Pass-Tuning: Towards Structure-Aware Parameter-Efficient Tuning for Code Representation Learning

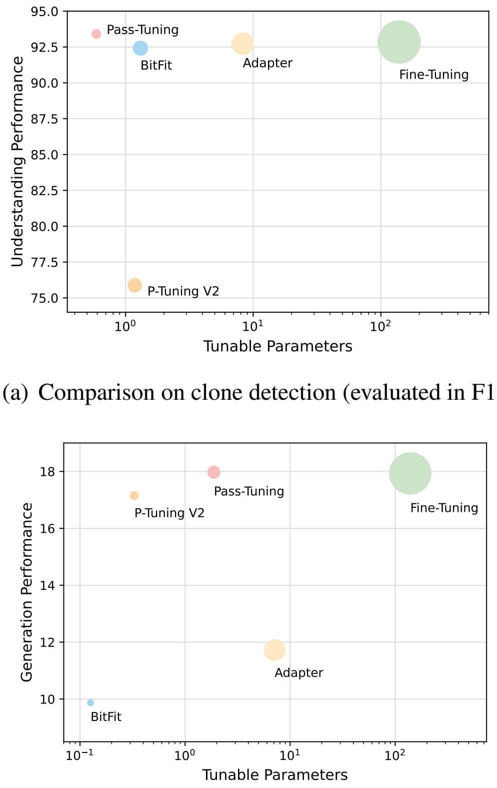
Nuo Chen[1], **Qiushi Sun**[1,2], **Jianing Wang**[1], **Xiang Li**[1]*, **Ming Gao**[1,3]
[1] School of Data Science and Engineering, East China Normal University
[2] National University of Singapore
[3] KLATASDS-MOE, School of Statistics, East China Normal University
{nuochen,qiushisun}@stu.ecnu.edu.cn, lygwjn@gmail.com
{xiangli,mgao}@dase.ecnu.edu.cn

Abstract

Code pre-trained models (CodePTMs) have recently become the de-facto paradigm for various tasks in the domain of code intelligence. To achieve excellent performance, the widely used strategy is to fine-tune all the parameters of CodePTMs. However, as the model size increases along with the number of downstream tasks, this strategy becomes excessively expensive. There are also some prior works that utilize Parameter-Efficient Learning (PEL) methods for model tuning in natural language processing to mitigate similar problems, but applying them directly to CodePTMs fails to capture the inherent structural characteristics of codes. To address the problem, in this paper, we propose *Pass-Tuning* for structure-aware Parameter-Efficient code representation learning. Specifically, a plug-and-play graph neural network module that can learn from Abstract Syntax Tree (AST) is employed as a tunable prefix. On the one hand, *Pass-Tuning* can further exploit the structural information of source code. On the other hand, it could serve as a replacement for full fine-tuning. We evaluate our method on multiple tasks across eight programming languages, including code understanding and generation. These results demonstrate the effectiveness, robustness, and universality of our method. Our codes and resources are available at https://github.com/nchen909/Pass-Tuning.

1 Introduction

Pre-trained language models (Devlin et al., 2019; Liu et al., 2019) have significantly boosted a series of natural language processing (NLP) tasks. These models mainly adopt deep transformer architecture (Vaswani et al., 2017), which are pre-trained on a large-scale unsupervised text corpus, and then fine-tuned on downstream tasks. When regarding a code snippet as a sequence of tokens, it innately lends itself to these transformer-based models (Yang et al., 2019; Lewis et al., 2020; Raffel et al., 2020) from NLP.



(a) Comparison on clone detection (evaluated in F1)

(b) Comparison on code summarization (evaluated in Smoothed BLEU-4 score)

Figure 1: *Pass-Tuning* performs far beyond popular Parameter-Efficient learning methods over both code understanding and generation tasks. It also achieves comparative performance to full fine-tuning with much fewer tunable parameters. The size of the circle is proportional to the number of tunable parameters. All the methods are evaluated on the PLBART backbone model.

Under the assumption of "Software Naturalness" (Hindle et al., 2016; Buratti et al., 2020) and inspired by the enormous success of pre-training, code pre-trained models (CodePTMs) have also been proposed and widely applied in the realm of code intelligence (Feng et al., 2020; Guo et al.,

*Corresponding Author.

Findings of the Association for Computational Linguistics: EMNLP 2023, pages 577–591
December 6-10, 2023 ©2023 Association for Computational Linguistics

2021; Wang et al., 2021; Ahmad et al., 2021; Guo et al., 2022). Similarly, a straightforward approach to adapting these large-scale CodePTMs to the downstream tasks (*e.g.*, code summarization, code clone detection) is fine-tuning the whole model for each task. However, the number of downstream tasks is ever-increasing, and it is often necessary to fine-tune the whole model for a specific programming language in real-world applications (Ayupov and Chirkova, 2022; Sun et al., 2023). The number of combinations of tasks and languages leads to standard full fine-tuning solutions being prohibitively expensive and cumbersome.

To reduce the cost of model tuning, researchers employ strategies like Prefix-Tuning (Li and Liang, 2021) to condition language models with frozen parameters to perform specific downstream tasks. Notwithstanding, we experimentally observe that current PEL methods (Ding et al., 2023) struggle to reach the same stunning effect in code representation learning as in the NLP domain[1]. Motivated to address the cost-performance dilemma for tuning CodePTMs, we believe that a new Parameter-Efficient Learning (PEL) approach with versatility should be built beyond the NLP domain.

In our work, we present a Parameter-Efficient tuning approach with Graph attention Network capturing code structural information from abstract syntax tree, namely *Pass-Tuning*, a versatile PEL method with minor tuning cost and competitive performance, to serve as a lightweight alternative to fine-tuning. Specifically, we first construct tunable Graph Attention Network (GAT) modules as tunable prefixes to capture the code structure information contained in Abstract Syntax Trees (ASTs). Then, we design a novel code retrieval strategy based on attention distribution and AST token distances for better initialization. In order to adapt our approach to more scenarios, we implement a PEL framework for CodePTMs with different architectures. Then, we evaluate *Pass-Tuning* on multiple tasks over eight different programming languages. Experiments show that our method can provide competitive (and in some tasks even better) performance while only modifying less than 1% of the parameters than full fine-tuning, substantially reducing the computational overhead for per-task/per-language when tuning CodePTM on downstream tasks. Our main contributions can be summarized as follows:

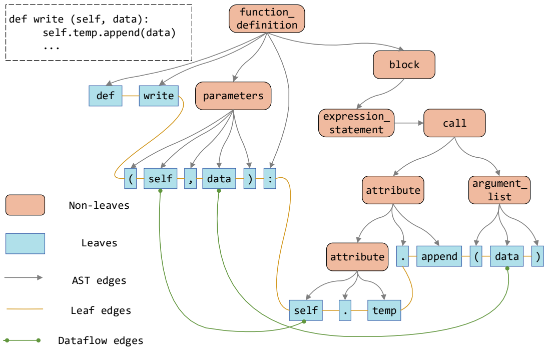
- We introduce *Pass-Tuning* framework for applying PEL on adapting CodePTMs to downstream tasks. It significantly reduces the tunable parameters in conjunction with the universality over models of different architectures.
- In *Pass-Tuning*, a plug-and-play graph attentional module is adopted to capture the structural information of source codes. We also design an ingenious code-retrieving method to improve our framework's performance by better parameter initialization based on token importance evaluation.
- As is depicted in Figure 1, our experiments on multiple tasks across different programming languages confirm the effectiveness of *Pass-Tuning*. Moreover, we can even achieve better results on some tasks than tuning all the parameters.

2 Preliminaries

This section introduces basic concepts and notations used in this paper.

2.1 Code Basics

Each program can be represented in two modals: the source code and the structure of code: Abstract Syntax Tree (AST), as is shown in Figure 3.



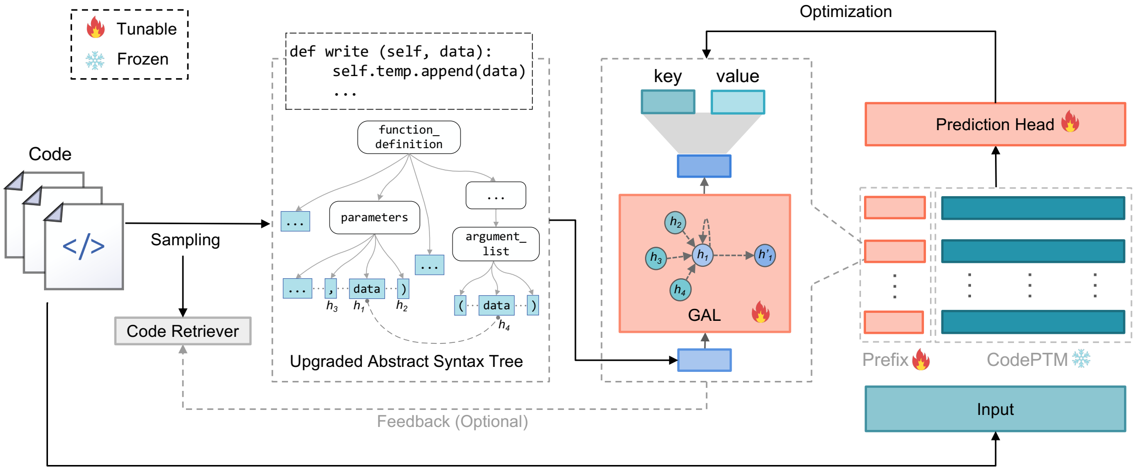
```
def write (self, data):
    self.temp.append(data)
    ...
```

Figure 3: A Python code snippet and its parsed U-AST, with connected Dataflow edges and Leaf edges. (Best viewed in color.)

We use Tree-sitter[2] to parse source codes. AST contains rich structural information, while it has a tree structure that may cause long-range problems due to the long distance between leaf nodes. Inspired by Wang et al. (2020) and Zhu et al. (2022), We add data flow edges to enhance the connectivity of the AST and name it as U-AST (upgraded AST).

[1] We take these PEL methods from NLP as strong baselines, the results are shown in section 4.4 and section 4.5.

[2] github.com/tree-sitter



```
def write (self, data):
    self.temp.append(data)
    ...
```

Figure 2: An illustration of *Pass-Tuning* based on a CodePTM, *e.g.*, CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021) with Transformer architecture. Graph Attentional Layers (GALs) designed to capture source codes' structural information are prepended as prefixes to each layer of the backbone model. In the process of model tuning, the parameters of the backbone model are frozen, and only these prefixes are modified. Besides, the Code Retriever will help the GAT to get better initialization through token importance evaluation, as is stated in section 3.3. (Best viewed in color.)

2.2 Code-related Tasks

Language Models' ability to understand and generate programs can boost developers' productivity. In order to better evaluate models' capacity, the following tasks are proposed to foster machine learning research for code.

Code Understanding Code understanding tasks assess models' ability to understand codes and their relationships. The most typical one is clone detection (Svajlenko et al., 2014; Mou et al., 2016), which measures the semantic similarity between code snippets. Another representative code understanding task is defect detection (Zhou et al., 2019). It identifies whether source codes contain defects that may be vulnerable to attacks.

Code Generation Code generation tasks evaluate the capacity to tackle sequence-to-sequence generation problems of models. These tasks could be further divided into code-code and code-text. For code-code tasks, code translation involves translating a code snippet from one programming language to another (Nguyen et al., 2015). Code completion's target is to predict the following tokens based on a code context (Raychev et al., 2016; Allamanis and Sutton, 2013). Code refinement (Tufano et al., 2019) is designed for automatic bug-fixing for source code. And for code-text tasks, code summarization (Iyer et al., 2016; Alon et al., 2019) aims to generate comments (natural language) for function-level code snippets.

2.3 Prefix-Tuning

Prefix-Tuning (Li and Liang, 2021) is a lightweight alternative to Fine-Tuning. Instead of modifying all parameters of a language model (LM), Prefix-Tuning focuses on optimizing a continuous task-specific vector (prefix) while keeping the LM's parameters frozen. This technique reduces the cost of adapting LM to downstream tasks and eliminates the necessity to store a full copy for each task.

2.4 Graph Attention Network (GAT)

Recently, graph neural network (Kipf and Welling, 2017) has emerged as a promising method for processing graph-structured data. GAT (Graph Attention Network) (Veličković et al., 2018) is a convolution-style graph neural network that leverages the attention mechanism for homogeneous graphs, which includes only one type of node or edge. It first computes the attention coefficient between different nodes and then produces the new feature of each node with neighborhoods' features through aggregation as output.

3 Pass-Tuning

In this section, we formally introduce our method: *Pass-Tuning* at length. An overview of *Pass-Tuning* is presented in Figure 2.

3.1 Overview

Compared with natural language, the most crucial feature of code is that it has its own structure. Directly migrating the PEL methods in NLP to the code domain might seriously impair the performance of several downstream tasks. Thus, we aim to pursue the cost-effectiveness of tuning while considering the code characteristics in this work. Inspired by Li and Liang (2021), using tunable prefixes to learn knowledge instead of modifying all parameters would be promising to develop a reliable PEL tuning method for CodePTMs. In practice, we design a Structure Knowledge Injector module (injector module) with graph attentional architecture to play the role of "tunable prefix". It will learn from the code structures instead of treating code as plain text, and then inject structure knowledge into the Transformer-based models.

As is presented in Figure 2, each (self-)attention layer of a Transformer block in the CodePTM is concatenated with a Structure Knowledge Injector. For the whole training procedure, only these modules are tuned while the parameters of the backbone model remain unchanged.

3.2 Structure Knowledge Injector

We hold the view that the problem of using PEL mentioned in section 3.1 is caused by a gap between texts and well-structured codes. Therefore, to make full use of the structural information, here we introduce Structure Knowledge Injector as a prefix for each layer of the model. For simplicity, we only show the key steps of injecting knowledge into attention modules.

Given a CodePTM G with encoder-only architecture that has n transformer layers, the l-th model layer is defined as $G(l)$. The task is to train Structure Knowledge Injector module f_l, as is demonstrated in Figure 4, to capture code structure information by the concatenation as follows:

$$\text{Concat}(f_l, G(l)) \tag{1}$$

Provided a code-related task dataset T with M samples, each code snippet T_m is first parsed into AST and then transformed into U-AST through the same process as section 2.1, notated as $S(T_m)$. After that, an $r \times r$ adjacency matrix $A_m^{r\times r}$ is constructed based on the connected edges in $S(T_m)$. Then, $A_m^{r\times r}$ will be the input of the GALs, and each node feature is represented as h_i.

Through the computation of attention coefficient α_{ij} and aggregation, as is shown in Equation 2, we get the updated representation h'_i, where $\mathcal{N}_i$ is the neighborhood of node i in the graph.

$$h'_i = \sigma(\sum_{j\in\mathcal{N}_i} \alpha_{ij} W h_j) \tag{2}$$

And given the input hidden states of the model as three vectors: Queries, Keys, and Values by projection matrices W_l^q, W_l^k, and W_l^v respectively.

$$W'k_l = \text{Concat}(f_l(h'_i), W_l^k) \tag{3}$$

$$W'v_l = \text{Concat}(f_l(h'_i), W_l^v) \tag{4}$$

After the concatenation, the original projection matrices will be equipped with the knowledge from the injector module in the specific layer l, represented as $W'_l v$ and $W'_l v$.

In the training stage, the cross-entropy loss will be computed based on the training sample T_m.

$$\sum_{l=1}^{n} loss_l \leftarrow \sum_{l=1}^{n} L_{T_m}(f_l, G(l)) \tag{5}$$

and each f_l will be optimized by backpropagation and $G(l)$ is frozen at all time.

$$f_l \leftarrow f_l - \alpha \nabla_{f_l} \sum_{l=1}^{n} loss_l \tag{6}$$

where α is the learning rate.

3.3 Code Retriever

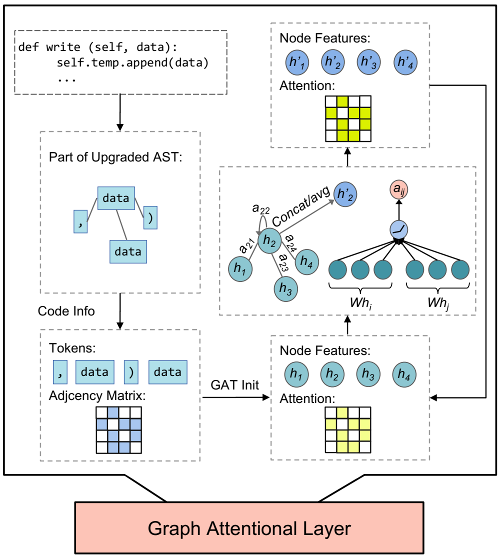
Initialization is a crucial factor in the graph neural networks' performance (Abboud et al., 2021). Inspired by previous works that leverage a reparameterization process for robustness, we employed a two-stage initialization strategy for *Pass-Tuning* that take backbone model, task, and code token type into account. Given m code snippets represented as S, each sample $S(i)$ is given a score $R(S(i))$ based on the relationship between token-level attention from the model and token distance of the U-AST (Chen et al., 2022). Then, this parsed U-AST first fed into the GAT as initialization. It is noteworthy that we sample the code snippet for initialization as follows:

$$P\left(S(i)\right) = \frac{\log|R(S(i))| + \delta}{\sum_{\tilde{k}=1}^{m} \log|R(S(\tilde{k}))| + \delta} \tag{7}$$

It is worth noticing that we regard codes as documents and first use BM25 (Stephen et al., 1994) to rank the input codes for the cold-start scenario.

3.4 Adaptation for Different Scenarios

Since there exist two kinds of CodePTMs: encoder-only and encoder-decoder, we implement two sets of *Pass-Tuning* for them respectively.



```
def write (self, data):
     self.temp.append(data)
     ...
```

Figure 4: An illustration of Injector modules with graph attentional architecture employed as tunable prefixes in *Pass-Tuning*, The input codes are parsed into U-ASTs for constructing adjacency matrices, which are fed into the GAT for the computation of the attention coefficient and aggregation.

3.4.1 Code Generation

For the code generation tasks, the knowledge injection process is similar to the description in section 3.2 for all CodePTMs.

3.4.2 Code Understanding

For the code understanding tasks that can be abstracted into classification tasks, we take different strategies for knowledge injection.

Encoder-Only. For CodePTMs like CodeBERT and GraphCodeBERT that only have transformer encoders, the knowledge of code structure is injected into the encoder modules layer-wise.

Encoder-Decoder. For models with Encoder-Decoder architecture, *e.g.*, CodeT5 and PLBART, we inject the knowledge provided by the GAL into each layer at the encoder modules, decoder modules, and cross-attention modules.

4 Experiments

In this section, we conduct extensive experiments to evaluate *Pass-Tuning* and compare it against full fine-tuning and strong baselines.

4.1 Datasets and Metrics

We evaluate *Pass-Tuning* on the following six datasets covering eight programming languages for both code understanding and generation.

Code Understanding For code understanding tasks, we employ BigCloneBench (Svajlenko et al., 2014) for clone detection and Devign (Zhou et al., 2019) for defect detection. The Accuracy and F1-score are reported for the two tasks respectively.

Code Generation For Code-Text generation, we use CodeSearchNet (Husain et al., 2019) dataset for code summarization and smoothed BLEU-4 (Lin and Och, 2004) as the evaluation metric. And for code generation, CONCODE dataset (Iyer et al., 2018) is involved with exact match (EM), the BLEU score (Papineni et al., 2002), and CodeBLEU (Ren et al., 2020) as evaluation metrics.

Then, for Code-Code generation, we first use Java-C# dataset (Nguyen et al., 2015) for code translation, we report the exact match accuracy (EM) and the BLEU score (Papineni et al., 2002). Secondly, we employ two Java datasets provided by Tufano et al. (2019): *Refine Small* and *Refine Medium* for code refinement tasks. BLEU-4 and EM are used for evaluation. Details of these four datasets are listed in Appendix A.

4.2 Experimental Setup

Pre-trained Language Models We select six representative pre-trained language models in our experiment, including five CodePTMs: GraphCodeBERT (Guo et al., 2021), PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021), and UniXcoder (Guo et al., 2022).

Experimental Details For a fair comparison, we adopt CodePTMs with the same number of transformer layers. To be specific, we choose GraphCodeBERT-base and UniXcoder-base from microsoft[3], PLBART-base from uclanlp[4], and CodeT5-base from Salesforce[5] as our model backbones. Then, we set the max length [6] of the U-AST token sequence fed into the graph attentional architecture as 32 and 64, represented by Pass-Tuning(32) and Pass-Tuning(64) in the following experiments. We choose Adam optimizer (Kingma and Ba, 2015) with a warm-up rate of 1,000 steps.

[3] https://huggingface.co/microsoft

[4] https://huggingface.co/uclanlp/plbart-base

[5] https://huggingface.co/Salesforce/codet5-base

[6] We find that the average length of the top K training samples of each task differs and falls within a range between 32 and 64. As such, we have covered both 32 and 64 as the lengths for the input code snippets. If the snippets exceed the token length, they are truncated; if they are shorter, then they are padded.

All the experiments are implemented by PyTorch 1.5.1, and models are trained and evaluated with 4 interconnected NVIDIA GTX 3090 GPUs. More details of the experimental implementation and hyperparameter settings of CodeBERT, GraphCodeBERT, and UniXcoder are given in Appendix B.

In our experiments, we aim to answer the following research questions: 1) As a PEL method, whether *Pass-Tuning* can achieve overall comparative performance while significantly diminishing the number of tunable parameters? 2) Can *Pass-Tuning* obtain surpassing full fine-tuning performance on some downstream tasks? 3) On which kind of code-related tasks *Pass-Tuning* excel? Do the tasks' characteristics or the CodePTMs' own properties cause this phenomenon?

4.3 Baselines

To further demonstrate the effectiveness of *Pass-Tuning*, we set the following PEL methods from NLP as strong baselines: BitFit (Ben Zaken et al., 2022), Adapter[7] (Houlsby et al., 2019), and Prefix-Tuning (Li and Liang, 2021). These methods all require minor modifications of parameters and keep the backbone model frozen. Since there are no ready-made implementations of these baseline tuning methods that can be directly applied to code representation learning, we implement them from scratch based on encoder-only and encoder-decoder architectures for various CodePTMs.

4.4 Performance of Code Generation

In this section, We evaluate *Pass-Tuning* on four generation tasks with different CodePTMs as backbones. The results of code summarization on six programming languages' bimodal[8] data of CodeSearchNet (CSN) are in Table 1, and then we make the following observations. 1) Even if we only tune less than 1% of the model parameters, *Pass-Tuning* can still reach state-of-the-art (SOTA) performance based on CodeT5 and PLBART in summarizing code for three and two subsets of languages respectively. 2) The experiments show that *Pass-Tuning* can outperform all previous PEL methods when using PLBART as the backbone model. 3) For CodeT5, compared to full fine-tuning and Adapter (with relatively richer parameters), there exists only marginal performance gaps. 4) In terms of absolute performance on code summarization, CodeT5 performs slightly better. We take the view that one potential reason for this is that CodeT5 utilizes the unimodal part of CSN during the pretraining stage, whereas PLBART does not.

From Table 2, employing *Pass-Tuning* with PLBART can reach SOTA results in EM, and comparable performance on other metrics for generation tasks. For using CodeT5 as the backbone, there exists a gap between *Pass-Tuning* and fine-tuning. However, our method can surpass all previous PEL methods although we have much lesser parameters.

Table 3 demonstrate the performance of code translation and code refinement. For the translation task, it is self-evident that the number of parameters is positively correlated with the performance, and it's hard for all PEL methods to conduct this seq2seq task since the number of parameters constraints models' capability. Nevertheless, there is no denying that *Pass-Tuning* outperforms both BitFit and P-Tuning V2 by a significant margin. The situation is different for code refinement, whether we use CodeT5 or PLBART as backbones, *Pass-Tuning* can surpass the performance of full fine-tuning on at least one of the metrics.

Moreover, Table 14 to Table 16 enumerate additional generation tasks using GraphCodeBERT and UniXcoder as backbones.

4.5 Performance of Code Understanding

The results of code understanding tasks: defect detection and clone detection, are shown in Table 4. We make the following observations that 1) For the CodeT5 backbone, by comparing with other strong PEL baselines, *Pass-Tuning* can reach similar performance but lower the cost of tuning further. 2) When using PLBART as the backbone, we can see that *Pass-Tuning* demonstrates excellent performance on Clone detection that surpasses full fine-tuning. 3) For the defect detection task, tuning all the parameters still exhibits dominant performance. We hold the view that this phenomenon is caused by the difficulty in understanding the semantics of defects in context, and there exists a bottom-line number of parameters that makes the representation learning of code defects feasible. Moreover, the amount of data in the Devign dataset is much smaller than that in BigCloneBench, which might lead to overfitting problems. Similarly, Table 17 lists the results of conducting the same experiments

[7] Concurrent with our work, Ayupov and Chirkova (2022) apply LoRA and adapters to CodeT5 and PLBART.

[8] Bimodal data means parallel data of NL-PL (Natural Language-Programming Langauge) pairs and unimodal stands for pure codes without NL texts.

Methods	Params	Ruby	JavaScript	Go	Python	Java	PHP	Overall
CodeT5								
Fine-Tuning	224M	15.24	**16.21**	19.53	19.90	20.34	**26.12**	19.56
BitFit	0.001M	1.75	1.05	1.19	2.15	1.40	0.97	1.42
Adapter	14.22M	15.45	16.04	19.40	20.22	20.19	24.90	19.37
P-Tuning V2	0.633M	15.22	15.63	18.92	20.18	19.71	25.43	19.18
Pass-Tuning(32)	3.068M	15.30	15.70	19.51	20.24	20.27	25.57	19.43
Pass-Tuning(64)	3.068M	**15.47**	15.92	**19.74**	**20.48**	**20.35**	25.83	**19.63**
PLBART								
Fine-Tuning	139M	13.97	14.13	**18.10**	**19.33**	**18.50**	23.56	17.93
BitFit	0.126M	8.38	5.21	12.18	12.78	9.08	11.57	9.87
Adapter	7.11M	3.91	2.05	11.62	15.20	13.54	**24.01**	11.72
P-Tuning V2	0.329M	13.43	13.93	17.18	18.16	16.96	23.21	17.15
Pass-Tuning(32)	1.879M	**14.30**	14.29	18.00	19.15	17.78	23.72	17.87
Pass-Tuning(64)	1.879M	14.23	**14.52**	17.84	19.13	18.30	23.82	**17.97**

Table 1: Performance on Code Summarization task.

Methods	Params	BLEU	EM	CodeBLEU
CodeT5				
Fine-Tuning	224M	**40.73**	22.25	**43.20**
BitFit	0.001M	0.13	0.00	12.36
Adapter	14.22M	33.28	21.20	39.81
P-Tuning V2	0.633M	28.87	19.50	32.02
Pass-Tuning(32)	3.068M	33.60	22.15	36.97
Pass-Tuning(64)	3.068M	34.12	**22.75**	37.38
PLBART				
Fine-Tuning	139M	**32.42**	16.55	**35.39**
BitFit	0.126M	2.70	0.25	0.67
Adapter	7.11M	4.40	6.35	13.72
P-Tuning V2	0.329M	26.92	16.75	30.65
Pass-Tuning(32)	1.879M	30.33	19.75	33.96
Pass-Tuning(64)	1.879M	29.86	**19.88**	33.01

Table 2: Performance on Code Generation task.

Methods	Params	Java to C#		C# to Java		Refine Small		Refine Medium	
		BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
CodeT5									
Fine-Tuning	224M	**84.15**	**65.30**	**79.12**	**66.40**	77.39	**21.35**	91.04	**7.82**
BitFit	0.001M	0.25	0.00	0.24	0.00	1.28	0.00	5.14	0.00
Adapter	14.22M	75.43	52.40	73.10	57.70	77.41	18.58	91.01	3.61
P-Tuning V2	0.633M	59.86	33.70	57.10	41.00	78.99	4.56	91.02	0.79
Pass-Tuning(32)	3.068M	75.46	52.30	75.38	60.70	79.51	11.85	**91.22**	5.72
Pass-Tuning(64)	3.068M	72.86	48.70	73.19	58.40	**79.69**	12.55	91.06	5.45
PLBART									
Fine-Tuning	139M	**77.05**	**62.60**	**79.29**	**62.80**	73.32	**12.71**	83.88	4.24
BitFit	0.126M	16.48	0.10	17.43	0.90	74.08	1.45	85.41	0.42
Adapter	7.11M	66.72	42.10	68.70	51.00	73.58	10.90	84.72	3.12
P-Tuning V2	0.329M	22.87	1.00	48.08	33.80	73.87	2.07	73.58	0.03
Pass-Tuning(32)	1.879M	64.95	44.00	64.14	52.20	74.37	5.07	86.38	6.09
Pass-Tuning(64)	1.879M	64.68	41.90	63.38	49.70	**74.45**	5.01	**87.26**	**6.24**

Table 3: Performance on Code Translation & Code Refinement Tasks.

Methods	Params	Defect	Clone
		Accuracy	F1
CodeT5			
Fine-Tuning	224M	**64.35**	**94.97**
BitFit	1.183M	55.05	69.52
Adapter	15.40M	59.74	94.47
P-Tuning V2	1.182M	54.61	79.83
Pass-Tuning(32)	0.591M	58.09	93.16
Pass-Tuning(64)	0.591M	56.83	88.25
PLBART			
Fine-Tuning	139M	**62.27**	92.85
BitFit	1.308M	56.30	92.42
Adapter	8.29M	61.60	92.74
P-Tuning V2	1.182M	53.81	75.88
Pass-Tuning(32)	0.591M	56.41	**93.41**
Pass-Tuning(64)	0.591M	56.09	92.75

Table 4: Performance on Code Clone Detection & Code Defect Detection Tasks.

on GraphCodeBERT and UniXcoder.

4.6 Ablation Study

We choose CodeT5 backbone[9] for ablation study.

Effectiveness of Injector Module. *Pass-Tuning* employs a Structure Knowledge Injector with GAT network architecture as the tunable prefixes in order to capture code structure. The results of ablation studies are shown in Table 5, and we can see that the two variants (removing the injector module and using GCN replacement) of prefix design lead to worse performance on code summarization tasks in most languages, and can no longer reach surpassing fine-tuning results. which indicates the effectiveness of our designation.

Effectiveness of Code Retriever. We design a code retriever in section 3.3 for better GAT initialization in *Pass-Tuning*. To confirm its effectiveness, we replace it by selecting code snippets randomly. Based on the observation in Table 5, it is clear that using random initialization causes slightly worse results among all programming languages.

Effectiveness of Modeling AST To demonstrate the necessity of explicitly modeling AST, we conduct comparative experiments for all involved tasks. As showcased in Table 12 and Table 13, modeling using AST achieves better results across all tasks compared to directly employing the code sequences.

4.7 Efficiency Analysis

Our approach offers a new path to get out of the cost-performance dilemma of using CodePTMs. To quantify our efficiency, Table 6 list the tunable parameters of all tuning strategies covered in this paper. Compared with PEL baselines, the tuning cost of our method is lower or at the same level, while *Pass-Tuning* evidently outperforms these methods across all the tasks. When compared with full fine-tuning, our approach can achieve overall comparative results by modifying less than 1% of the parameters, which further proves the cost-effectiveness of *Pass-Tuning*.

5 Related Works

Code Pre-trained Language Models. Transformer based models (Vaswani et al., 2017; Devlin et al., 2019) significantly advance the performance

[9] Experiments on other models are listed in Appendix C.3

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
Full-Tuning	15.24	**16.21**	19.53	19.90	20.34	**26.12**	19.56
Pass-Tuning(64)	**15.47**	15.92	**19.74**	**20.48**	**20.35**	25.83	**19.63**
w/o. Code Retriever	15.24	15.70	19.38	20.01	20.32	25.41	19.34
w/o. Knowl. Injector	15.22	15.63	18.92	20.18	19.71	25.43	19.18
with GCN	15.41	15.83	19.44	20.20	19.76	25.48	19.35

Table 5: Ablation study of *Pass-Tuning* based on CodeT5 in code summarizaton tasks.

Methods	PLBART		CodeT5	
	CLS MB	GEN MB	CLS MB	GEN MB
Fine-Tuning	139M	139M	224M	224M
Bitfit	1.308M	0.126M	1.183M	0.001M
Adapter	8.29M	7.11M	15.40M	14.22M
P-Tuning V2	1.182M	0.329M	1.182M	0.633M
Pass-Tuning	0.591M	1.879M	0.591M	3.068M

Table 6: Comparison of the number of learnable parameters between *Pass-Tuning* and other PEL strategies.

of various natural language processing (NLP) tasks. With the brilliant achievements of these pre-trained models in the field of NLP, recent works attempt to apply them to codes in order to boost the development of software engineering and code intelligence. CodeBERT (Feng et al., 2020) is pre-trained on NL-PL data. It follows RoBERTa (Liu et al., 2019), which uses multi-layer bidirectional Transformers as the architecture with masked language modeling (MLM) and replaced token detection (RTD) (Yang et al., 2019) pre-training tasks. GraphCodeBERT (Guo et al., 2021) is a variant of CodeBERT that integrates the data-flow to facilitate code representation learning. PLBART (Ahmad et al., 2021) is based on BART (Lewis et al., 2020) with a denoising objective in pre-training. CodeT5 (Wang et al., 2021) utilizes the T5 (Raffel et al., 2020) architecture, leveraging code semantics through identifier tokens and applying multi-task learning (MTL) to a unified framework. UniXcoder (Guo et al., 2022) adapts the UniLM (Dong et al., 2019) architecture and is pre-trained on unified cross-modal data to support both code understanding and generation tasks.

Parameter-Efficient Learning. The idea of Parameter-Efficient Learning (PEL) (Ding et al., 2023) is to optimize a small portion of parameters while keeping the model backbone frozen. As long as the data is sufficient, PEL can reach comparable performance to full model tuning (He et al., 2022). Houlsby et al. (2019) insert task-specific neural modules called *adapters* into the transformer-based models, and only these *adapters* are trained during fine-tuning. Mahabadi et al. (2021) propose a better trade-off between performance and the number of tunable parameters through combining adapters and low-rank optimization. Inspired by the success of prompting methods that guide large language models (Brown et al., 2020) through textual prompt (Liu et al., 2021), Prefix-Tuning (Li and Liang, 2021) concatenate tunable prefix vectors with the keys and values of each attention layer inside the model, and only train these soft prompts when being fine-tuned. Then, it is further simplified by prompt-tuning (Lester et al., 2021) that only prepends to the input in the first layer. After that, BitFit (Ben Zaken et al., 2022) employs a sparse method that only tunes the bias terms of the model. LoRA (Hu et al., 2022) has utilized low-rank matrices for approximating parameter updates. Recently, Wang et al. (2023) employ uncertainty estimation in PEL for cost-effective self-training. Compared to full fine-tuning, these techniques have all demonstrated competitive performance on a series of NLP tasks while only updating less than 10% of the model parameters.

6 Conclusion

In this work, we present a novel *Pass-Tuning* framework for adapting CodePTMs with different architectures to downstream tasks. To our best knowledge, we are the first to propose a lightweight alternative to full fine-tuning in the domain of code representation learning and significantly reduce the number of trainable parameters for a series of tasks. Besides, we consider source codes' structure information and employ tunable prefixes with GAL architecture. Moreover, a distinct initialization strategy is designed for prefixes to exploit the characteristics of codes. Extensive experiments have demonstrated the effectiveness of our method and *Pass-Tuning* can steadily outperform all PEL baselines. In comparison with full Fine-Tuning, we only modify less than 1% of the parameters while achieving competitive results. Further ablation studies indicate the robustness and rationality of our strategies. In our future work, we will explore more Parameter-Efficient learning approaches for code and move one step forward to further utilize code structure information for tuning CodePTMs.

Limitations

Our method has mainly two limitations. First, when performing an NL-PL task such as code summarization, our approach cannot provide additional information to natural language comments. Secondly, we obtain ASTs with high connectivity by adding data-flow edges and using them as input. However, different token types have different salience for various CodePTMs (Chen et al., 2022). Thus, feature engineering for AST may further enhance the performance, which we leave as future work.

Broader Impact and Ethical Consideration

To our best knowledge, we are the first to propose a Parameter-Efficient tuning method for CodePTMs while considering code structure. *Pass-Tuning* will not introduce additional model bias and does not involve misuse of code and natural language comments. Our approach significantly reduces the computation and operational costs when applying pre-trained models to downstream tasks. We believe it will be beneficial to the NLP community.

Acknowledgement

This work has been supported by the National Natural Science Foundation of China under Grant No.U1911203, and the National Natural Science Foundation of China under Grant No.62377012. And the authors would like to thank all the anonymous reviewers for their constructive and insightful comments on this paper.

References

Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. 2021. The surprising power of graph neural networks with random node initialization. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2112–2118. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. 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*, pages 2655–2668, Online. Association for Computational Linguistics.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE.

Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Shamil Ayupov and Nadezhda Chirkova. 2022. Parameter-efficient finetuning of transformers for source code. In *Efficient Natural Language and Speech Processing (ENLSP-II) workshop of the 36th Conference on Neural Information Processing Systems (NeurIPS 2022)*. Curran Associates, Inc.

Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. 2022. 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)*, pages 1–9, Dublin, Ireland. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring software naturalness through neural language models.

Nuo Chen, Qiushi Sun, Renyu Zhu, Xiang Li, Xuesong Lu, and Ming Gao. 2022. CAT-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure. In *EMNLP 2022*, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, Volume 1 (Long and Short Papers)*, pages

4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. 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*, pages 13042–13054.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2022. Towards a unified view of parameter-efficient transfer learning. In *International Conference on Learning Representations*.

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

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880. Association for Computational Linguistics.

Xiang Lisa Li and Percy Liang. 2021. 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 (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *CoRR*, abs/2107.13586.

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

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. 2021. Compacter: Efficient low-rank hypercomplex adapter layers. In *Advances in Neural Information Processing Systems*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596. IEEE.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, pages 731–747.

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

Robertson Stephen, Walker Steve, Jones Susan, Hancock-Beaulieu Micheline, and Gatford Mike. 1994. Okapi at trec.

Qiushi Sun, Nuo Chen, Jianing Wang, Xiang Li, and Ming Gao. 2023. Transcoder: Towards unified transferable code representation learning inspired by human skills.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, 28(4).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *International Conference on Learning Representations*.

Jianing Wang, Qiushi Sun, Nuo Chen, Chengyu Wang, Jun Huang, Ming Gao, and Xiang Li. 2023. Uncertainty-aware parameter-efficient self-training for semi-supervised language understanding.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Renyu Zhu, Lei Yuan, Xiang Li, Ming Gao, and Wenyuan Cai. 2022. A neural network architecture for program understanding inspired by human behaviors. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5142–5153, Dublin, Ireland. Association for Computational Linguistics.

A Statistics of Datasets

A.1 Code Generation & Translation

Dataset	Language	Training	Dev	Testing
CONCODE	NL - Java	100,000	2,000	2,000
CodeTrans	Java - C#	10,300	500	1,000

Table 7: CONCODE (Iyer et al., 2018) and CodeTrans (Nguyen et al., 2015) datasets statistics for code generation and code translation tasks.

A.2 Code Refinement

Dataset	Language	Training	Dev	Testing
Refinement$_{\text{Small}}$	Java	46,680	5,835	5,835
Refinement$_{\text{Medium}}$	Java	52,364	6,545	6,545

Table 8: Code Refinement (Tufano et al., 2019) dataset statistics.

A.3 Defect Detection & Clone Detection

Dataset	Language	Training	Dev	Testing
BigCloneBench	Java	900K	416K	416K
Devign	C	21K	2.7K	2.7K

Table 9: BigCloneBench (Svajlenko et al., 2014) and Devign (Zhou et al., 2019) datasets statistics for Clone detection and Defect Detection tasks.

A.4 Code Summarization

Language	Training	Dev	Testing
Go	167,288	7,325	8,122
Java	164,923	5,183	10,955
JavaScript	58,025	3,885	3,291
PHP	241,241	12,982	14,014
Python	251,820	13,914	14,918
Ruby	24,927	1,400	1,261

Table 10: CodeSearchNet (Husain et al., 2019) data statistics for the code summarization task.

Hyperparameter	**value**
Batch Size	8,16,32
Learning Rate	{8e-6, 2e-5, 1e-4, 5e-4}
Max Source Length	{130, 240, 256, 320, 512}
Max Target Length	{3, 120, 150, 240, 256, 512}
GAL max Length	{32, 64}
Epoch	{2, 30, 50, 100}
Smoothing Factor δ	{0.05, 0.1}

Table 11: Hyperparameters for Pass-Tuning

B Implementation Details

C Supplementary Experiments

In this section, we provide additional experiments that are not demonstrated in section 4.

C.1 Supplementary Analysis of Explicitly Modeling AST

To demonstrate the necessity of explicitly modeling the AST, we design additional experiments to compare its performance improvement relative to simply modeling code tokens (directly constructing sequences in the order of code). In the experiments, we set the maximum sequence length to 32.

Tasks	Clone	Defect	Java → C#		C# → Java	
Metrics	F1	Acc	BLEU	EM	BLEU	EM
CodeT5						
Token Modeling	92.97	55.82	74.60	52.10	74.64	59.50
AST Modeling	**93.16**	**58.09**	**75.46**	**52.30**	**75.38**	**60.70**
PLBART						
Token Modeling	93.02	55.60	64.65	41.20	**64.69**	**52.80**
AST Modeling	**93.41**	**56.41**	**64.95**	**44.00**	64.14	52.20

Table 13: Effectiveness of explicitly modeling AST for clone detection, defect detection, and code translation tasks.

C.2 Supplementary Experiments on More Backbones

This section provides supplementary experiments on more backbones in Table 14, Table 15, Table 16, and Table 17.

C.3 Supplementary Ablation Studies

This section provides supplementary experiments of detailed ablation studies in Table 18, Table 19, Table 20, and Table 21.

Languages	Ruby	JavaScript	Go	Python	Java	PHP	Overall
CodeT5							
Token Modeling	15.17	15.64	19.33	19.92	20.08	25.52	19.27
AST Modeling	**15.30**	**15.70**	**19.51**	**20.24**	**20.27**	**25.57**	**19.43**
PLBART							
Token Modeling	14.28	13.17	17.33	17.08	17.41	23.54	17.13
AST Modeling	**14.30**	**14.29**	**18.00**	**19.15**	**17.78**	**23.72**	**17.87**

Table 12: Effectiveness of explicitly modeling AST for code summarization.

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
GraphCodeBERT							
Fine-Tuning	11.94	**15.05**	**18.43**	**19.27**	18.72	25.37	18.13
Pass-Tuning(32)	**12.94**	14.19	18.34	19.19	**18.75**	**25.51**	**18.15**
UniXcoder							
Fine-Tuning	14.66	**15.39**	19.01	19.75	**20.19**	**26.08**	**19.18**
Pass-Tuning(32)	**14.69**	14.82	**19.84**	**20.10**	19.42	24.97	18.97

Table 14: Performance on Code Summarization task based on GraphCodeBERT and UniXcoder.

Methods	BLEU	EM	CodeBLEU
GraphCodeBERT			
Fine-Tuning	**31.08**	**18.35**	**35.00**
Pass-Tuning(32)	30.58	17.10	26.05
UniXcoder			
Fine-Tuning	**31.35**	18.80	**35.41**
Pass-Tuning(32)	29.72	**19.23**	34.69

Table 15: Performance on Code Generation tasks based on GraphCodeBERT and UniXcoder.

Methods	Java to C#		C# to Java		Refine Small		Refine Medium	
	BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
GraphCodeBERT								
Fine-Tuning	**74.69**	**54.10**	**69.94**	**57.40**	78.44	**16.13**	90.68	**7.74**
Pass-Tuning(32)	63.71	45.40	58.99	49.20	**79.80**	13.22	**90.93**	5.44
UniXcoder								
Fine-Tuning	**77.21**	**61.00**	**72.37**	**62.30**	**64.05**	**14.34**	75.12	5.78
Pass-Tuning(32)	65.90	48.20	62.76	46.70	63.75	7.78	**76.44**	**5.92**

Table 16: Performance on Code Translation & Code Refinement Tasks.

Methods	Defect	Clone
	Accuracy	F1
GraphCodeBERT		
Fine-Tuning	**62.88**	**95.30**
Pass-Tuning(32)	57.28	93.14
UniXcoder		
Fine-Tuning	**62.34**	**91.36**
Pass-Tuning(32)	54.28	87.74

Table 17: Performance on Code Clone Detection & Code Defect Detection Tasks.

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
CodeT5							
Fine-Tuning	15.24	**16.21**	19.53	19.90	20.34	**26.12**	19.56
Pass-Tuning(64)	**15.47**	15.92	**19.74**	**20.48**	**20.35**	25.83	**19.63**
w/o. Code Retriever	15.24	15.70	19.38	20.01	20.32	25.41	19.34
w/o. GAT Module	15.22	15.63	18.92	20.18	19.71	25.43	19.18
with GCN	15.41	15.83	19.44	20.20	19.76	25.48	19.35
PLBART							
Fine-Tuning	13.97	14.13	**18.10**	**19.33**	**18.50**	23.56	17.93
Pass-Tuning(64)	**14.23**	**14.52**	17.84	19.13	18.30	**23.82**	**17.97**
w/o. Code Retriever	14.00	14.42	17.31	18.72	18.28	23.65	17.73
w/o. GAT Module	13.43	13.93	17.18	18.16	16.96	23.21	17.15
with GCN	14.10	14.40	17.91	18.85	17.22	23.58	17.71

Table 18: Detailed comparison of code summarization tasks based on PLBART and CodeT5 with variant *Pass-Tuning* implementation

Methods	BLEU	EM	CodeBLEU
CodeT5			
Fine-Tuning	**40.73**	22.25	**43.20**
Pass-Tuning(64)	34.12	**22.75**	37.38
w/o. Code Retriever	30.51	20.85	33.92
w/o. GAT Module	28.87	19.50	32.02
with GCN	31.51	20.65	35.11
PLBART			
Fine-Tuning	**32.42**	16.55	**35.39**
Pass-Tuning(32)	30.33	**19.75**	33.96
w/o. Code Retriever	27.42	18.70	30.97
w/o. GAT Module	26.92	16.75	30.95
with GCN	27.47	18.94	31.08

Table 19: Detailed comparison of code generation tasks based on PLBART and CodeT5 with variant *Pass-Tuning* implementation

Methods	Java to C#		C# to Java		Refine Small		Refine Medium	
	BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
CodeT5								
Fine-Tuning	**84.15**	**65.30**	**79.12**	**66.40**	77.39	**21.35**	91.04	**7.82**
Pass-Tuning(32)	75.46	52.30	75.38	60.70	79.51	11.85	**91.22**	5.72
w/o. Code Retriever	64.19	49.60	63.60	49.00	79.44	7.69	91.02	1.70
w/o. GAT Module	59.86	33.70	57.10	41.00	78.99	4.56	91.02	0.79
with GCN	69.37	43.80	67.91	51.80	**79.59**	8.68	91.10	2.46
PLBART								
Fine-Tuning	**77.05**	**62.60**	**79.29**	**62.80**	73.32	**12.71**	83.88	4.24
Pass-Tuning(32)	64.95	44.00	64.14	52.20	74.37	5.07	**86.38**	**6.09**
w/o. Code Retriever	55.55	29.10	55.67	38.60	74.37	3.18	57.00	0.09
w/o. GAT Module	22.87	1.00	48.08	33.80	73.87	2.07	73.58	0.03
with GCN	52.20	27.80	56.64	42.30	**74.63**	3.48	79.04	0.05

Table 20: Detailed comparison of code translation and refinement tasks based on PLBART and CodeT5 with variant *Pass-Tuning* implementation

Methods	Defect	Clone
	Accuracy	F1
CodeT5		
Fine-Tuning	**64.35**	**94.97**
Pass-Tuning(32)	58.09	93.16
w/o. Code Retriever	54.98	92.20
w/o. GAT Module	54.61	79.83
with GCN	54.94	92.96
PLBART		
Fine-Tuning	**62.27**	92.45
Pass-Tuning(32)	56.41	**93.41**
w/o. Code Retriever	53.66	77.46
w/o. GAT Module	53.81	75.88
with GCN	53.37	78.37

Table 21: Detailed comparison of code understanding tasks based on PLBART and CodeT5 with variant *Pass-Tuning* implementation

Dataset	CodeTrans				Bugs2Fix			
Task	Code Translation				Code Refinement			
Language	Java (java-cs)		CSharp(cs-java)		Java(small)		Java(medium)	
Sample Size	10300		10300		46680		52364	
Rank	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency
1	public	10300	identifier	10300	identifier	46680	identifier	52364
2	identifier	10300	{	10300	(	46680	;	52364
3	;	10300	}	10296	)	46680	(	52364
4	(	10300	;	10294	;	46674	)	52364
5	)	10300	(	10293	{	45800	{	52360
6	{	10300	)	10293	}	45800	}	52360
7	}	10278	public	10286	.	41536	.	51122
8	type_identifier	8693	.	8020	public	35972	type_identifier	46686
9	return	7329	return	7280	type_identifier	34420	=	42125
10	=	6174	=	6866	void_type	27452	public	37333
11	.	4967	,	5933	,	20905	,	36045
12	,	3189	new	5176	return	18961	void_type	30241
13	int	2447	predefined_type	4986	=	17800	if	28548
14	new	2315	virtual	4458	if	9985	return	24441

Table 22: Frequent token types for CodeTrans and Bugs2Fix datasets.

Dataset	CodeSearchNet											
Task	Code Summarization											
Language	Ruby		JavaScript		Go		Python		Java		PHP	
Sample Size	24927		58025		167288		251820		164923		241241	
Rank	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency	Token Type	Frequency
1	def	24927	function	58025	func	167288	def	251820	identifier	164923	function	241241
2	identifier	24927	(	58025	identifier	167288	identifier	251820	;	164923	name	241241
3	end	24149	)	58024	(	167288	(	251820	(	164923	(	241241
4	)	23963	{	58024	)	167288	)	251820	)	164923	)	241240
5	(	23961	identifier	57920	{	167288	:	251820	{	164923	{	241237
6	.	23694	}	57830	}	167256	.	240830	}	164923164893	}	241189
7	=	20458	;	56535	type_identifier	167124	,	231582	type_didentifier	155261	;	241177
8	,	19763	property_identifier	56230	.	160184	=	224000	.	154207	$	240579
9	constant	16005	.	55902	field_identifier	153297	"	170662	,	122881	->	210556
10	"	13367	,	49939	,	138470	]	134779	=	119607	=	207844
11	string_content	13281	=	49366	return	133661	[	134778	return	115325	return	193817
12	}	13156	return	40431	*	132700	if	134518	public	112995	,	188558
13	if	12390	if	39780	:=	107524	return	119020	if	96105	string_value	165950
14	]	11759	string_fragment	34220	package_identifier	101431	in	105395	new	79605	public	163542

Table 23: Frequent token types for CodeSearchNet dataset.

Dataset	BigCloneBench		Devign		CONCODE	
Task	Clone Detection		Defect Detection		Code Generation	
Language	Java		C		Java	
Sample Size	901028		21800		100000	
Rank	Token type	frequency	Token type	frequency	Token type	frequency
1	identifier	901028	identifier	21800	identifier	100000
2	;	901028	(	21800	type_identifer	99834
3	(	901028	)	21798	;	98005
4	)	901028	{	21783	.	78004
5	{	901028	;	21602	void_type	61779
6	}	900430	primitive_type	20988	>	33964
7	.	899241	*	20741	<	33911
8	type_identifier	895508	}	20352	boolean_type	33322
9	=	894986	type_identifier	20264	int	30689
10	new	898876	,	20056	,	25166
11	,	792560	=	19084	]	18274
12	string_literal	712764	number_liternal	17494	[	18272
13	try	665816	field_identifier	16811	long	11661
14	public	623123	->	16188	ERROR	10146

Table 24: Frequent token types for BigCloneBench, Devign, and CONCODE dataset.