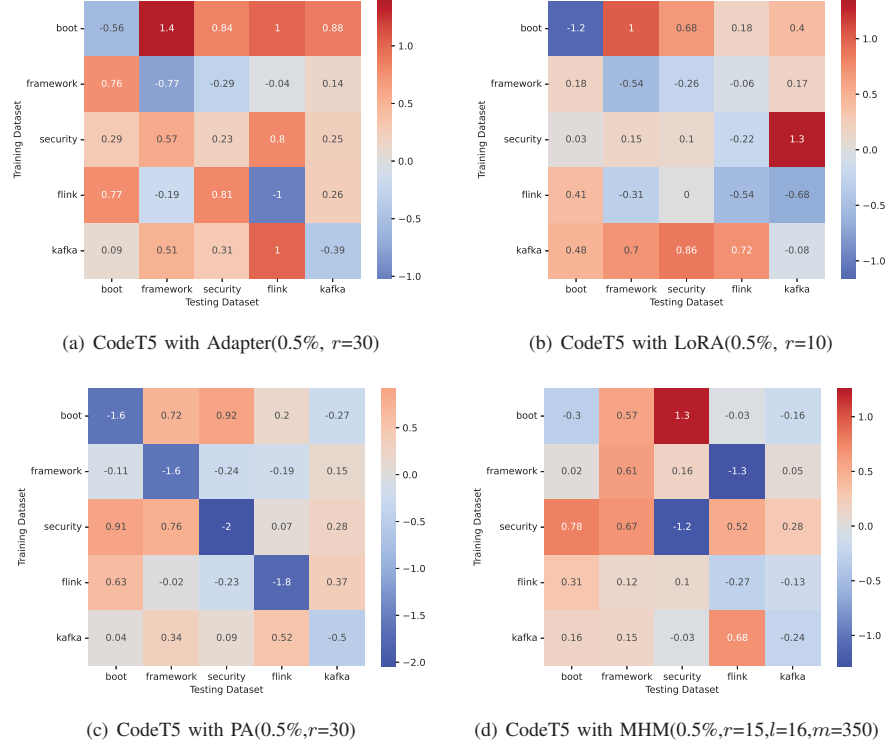


Fig. 4. Relative BLEU-4 improvement of PEFT methods over full fine-tuning on cross-project code summarization task. The horizontal and the vertical axes represent the different training sets and test sets, respectively, and each row corresponds to a fine-tuned model testing on different datasets.

may underperform full fine-tuning, when tuning and testing on the same code summarization dataset.

However, when fine-tuning CodeT5 with Adapter on the Spring-boot dataset and testing on Spring-framework, Spring-security, Apache-flink, and Apache-kafka datasets, the relative improvement is 1.40%, 0.84%, 1.00%, and 0.88%, respectively. And when fine-tuning CodeT5 with Lora on the Apache-kafka dataset and testing on Spring-boot, Spring-framework, Spring-security and Apache-flink datasets, the relative improvement is 0.48%, 0.70%, 0.86%, and 0.72%, respectively. It demonstrates that PEFT methods tend to show better results than full fine-tuning in cross-project scenarios. This suggests that PEFT methods can enhance the model's transfer ability across projects while maintaining comparable results to full fine-tuning. One possible explanation for the

superior performance of PEFT methods is their ability to leverage the frozen pre-trained parameters, which can enhance their transfer learning capability in cross-project scenarios. For further comparison, we directly conduct inference using the pre-trained CodeT5 model without any fine-tuning (*w/o-FT*) and present the performance in Table VII. We can find that only frozen pre-trained parameters are not suffice for achieving satisfactory evaluation performance.

**Finding 5.** In cross-project scenarios, PEFT methods can leverage the frozen pre-trained model parameters to achieve better transfer ability compared to full fine-tuning.

## V. LIMITATIONS AND THREATS TO VALIDITY

**External Validity** concerns the generalizability of study findings to other settings and datasets. In this study, we use four datasets comprising two code understanding and two code generation tasks involving various programming languages to assess the effectiveness of several state-of-the-art PEFT methods. However, due to resource constraints, we do not include other promising methods, such as prompt tuning [39] and BitFit [40], which may affect the generalizability of our results. Additionally, since there are various versions of PEFT methods, such as LoRA that can be applied to both FFN and MHA modules, we only test the typical design of these methods. Therefore, future research can investigate more PEFT methods and their variations to further validate our findings. Another potential threat is the poor choice of performance measures. We adopt BLEU for code translation task, while it was initially designed for natural language generation and is poor in measuring the correctness of programs [38]. Although we have also utilized CodeBLEU [38] to evaluate different models, there are still some metrics (e.g., functional correctness [41] and CrystalBLEU [42]) that could be considered in the future work.

**Internal Validity** concerns unanticipated relationships that could impact the study's results. One potential threat is the number of trainable parameters. We initially utilize PEFT methods that updated only 0.5% additional parameters of the overall parameters. Even though we have adjusted the trainable parameters to evaluate the performance changes of these methods, further parameter settings are still required to assess these methods comprehensively. Another potential threat is the limited availability of resources. Although fine-tuning is less resource-intensive than training models on source code corpora, it still necessitates significant time on one or more GPUs, and a single run can take hours. This limitation restricts the number of experiments and training epochs that can be performed. Specifically, we conduct all of our experiments with three different random seeds and limited epochs.

## VI. RELATED WORK

### A. Pre-trained Code Models

Transformer was first applied to various NLP tasks, where large pre-trained language models have achieved notable improvements [7] [43]. In recent years, there has been a growing interest among researchers in transferring pre-trained language models to software engineering, leading to the development of several pre-trained code models for source code understanding and generation [44] [45]. CodeBERT, a BERT-style encoder model, is a bimodal pre-trained model designed to process both natural language text and source code. It employs the same architecture as RoBERTa and is pre-trained on the CodeSearchNet corpus, which comprises millions of multilingual source code snippets collected from GitHub. CodeBERT inputs code snippets as a sequence of tokens and is pre-trained through a self-supervised learning paradigm, using Masked Language Modeling (MLM) and Replaced Token Detection

objectives. Similarly, CodeT5, following the same architecture as T5, is a unified pre-trained encoder-decoder transformer model. In addition to the original pre-training tasks (e.g. MLM), CodeT5 proposes a novel identifier-aware pre-training task to leverage code-specific structural information, as well as a bimodal dual-generation pre-training task for augmenting natural language programming language alignment. As an encoder-decoder model, CodeT5 formalizes all tasks as text-to-text tasks and is commonly applied to code-related sequence generation tasks. These pre-trained code models have demonstrated substantial improvements in various software engineering tasks [46] [47], benefiting from their powerful, general-purpose linguistic representations.

### B. Parameter-efficient Fine-tuning Methods

Parameter-efficient fine-tuning methods, such as Adapter and LoRA, were proposed to alleviate the computationally expensive of full fine-tuning. PEFT methods introduce only a small number of additional parameters (e.g., 0.5%) and fine-tune solely these parameters, leaving the pre-trained model parameters frozen. Wang et al. [48] conducted prompt tuning on CodeBERT and CodeT5, and experimental found that prompt tuning [39] consistently outperforms fine-tuning in defect prediction, code summarization, and code translation tasks. However, due to resource constraints, we adopt prefix tuning [21] other than prompt tuning to compare with full fine-tuning method, since prompt tuning prepends soft tokens only in source input instead of each transformer layer, and can be seen as a special prefix tuning. Ayupov et al. [13] have investigated the effectiveness of two widely used approaches, adapters and LoRA [14], on four software engineering tasks, finding that parameter-efficient fine-tuning methods can achieve comparable or even superior performance to full fine-tuning for code understanding tasks.

## VII. CONCLUSION

This paper presents an empirical study of parameter-efficient fine-tuning methods applied to code understanding and code generation tasks. Our results suggest that methods such as Adapter and LoRA achieve comparable performance to full fine-tuning by updating only 0.5% additional parameters. Moreover, as the number of trainable parameters increases, these methods can outperform full fine-tuning on these tasks. However, in low-resource scenarios, performance degradation may occur, which can be mitigated by increasing the number of training samples. Our study also demonstrates that PEFT methods can improve the transfer ability of code models in cross-language and cross-project scenarios.

Furthermore, Adapter and LoRA exhibit better performance than other PEFT methods, while their variants, such as Parallel Adapter and MHM, sometimes show promising results. Our findings suggest that Adapter and LoRA can be applied to transfer pre-trained language models to software engineering tasks with less training cost and memory. With their superior transfer ability, these methods could enable fine-tuning models to handle more programming languages and projects.

## REFERENCES

- [1] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- [2] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: <https://doi.org/10.18653/v1/2021.emnlp-main.685>
- [3] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 1643–1652. [Online]. Available: <https://doi.org/10.18653/v1/d18-1192>
- [4] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning source code representations," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1–13. [Online]. Available: <https://doi.org/10.1145/3510003.3510096>
- [5] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2291–2302. [Online]. Available: <https://doi.org/10.1145/3510003.3510621>
- [6] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 287–298. [Online]. Available: <https://doi.org/10.1145/3510003.3510062>
- [7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: <https://doi.org/10.18653/v1/n19-1423>
- [8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/hash/1457c0d6fbcb4967418bfb8ac142f64a-Abstract.html>
- [9] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2136–2148. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00180>
- [10] R. Xie, T. Hu, W. Ye, and S. Zhang, "Low-resources project-specific code summarization," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 68:1–68:12. [Online]. Available: <https://doi.org/10.1145/3551349.3556909>
- [11] D. Goel, R. Grover, and F. H. Fard, "On the cross-modal transfer from natural language to code through adapter modules," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, A. Rastogi, R. Tufano, G. Bavota, V. Arnaoudova, and S. Haiduc, Eds. ACM, 2022, pp. 71–81. [Online]. Available: <https://doi.org/10.1145/3524610.3527892>
- [12] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 5–16. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00013>
- [13] S. Ayupov and N. Chirkova, "Parameter-efficient finetuning of transformers for source code," *CoRR*, vol. abs/2212.05901, 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2212.05901>
- [14] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. [Online]. Available: <https://openreview.net/forum?id=nZeVKeeFYf9>
- [15] X. Luo, Y. Xue, Z. Xing, and J. Sun, "PRCBERT: prompt learning for requirement classification using bert-based pretrained language models," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 75:1–75:13. [Online]. Available: <https://doi.org/10.1145/3551349.3560417>
- [16] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, "Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 79:1–79:13.
- [17] C. Niu, C. Li, V. Ng, and B. Luo, "Crosscodebench: Benchmarking cross-task generalization of source code models," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 537–549. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00055>
- [18] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 49–60. [Online]. Available: <https://doi.org/10.1145/3106237.3106256>
- [19] J. A. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Transactions on Software Engineering*, 2021.
- [20] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, "Towards a unified view of parameter-efficient transfer learning," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. [Online]. Available: <https://openreview.net/forum?id=0RDcd5Axok>
- [21] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 4582–4597.
- [22] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for NLP," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 2790–2799. [Online]. Available: <http://proceedings.mlr.press/v97/houlsby19a.html>
- [23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.
- [24] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: <http://arxiv.org/abs/1907.11692>

- [25] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: <http://jmlr.org/papers/v21/20-074.html>
- [26] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *NeurIPS*, 2022. [Online]. Available: [http://papers.nips.cc/paper\\_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html)
- [27] S. Fakhoury, V. Arnaudova, C. Noisoux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 602–611.
- [28] J. Pfeiffer, A. Kamath, A. Rücklé, K. Cho, and I. Gurevych, "Adapterfusion: Non-destructive task composition for transfer learning," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, P. Merlo, J. Tiedemann, and R. Tsarfaty, Eds. Association for Computational Linguistics, 2021, pp. 487–503. [Online]. Available: <https://doi.org/10.18653/v1/2021.eacl-main.39>
- [29] T. Ahmed and P. T. Devanbu, "Multilingual training for software engineering," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1443–1455. [Online]. Available: <https://doi.org/10.1145/3510003.3510049>
- [30] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 39–51. [Online]. Available: <https://doi.org/10.1145/3533767.3534390>
- [31] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 197–10 207. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html>
- [32] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 476–480.
- [33] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: <http://arxiv.org/abs/1909.09436>
- [34] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>
- [35] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: <https://openreview.net/forum?id=jLoC4ez43PZ>
- [36] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: <https://doi.org/10.18653/v1/2021.emnlp-main.685>
- [37] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Association for Computational Linguistics, 2021, pp. 2655–2668. [Online]. Available: <https://doi.org/10.18653/v1/2021.naacl-main.211>
- [38] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: <https://arxiv.org/abs/2009.10297>
- [39] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 3045–3059. [Online]. Available: <https://doi.org/10.18653/v1/2021.emnlp-main.243>
- [40] E. B. Zaken, Y. Goldberg, and S. Ravfogel, "Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 1–9.
- [41] B. Rozière, M. Lachaux, L. Chantussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020.
- [42] A. Eghbali and M. Pradel, "Crystalbleu: Precisely and efficiently measuring the similarity of code," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 341–342. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE-Companion55297.2022.9793747>
- [43] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [44] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, "Codetrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing," *CoRR*, vol. abs/2104.02443, 2021. [Online]. Available: <https://arxiv.org/abs/2104.02443>
- [45] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225. [Online]. Available: <https://doi.org/10.18653/v1/2022.acl-long.499>
- [46] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: <https://doi.org/10.18653/v1/p16-1195>
- [47] M. R. I. Rabin, A. Hussain, V. J. Hellendoorn, and M. A. Alipour, "Memorization and generalization in neural code intelligence models," *CoRR*, vol. abs/2106.08704, 2021. [Online]. Available: <https://arxiv.org/abs/2106.08704>
- [48] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 382–394. [Online]. Available: <https://doi.org/10.1145/3540250.3549113>