Appendix

Anonymous Author(s)

The Appendix is structured as follows. In Section 1, we provide an introduction to the definition, implementation approaches, evaluated metrics, and experimental settings of four probing tasks discussed in the main text. In Section 2, we present definitions, evaluated datasets and metrics, and experimental settings for five different downstream tasks. Sections 3 and 4 are experiment results and analysis on UniXcoder and GraphCodeBERT, respectively.

1 PROBING TASKS

We design four probing tasks related to lexical, syntactical, semantic, and structural code properties. We introduce them one by one in detail as follows.

1.1 Lexical Probing

Lexical probing aims to measure how well contextual representations encode the lexical properties of source code. To achieve this, we first use the contextual representations of pre-trained code models as frozen features, then feed them to a linear classifier, and finally train it to predict the type of each code token. There are five code token types shown in Figure 1. We use a python library named *tokenize* [1] to tokenize source code and obtain the type of each code token. The linear classifier is the 768x5 matrix.

1.2 Syntactic Probing

Syntactic probing is designed to investigate how well contextual representations perceive the syntactic properties of source code. Follow the previous work [11], we first obtain the AST of the code snippet, then remove the values of terminal nodes, and finally flatten the AST using structural-based traverse [6] to obtain the AST-Only. The true pairs are constructed by pairing the codes with the corresponding parsed AST-Onlys, while the false pairs are constructed by pairing the codes with AST-Onlys parsed by other different codes. The linear classifier is a 768x768 matrix. It is used to map the representations of code snippets and AST-Only into a same latent space. The similarity of code and AST-only pairs are measure by the cosine similarity. The cosine similarity score larger than 0.5 indicates the code and AST-only are paired otherwise they are not paired.

1.3 Semantic Probing

Semantic probing is to examine the ability to identify code snippets with the same semantics but different implementations. Specifically, we use POJ-104 [15] dataset, which consists of 104 problems and 500 C/C++ implementations for each problem, as the evaluated dataset. The linear mapper is a 768x768 matrix which is trained to map semantically similar code snippets into similar embeddings.

[1] https://docs.python.org/3/library/tokenize.html

1.4 Structural Probing

We use the cyclomatic complexity prediction as the structural probing task to investigate how well contextual representations understand the structural property of source code. We use a popular python library named *radon* [2] to obtain the cyclomatic complexity value of one code snippet. The complexity value greater than five is labeled *Other*. Thus, the value of cyclomatic complexity value is in one of the six values ([1, 2, 3, 4, 5,*Other*]). The linear classifier is a 768x6 matrix and is trained to predict the cyclomatic complexity of one code snippet.

1.5 Evaluation Metrics

The results of lexical, syntactical, and structural probing are evaluated by accuracy. The results of semantic probing are evaluated by mean average precision (MAP) [17]. We introduce these two metrics as follows.

Accuracy is a widely used metric in classification tasks [10]. It is the ratio of correct predictions to total predictions. Mathematically, it is calculated by:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \times 100\% \quad (1)$$

MAP is a commonly used metric for evaluating information retrieval systems [17]. It takes into account both the ranking of relevant/targeted results and the number of relevant/targeted results returned. On the evaluation of semantic probing, it can be calculated by:

$$\text{MAP} = \frac{1}{C}\sum_{i=1}^{C}\frac{1}{n_i}\sum_{j=1}^{n_i}\text{Precision}(j) \times \text{rel}(j) \quad (2)$$

where, C is the number of input code snippets, and n_i is the number of code snippets returned for the i-th input code snippet. Precision(j) is the proportion of code snippets implementing the same semantic with the input code snippet among the top j returned results. rel(j) is an indicator function equaling 1 if the j-th returned code snippet implements the same semantic with the input code snippet, zero otherwise.

2 DOWNSTREAM TASKS

Our experiments are conducted on five diverse downstream tasks including code search, code detection, code summarization, code generation and line-level code completion. We introduce them one by one as follows.

2.1 Code Search

Code search aims to find the most semantically relevant code snippets for a given natural language query from open-source repositories such as GitHub or from a large local codebase. In practice, following the previous study [4], we use the average of last-layer

[2] https://github.com/rubik/radon

Type	Description
Identifier	Identifiers are usually named by developers including the variable name and API name.
Keyword	Keywords are the reserved words including *False, await, else, import, pass, None, break, except, in, raise, True, class, finally, is, return, and, continue, for, lambda, try, as, def, from, nonlocal, while, assert, del, global, not, with, async, elif, if, or, yield*
Operator	Operators include !=, %, %=, &, &=, (,), *, **, **=, *=, +, +=, ,, -, -=, ->, ., ..., /, //, //=, /=, :, :=, ;, <, <<, <<=, <=, =, ==, >, >=, >>, >>=, @, @=, [,], ^, ^=, {, |, |=, }, ~
Number	Like integer, decimal, etc.
String	The content between quotation marks.

Figure 1: An overview of five types of code tokens.

contextual representations as the overall representational vectors of the code snippet or query. The similarity of the code and query is measured by the vector distance such as cosine similarity. We fine-tune the pre-trained model to pull together the paired code snippets and queries and pull apart the unpaired ones.

We conduct the experiment on a widely-used CodeSearchNet [8] dataset with Python and Ruby languages. The training data contains code-query pairs. For test and validation set, the ground-truth codes are stored in *Candidate Codes*. Following previous studies [3, 5, 7], the model is to retrieve the correct code snippets from the *Candidate Codes* for the given queries when performing the evaluation.

We use commonly-used mean reciprocal rank (MRR) and top-k recall (R@k, k=1,5,10) as evaluated metrics. MRR is the average of reciprocal ranks of the correct code snippets for given queries Q. R@k measures the percentage of queries that the paired code snippets that exist in the top-k returned ranked lists. They are calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank_i}, \ \text{R@k} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\delta\left(Rank_i \le k\right) \tag{3}$$

where $Rank_i$ is the rank of the paired code snippet related to the i-th query. δ is an indicator function that returns 1 if $Rank_i \le k$ otherwise returns 0.

We use a pre-trained UniXcoder [4] to obtain the overall representation of code snippets and natural language queries. Following the experimental settings of UniXcoder, the maximum lengths of code snippets and natural language queries are set to 256 and 128, respectively. The maximum epoch is set to 10. The batch size is set to 128 for accelerating training. We adopt the Adam optimizer with learning rate of 2e-5 and perform early stopping on the validation set.

2.2 Clone Detection

Clone detection is to detect whether two code snippets are functional equivalence. In practice, following the previous studies [4], we first average the contextual representation of the last layer of one model as the overall representations of the two code snippets. Then, we measure the semantic similarities of the two representations by the cosine similarity. Finally, we fine-tune the pre-trained model to pull together the true clones and pull apart the false clones.

We conduct the experiment on the widely-used BigCloneBench [20] dataset, which consists of known true and false clones from a large Java repository. The sizes of training, validation, and testing are 901,724, 416,328, and 416,328. We treat this task as a binary classification to fine-tune the pre-trained code models.

We use the commonly-used Precision, Recall and F1-score as evaluation metrics. Suppose TP and FP are the numbers of predicted true positive clones and false positive clones respectively, and FN are the number of predicted false negative clones. Then, Precision (P), Recall (R), and F1-score (F1) are calculated by:

$$P = \frac{TP}{TP+FP}, \ R = \frac{TP}{TP+FN}, \ F1 = 2 * \frac{P*R}{Pn+R} \tag{4}$$

We use a pre-trained UniXcoder [4] to obtain the overall representation of code snippets. Following the fine-tuning experimental settings of UniXcoder, the maximum length of code snippets is set to 512. The batch size and maximum epoch are set to 16 and 10, respectively. We adopt the Adam optimizer with the learning rate of 5e-5 and perform early stopping on the validation set.

2.3 Code Summarization

Code summarization aims to generate the concise natural language description of source code and plays a vital roles in the software development and maintenance. In practice, we employ an encoder to encode the information of input source code and use a decoder to generate the concise summaries word by word according to the information of encoder. We fine-tune the pre-trained model to generate more natural and correct summaries.

We conduct the experiments on the widely-used CodeSearchNet [8] data with Python and Ruby dataset. Similar to previous works [18, 19], we evaluate our models based on four widely-used metrics including sentence-level smoothing BLEU [16], METEOR [2], ROUGE-L [12] and CIDER [22].

BLEU measures the average n-gram precision between the reference sentences and generated sentences, with brevity penalty for short sentences. The formula to compute BLEU is:

$$\text{BLEU} = BP \cdot \exp\sum_{n=1}^{4}\omega_n \log p_n, \tag{5}$$

where p_n (n-gram precision) is the fraction of n-grams in the generated sentences which are present in the reference sentences, and ω_n is the uniform weight 1/N. Since the generated summary is very short, high-order n-grams may not overlap. We use the +1 smoothing function [13]. BP is brevity penalty given as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{6}$$

Here, c is the length of the generated summary, and r is the length of the reference sentence.

Based on longest common subsequence (LCS), *ROUGE-L* is widely used in text summarization. Instead of using only recall, it uses F-score which is the harmonic mean of precision and recall values. Suppose A and B are generated and reference summaries of lengths c and r respectively, we have:

$$\begin{cases} P_{ROUGE-L} = \frac{LCS(A,B)}{c} \\ R_{ROUGE-L} = \frac{LCS(A,B)}{r} \end{cases} \tag{7}$$

$F_{ROUGE-L}$, which indicates the value of ROUGE-L, is calculated as the weighted harmonic mean of $P_{ROUGE-L}$ and $R_{ROUGE-L}$:

$$F_{ROUGE-L} = \frac{\left(1+\beta^2\right) P_{ROUGE-L} \cdot R_{ROUGE-L}}{R_{ROUGE-L} + \beta^2 P_{ROUGE-L}} \tag{8}$$

β is set to 1.2 as in [23, 24].

METEOR is a recall-oriented metric that measures how well the model captures the content from the references in the generated sentences and has a better correlation with human judgment. Suppose m is the number of mapped unigrams between the reference and generated sentence with lengths c and r respectively. Then, precision, recall and F are given as:

$$P = \frac{m}{c}, \; R = \frac{m}{r}, \; F = \frac{PR}{\alpha P + (1-\alpha)R} \tag{9}$$

The sequence of mapping unigrams between the two sentences is divided into the fewest possible number of "chunks". This way, the matching unigrams in each "chunk" are adjacent (in two sentences) and the word order is the same. The penalty is then computed as:

$$\text{Pen} = \gamma \cdot \text{ frag}^{\beta} \tag{10}$$

where frag is a fragmentation fraction: frag = ch/m, where ch is the number of matching chunks and m is the total number of matches. The default values of α, β, γ are 0.9, 3.0 and 0.5 respectively.

CIDER is a consensus-based evaluation metric used in image captioning tasks. The notions of importance and accuracy are inherently captured by computing the TF-IDF weight for each n-gram and using cosine similarity for sentence similarity. To compute CIDER, we first calculate the TF-IDF weighting $g_k(s_i)$ for each n-gram ω_k in reference sentence s_i. Here ω is the vocabulary of all n-grams. Then we use the cosine similarity between the generated sentence and the reference sentences to compute CIDER_n score for n-grams of length n. The formula is given as:

$$\text{CIDER}_n\left(c_i, s_i\right) == \frac{< \boldsymbol{g^n}\left(c_i\right), \boldsymbol{g^n}\left(s_i\right) >}{\left\|\boldsymbol{g^n}\left(c_i\right)\right\| \left\|\boldsymbol{g^n}\left(s_i\right)\right\|} \tag{11}$$

where $\boldsymbol{g^n}(s_i)$ is a vector formed by $g_k(s_i)$ corresponding to all the n-grams (n varying from 1 to 4). c_i is the i^{th} generated sentence. Finally, the scores of various n-grams can be combined to calculate CIDER as follows:

$$\text{CIDER}\left(c_i, s_i\right) = \sum_{n=1}^{N} w_n \, \text{CIDER}_n\left(c_i, s_i\right) \tag{12}$$

We use a pre-trained UniXcoder [4] to encode the input source code and use a decoder similar to the encoder to generate the summaries according to encoded input information. Following the experimental settings of UniXcoder [4], the encoder and decoder share the same parameters. The maximum lengths of code snippets and summaries are set to 256 and 128, respectively. The maximum epoch and batch size are set to 10 and 32, respectively. We adopt the Adam optimizer with the learning rate of 5e-5 and perform early stopping on the validation set. During inference, we adopt the beam search with beam size 10 to generate the better summaries.

2.4 Code Generation

This task aims to generate a function-level code snippet for the given natural language description and some context of the targeted code snippets such as member variables and other member functions in one class. In practice, we employ an encoder to encode the information of the input natural language description and code context and use a decoder to generate the corresponding code snippet token by token according to the information of the encoder. We fine-tune the pre-trained model to generate better code snippets.

We conduct the experiments on the widely-used CONCODE [9] dataset, which is collected from about 33K Java repositories on Github. The training, validation, and testing set contains 100K, 2K, and 2k samples. Each sample includes a natural language description, code context, and code snippets.

Similar to previous work [4], we evaluate the performance of our proposed model by sentence-level smoothing BLEU [16] and Exact Match accuracy (EM). The calculation of BLEU is introduced in Section 2.3. EM is the fraction of exact matched results.

$$EM = \frac{\#exact\ matched\ results}{\#total\ results} \tag{13}$$

where #exact matched results is the number of the exact matched results and #total results is the number of all the generated results.

We use a pre-trained UniXcoder [4] as an encoder to encode the input source code and use a decoder similar to the encoder to generate the summaries according to encoded input information. Following the experimental settings of UniXcoder [4], the encoder and decoder share the same parameters. The maximum lengths of the input and output sequences are set to 450 and 150, respectively. The maximum epoch and batch size are set to 30 and 32, respectively. We adopt the Adam optimizer with the learning rate of 5e-5 and perform early stopping on the validation set. During inference, we adopt the beam search with beam size 3 to generate better code snippets.

2.5 Line-level Code Completion

This task aims to predict the next line of code for the given previous context. In practice, we employ a decoder to generate the next line of code token by token according to the given previous code snippets. We fine-tune the pre-trained model to predict better code snippets.

We conduct the experiments on a large dataset named *Github Java Corpus* [1] in CodeXGLUE [14]. The dataset collects over 13K Java repositories on Github. We split the training, validation, and testing set by 8:1:1. Thus, the training, validation, and testing set contains about 12K, 1.5k, and 1.5k samples. Similar to previous work [14], we evaluate the performance of our models by Exact Match Accuracy (EM) and Levenshtein edit similarity (Edit sim) [21].

We use a pre-trained UniXcoder [4] as the decoder to predict the next line of code. Following the experimental settings of UniXcoder [4], the maximum length of the generated line of code is 64. The maximum epoch and batch size are set to 10 and 32, respectively. We adopt the Adam optimizer with the learning rate of 2e-5 and perform early stopping on the validation set. During inference, we adopt the beam search with beam size 5 to generate better code snippets.

3 EXPERIMENTAL RESULTS

3.1 Code Search

We study the performance of 12 Telly-K variants on code search. We show the results on Ruby dataset here.

- For all variant models, both the training time cost (especially the convergence time cost) and the training parameters are significantly reduced compared with the base model, while the performance does not change much across the four metrics.
- For Telly-K ($0 \leq K \leq 8$), they reduce the training parameters by 32% to 77%, correspondingly saving about 42% to 86% of training time, with the performance increment of 0% to 5% in terms of four metrics.
- When freezing the bottom 9 layers, there is an 83% reduction in training parameters and a corresponding 88% training time saving with a slight change in performance.
- For Telly-K ($K \geq 10$), the performance consistently drops across four metrics. However, even freezing the bottom 11 layers, the performance does not drop significantly, while training parameters and corresponding training time are greatly reduced.

3.2 Code Summarization

We conduct the experiments with different Telly-K on code summarization and the results are shown in Table 2 and 3. From these tables, we can see that for all variant models, both training time costs and parameters are significantly reduced compared to the base model, while the performance does not change much for all metrics. When Telly-K ($K \geq 10$), the performance slightly drops in terms of four metrics. However, even freezing the bottom 11 layers, the performance does not drop significantly, while both the training parameters and the corresponding time cost are extremely reduced.

4 EXPERIMENTAL RESULTS ON GRAPHCODEBERT

4.1 RQ1: What code properties are encoded in layer-wise pre-trained representations?

In this section, we investigate what code properties are encoded by layer-wise pre-trained representations via four probing tasks. We conduct experiments with pre-trained and fine-tuned GraphCodeBERT, and the experimental results are shown in Figure 2.

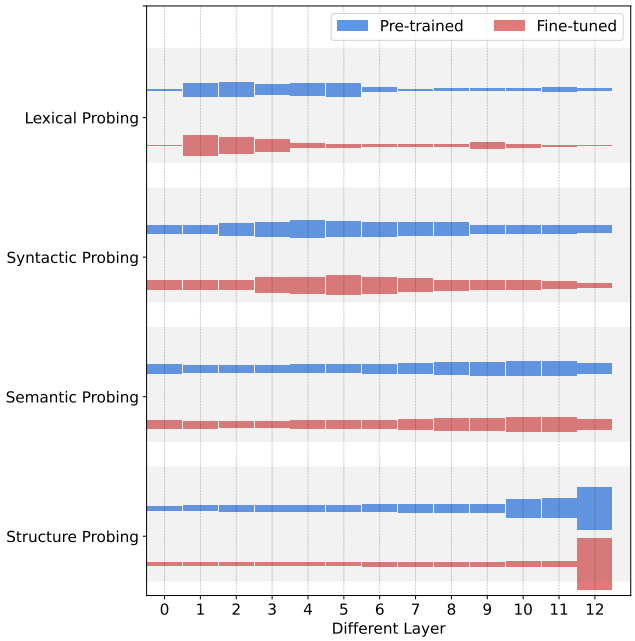

Figure 2: Layer-wise contributions on different probing tasks for the pre-trained and fine-tuned GraphCodeBERT.

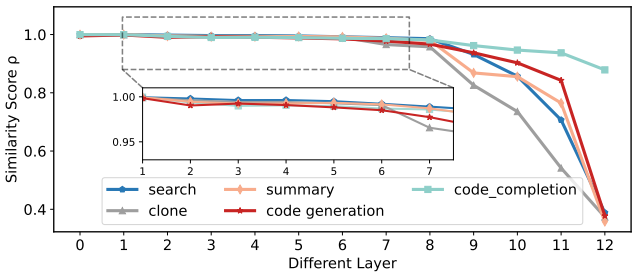

Figure 3: Similarity scores between pre-trained and fine-tuned GraphCodeBERT for five downstream tasks.

From Figure 2, we can see that, for pre-trained layer-wise representations, lexical, syntactic, and structural properties of source code are mainly captured by the lower, intermediate and, higher layers, respectively, while the semantic property almost spans across the entire model. The findings are consistent with the results of UniXcoder in the main text.

4.2 RQ2: What happens to the layer-wise representations during fine-tuning?

We also conduct extensive experiments on representational similarity analysis (RSA) to study what happens to the layer-wise representations of pre-trained GraphCodeBERT during the fine-tuning for five different downstream tasks. The results are shown in Figure 3. From the presented results, we can see that the correlation

Table 1: Experimental results of code search on Ruby dataset based on UniXcoder. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	Note	#Params	Training time		Performance			
			Each epoch	Convergence	MRR	R@1	R@5	R@10
Base	Fine-tuning all parameters	125.93M	02m45s	0h24m45s	0.740	0.642	0.850	0.892
Telly-0	Freezing the first 0 layers	85.0M(↓32%)	1m47s(↓35%)	0h14m16s(↓42%)	0.751(↑1%)	0.665(↑4%)	0.862(↑1%)	0.904(↑1%)
Telly-1	Freezing the first 1 layers	78.0M(↓38%)	01m42s(↓38%)	0h13m36s(↓45%)	0.753(↑2%)	0.668(↑4%)	0.864(↑2%)	0.902(↑1%)
Telly-2	Freezing the first 2 layers	70.9M(↓44%)	01m36s(↓42%)	0h12m48s(↓48%)	0.75(↑1%)	0.662(↑3%)	0.863(↑2%)	0.899(↑1%)
Telly-3	Freezing the first 3 layers	63.8M(↓49%)	01m28s(↓47%)	0h04m24s(↓82%)	0.755(↑2%)	0.671(↑5%)	0.864(↑2%)	0.902(↑1%)
Telly-4	Freezing the first 4 layers	56.7M(↓55%)	01m21s(↓51%)	0h05m24s(↓78%)	0.746(↑1%)	0.66(↑3%)	0.854(↑0%)	0.898(↑1%)
Telly-5	Freezing the first 5 layers	49.6M(↓61%)	01m17s(↓53%)	0h03m51s(↓84%)	0.756(↑2%)	0.676(↑5%)	0.857(↑1%)	0.902(↑1%)
Telly-6	Freezing the first 6 layers	42.5M(↓66%)	01m13s(↓56%)	0h03m39s(↓85%)	0.752(↑2%)	0.67(↑4%)	0.855(↑1%)	0.898(↑1%)
Telly-7	Freezing the first 7 layers	35.4M(↓72%)	01m08s(↓59%)	0h03m24s(↓86%)	0.747(↑1%)	0.661(↑3%)	0.86(↑1%)	0.897(↑1%)
Telly-8	Freezing the first 8 layers	28.4M(↓77%)	01m02s(↓62%)	0h04m08s(↓83%)	0.74(↓0%)	0.649(↑1%)	0.854(↑0%)	0.898(↑1%)
Telly-9	Freezing the first 9 layers	21.3M(↓83%)	00m57s(↓65%)	0h02m51s(↓88%)	0.734(↓1%)	0.637(↓1%)	0.849(↓0%)	0.892(0%)
Telly-10	Freezing the first 10 layers	14.2M(↓89%)	00m51s(↓69%)	0h02m33s(↓90%)	0.723(↓2%)	0.627(↓2%)	0.847(↓0%)	0.891(↓0%)
Telly-11	Freezing the first 11 layers	7.1M(↓94%)	00m46s(↓72%)	0h02m18s(↓91%)	0.699(↓6%)	0.592(↓8%)	0.836(↓2%)	0.875(↓2%)

Table 2: Freezing experimental results of code summarization on Python dataset based on UniXcoder. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance			
		Each epoch	Convergence	BLEU	METEOR	ROUGH-L	CIDER
Base	125.93M	22m44s	1h53m40s	19.15	17.26	41.31	1.53
Telly-0	85.0M(↓32%)	21m30s(↓5%)	1h47m30s(↓5%)	19.19(↑0%)	17.32(↑0%)	41.50(↑0%)	1.53(0%)
Telly-1	78.0M(↓38%)	20m15s(↓11%)	1h41m15s(↓11%)	19.21(↑0%)	17.33(↑0%)	41.57(↑1%)	1.54(↑1%)
Telly-2	70.9M(↓44%)	19m11s(↓16%)	1h35m55s(↓16%)	19.17(↑0%)	17.3(↑0%)	41.51(↑0%)	1.54(↑1%)
Telly-3	63.8M(↓49%)	17m59s(↓21%)	0h53m57s(↓21%)	19.16(↑0%)	17.26(0%)	41.50(↑0%)	1.54(↑1%)
Telly-4	56.7M(↓55%)	17m07s(↓25%)	0h51m21s(↓25%)	19.13(↓0%)	17.26(0%)	41.53(↑1%)	1.54(↑1%)
Telly-5	49.6M(↓61%)	16m18s(↓28%)	0h48m54s(↓28%)	19.18(↑0%)	17.26(0%)	41.55(↑1%)	1.54(↑1%)
Telly-6	42.5M(↓66%)	15m10s(↓33%)	0h45m30s(↓33%)	19.36(↑1%)	17.35(↑1%)	41.78(↑1%)	1.56(↑2%)
Telly-7	35.4M(↓72%)	14m08s(↓38%)	0h28m16s(↓38%)	19.37(↑1%)	17.28(↑0%)	41.74(↑1%)	1.56(↑2%)
Telly-8	28.4M(↓77%)	12m59s(↓43%)	0h25m58s(↓43%)	19.34(↑1%)	17.26(0%)	41.69(↑1%)	1.55(↑1%)
Telly-9	21.3M(↓83%)	11m28s(↓50%)	0h22m56s(↓50%)	19.18(↑0%)	17.22(↓0%)	41.33(↑0%)	1.53(0%)
Telly-10	14.2M(↓89%)	10m19s(↓55%)	0h10m19s(↓55%)	19.11(↓0%)	17.18(↓0%)	41.23(↓0%)	1.53(0%)
Telly-11	7.1M(↓94%)	09m14s(↓59%)	0h09m14s(↓59%)	19.10(↓0%)	17.20(↓0%)	41.27(↓0%)	1.53(0%)

coefficients (similarity score in Figure 3) of the bottom 9 layers are all greater than 0.8. It means that representations of the bottom 9 layers are similar between pre-trained and fine-tuned models for the five downstream tasks. The representations of the top layer are dissimilar ($\rho \leq 0.5$) except for code completion. Furthermore, we find that the representation of the bottom 7 layers ($\rho \geq 0.9$) are greatly related.

4.3 RQ3: Are there efficient alternatives to fine-tuning?

In this section, we conduct experiments with Telly-K based on GraphCodeBERT. Specifically, Telly-K freezes the bottom K layers of the pre-trained GraphCodeBERT when fine-tuning it on the five different downstream tasks, namely, code search, clone detection, code summarization, code generation and line-level code completion.

4.3.1 Code search. We conduct the experiments using Telly-K based on GraphCodeBERT on code search, and the results are presented in Table 4. The table shows the number of trained parameters, training time costs, and performance for the base model and 12 model variants. The base model fine-tunes all parameters of pre-trained GraphCodeBERT, while the variant models would freeze partial parameters. From Table 4, we can see that, for all variant models, both training time costs and the number of tuned parameters are significantly reduced compared to the base model, while the performance does not change much for all metrics. In detail, the performance of Telly-K increases for $0 \leq K \leq 8$, changes slightly for $K = 9$, and decreases lightly for $K \geq 10$ compared to the base model.

4.3.2 Clone detection. We conduct the experiments on clone detection with different GraphCodeBERT-based Telly-K, and the results are shown in Table 5. We can see that, compared to the base mode,

Table 3: Freezing experimental results of code summarization on Ruby dataset based on UniXcoder. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance			
		Each epoch	Convergence	BLEU	METEOR	ROUGH-L	CIDER
Base	125.93M	02m19s	0h11m35s	14.87	13.45	32.96	1.12
Telly-0	85.0M(↓32%)	01m56s(↓17%)	0h09m40s(↓17%)	14.91(↑0%)	13.42(↓0%)	32.83(↓0%)	1.13(↑1%)
Telly-1	78.0M(↓38%)	01m49s(↓22%)	0h09m05s(↓22%)	14.78(↓1%)	13.43(↓0%)	32.87(↓0%)	1.13(↑1%)
Telly-2	70.9M(↓44%)	01m44s(↓25%)	0h08m40s(↓25%)	14.74(↓1%)	13.36(↓1%)	32.77(↓1%)	1.12(0%)
Telly-3	63.8M(↓49%)	01m38s(↓30%)	0h04m54s(↓30%)	14.68(↓1%)	13.40(↓0%)	32.7(↓1%)	1.11(↓1%)
Telly-4	56.7M(↓55%)	01m32s(↓34%)	0h04m36s(↓34%)	14.79(↓1%)	13.45(↓0%)	32.73(↓1%)	1.12(0%)
Telly-5	49.6M(↓61%)	01m26s(↓38%)	0h04m18s(↓38%)	14.7(↓1%)	13.43(↓0%)	32.72(↓1%)	1.12(0%)
Telly-6	42.5M(↓66%)	01m21s(↓42%)	0h04m03s(↓42%)	14.86(↓0%)	13.44(↓0%)	32.95(↓0%)	1.12(0%)
Telly-7	35.4M(↓72%)	01m14s(↓47%)	0h02m28s(↓47%)	15.02(↑1%)	13.48(↑0%)	33.11(↑0%)	1.13(↑1%)
Telly-8	28.4M(↓77%)	01m09s(↓50%)	0h02m18s(↓50%)	15.18(↑2%)	13.51(↑0%)	33.26(↑1%)	1.14(↑2%)
Telly-9	21.3M(↓83%)	01m02s(↓55%)	0h02m04s(↓55%)	14.91(↑0%)	13.45(↓0%)	32.8(↓0%)	1.12(0%)
Telly-10	14.2M(↓89%)	00m56s(↓60%)	0h00m56s(↓60%)	14.92(↑0%)	13.43(↓0%)	32.7(↓1%)	1.12(0%)
Telly-11	7.1M(↓94%)	00m50s(↓64%)	0h00m50s(↓64%)	14.72(↓1%)	13.37(↓1%)	32.75(↓1%)	1.11(↓1%)

Table 4: Experimental results on code search based on GraphCodeBERT. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	Note	#Params	Training time		Performance			
			Each epoch	Convergence	MRR	R@1	R@5	R@10
Base	None	124.65M	02m05s	0h18m45s	0.703	0.607	0.824	0.872
Telly-0 (GraphCodeBERT)	Freezing the first 0 layers	85.0M(↓32%)	1m37s(↓22%)	0h12m56s(↓31%)	0.725(↑3%)	0.626(↑3%)	0.849(↑3%)	0.889(↑2%)
Telly-1 (GraphCodeBERT)	Freezing the first 1 layers	78.0M(↓37%)	01m32s(↓26%)	0h12m16s(↓35%)	0.727(↑3%)	0.63(↑4%)	0.846(↑3%)	0.887(↑2%)
Telly-2 (GraphCodeBERT)	Freezing the first 2 layers	70.9M(↓43%)	01m27s(↓30%)	0h11m36s(↓38%)	0.727(↑3%)	0.63(↑4%)	0.841(↑2%)	0.886(↑2%)
Telly-3 (GraphCodeBERT)	Freezing the first 3 layers	63.8M(↓49%)	01m23s(↓34%)	0h04m09s(↓78%)	0.728(↑4%)	0.63(↑4%)	0.848(↑3%)	0.887(↑2%)
Telly-4 (GraphCodeBERT)	Freezing the first 4 layers	56.7M(↓55%)	01m17s(↓38%)	0h05m08s(↓73%)	0.728(↑4%)	0.632(↑4%)	0.847(↑3%)	0.892(↑2%)
Telly-5 (GraphCodeBERT)	Freezing the first 5 layers	49.6M(↓60%)	01m13s(↓42%)	0h03m39s(↓81%)	0.727(↑3%)	0.63(↑4%)	0.846(↑3%)	0.891(↑2%)
Telly-6 (GraphCodeBERT)	Freezing the first 6 layers	42.5M(↓66%)	01m09s(↓45%)	0h03m27s(↓82%)	0.725(↑3%)	0.627(↑3%)	0.845(↑3%)	0.89(↑2%)
Telly-7 (GraphCodeBERT)	Freezing the first 7 layers	35.4M(↓72%)	01m04s(↓49%)	0h03m12s(↓83%)	0.721(↑3%)	0.623(↑3%)	0.841(↑2%)	0.888(↑2%)
Telly-8 (GraphCodeBERT)	Freezing the first 8 layers	28.4M(↓77%)	01m00s(↓52%)	0h04m00s(↓79%)	0.712(↑1%)	0.609(↑0%)	0.836(↑1%)	0.890(↑2%)
Telly-9 (GraphCodeBERT)	Freezing the first 9 layers	21.3M(↓83%)	00m54s(↓57%)	0h02m42s(↓86%)	0.71(↑1%)	0.604(↓0%)	0.841(↑2%)	0.889(↑2%)
Telly-10 (GraphCodeBERT)	Freezing the first 10 layers	14.2M(↓89%)	00m48s(↓62%)	0h02m24s(↓87%)	0.699(↓1%)	0.601(↓1%)	0.825(↓1%)	0.880(↑1%)
Telly-11 (GraphCodeBERT)	Freezing the first 11 layers	7.1M(↓94%)	00m43s(↓66%)	0h02m09s(↓89%)	0.696(↓1%)	0.593(↓2%)	0.823(↓0%)	0.869(↓1%)

all variant models significantly reduce both training time cost and tuned parameters, while preserving the performance across all metrics. Specifically, the performance changes of all variant models vary from 1% to 2% compared to the base model, while the numbers of training parameters are reduced by 32% to 94% and the training time is saved by 5% to 59%.

4.3.3 Code summarization. We conduct the experiments on code summarization with different Telly-K, and the results are shown in Table 6. We can see that, for all variant models, both training time costs and parameters are significantly reduced compared to the base model, while the performance does not change much for all metrics. In detail, on the code summarization task, the performance of Telly-K generally increases for $0 \leq K \leq 9$, and decreases for $K \geq 10$ compared to the base model.

4.3.4 Code generation. We conduct the experiments with different GraphCodeBERT-based Telly-K on code generation, and the results are shown in Table 7. We can see that, for all variant models, both training time costs and parameters are significantly reduced compared to the base model, while the performance does not change much for all metrics except for Telly-K ($K \geq 9$). Specifically, the performance of Telly-K ($0 \leq K \leq 7$) are generally stable in terms of BLEU scores, while the performance noticeably increases in terms of EM scores. The performance of Telly-9 is unchanged for EM score and significantly drops for BLEU scores. In particular, when $K \geq 10$, the performance significantly drops for all metrics compared to the base model.

4.3.5 Line-level code completion. We conduct the experiments with all Telly-K on line-level code completion and the results are shown in Table 8. We can see that, for all variant models, both the training time cost and parameters are greatly reduced compared to the base model, while the performance does not change significantly for all metrics except for Telly-11. In detail, the performance of Telly-K (

Table 5: Freezing experimental results of clone detection based on GraphCodeBERT. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance		
		Each epoch	Convergence	Recall	Precision	F1-score
Base	125.83M	20m12s	1h41m00s	0.95	0.94	0.94
Telly-0(GraphCodeBERT)	86.2M(↓31%)	19m19s(↓4%)	1h17m16s(↓24%)	0.95(0%)	0.93(↓1%)	0.94(0%)
Telly-1(GraphCodeBERT)	79.2M(↓37%)	18m25s(↓9%)	1h13m40s(↓27%)	0.95(0%)	0.94(0%)	0.94(0%)
Telly-2(GraphCodeBERT)	72.1M(↓43%)	17m22s(↓14%)	1h09m28s(↓31%)	0.95(0%)	0.93(↓1%)	0.94(0%)
Telly-3(GraphCodeBERT)	65.0M(↓48%)	15m10s(↓25%)	1h00m40s(↓40%)	0.95(0%)	0.94(0%)	0.94(0%)
Telly-4(GraphCodeBERT)	57.9M(↓54%)	14m17s(↓29%)	0h57m08s(↓43%)	0.95(0%)	0.95(↑1%)	0.95(↑1%)
Telly-5(GraphCodeBERT)	50.8M(↓60%)	13m15s(↓34%)	1h06m15s(↓34%)	0.96(↑1%)	0.93(↓1%)	0.94(0%)
Telly-6(GraphCodeBERT)	43.7M(↓65%)	12m26s(↓38%)	0h49m44s(↓51%)	0.96(↑1%)	0.94(0%)	0.95(↑1%)
Telly-7(GraphCodeBERT)	36.6M(↓71%)	11m32s(↓43%)	0h57m40s(↓43%)	0.96(↑1%)	0.93(↓1%)	0.94(0%)
Telly-8(GraphCodeBERT)	29.5M(↓77%)	10m31s(↓48%)	0h52m35s(↓48%)	0.95(0%)	0.95(↑1%)	0.95(↑1%)
Telly-9(GraphCodeBERT)	22.4M(↓82%)	9m46s(↓52%)	0h29m18s(↓71%)	0.95(0%)	0.95(↑1%)	0.95(↑1%)
Telly-10(GraphCodeBERT)	15.4M(↓88%)	8m40s(↓57%)	0h34m40s(↓66%)	0.94(↓1%)	0.95(↑1%)	0.95(↑1%)
Telly-11(GraphCodeBERT)	8.3M(↓93%)	7m49s(↓61%)	0h31m16s(↓69%)	0.95(0%)	0.92(↓2%)	0.93(↓1%)

Table 6: Freezing experimental results of code summarization based on GraphCodeBERT. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance			
		Each epoch	Convergence	BLEU	METEOR	ROUGH-L	CIDER
Base	124.65M	02m09s	0h10m45s	13.39	12.25	31.6	0.99
Telly-0 (GraphCodeBERT)	85.0M(↓32%)	01m46s(↓18%)	0h08m50s(↓18%)	13.79(↑3%)	12.82(↑5%)	32.18(↑2%)	0.98(↓1%)
Telly-1 (GraphCodeBERT)	78.0M(↓37%)	01m45s(↓19%)	0h08m45s(↓19%)	14.0(↑5%)	13.03(↑6%)	32.52(↑3%)	0.99(0%)
Telly-2 (GraphCodeBERT)	70.9M(↓43%)	01m40s(↓22%)	0h08m20s(↓22%)	14.38(↑7%)	13.16(↑7%)	33.0(↑4%)	1.03(↑4%)
Telly-3 (GraphCodeBERT)	63.8M(↓49%)	01m34s(↓27%)	0h04m42s(↓27%)	14.34(↑7%)	13.15(↑7%)	33.07(↑5%)	1.04(↑5%)
Telly-4 (GraphCodeBERT)	56.7M(↓55%)	01m28s(↓32%)	0h04m24s(↓32%)	14.18(↑6%)	13.07(↑7%)	32.77(↑4%)	1.03(↑4%)
Telly-5 (GraphCodeBERT)	49.6M(↓60%)	01m23s(↓36%)	0h04m09s(↓36%)	14.23(↑6%)	13.04(↑6%)	32.68(↑3%)	1.03(↑4%)
Telly-6 (GraphCodeBERT)	42.5M(↓66%)	01m17s(↓40%)	0h03m51s(↓40%)	13.91(↑4%)	12.97(↑6%)	31.99(↑1%)	0.98(↓1%)
Telly-7 (GraphCodeBERT)	35.4M(↓72%)	01m12s(↓44%)	0h02m24s(↓44%)	13.92(↑4%)	12.88(↑5%)	32.44(↑3%)	1.0(↑1%)
Telly-8 (GraphCodeBERT)	28.4M(↓77%)	01m06s(↓49%)	0h02m12s(↓49%)	13.91(↑4%)	12.56(↑3%)	32.33(↑2%)	1.01(↑2%)
Telly-9 (GraphCodeBERT)	21.3M(↓83%)	01m00s(↓53%)	0h02m00s(↓53%)	13.68(↑2%)	12.64(↑3%)	31.81(↑1%)	1.0(↑1%)
Telly-10 (GraphCodeBERT)	14.2M(↓89%)	00m54s(↓58%)	0h00m54s(↓58%)	13.34(↓0%)	12.2(↓0%)	31.57(↓0%)	0.98(↓1%)
Telly-11 (GraphCodeBERT)	7.1M(↓94%)	00m49s(↓62%)	0h00m49s(↓62%)	12.99(↓3%)	11.61(↓5%)	30.62(↓3%)	0.96(↓3%)

$0 \leq K \leq 3$) noticeably increases especially for EM metrics. When $4 \leq K \leq 9$, the performance of Telly-K slightly changes across all metrics. When $K \geq 10$, the performance of variants significantly drops for two metrics.

REFERENCES

[1] 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)*. IEEE, 207–216.

[2] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *IEEvaluation@ACL*.

[3] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. ACM, 933–944.

[4] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *ACL (1)*. Association for Computational Linguistics, 7212–7225.

[5] 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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. 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. https://openreview.net/forum?id=jLoC4ez43PZ

[6] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*.

[7] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20, 000+ Web Queries for Code Search and Question Answering. In *ACL*.

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[9] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[10] Nathalie Japkowicz and Mohak Shah. 2011. *Evaluating learning algorithms: a classification perspective*. Cambridge University Press.

[11] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*. 795–806.

[12] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*.

Table 7: Freezing experimental results of code generation based on GraphCodeBERT. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance	
		Each epoch	Convergence	BLEU	EM
Base	124.65M	13m14s	4h24m40s	30.59	15.6
Telly-0(GraphCodeBERT)	85.0M(↓32%)	11m41s(↓12%)	3h41m59s(↓16%)	31.7(↑4%)	17.0(↑9%)
Telly-1(GraphCodeBERT)	78.0M(↓37%)	11m04s(↓16%)	3h19m12s(↓25%)	30.89(↑1%)	17.3(↑11%)
Telly-2(GraphCodeBERT)	70.9M(↓43%)	10m30s(↓21%)	2h06m00s(↓52%)	30.91(↑1%)	17.1(↑10%)
Telly-3(GraphCodeBERT)	63.8M(↓49%)	09m49s(↓26%)	2h07m37s(↓52%)	30.77(↑1%)	17.3(↑11%)
Telly-4(GraphCodeBERT)	56.7M(↓55%)	09m14s(↓30%)	1h50m48s(↓58%)	30.58(↓0%)	16.6(↑6%)
Telly-5(GraphCodeBERT)	49.6M(↓60%)	08m41s(↓34%)	1h52m53s(↓57%)	30.35(↓1%)	16.7(↑7%)
Telly-6(GraphCodeBERT)	42.5M(↓66%)	08m12s(↓38%)	1h46m36s(↓60%)	30.45(↓0%)	16.8(↑8%)
Telly-7(GraphCodeBERT)	35.4M(↓72%)	07m40s(↓42%)	1h39m40s(↓62%)	29.96(↓2%)	16.9(↑8%)
Telly-8(GraphCodeBERT)	28.4M(↓77%)	07m06s(↓46%)	1h32m18s(↓65%)	29.93(↓2%)	15.8(↑1%)
Telly-9(GraphCodeBERT)	21.3M(↓83%)	06m40s(↓50%)	1h26m40s(↓67%)	29.59(↓3%)	15.6(0%)
Telly-10(GraphCodeBERT)	14.2M(↓89%)	06m02s(↓54%)	1h12m24s(↓73%)	27.8(↓9%)	15.0(↓4%)
Telly-11(GraphCodeBERT)	7.1M(↓94%)	05m20s(↓60%)	1h09m20s(↓74%)	26.51(↓13%)	14.0(↓10%)

Table 8: Freezing experimental results of code completion based on GraphCodeBERT. #Params is short for the number of training parameters. M is short for million. The changing ratios compared to the base model are shown in parentheses.

Model	#Params	Training time		Performance	
		Each epoch	Convergence	Edit sim	EM
Base	124.65M	03m12s	0h28m48s	38.2	11.33
Telly-0(GraphCodeBERT)	85.0M(↓32%)	02m50s(↓11%)	0h22m40s(↓21%)	38.99(↑2%)	12.02(↑6%)
Telly-1(GraphCodeBERT)	78.0M(↓37%)	03m39s(↑14%)	0h18m15s(↓37%)	38.78(↑2%)	11.93(↑5%)
Telly-2(GraphCodeBERT)	70.9M(↓43%)	03m28s(↑8%)	0h31m12s(↑8%)	38.41(↑1%)	11.82(↑4%)
Telly-3(GraphCodeBERT)	63.8M(↓49%)	03m19s(↑4%)	0h19m54s(↓31%)	38.47(↑1%)	11.81(↑4%)
Telly-4(GraphCodeBERT)	56.7M(↓55%)	03m08s(↓2%)	0h25m04s(↓13%)	38.01(↓0%)	11.51(↑2%)
Telly-5(GraphCodeBERT)	49.6M(↓60%)	02m57s(↓8%)	0h08m51s(↓69%)	38.13(↓0%)	11.41(↑1%)
Telly-6(GraphCodeBERT)	42.5M(↓66%)	02m48s(↓12%)	0h11m12s(↓61%)	38.5(↑1%)	11.27(↓1%)
Telly-7(GraphCodeBERT)	35.4M(↓72%)	02m39s(↓17%)	0h13m15s(↓54%)	38.36(↑0%)	11.13(↓2%)
Telly-8(GraphCodeBERT)	28.4M(↓77%)	02m28s(↓23%)	0h12m20s(↓57%)	38.48(↑1%)	11.1(↓2%)
Telly-9(GraphCodeBERT)	21.3M(↓83%)	02m19s(↓28%)	0h11m35s(↓60%)	37.29(↓2%)	11.08(↓2%)
Telly-10(GraphCodeBERT)	14.2M(↓89%)	02m08s(↓33%)	0h08m32s(↓70%)	36.91(↓3%)	10.51(↓7%)
Telly-11(GraphCodeBERT)	7.1M(↓94%)	01m57s(↓39%)	0h05m51s(↓80%)	36.51(↓4%)	10.37(↓8%)

[13] Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a Method for Evaluating Automatic Evaluation Metrics for Machine Translation. In *COLING*.

[14] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[15] 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*. 1287–1293.

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*.

[17] Mark Sanderson. 2010. Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008. ISBN-13 978-0-521-86571-5, xxi+ 482 pages. *Natural Language Engineering* 16, 1 (2010), 100–103.

[18] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the Evaluation of Neural Code Summarization. In *ICSE*.

[19] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees. In *EMNLP*.

[20] 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*. IEEE, 476–480.

[21] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[22] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *CVPR*.

[23] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *ASE*.

[24] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *ICSE*. IEEE / ACM.