

Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond

Ensheng Shi^{a,†} Yanlin Wang^{b,§,†} Hongyu Zhang^c
Lun Du^d Shi Han^d Dongmei Zhang^d Hongbin Sun^{a,§}

^aXi'an Jiaotong University ^bSchool of Software Engineering, Sun Yat-sen University
^cChongqing University ^dMicrosoft
s1530129650@stu.xjtu.edu.cn, wangylin36@mail.sysu.edu.cn, hyzhang@cqu.edu.cn
{lun.du, shihan, dongmeiz}@microsoft.com, hsun@mail.xjtu.edu.cn

ABSTRACT

Recently, fine-tuning pre-trained code models such as CodeBERT on downstream tasks has achieved great success in many software testing and analysis tasks. While effective and prevalent, fine-tuning the pre-trained parameters incurs a large computational cost. In this paper, we conduct an extensive experimental study to explore what happens to layer-wise pre-trained representations and their encoded code knowledge during fine-tuning. We then propose efficient alternatives to fine-tune the large pre-trained code model based on the above findings. Our experimental study shows that (1) lexical, syntactic and structural properties of source code are encoded in the lower, intermediate, and higher layers, respectively, while the semantic property spans across the entire model. (2) The process of fine-tuning preserves most of the code properties. Specifically, the basic code properties captured by lower and intermediate layers are still preserved during fine-tuning. Furthermore, we find that only the representations of the top two layers change most during fine-tuning for various downstream tasks. (3) Based on the above findings, we propose **Telly** to efficiently fine-tune pre-trained code models via layer freezing. The extensive experimental results on five various downstream tasks demonstrate that training parameters and the corresponding time cost are greatly reduced, while performances are similar or better.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**; *Reusability*.

KEYWORDS

Empirical study, Pre-Trained Language Models, Efficient Fine-tuning, Probing Techniques, Representational Similarity Analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA 2023, 17-21 July, 2023, Seattle, USA

© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/XXXXXXXXXXXXXX>

1 INTRODUCTION

Recently, the pre-training with fine-tuning paradigm [2, 15, 17, 34] has achieved substantial improvement in many software testing and analysis tasks such as vulnerability detection [13, 17], patch generation [9, 18], automatic program repair [27], code review [42, 55], code generation [2, 9, 17], and clone detection [13, 17, 18]. They first pre-train large Transformer-base models to learn the general-purpose code representations on a large amount of data. Then, to adapt these models to the downstream tasks, they usually fine-tune them on targeted tasks [2, 15, 17, 18, 40, 55, 63].

In this paradigm, fine-tuning pre-trained code models usually achieves greatly better results on the downstream tasks. Although effective, fine-tuning the pre-trained parameters incurs a large computational cost with similarly large energy consumption. As reported in CodeXGLUE [34], they usually require more than 10 hours to fine-tune the pre-trained model on a machine with two P100 cards for downstream tasks. In particular, as pre-trained models or fine-tuned datasets become larger, the computational cost becomes more expensive. For example, CodeT5 [63] which has about 220MB of parameters spends over 40 hours fine-tuning it on CONCODE [24] dataset for code generation. In fact, these actions are contrary to low-carbon deep learning [48]. In the software engineering area, there are only a few studies that explore what would happen to pre-trained code models during the fine-tuning process. Most related studies [15, 21, 30, 54, 58] aim to understand what pre-trained code models know about source code. **There is a clear need to understand what happens to the pre-trained code models during fine-tuning and further efficiently adapt the pre-trained models to downstream tasks with less computational cost.**

In this paper, we first explore what code properties are encoded in layer-wise representations of pre-trained code models and what happens to these representations during fine-tuning. Then, we propose some efficient alternatives to fine-tuning for pre-trained code models based on the above findings. Specifically, first, inspired by the compilation process [3] and static program analysis techniques [38], we propose four probing tasks (introduced in Section 3.2.1) involving the lexical, syntactical, semantic, and structural code properties. Next, we conduct an empirical study to explore what code properties are encoded in pre-trained code models and what contributions of different layers are to the understanding of

[§]Yanlin Wang and Hongbin Sun are the corresponding authors.

[†]Work done during the author's employment at Microsoft Research Asia.

the encoded properties. Furthermore, we conduct an extensive experimental study to delve into what happens to layer-wise representations during fine-tuning on five diverse downstream tasks (shown in Table 1) including code search [16, 46], clone detection [11, 49], code summarization [47, 61], code generation [25], and line-level code completion [34]. Through extensive experiments, we obtain the following major findings about pre-trained code models.

The *first major finding* is that pre-trained code models encode the lexical property of source code mainly in the lower layers, recognize syntactical property mainly in the intermediate layers, and understand structural property mainly in higher layers. The semantic properties are perceived across layers in the entire model. The *second major finding* is that the process of fine-tuning preserves most of the code properties. That is, during fine-tuning, the basic code knowledge (or properties) encoded in lower and intermediate layers is still preserved. Only the knowledge captured by higher layers varies the most. In addition, our experimental study demonstrates that, when fine-tuning the pre-trained models on five diverse downstream tasks, the representations of lower layers change slightly, with only the top two layers showing substantial changes.

Based on the above findings, we propose **Telly- K** , for efficient fine-Tuning of pre-trained code models via layer freezing. Different K values mean different variants of our approach. Specifically, we decrease the trained parameters via freezing the pre-trained parameters of the bottom K layers that change insignificantly during fine-tuning, where $K \in [0, 1, 2, 3, \dots, L-1]$, the 0-th layer is the embedding layer, and L is the maximum number (typically 12) of layers of the pre-trained code model. Thus, Telly-1 means freezing the embedding and the 1-st encoder layer. We conduct extensive experiments on five different downstream tasks from three aspects including training parameters, time cost, and performance. The evaluated pre-trained code models have 12 hidden layers. The experimental results show that (1) for almost all Telly- K ($0 \leq K \leq 11$), the training time cost and parameters are substantially reduced, without significant changes in model performance. (2) When freezing the bottom K ($0 \leq K \leq 5$) layers, training parameters are reduced by about 30% to 65%, and the training time is saved accordingly by about 10% to 75%. The model performance generally increases by 1% to 4% for different downstream tasks. (3) When freezing the bottom K ($6 \leq K \leq 9$) layers, the training parameters are reduced by 65% to 80%, correspondingly saving about 50% to 80% of training time, while the model performance only changes slightly. (4) When the number of frozen layer is greater than nine ($10 \leq K \leq 11$), training parameters and corresponding training time cost are tremendously reduced, while the model performance also drops significantly.

Our main contributions are summarized as follows:

- We propose four probing tasks related to lexical, syntactic, semantic, and structural code properties. We explore what and how code properties are encoded in layer-wise representations through the above probing tasks.
- To the best of our knowledge, we are the first to conduct an extensive experimental study to analyze what happens to layer-wise representations and their encoded code properties during fine-tuning of pre-trained code models.
- We propose an efficient approach to fine-tune pre-trained code models to downstream tasks via layer freezing. In addition, we conduct extensive experiments on five different downstream tasks to demonstrate the efficiency of our approach.

The rest of this paper is organized as follows. Section 2 introduces the relevant background knowledge. Then, we conduct an experimental study to understand what happens to pre-trained code models during fine-tuning in Section 3. Based on the above findings, in Section 4, we propose Telly- K and conduct the extensive experiments on five different downstream tasks to show its superiority. Section 5 discusses the importance of reducing fine-tuning time, actionable guideline to better fine-tuning, and the generality of our experimental findings, and identifies some threats to validity. Section 6 presents related work. Finally, we summarize our paper and discuss the future work in Section 7.

2 BACKGROUND

2.1 Pre-trained Code Models

Large pre-trained models have achieved substantial results in many areas including natural language processing [12, 33], computer vision [7, 19] and software engineering [2, 15, 17, 18, 64]. In the software engineering community, generally, they firstly pre-train large models on amounts of source code-related data, and then fine-tune them on downstream tasks to improve their performance. Recently, many pre-trained code models [2, 15, 17, 18, 40, 63] have been proposed and shown the surprisingly promising results on many software engineering tasks involving software testing, security, maintenance, and development [13, 17, 34, 42, 55]. We introduce these models from three aspects as follows.

Basic architecture. Most recent pre-trained code models adopt the multi-layer Transformer model [56] (typically, a Transformer encoder) as the basic architecture. A Transformer encoder is essentially composed of an embedding layer, a positional encoder, and a stack of encoder layers. In general, given an input code snippet, it is firstly embedded by the embedding layer and the positional encoder to obtain the initial word embeddings. Next, they are fed to the multiple stacked encoder layers to encode the input information layer by layer.

Mathematically, we denote the input tokens as $T = [t_1, t_2, \dots, t_n]$, where n is the length of the input token sequence. The embedding layer maps each token to a high-dimensional semantic space. The positional encoder is used to encode the positional information and then injected it into input embedding by:

$$w_i = \text{embed}(t_i) + \text{pos}(t_i), \quad i = 1, 2, \dots, n \quad (1)$$

where $\text{embed}(\ast)$ and $\text{pos}(\ast)$ denote the embedding layer and positional encoder, respectively. $W = [w_1, w_2, \dots, w_n]$ is the initial word embeddings. Next, the multiple stacked encoder layers produce a set of layer-wise contextual representations H^0, H^1, \dots, H^L by:

$$\begin{aligned} H^0 &= [w_1, w_2, \dots, w_n] \\ H^l &= \text{encoder}^l(H^{l-1}), \quad l = 1, 2, \dots, L \end{aligned} \quad (2)$$

where L is the number of stacked layers, and $\text{encoder}^l(\ast)$ denotes the l -th encoder layer. $H^l = [h_1^l, h_2^l, \dots, h_n^l]$ denotes the contextual representations of the l -th layer, and H^0 is the initial word embeddings.

Pre-training. Pre-training techniques, as one of the self-supervised learning approaches, can leverage a big model to learn the general representations with amounts of unlabeled dataset [12, 19, 33, 43]. Typically, such techniques usually automatically generate virtual labels from the unlabeled samples to reformulate an unsupervised learning problem as one that is supervised learning. The corresponding supervised tasks are named pre-trained tasks. For example, CodeBERT [15] uses a 12-layer Transformer encoder with 768-dimensional embedding and pre-trains all parameters on a large-scale dataset named CodeSearchNet [23] (contains 2.1M bimodal data (code functions paired with natural language comments) and 6.4M unimodal codes across six programming languages (Ruby, JavaScript, Go, Python, Java, PHP) with two pre-trained tasks, namely, masked language modeling and replaced token detection. The former task is to predict the original tokens of the masked positions, while the latter is to identify whether a token is the original one or not. UniXcoder [17] takes code/text sequence as input and is pre-trained on the C4 dataset from T5 [43] and 4.1M unimodal code from CodeSearchNet with five different pre-trained tasks.

Fine-tuning. After pre-training on the massive dataset, they adapt pre-trained models to downstream tasks by fine-tuning all the pre-trained parameters on the targeted dataset [2, 14, 15, 17, 18, 40, 63]. For example, in code search, previous studies [17, 34] average the last-layer contextual representations (e.g. H^L) of models as the overall representation, measure the similarity between representations of the source code and the query by vector distance, and fine-tune them by pulling together the paired code and query and pushing apart the unpaired code and query. Compared with pre-trained models, the MRR values of fine-tuned models on code search are improved from 0.001 and 0.156 to 0.694 and 0.713 for CodeBERT and GraphCodeBERT, respectively.

2.2 Probing Techniques

Probing techniques have been extensively used in the NLP community to study what linguistic properties are captured by pre-trained language models. Specifically, they extract contextual representations (such as H^1, H^2) from the pre-trained model as frozen features, feed them to a linear classifier, and only train it to predict probing tasks relevant to linguistic properties. They also take random representations as a baseline to demonstrate the ability of the pre-trained representations to encode linguistic attributes for comparison. For example, Tenney et al. [53] utilize different probing tasks such as part-of-speech tagging, dependency parsing, semantic role labeling, and coreference resolution labeling to examine the ability of pre-trained models to understand linguistic properties, such as parts of speech, dependencies, semantics, and coreferences. In software engineering area, most related studies [21, 30, 58] aim to understand what pre-trained code models know about source code. For example, Wan et al. [58] propose a new probing task to investigate whether the structure of ASTs is encoded in the representations of pre-trained code models. Specifically, they extract layer-wised pre-trained representations of two input code tokens from code pre-trained models, feed them to a matrix, and train the matrix to reconstruct the distance between two corresponding terminal nodes in the AST, where the distance is used to represent the syntactical structure. López et al [21] propose a new probing task named

AST-Probe, which recovers ASTs from hidden representations of pre-trained language models. Specifically, AST-probe first maps the layer-wise representations of pre-trained code models to a latent space, called syntactic subspace, using the orthogonal projection and then uses geometry of this space to predict the AST of the input code snippet. Karmakar et al [30] also propose some new probing tasks to investigate whether pre-trained language models can understand some simple code properties. Motivated by the compilation process [3] and static analysis techniques [38], we propose four probing tasks (Section 3.2.1) related to different code properties. Besides probing pre-trained code models, we also study their layer-wise representations during fine-tuning.

2.3 Representational Similarity Analysis

Representational similarity analysis (RSA) is originally used in cognitive neuroscience [31] to study the relation between the neural activation patterns in human brains and representations of a computational model for given a set of stimuli. Recently, it is adopted to measure the similarity between two representational spaces [1, 10, 37]. For example, given a set of inputs, different models generating different representational spaces would generate different representations. Merchant et al. [37] construct two distance matrices. Each records the vector distances (such as cosine similarity) between representations in one representational space. Then the representational similarity of two representational spaces is measured by the Pearson’s correlation [45] of these two distance matrices. In general, a value of the correlation coefficient between 0.8 and 1 indicates that the two representational spaces are fairly similar, while a value lower than 0.5 means the two representational spaces are dissimilar [4, 45]. In this paper, we conduct representational similarity analysis to study the similarity of layer-wise representations between pre-trained and fine-tuned models in Section 3.3 and 3.5

3 AN EXPERIMENTAL STUDY ON PRE-TRAINED CODE MODEL

3.1 Research Questions

While effective and prevalent, fine-tuning pre-trained code models incur a large computational cost. In this work, we first conduct an experimental study to investigate what code properties are encoded in layer-wise pre-trained representations and what happens to these representations during fine-tuning. The research questions on the experimental study are introduced in detail as follows.

RQ1: What code properties are encoded in layer-wise pre-trained representations? Probing techniques have been extensively used in the NLP community to analyze and interpret pre-trained language models. Motivated by the compilation process [3] and static analysis techniques [38], we first propose four probing tasks related to lexical, syntactic, semantic, and structural properties of source code. They are introduced in detail in Section 3.2.1. Then, we investigate what code properties are encoded by layer-wise pre-trained representations via the above probing tasks in Section 3.2.2. At the same time, we study how much representations of each layer contribute to understanding these code properties. Further, we perform the same probing experiments for the model fine-tuned in a

downstream task in Section 3.5, and intuitively understand what happens to the code properties captured by the pre-trained model during fine-tuning.

RQ2: What happens to the layer-wise representations during fine-tuning? In the RQ1, we aim to roughly understand what happens to a pre-trained code model when fine-tuning with the help of probing tasks. We further conduct the extensive representation similarity analysis (RSA) to study what happens to pre-trained representations layer-by-layer when fine-tuning them on downstream tasks. RSA introduced in Section 2.3 is a task-agnostic technique and requires no prior knowledge of probing tasks. How to apply RSA to the pre-trained and fine-tuned models is described in Section 3.3. To ensure the generality of our experimental findings, we conduct experiments on five diverse downstream tasks including code search, clone detection, code summarization, code generation, and line-level code completion.

3.2 Probing Pre-trained Code Models

We introduce the four code-related probing tasks and probing pipeline as follows.

3.2.1 Four probing tasks. We design and show the four probing tasks in Figure 1. They are related to lexical, syntactical, semantic and structural code properties. We introduce them one by one in detail.

Lexical probing. Lexical probing aims to measure how well contextual representations encode the lexical properties of source code. As we all know, when the source code is compiled, the first step is lexical analysis, which tokenizes the source code string and determines the type (such as *Identifier*, *Keywords*) of each code token. Different types play very different semantic or syntactic roles in subsequent program analysis and compilation. Thus, it is important to understand whether pre-trained code models have captured the lexical information of source code by contextual representations. 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. As shown in Figure 1(d), each token belongs to one of the five types including *Identifier*, *Keyword*, *Operator*, *Number*, and *String*. Due to space limitation, the detailed definition of each type and description of lexical probing can be found in the online Appendix of the replication package [51].

Syntactic probing. Syntax analysis typically comes after the lexical analysis in the process of program compilation [3], where a parser takes token sequence generated by the lexer as input and produces data structures like parse tree or abstract syntax tree (AST). Similarly, syntactic probing is designed to investigate how well contextual representations perceive the syntactic properties of source code. The basic idea is to identify whether a code and an anonymous AST (named AST-Only shown in Figure 1(b)) are paired. Specifically, we first parse the source code to obtain the corresponding AST, which includes the non-terminal and terminal nodes. The non-terminal nodes represent the syntactic information, while terminal nodes consist of types corresponding to the syntactic elements and values corresponding to code tokens in the source code. It is easy for one model to identify whether the AST is

parsed by one code snippet according to the overlap of code tokens. Therefore, as shown in Figure 1(b), we construct the AST-Only by removing the values of terminal nodes. Next, we train a linear classifier to determine whether the given AST-Only is parsed from the given code snippet or not. The true pairs are constructed by pairing the code with the corresponding parsed AST-Only, while false pairs are constructed by pairing the code with an AST-Only parsed by other different codes. Detailed descriptions of syntactic probing can be found in the online Appendix [51].

Semantic probing. To understand to what extent pre-trained code models are aware of code semantics, we perform semantic probing (Figure 1(e)), which examines the ability to identify code snippets with the same semantics but different implementations. Specifically, we use POJ-104 [39] dataset, which consists of 104 problems and 500 C/C++ implementations for each problem, as the evaluated dataset. We train a linear mapper taking the pre-trained representations as input to map semantically similar code snippets into similar embeddings. Thus, the implementations with the same semantic can be easily recalled by the vector distance of them. Detailed descriptions of semantic probing can be found in the online Appendix [51].

Structural probing. In addition to lexical, syntactic, and semantic properties, structural properties are also important for code analysis. Cyclomatic complexity [36], which indicates the complexity of a program and can be referred to control flow graph (CFG), can be used as a structural property of code. Mathematically, the cyclomatic complexity M can be calculated based on the CFG of source code by:

$$M = E - N + 2P \quad (3)$$

where E and N are the number of edges and nodes of the graph, respectively. P is the number of connected components. Its value is typically 1 because the CFG is a connected graph. As shown in Figure 1(c), the CFG has 7 nodes and 7 edges, hence the cyclomatic complexity of the code snippet is $7 - 7 + 2 = 2$. We use the cyclomatic complexity prediction as the structural probing task to investigate how well contextual representations understand the structural property of source code. Detailed descriptions of structural probing can be found in the online Appendix [51].

3.2.2 Probing pipeline. Following previous studies [37, 52], to investigate what code properties are encoded in layer-wise representations of the pre-trained model, we train a classifier that takes these layer-wise representations as input to predict the probing task associated with one of code properties. At the same time, we learn a linear combination of contextual representations of all layers to study how much representations of each layer contribute to understanding these code properties. Mathematically, for the pre-trained layer-wise representations H^0, H^1, \dots, H^L , we combine them by:

$$F = \sum_{l=1}^L \lambda^l H^l, \quad \lambda^l = \frac{\exp a_l}{\sum_{i=0}^L \exp a_i} \quad (4)$$

where the layer-wise weight a_l are jointly learned with the probing classifier. On the one hand, we compare the performance between combined representations F and randomly initialized representations to study how well the pre-trained contextual representations

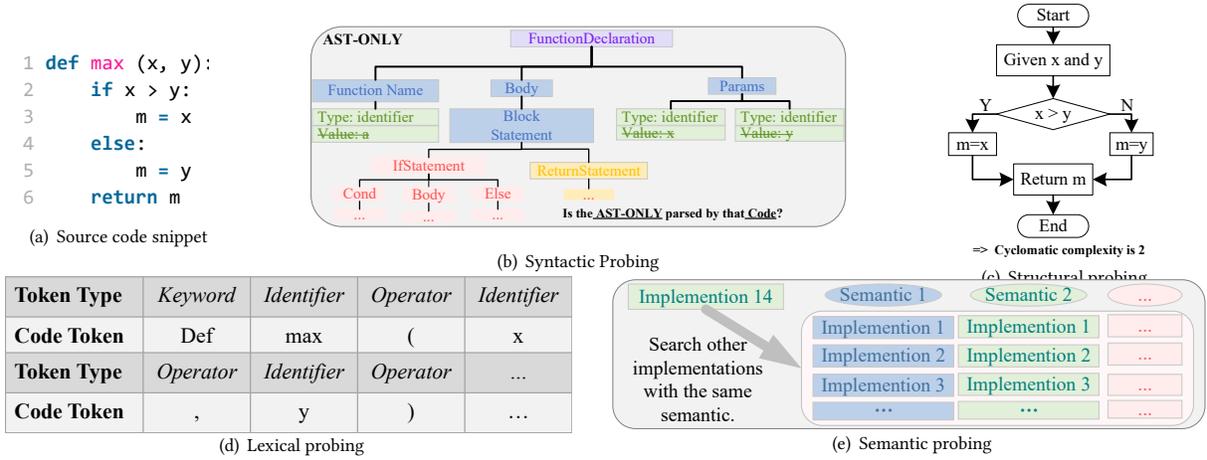


Figure 1: An example of source code and probing tasks

Table 1: An overview of downstream tasks, which includes descriptions, and evaluated datasets, programming languages and metrics. #Size shows the sizes of train, validation and test sets in order. For metrics, P, R, F1 are short for precision, recall and F1-score, respectively. EM and Edit sim are short for Exact Match accuracy and Levenshtein edit similarity, respectively.

Task	Description	Dataset Name	Language	#Size	Metrics
Code search	Search semantically relevant code snippets for a given natural language query.	CodeSearchNet [23]	Python Ruby	251K/9.6K/1K 24.9K/1.4K/1.3K	MRR, R@1 R@5, R@10
Clone detection	Detect whether two code snippets are functional equivalence.	BigCloneBench [49]	Java	901K/416K/416K	P, R, F1
Code summarization	Generate the concise natural language description for the given code snippet.	CodeSearchNet [23]	Python Ruby	251K/9.6K/1K 24.9K/1.4K/1.3K	BLEU, Meteor, Rouge-L, Cider
Code generation	Generate a function-level code snippet for the given natural language description.	CONCODE [24]	Java	100K/2K/2K	BLEU, EM
Code completion	Predict the next line of code for the given previous code context.	Github Java Corpus [5]	Java	12K/1.5K/1.5K	Edit sim, EM

encode the properties of source code. We also compare the performance between pre-trained and fine-tuned representations to study what happens to the code properties during fine-tuning. On the other hand, to investigate how much the representation of each layer contributes to encoding a code property and to explore the differences between pre-trained and fine-tuned models, we present layer-wise weight a_l of pre-trained and fine-tuned models for each probing task and further analyze the experimental results Section 3.5.1.

3.3 Representational Similarity Analysis

Following the previous study [37], we randomly sample N code snippets and obtain the layer-wise representations of the pre-trained and fine-tuned model. Then, for each layer, we obtain distance matrix A^l (introduced in Section 2.3 and the size is $N \times N$) for the l -th layer by calculating the cosine similarity between this layer’s representational vectors of any two code snippets. Representational vectors are obtained by averaging that layer’s contextual representations. Mathematically, we denote the l -th contextual representations of the pre-trained or fine-tuned code model for the

k -th code snippet as H_k^l . The distance matrix A^l is calculated by:

$$A_{i,j}^l = \frac{v_i^l \cdot v_j^l}{\|v_i^l\| \|v_j^l\|}, \quad v_k^l = \text{mean}(H_k^l), \quad i, j, k \in [1, 2, \dots, N] \quad (5)$$

Next, for the l -th layer, we calculate the Pearson’s correlation coefficients ρ^l between the two distance matrices obtained from pre-trained and fine-tuned models, respectively. In particular, we conduct experiments on five diverse downstream tasks including code search, clone detection, code summarization, code generation, and line-level code completion. The overview of them is in Table 1. The experimental results are shown in Section 3.5.2.

3.4 Experimental Settings

In this study, we analyze the state-of-art pre-trained code models UniXcoder [17] and GraphCodeBERT [18]. Both of them are 12-layer Transformer with 768 dimensions and the total parameters are about 120 MB. UniXcoder is a unified pre-trained code model and can be used as an encoder, a decoder, or an encoder-decoder architecture by a special indication token. GraphCodeBERT considers the data flow information and pre-trains a large model using a lot of bimodal data (code functions paired with natural

Table 2: The performance of probing tasks for random, pre-trained and fine-tuned representations.

Probing Task	Performance		
	Random	Pre-trained	Fine-tuned
Lexical probing	76.14	99.98	99.95
Syntactic probing	64.70	95.60	95.10
Semantic probing	46.10	73.25	71.55
Structural probing	34.80	89.80	62.20

language comments) and unimodal codes data. We conduct the experiment on UniXcoder and GraphCodeBERT because they all achieve promising results on many code intelligence tasks.

For the experiments on probing, we construct the evaluation datasets through CodeSeachNet and POJ-104 [39] shown in Table 1. The lexical, syntactic, and structural probeings employ the CodeSeachNet dataset with Python, and semantic probing uses the POJ-104 [39]. Following the fine-tuning experimental settings of UniXcoder/GraphCodeBERT on code search, We fine-tune the pre-trained model on CodeSeachNet dataset with Python, and probe the pre-trained and fine-tuned model with the four probing tasks. When probing, the maximum length of code snippets is set to 512. The maximum epoch and batch size are set to 30 and 32, respectively. We adopt the Adam optimizer with a learning rate of $1e-4$ and perform early stopping on the validation set. We run the experiments 3 times with random seeds 0,1,2 and display the mean value in the paper.

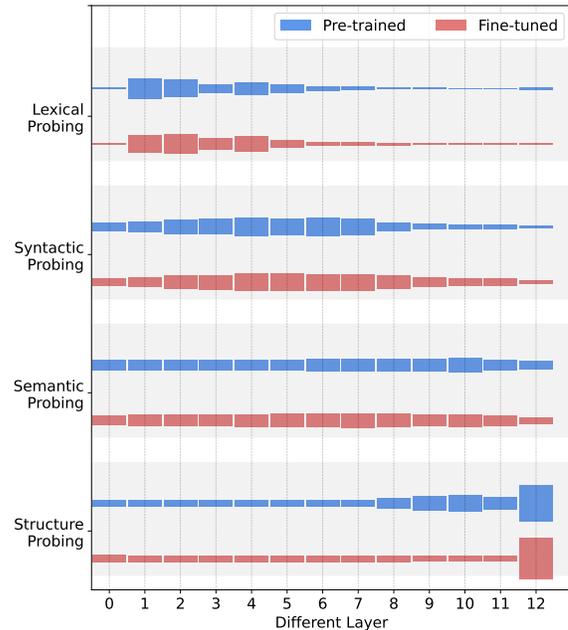
For the representational similarity analysis, following the previous study [37], N is set to 5,000. Following the fine-tuning experimental settings of UniXcoder/GraphCodeBERT, we fine-tune it on the five downstream tasks shown in Table 1.

3.5 Experimental Findings

In this section, we present and analyze the results of the above two research questions. We present the results of UniXcoder-based Telly- K only due to space limitation and put results of GraphCodeBERT-based Telly- K in the online Appendix [51]. Conclusions and findings that hold on UniXcoder generally hold for GraphCodeBERT.

3.5.1 RQ1: What code properties are encoded in layer-wise pre-trained representations? We use the four probing tasks related to lexical, syntactic, semantic and structural properties to explore what code properties are encoded in layer-wise pre-trained representations and how much representations of each layer contribute to understanding these code properties. At the same time, we also compare pre-trained and fine-tuned layer-wise representations in the same setting. The performance on probing tasks is shown in Table 2 and the layer-wise contribution are presented in Figure 2.

In Table 2, the results of lexical, syntactic, and structural probing are measured by accuracy. The results of semantic probing are measured by mean average precision (MAP) [44]. For more detailed descriptions of the accuracy and MAP metrics, please refer to the online Appendix [51]. From Table 2, we can find that (1) pre-trained and fine-tuned representations better understand code properties than random representations; (2) after fine-tuning, lexical, syntactic and semantic code properties are still well captured, while the ability to capture the structural property significantly declines. The first finding is expected because pre-trained code

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

models can take advantage of larger datasets and model sizes to encode the basic code knowledge into their representations. After fine-tuning, some code properties are still preserved. This may be related to the characteristics of downstream tasks since code search mainly relies on the lexical, syntactic, and semantic information of the code rather than the structural information. It may also be the result of vanishing gradients [8] because code properties encoded in the lower layer change very little, and code properties encoded in the higher layer change obviously. Actually, vanishing gradients has little effect on the optimization of the pre-trained code model as the basic architecture employed by the model uses the residual connection [20, 56] which can effectively avoid the vanishing problem.

Figure 2 displays the layer-wise contributions (λ^l in Eq. 4) for the pre-trained and fine-tuned model. From Figure 2, *on the one hand*, we can observe that for pre-trained or fine-tuned models, lexical, syntactic, and structural properties of source code are mostly captured in the lower, intermediate, and higher layers, respectively, while the semantic property almost spans across the entire model. We conduct the significance testing¹ to examine the significance of the contribution differences. The experimental result shows that for lexical probing, the contributions of the 1-th, 2-th, and 4-th layers are significantly greater than other layers. For syntactic probing, the contributions of the 4-th to 7-th layers are significantly greater than other layers. For semantic probing, the contributions among different layers are not significantly different. For structural probing, the contribution of the last layers of the pre-trained and fine-tuned model is significantly greater than others. *On the other hand*, we find that the process of fine-tuning preserves most of the code properties. Specifically, basic code properties captured by lower

¹The results are put in the online Appendix [51] due to space limitation

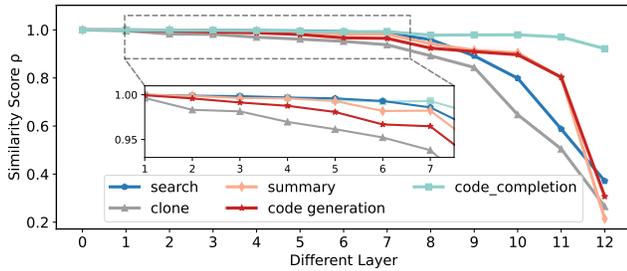


Figure 3: Similarity scores between pre-trained and fine-tuned models for five downstream tasks.

and intermediate layers are still preserved during fine-tuning. Only the performance of structural probing task changes obviously.

Summary. 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. Meanwhile, the basic code properties captured by lower and intermediate layers are still preserved during fine-tuning.

3.5.2 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 the pre-trained model during the fine-tuning for five diverse downstream tasks without the help of probing tasks. The results are shown in Figure 3. From the presented results, we can see that the correlation coefficients (similarity scores 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. This is because the top layer of the pre-trained model UniXcoder are used to predict the masked tokens, which is similar to the experimental setting of code completion. Furthermore, we find that the representation of the bottom 7 layers ($\rho \geq 0.9$) are greatly related and the bottom 5 layers ($\rho \geq 0.95$) are strongly similar.

Summary. The representations of the bottom nine layers are similar between the pre-trained and fine-tuned models for the five downstream tasks. Only the representations of the top two layers change greatly during fine-tuning.

4 EFFICIENT FINE-TUNING OF PRE-TRAINED CODE MODELS

4.1 Research Question

RQ3: Are there efficient alternatives to fine-tuning? Based on the results of the experimental study, we investigate more efficient alternatives to fine-tune pre-trained code models. Our primary motivation is to freeze the pre-trained parameters of those layers that change only slightly during the fine-tuning of downstream tasks. We propose **Telly-K**, which stands for efficient fine-tuning of pre-trained code models via layer freezing. Telly-K means freezing the pre-trained parameters of the bottom K layers and different K means different variants of our approach. The 0-th layer is the

embedding layer, and the maximum number of layers of our studied pre-trained code model is 12. If K is set to 12, then Telly-12 will freeze all parameters and the model will be reduced to the vanilla pre-trained model. Therefore, for the comprehensiveness of the experiments, we vary K from 0 to 11 and conduct extensive experiments on five downstream tasks for these 12 model variants. Next, we introduce the experimental settings, results, and analysis.

4.2 Experimental Settings

Our experiments are conducted on five diverse downstream tasks including code search, code detection, code summarization, code generation, and line-level code completion. The overview of these tasks is presented in Table 1. *Code search* is evaluated on the widely-used CodeSearchNet dataset with Python and Ruby and the performance is measured by mean reciprocal rank (MRR) and top-k recall ($R@k$, $k=1,5,10$) [15, 18, 23]. For *Clone detection*, we experiment on the commonly-used BigCloneBench dataset and use the precision (P), recall (R), and F1-score (F1) as evaluation metrics [17, 18]. For *code summarization*, we fine-tune pre-trained models [17, 18] on CodeSearchNet dataset with Python and Ruby. The evaluation metrics are sentence-level smoothing BLEU [41], Meteor [6], Rouge-L [32] and Cider [57]. *Code generation* is evaluated on the widely-used CONCODE dataset, and the performance is measured by sentence-level smoothing BLEU [41] and Exact Match accuracy (EM). For *line-level code completion*, We conduct the experiments on the large dataset (named GitHub Java Corpus) in CodeXGLUE. Similar to previous work [34], the performance is measured by EM and Levenshtein edit similarity (Edit sim) [50]. The fine-tuning of line-level code completion according to experimental settings in CodeXGLUE [34]. Other tasks follow the settings of previous studies [17, 18]. We adopt the Adam optimizer with the maximum epoch of 30 and perform early stopping on the validation set. We run the experiments 3 times with random seeds 0,1,2 and display the mean value in the paper. All experiments are conducted on a machine with Tesla A100 GPU. Detailed experimental settings can be found in the online Appendix of the replication package [51].

4.3 Experimental Results

We conduct experiments with Telly- K , which freezes the bottom K layers of the pre-trained code model when fine-tuning it on the five downstream tasks. We first present and analyze the experimental results including training parameters, training time cost and performance task by task, and then summarize the findings on the downstream tasks. Due to space limitations, we only present the results of UniXcoder-based Telly- K and we put results of GraphCodeBERT-based Telly- K in the online Appendix [51]. Conclusions and findings that hold for UniXcoder also generally hold for GraphCodeBERT.

4.3.1 Code search. We study the performance of 12 Telly- K variants on code search. The results are shown in Table 3. We only show the results on Python dataset and put the results on Ruby (which is similar to the results on Python) in online Appendix [51].

In Table 3, we present the numbers of trained parameters, training time and performance for the base model and 12 model variants. The base model is to fine tune all pre-trained parameters, while the different variants would freeze partial parameters. We report both the training time of each epoch and time til model convergence.

Table 3: Experimental results on code search. #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	17m14s	2h35m06s	0.720	0.612	0.838	0.889
Telly-0	Freezing the bottom 0 layers	85.0M(↓32%)	15m49s(↓8%)	2h06m32s(↓18%)	0.727(↑1%)	0.630(↑3%)	0.846(↑1%)	0.895(↑1%)
Telly-1	Freezing the bottom 1 layers	78.0M(↓38%)	14m53s(↓14%)	1h59m04s(↓23%)	0.727(↑1%)	0.629(↑3%)	0.848(↑1%)	0.895(↑1%)
Telly-2	Freezing the bottom 2 layers	70.9M(↓44%)	14m06s(↓18%)	1h52m48s(↓27%)	0.727(↑1%)	0.630(↑3%)	0.849(↑1%)	0.896(↑1%)
Telly-3	Freezing the bottom 3 layers	63.8M(↓49%)	13m12s(↓23%)	0h39m36s(↓74%)	0.724(↑1%)	0.626(↑2%)	0.842(↑0%)	0.895(↑1%)
Telly-4	Freezing the bottom 4 layers	56.7M(↓55%)	12m12s(↓29%)	0h48m48s(↓69%)	0.724(↑1%)	0.626(↑2%)	0.844(↑1%)	0.896(↑1%)
Telly-5	Freezing the bottom 5 layers	49.6M(↓61%)	11m42s(↓32%)	0h35m06s(↓77%)	0.726(↑1%)	0.628(↑3%)	0.848(↑1%)	0.896(↑1%)
Telly-6	Freezing the bottom 6 layers	42.5M(↓66%)	10m50s(↓37%)	0h32m30s(↓79%)	0.727(↑1%)	0.629(↑3%)	0.848(↑1%)	0.896(↑1%)
Telly-7	Freezing the bottom 7 layers	35.4M(↓72%)	9m57s(↓42%)	0h29m51s(↓81%)	0.725(↑1%)	0.627(↑2%)	0.847(↑1%)	0.896(↑1%)
Telly-8	Freezing the bottom 8 layers	28.4M(↓77%)	9m04s(↓47%)	0h36m16s(↓77%)	0.723(↑0%)	0.625(↑2%)	0.844(↑1%)	0.894(↑1%)
Telly-9	Freezing the bottom 9 layers	21.3M(↓83%)	8m14s(↓52%)	0h24m42s(↓84%)	0.718(↓0%)	0.620(↑1%)	0.838(0%)	0.888(↓0%)
Telly-10	Freezing the bottom 10 layers	14.2M(↓89%)	7m10s(↓58%)	0h21m30s(↓86%)	0.710(↓1%)	0.612(↓0%)	0.829(↓1%)	0.882(↓1%)
Telly-11	Freezing the bottom 11 layers	7.1M(↓94%)	6m21s(↓63%)	0h19m03s(↓88%)	0.694(↓4%)	0.593(↓3%)	0.815(↓3%)	0.871(↓2%)

Table 4: Experimental results on clone detection and code summarization. #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	Clone Detection						Code Summarization				
	#Params	Training time		Performance			#Params	Training time		Performance	
		Each epoch	Convergence	Recall	Precision	F1-score		Each epoch	Convergence	BLEU	METEOR
Base	127.1M	20m21s	1h41m45s	0.95	0.95	0.95	125.93M	22m44s	1h53m40s	19.15	17.26
Telly-0	86.2M(↓32%)	19m29s(↓4%)	1h17m56s(↓23%)	0.96(↑1%)	0.94(↓1%)	0.95(0%)	85.0M(↓32%)	21m30s(↓5%)	1h47m30s(↓5%)	19.19(↑0%)	17.32(↑0%)
Telly-1	79.2M(↓38%)	18m36s(↓9%)	1h14m24s(↓27%)	0.95(0%)	0.95(0%)	0.95(0%)	78.0M(↓38%)	20m15s(↓11%)	1h41m15s(↓11%)	19.21(↑0%)	17.33(↑0%)
Telly-2	72.1M(↓43%)	17m34s(↓14%)	1h10m16s(↓31%)	0.95(0%)	0.95(0%)	0.95(0%)	70.9M(↓44%)	19m11s(↓16%)	1h35m55s(↓16%)	19.17(↑0%)	17.30(↑0%)
Telly-3	65.0M(↓49%)	15m20s(↓25%)	1h01m20s(↓40%)	0.95(0%)	0.95(0%)	0.95(0%)	63.8M(↓49%)	17m59s(↓21%)	0h53m57s(↓53%)	19.16(↑0%)	17.26(0%)
Telly-4	57.9M(↓54%)	14m28s(↓29%)	0h57m52s(↓43%)	0.94(↓1%)	0.96(↑1%)	0.95(0%)	56.7M(↓55%)	17m07s(↓25%)	0h51m21s(↓55%)	19.13(↓0%)	17.26(0%)
Telly-5	50.8M(↓60%)	13m25s(↓34%)	1h07m05s(↓34%)	0.96(↑1%)	0.94(↓1%)	0.95(0%)	49.6M(↓61%)	16m18s(↓28%)	0h48m54s(↓57%)	19.18(↑0%)	17.26(0%)
Telly-6	43.7M(↓66%)	12m35s(↓38%)	0h50m20s(↓51%)	0.96(↑1%)	0.95(0%)	0.95(0%)	42.5M(↓66%)	15m10s(↓33%)	0h45m30s(↓60%)	19.36(↑1%)	17.35(↑1%)
Telly-7	36.6M(↓71%)	11m44s(↓42%)	0h58m40s(↓42%)	0.95(0%)	0.93(↓2%)	0.94(↓1%)	35.4M(↓72%)	14m08s(↓38%)	0h28m16s(↓75%)	19.37(↑1%)	17.28(↑0%)
Telly-8	29.5M(↓77%)	10m41s(↓48%)	0h53m25s(↓48%)	0.95(0%)	0.94(↓1%)	0.95(0%)	28.4M(↓77%)	12m59s(↓43%)	0h25m58s(↓77%)	19.34(↑1%)	17.26(0%)
Telly-9	22.4M(↓82%)	9m55s(↓51%)	0h29m45s(↓71%)	0.95(0%)	0.92(↓3%)	0.93(↓2%)	21.3M(↓83%)	11m28s(↓50%)	0h22m56s(↓80%)	19.18(↑0%)	17.22(↓0%)
Telly-10	15.4M(↓88%)	8m51s(↓57%)	0h35m24s(↓65%)	0.97(↑2%)	0.92(↓3%)	0.94(↓1%)	14.2M(↓89%)	10m19s(↓55%)	0h10m19s(↓91%)	19.11(↓0%)	17.18(↓0%)
Telly-11	8.3M(↓93%)	8m00s(↓61%)	0h32m00s(↓69%)	0.96(↑1%)	0.92(↓3%)	0.94(↓1%)	7.1M(↓94%)	09m14s(↓59%)	0h09m14s(↓92%)	19.10(↓0%)	17.20(↓0%)

The changing ratios of different variant models compared to base model are shown in parentheses. From the results of Table 3, we can find that:

- 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. Especially, for Telly-11 that freezes the bottom 11 layers, the time cost of model convergence and the trained parameters are reduced by 88% and 94%, respectively, while the performance only drops by about 3%.
- For Telly- K ($0 \leq K \leq 8$), they reduce the training parameters by 32% to 77%, correspondingly saving about 18% to 81% of training time, with the performance increment of 0% to 3% across all metrics.
- When freezing the bottom 9 layers, there is an 83% reduction in training parameters and a corresponding 84% training time saving with a slight change in performance. For example, compared with the base model, the values of MRR and R@10 for Telly-9

decrease by less than 1%, R@1 increases by about 1%, and the R@5 is unchanged.

- 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.

Summary. In the code search task, 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 experiments with different Telly- K variants on clone detection and the results are shown in the left half of Table 4. As the pre-trained code model adopts 2-layer MLP as the classifier to determine whether two codes are clones or not, the total parameters are about 127.1 million. The performance is evaluated by precision (P), recall (R), and F1-score (F1) and they are in the range of [0, 1]. From Table 4, we can find that:

- For all variant models, training costs and parameters are significantly reduced compared to the base model, while there is no significant change in performance. In particular, when freezing the bottom 11 layers, the convergence time cost and the training parameters are reduced by 88% and 94%, respectively, while the performance changes by only 1-3%.
- For Telly- K ($0 \leq K \leq 8$), the training parameters are reduced by 32% to 77% and correspondingly about 23% to 51% training time costs are saved. In addition, the performance is generally stable for F1-score and slightly changes for precision and recall.
- When ($K \geq 9$), the performance of the different variants consistently decreases in precision and F1-score but increases in recall. However, even for Telly-11, all evaluation metrics have high scores (greater than 0.9), while the training parameters and the corresponding time cost are greatly reduced.

Summary. In the clone detection task, the performance of Telly- K is generally stable for $0 \leq K \leq 8$ but lightly changes for $K \geq 9$ compared to the base model.

4.3.3 Code summarization. We conduct the experiments with different Telly- K on code summarization and the results are shown in the right half of Table 4. The results of Rouge-L and Cider on Python and all experimental results on Ruby are put in online Appendix [51] due to space limit. The reported metrics including BLEU and METEOR are in the range of [0, 100]. From Table 4, we can find 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.
- For Telly- K ($0 \leq K \leq 5$), they reduce the training parameters by 32% to 61%, correspondingly saving about 5% to 57% of training time, with stable performance. In particular, the performance change is less than 1% for all evaluation metrics.
- For Telly- K ($6 \leq K \leq 9$), training parameters are reduced by 66% to 83%, corresponding to 60% to 80% saving in training time with a generally slight increase in performance.
- 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.

Summary. On the code summarization task, the performance of Telly- K is stable for ($0 \leq K \leq 5$), slightly increases for $6 \leq K \leq 9$, but lightly decreases when $K \geq 10$ compared to the base model.

4.3.4 Code generation. We conduct experiments with different Telly- K on code generation and the results are shown in the left half of Table 5. The evaluation metrics including BLEU and EM are in the range of [0, 100]. From Table 5, we can find 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 ($K \geq 8$).
- For Telly- K ($0 \leq K \leq 5$), they reduce the training parameters by 32% to 61%, correspondingly saving about 10% to 55% of training

time, and the performance is slightly improved. Specifically, the BLEU scores are increased by 0% to 2%, and the EM scores are increased by 3% to 10%.

- For Telly- K ($6 \leq K \leq 9$), training parameters are reduced by 66% to 83%, corresponding to 57% to 65% saving in training time. The performance of these variants significantly drops under BLEU but generally increases under EM. This is because BLEU combines the n-gram precision ($n=1,2,3,4$) between the generated code snippet and the ground truth for one sample, while EM is 1 if they are exactly the same, 0 otherwise. However, for the entire set, only about 18% of the code snippets can be generated exactly the same as ground truth. The other 82% samples also affect the final result of BLEU scores. Therefore, the performance changes for BLEU and EM behave differently.
- When Telly- K ($K \geq 10$), training parameters and the corresponding time cost are greatly reduced, and the performance of variants also significantly drops on both metrics.

Summary. On the code generation task, the performance of Telly- K is slightly improved for Telly- K ($0 \leq K \leq 5$), obviously changes for $6 \leq K \leq 9$, and significantly drops when $K \geq 10$ 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 the right half of Table 5. The evaluated metrics including Edit Sim and EM are in the range of [0, 100]. From Table 5, we can find 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 except the Telly-11.
- For Telly- K ($0 \leq K \leq 7$), they reduce the training parameters by 32% to 72%, correspondingly saving about 13% to 75% of training time. In addition, the performance of Telly- K ($0 \leq K \leq 3$) is lightly improved and the performance of Telly- K ($4 \leq K \leq 7$) slightly drops.
- For Telly- K ($K \geq 8$), training parameters and corresponding time cost are hugely reduced. The performance of variants generally drops for two metrics. Especially, when $K = 11$, the performance significantly drops. Therefore, the pre-trained parameters of the last layers need to be fine-tuned to learn to predict the next line of code.

Summary. On code completion task, the performance of Telly- K is slightly improved for ($0 \leq K \leq 10$ and significantly drops when $K = 11$ compared to the base model.

4.3.6 Findings across all downstream tasks. After analyzing the experimental results task by task, we summarize the general findings across various tasks as follows.

- For all Telly- K , both the training time cost (especially the convergence time cost) and the training parameters are significantly reduced compared to the base model. The performance does not change much, except for Telly-10 and Telly-11 on code generation and Telly-11 on code completion.
- When $0 \leq K \leq 5$, Telly- K generally reduces the training parameters by 30% to 65%, correspondingly saving about 10% to 70% of training time, with the performance generally increasing varying

Table 5: Experimental results on code generation and line-level code completion. #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	Code Generation					Line-Level Code Completion				
	#Params	Training time		Performance		#Params	Training time		Performance	
		Each epoch	Convergence	BLEU	EM		Each epoch	Convergence	Edit sim	EM
Base	125.93M	12m25s	4h08m20s	33.82	17.4	125.93M	04m12s	0h37m48s	51.92	20.40
Telly-0	85.0M(↓32%)	11m43s(↓6%)	3h42m37s(↓10%)	33.88(↑0%)	18.1(↑4%)	85.0M(↓32%)	03m59s(↓5%)	0h31m52s(↓16%)	52.58(↑1%)	21.07(↑3%)
Telly-1	78.0M(↓38%)	11m07s(↓10%)	3h20m06s(↓19%)	34.43(↑2%)	17.9(↑3%)	78.0M(↓38%)	03m47s(↓10%)	0h18m55s(↓50%)	52.35(↑1%)	20.87(↑2%)
Telly-2	70.9M(↓44%)	10m32s(↓15%)	2h06m24s(↓49%)	33.85(↑0%)	19.1(↑10%)	70.9M(↓44%)	03m38s(↓13%)	0h32m42s(↓13%)	52.31(↑1%)	20.93(↑3%)
Telly-3	63.8M(↓49%)	09m52s(↓21%)	2h08m16s(↓48%)	34.24(↑1%)	19.0(↑9%)	63.8M(↓49%)	03m29s(↓17%)	0h20m54s(↓45%)	51.81(↓0%)	20.73(↑2%)
Telly-4	56.7M(↓55%)	09m15s(↓26%)	1h51m00s(↓55%)	34.02(↑1%)	18.6(↑7%)	56.7M(↓55%)	03m18s(↓21%)	0h26m24s(↓30%)	51.62(↓1%)	20.27(↓1%)
Telly-5	49.6M(↓61%)	08m44s(↓30%)	1h53m32s(↓54%)	34.36(↑2%)	18.6(↑7%)	49.6M(↓61%)	03m07s(↓26%)	0h09m21s(↓75%)	51.66(↓0%)	20.40(0%)
Telly-6	42.5M(↓66%)	08m13s(↓34%)	1h46m49s(↓57%)	32.90(↓3%)	18.1(↑4%)	42.5M(↓66%)	02m57s(↓30%)	0h11m48s(↓69%)	51.36(↓1%)	20.27(↓1%)
Telly-7	35.4M(↓72%)	07m41s(↓38%)	1h39m53s(↓60%)	32.92(↓3%)	18.0(↑3%)	35.4M(↓72%)	02m58s(↓29%)	0h14m50s(↓63%)	51.32(↓1%)	20.40(0%)
Telly-8	28.4M(↓77%)	07m08s(↓43%)	1h32m44s(↓63%)	32.12(↓5%)	17.4(0%)	28.4M(↓77%)	02m59s(↓29%)	0h14m55s(↓65%)	50.95(↓2%)	20.27(↓1%)
Telly-9	21.3M(↓83%)	06m42s(↓46%)	1h27m06s(↓65%)	31.33(↓7%)	18.1(↑4%)	21.3M(↓83%)	02m58s(↓29%)	0h14m50s(↓67%)	51.13(↓2%)	20.73(↑2%)
Telly-10	14.2M(↓89%)	06m03s(↓51%)	1h12m36s(↓71%)	30.43(↓10%)	16.6(↓5%)	14.2M(↓89%)	02m57s(↓30%)	0h11m48s(↓76%)	50.70(↓2%)	20.53(↑1%)
Telly-11	7.1M(↓94%)	05m22s(↓57%)	1h09m46s(↓72%)	27.71(↓18%)	14.3(↓18%)	7.1M(↓94%)	02m06s(↓50%)	0h06m18s(↓83%)	49.49(↓5%)	19.67(↓4%)

from 1% to 4% in terms for various downstream tasks. Therefore, Telly-5 is usually the best choice among the variants which reduce resource consumption and performance improvement.

- When $6 \leq K \leq 9$, Telly- K generally reduces the training parameters by 65% to 80%, correspondingly saving about 50% to 80% of training time, with the performance generally marginally changes for various downstream tasks. Specifically, the performance of code search and code generation varies from 1% to 5%, clone detection and code summarization varies from 0% to 1%, and code completion varies from 0% to 3%. Therefore, Telly-9 is usually the best choice among the variants which reduce resource consumption with a small change in performance.
- When $K \geq 10$, both training parameters and time cost are greatly reduced. The performance of Telly- K drops but not significantly except for code generation and code completion.

The performance of models increases despite freezing some layers in the above experiments. One possible explanation is that the reduction in the number of parameters alleviates overfitting, allowing parameters fine-tuned on the training set to better generalize to the testing set. However, the exact mechanism behind this phenomenon is still not well understood and requires further investigations, such as studying more datasets with different distributions between training and testing sets, to verify our conjecture.

Summary. For Telly- K across various downstream tasks, both the training parameters and the time cost are extremely reduced compared to the base model. In general, the performance of Telly- K increases by 1% to 4% for $0 \leq K \leq 5$, slight changes for $6 \leq K \leq 9$, and obviously drops when $K \geq 10$ compared to the base model.

5 DISCUSSIONS AND THREATS TO VALIDITY

5.1 Importance of Reducing Fine-tuning Time Costs and the Advantages of Telly- K .

It is important to reduce fine-tuning costs especially time costs because (1) the pre-training with fine-tuning paradigm shows increasing adoption in many software engineering tasks [2, 15, 17, 34]. Reducing time cost is important as it typically takes much time

to fine-tune a model, especially on larger datasets. Reducing fine-tuning time costs can also improve the development efficiency of pre-trained code models, especially when developers need to meet product launching deadlines, thereby saving costs and reducing unexpected losses. (2) GPUs are usually expensive computing resources. We can save GPU resources with reduced time costs. Moreover, Training deep-learning-based models, including fine-tuning pre-trained models, can emit over 600K tons of CO2 each year [48]. By reducing time costs, carbon emissions can be reduced.

In fact, many approaches [22, 26, 29, 35, 59, 60] have been proposed to save fine-tuning time costs. Among them, distillation techniques [29, 35, 60] are promising and popular. Specifically, they propose different approaches to compress a large pre-trained model into a smaller model, and fine-tune the smaller model to perform downstream tasks. However, these techniques need to carefully design and tune the architectures of small models and loss functions to distill knowledge from big models. Therefore, generally the process of distillation requires more manual design and is more laborious. Our approach Telly- K is simple and can directly decrease the training cost via layer freezing.

5.2 How to help better fine-tuning in the future

In this paper, we have not concluded a “one-method-to-rule-them-all” suggestion for different tasks. However, our study of RQ2 and RQ3 show that Telly-7, which freezes the bottom 7 layers, achieves a good trade-off between efficiency and performance. Specifically, (1) from Figure 3, we can see that the representations of the bottom 7 layers between pre-trained and fine-tuned models are very similar; (2) from Tables 3, 4 and 5, we can see that Telly-7 significantly reduces the training time and parameters while maintaining similar performance for all downstream tasks. Therefore, it is recommended to use Telly-7 in practice. In future work, we aim to investigate automated layer freezing strategies to use layer-wise representations more efficiently. For example, we plan to design algorithms to automatically select features from different layers and aggregate them to perform different downstream tasks.

5.3 Threats to Validity

We have identified the following threats to our study:

Program Languages. We conduct experiments on five programming languages (Python, Java, Ruby, C, and C++). Although in principle, our studied models are not specifically designed for certain languages, models may perform differently on different programming languages. Therefore, more experiments are needed to confirm the generality of our findings and conclusions. In the future, we will extend our study to more programming languages.

Evaluation Datasets. We conduct the experiments on widely-used datasets. Besides, there are other datasets for each downstream task. They are different in some aspects such as construction methods and corpus sizes. Model may perform differently on different datasets. Thus, we will conduct experiments on more datasets to confirm the generality of our findings and conclusions.

Evaluation Metrics. We use as many commonly-used metrics as possible to evaluate model performance in this study. However, these metrics may have their inherent limitations. For example, BLEU and METEOR are textual similarity-based metrics and cannot measure the semantic similarity of two sentences. In the future, we will use more metrics and human evaluation to confirm the findings and conclusions in this study.

Pre-trained Code Models. Due to computational resource constraints, we focus on the state-of-art pre-trained code model UniX-coder and GraphCodeBERT in this study. Other pre-trained code models such as CodeGPT [34] and CodeT5 [63] are yet to be studied.

6 RELATED WORK

6.1 Probing Pre-trained Models

In the natural language processing community, many studies [1, 10, 37, 52, 53] have investigated how pre-trained language models understand natural language and what happens when fine-tuning them. They are typically divided into two categories. The first employs probing techniques to study what linguistic properties are captured by pre-trained language models [52, 53]. The second is representational similarity analysis [1, 10, 37]. It is a task-agnostic analysis and is used to measure the similarity between two different representational spaces. However, in software engineering field, few studies explore what happens to pre-trained code models *during the fine-tuning process*. Most related studies [15, 21, 30, 54, 58] aim to understand what pre-trained code models know about source code. For example, Wan et al. [58] conduct a structural analysis to demonstrate that the pre-trained models are aware of syntactic structure. López et al [21] recover ASTs from hidden representations of pre-trained language models. Karmakar et al [30] also propose some new probing tasks to investigate what pre-trained code models know about code. Inspired by the compilation process and static analysis, we first propose four probing tasks involving the lexical, syntactical, semantic, and structural code properties. Then we investigate what code properties are encoded in layer-wise pre-trained representations and what happens to these representations during fine-tuning.

6.2 Accelerating the Fine-tuning Process

There are many studies on accelerating fine-tuning process [22, 26, 29, 35, 59, 60]. These studies can be roughly categorized into two categories. The first is to use the knowledge distillation technique to compress large-scale pre-trained language models [29, 35, 60].

For example, Jiao et al. [29] propose TinyBERT to distill BERT and only use about 28% parameter for natural language understanding. The second is the adapter-based fine-tuning approach [22, 59], where adapters are new trainable modules added between layers of pre-trained models. For example, Houlsby et al. [22] design some adapters with two orders of magnitude fewer parameters to fine-tune compared to full models and achieve similar performance with fine-tuning all parameters of the pre-trained mode. In addition, there are some studies on efficient neural network training from scratch with layer freezing [28, 62]. For example, Wang et al. [62] leverage the knowledge from a reference model to accurately evaluate individual layers' training plasticity, freeze the converged ones and unfreeze the frozen layers to continue training. Our study could motivate researchers to come up with more efficient fine-tuning approaches.

7 CONCLUSION

In this paper, we firstly conduct extensive experimental study to explore what happens to layer-wise code knowledge and pre-trained representations during fine-tuning. We then propose efficient alternatives to fine-tune the large pre-trained code model based on the above findings. Our experimental study shows that the lexical, syntactic, and structural properties of source code are mainly captured in the lower, intermediate, and higher layers, respectively, while the semantic property spans across the entire model. The basic code properties captured by lower and intermediate layers are still preserved during fine-tuning. Furthermore, we find that only the representations of the top two layers change the most during fine-tuning for various downstream tasks. Based on the above findings, we propose Telly- K that efficiently fine-tunes pre-trained code models via selective layer freezing. The extensive experiments on five various downstream tasks demonstrate that both training parameters and time costs can be greatly reduced, while performance is similar or even better.

Furthermore, our experimental study shows many useful findings and promising directions for efficient fine-tuning of pre-trained code models. For example, we find that pre-trained code models encode syntactic properties into intermediate layers. Therefore, when the downstream tasks are syntax-aware such as code generation, we could design some modules to make use of the contextual representations of intermediate layers. In the future, we will continue to explore various promising directions and propose more efficient fine-tuning approaches.

Replication package including source code, datasets, and online Appendix is available at: <https://github.com/DeepSoftwareAnalytics/Telly>.

ACKNOWLEDGEMENT

We thank reviewers for their valuable comments on this work. This research was supported by National Key R&D Program of China (No. 2017YFA0700800) and Fundamental Research Funds for the Central Universities under Grant xtr072022001.

REFERENCES

- [1] Samira Abnar, Lisa Beinborn, Rochelle Choenni, and Willem H. Zuidema. 2019. Blackbox Meets Blackbox: Representational Similarity & Stability Analysis of

- Neural Language Models and Brains. In *BlackboxNLP@ACL*. Association for Computational Linguistics, 191–203.
- [2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *NAACL-HLT*. Association for Computational Linguistics, 2655–2668.
 - [3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools*. Pearson Education India.
 - [4] Haldun Akoglu. 2018. User’s guide to correlation coefficients. *Turkish journal of emergency medicine* 18, 3 (2018), 91–93.
 - [5] 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.
 - [6] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *IEE Evaluation@ACL*.
 - [7] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. 2022. BEiT: BERT Pre-Training of Image Transformers. In *ICLR*. OpenReview.net.
 - [8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.
 - [9] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by “Naturalizing” source code. (2022).
 - [10] Grzegorz Chrupala and Afra Alishahi. 2019. Correlating Neural and Symbolic Representations of Language. In *ACL (1)*. Association for Computational Linguistics, 2952–2962.
 - [11] Yingnong Dang, Song Ge, Ray Huang, and Dongmei Zhang. 2011. Code clone detection experience at Microsoft. In *Proceedings of the 5th International Workshop on Software Clones*. 63–64.
 - [12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.
 - [13] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *ACL (1)*. Association for Computational Linguistics, 6300–6312.
 - [14] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning Approach for Semantic Code Search. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2994–2998.
 - [15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiao Cheng 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 *EMNLP (Findings)*.
 - [16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. ACM, 933–944.
 - [17] 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.
 - [18] 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>
 - [19] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross B. Girshick. 2022. Masked Autoencoders Are Scalable Vision Learners. In *CVPR*. IEEE, 15979–15988.
 - [20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
 - [21] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.
 - [22] 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 *ICML (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 2790–2799.
 - [23] 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)
 - [24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).
 - [25] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *EMNLP*. Association for Computational Linguistics, 1643–1652.
 - [26] Junguang Jiang, Yang Shu, Jianmin Wang, and Mingsheng Long. 2022. Transferability in Deep Learning: A Survey. *arXiv preprint arXiv:2201.05867* (2022).
 - [27] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *ICSE*. IEEE, 1161–1173.
 - [28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.
 - [29] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *EMNLP (Findings) (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 4163–4174.
 - [30] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1332–1336.
 - [31] Nikolaus Kriegeskorte, Marieke Mur, and Peter A Bandettini. 2008. Representational similarity analysis—connecting the branches of systems neuroscience. *Frontiers in systems neuroscience* (2008), 4.
 - [32] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *ACL*.
 - [33] 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. *CoRR* abs/1907.11692 (2019).
 - [34] 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).
 - [35] Wenhao Lu, Jian Jiao, and Ruofei Zhang. 2020. Twinbert: Distilling knowledge to twin-structured compressed bert models for large-scale retrieval. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2645–2652.
 - [36] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software engineering* 4 (1976), 308–320.
 - [37] Amil Merchant, Elahe Rahimtoroghi, Ellie Pavlick, and Ian Tenney. 2020. What Happens To BERT Embeddings During Fine-tuning?. In *BlackboxNLP@EMNLP*. Association for Computational Linguistics, 33–44.
 - [38] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis. *Notes. Feb* (2012).
 - [39] 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.
 - [40] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *ICSE*. ACM, 1–13.
 - [41] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*. ACL, 311–318.
 - [42] Julian Aron Prener and Romain Robbes. 2021. Automatic Program Repair with OpenAI’s Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922* (2021).
 - [43] 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. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.
 - [44] 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.
 - [45] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.
 - [46] Ensheng Shi, Wenchao Gub, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation. *arXiv preprint arXiv:2204.03293* (2022).
 - [47] 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*.
 - [48] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and Policy Considerations for Modern Deep Learning Research. In *AAAI*. AAAI Press, 13693–13696.
 - [49] 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.

- [50] 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.
- [51] Telly. 2023. Replication Package. *ISSTA (2023)*. <https://github.com/DeepSoftwareAnalytics/Telly>
- [52] Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT Rediscovered the Classical NLP Pipeline. In *ACL (1)*. Association for Computational Linguistics, 4593–4601.
- [53] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R. Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R. Bowman, Dipanjan Das, and Ellie Pavlick. 2019. What do you learn from context? Probing for sentence structure in contextualized word representations. In *ICLR (Poster)*. OpenReview.net.
- [54] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Code. *arXiv preprint arXiv:2202.08975 (2022)*.
- [55] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *ICSE*. ACM, 2291–2302.
- [56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.
- [57] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *CVPR*.
- [58] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture? - A Structural Analysis of Pre-Trained Language Models for Source Code. In *ICSE*. ACM, 2377–2388.
- [59] Ruize Wang, Duyu Tang, Nan Duan, Zhongyu Wei, Xuanjing Huang, Guihong Cao, Daxin Jiang, Ming Zhou, et al. 2020. K-adapter: Infusing knowledge into pre-trained models with adapters. *arXiv preprint arXiv:2002.01808 (2020)*.
- [60] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems* 33 (2020), 5776–5788.
- [61] Yanlin Wang, Lun Du, Ensheng Shi, Yuxuan Hu, Shi Han, and Dongmei Zhang. 2020. *Cocogum: Contextual code summarization with multi-relational gnn on umls*. Technical Report. Microsoft, MSR-TR-2020-16. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/cocogum-contextual-code-summarization-with-multi-relational-gnn-on-umls>.
- [62] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. 2022. Efficient DNN Training with Knowledge-Guided Layer Freezing. *CoRR* abs/2201.06227 (2022).
- [63] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP (1)*. Association for Computational Linguistics, 8696–8708.
- [64] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *ISSTA*. ACM, 39–51.