# HGAdapter: Hypergraph-based Adapters in Language Models for Code Summarization and Clone Detection

**Guang Yang**[1,2*], **Yujie Zhu**[2]
[1]Data Technology Group, Technology Research and Development Department,
Guotai Haitong Securities, China
[2]School of Computer Science and Technology, East China Normal University, China
{51215901104, 52205901006}@stu.ecnu.edu.cn

## Abstract

Pre-trained language models (PLMs) are increasingly being applied to code-related tasks. Although PLMs have achieved good results, they do not take into account potential high-order data correlations within the code. We propose three types of high-order correlations in code tokens, i.e. abstract syntax tree family correlation, lexical correlation, and line correlation. We design a tokens and hyperedges generator to capture these high-order data correlations. We improve the architecture of hypergraph neural networks and combine it with adapter tuning to propose a novel hypergraph-based adapter (HGAdapter) to fine-tune PLMs. HGAdapter can encode high-order data correlations and is allowed to be inserted into various PLMs to enhance performance. Experiments were conducted on several public datasets, including six languages of code summarization and code clone detection tasks. Our methods improved the performance of PLMs in datasets to varying degrees. Experimental results validate the introduction of high-order data correlations that contribute to improved effectiveness.

## 1 Introduction

In recent years, the emergence of pre-trained language models (PLMs), large language models (LLMs), and small language models (SLMs) has driven the application of associated techniques to code-related tasks, including code clone detection, code classification, code summarization, and more (Niu et al., 2022; Zhang et al., 2024b). Unlike natural language, code inherently contains richer structural information. Consequently, in code-related tasks, numerous language models incorporate syntactic and semantic code structures to enhance task performance (Feng et al., 2020; Guo et al., 2021, 2022). However, the high-order data correlation inherent in code remains underexplored in current language models. High-order data correlation refers to the relationship among multiple entities as a unified unit, usually involving more than two entities, distinguishing it from a pairwise relationship (Feng et al., 2019; Berge, 1973). Current language models mainly employ Transformer-based encoder or decoder architectures, where the self-attention mechanism considers pairwise relationships between all tokens. However, this framework lacks the capability to capture cohesive group-level features where multiple tokens can be treated as an integrated feature unit. Our research reveals that in the source code, there exist high-order data correlations that enable the extraction of such integrated feature units.

The first category of these correlations originates from the cohesive relationships among syntax tree nodes that share a common parent node in the abstract syntax tree (AST). The code can be parsed into an AST which represents the structure of the code. The text of the code corresponds to the leaf nodes of the AST. Tokens that belong to the same AST parent node can have their features computed as a unified whole to capture structural semantics, indicating that they together form a specific code structure. We call it AST family correlation. An example of AST family correlation is shown in Figure 1.

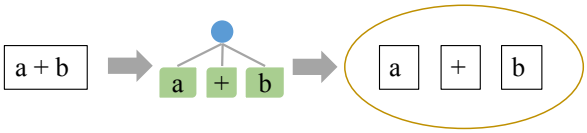



Figure 1: AST family correlation. The tokens 'a', '+', and 'b' belong to the same AST parent node, forming an addition operation, and can thus be treated as a whole unit.

The second category are correlations within lexical units. In code, there are numerous long lexical

* Guang Yang is the corresponding author. E-mail: 51215901104@stu.ecnu.edu.cn or yangguang8@gtht.com
Code is available at https://github.com/qiankunmu/HGAdapter



*Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 14823–14833
November 4-9, 2025 

constructs, such as function names, variable names, and class names, which frequently comprise multiple semantic components concatenated via camel case or underscore delimiters. When undergoing tokenization, these unified lexical units may be fragmented into discrete tokens. The resulting split tokens can have their features computed as a whole to preserve the characteristics of their parent lexical unit. We call this lexical correlation. An example of lexical correlation is shown in Figure 2.

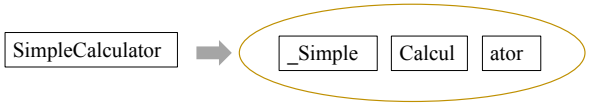



Figure 2: Lexical correlation. We use Llama BBPE-based tokenizer for tokenization. SimpleCalculator is a class name that was split into three tokens. These tokens can be viewed as a whole to extract features from their original class name.

The third category involves tokens within the same line of code. Programmers typically write code with lines as the basic unit. Tokens on the same line of original code can be processed as a whole, providing additional granularity to represent code organization patterns. We call it line correlation. An example of line correlation is shown in Figure 3. To extract the three types of high-order

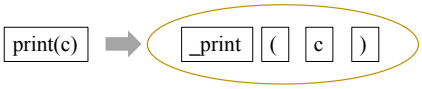



Figure 3: Line correlation. These tokens can be treated as a line to extract features.

data correlation from the code, we design a tokens and hyperedges generator.

Existing language models only consider tokens as pairwise relationships, not as collective units for the above high-order correlation feature encoding. To address this gap, we propose a novel hypergraph-based adapter (HGAdapter). Hypergraphs are the standard framework for representing high-order correlations. Unlike the edges in a standard graph, which only connect two entities, the edges in a hypergraph can connect any number of entities called the hyperedge (Berge, 1973). Hypergraph neural networks (HGNNs) have emerged as the most widely used neural networks to capture high-order data correlations (Feng et al., 2019; Yadati et al., 2019; Kim et al., 2020; Huang and Yang, 2021). Adapters are lightweight modules inserted into pre-trained models for fine-tuning (Rebuffi et al., 2017; Houlsby et al., 2019). We improved the two-stage architecture of HGNNs and combined it with adapters to propose HGAdapter, we implement it as an adapter module inserted into language models, enabling the extraction and computation of the high-order features to enhance language models performance.

We conduct experiments on code summarization, a generation task, and code clone detection, an understanding task. We used public datasets of the two tasks to evaluate our method. Experimental results validate that our method improves the performance of language models, with the introduction of high-order data correlations contributing to an improvement in effectiveness.

The main contributions of this paper are:

- We propose to introduce high-order data correlations within code into PLMs. We propose AST family correlation, lexical correlation, and line correlation. We propose a tokens and hyperedges generator to capture these high-order data correlations.
- We propose a novel HGAdapter to encode high-order data correlations in PLMs that is improved from the architecture of hypergraph neural networks and combined with adapter tuning.
- We conducted experiments on public datasets of code summarization and code clone detection to validate our method.

## 2 Related Work

### 2.1 Code Summarization and Clone Detection

Code summarization is a task that takes the code as input and outputs a description of that code, which can be considered as a generation task. Code clone detection is a task that takes two code samples as input, determines whether the two code fragments are functionally equivalent, and can be considered as a binary classification task. In the early stages, most approaches designed different neural networks to extract code structural features for such tasks (Alon et al., 2019; Chen et al., 2019; Wang et al., 2020). Following the success of PLMs such as BERT (Devlin et al., 2019) and GPT (Radford and Narasimhan, 2018) in natural language processing, many researchers have developed PLMs for code-related tasks such as CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021). Compared to natural language, code has more structural

features, so some PLMs will consider structural information of code, such as GraphCodeBERT (Guo et al., 2021) incorporates data flows and UniXcoder (Guo et al., 2022) learn the text of AST. Currently, due to the success of LLMs, some LLMs for code have emerged, including StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), DeepSeek series (DeepSeek-AI et al., 2024), Qwen Coder series (Hui et al., 2024; Yang et al., 2025), and more.

These PLMs are typically built on the Transformer (Vaswani et al., 2017) encoder-only, decoder-only, or encoder-decoder architecture (Niu et al., 2022; Zhang et al., 2024b). For code summarization, encoder-only PLMs require training a decoder to generate results in a seq2seq manner, whereas decoder-only models can directly produce output. In code clone detection with encoder-only models, the hidden state vector corresponding to the starting position of the PLM output is used as the code representation. Two code vectors are concatenated and fed into a fully-connected neural network classifier for binary classification. Our method follows this process as established in previous work.

### 2.2 Hypergraph Neural Networks

HGNNs are proposed to encode high-order data correlations. Similarly to graph neural networks (GNNs), an HGNN consists of multiple layers, each layer updating the hidden vectors of the nodes. HGNNs employ a two-stage message passing process at each layer: aggregating messages from nodes to their connected hyperedges and aggregating messages from hyperedges to their connected nodes to update nodes vectors. Representative works are HGNN (Feng et al., 2019), HyperGCN (Yadati et al., 2019), HGAN (Kim et al., 2020), UniGNN (Huang and Yang, 2021), etc. In code-related tasks, HEAT (Georgiev et al., 2022) and HDHGN (Yang et al., 2023) have mined different high-order data correlations in code and improved HGNNs.

### 2.3 Adapter Tuning

Adapter tuning is a kind of parameter-efficient fine-tuning (PEFT) method that inserts small-scale parameters between PLM layers. During fine-tuning, the PLM parameters are frozen and not updated, while only the inserted parameters are trained. This approach can achieve or even surpass the performance of full fine-tuning of the PLM. Such methods require the storage of only a small number of parameters and reduce the resources needed for training. Representative works are adapter (Rebuffi et al., 2017), adapter for NLP (Houlsby et al., 2019), AdapterFusion (Pfeiffer et al., 2021), MADX (Pfeiffer et al., 2020), etc. Structural adapter (Ribeiro et al., 2021; Montella et al., 2023) combines the GNNs and adapter tuning.

## 3 Methods

Our methodology has two core components: the tokens and hyperedges generator module and the HGAdapter module. Before code text is input into the language model, it needs to be tokenized. To capture high-order data correlation in the code, we design a tokens and hyperedges generator instead of directly tokenizing the code. To encode high-order data correlation, we design the HGAdapter, which is integrated as an adapter module between the PLM layers.

### 3.1 Tokens and Hyperedges Generator

A tokens and hyperedges generator is designed to extract high-order data correlations within code text. Since we use hyperedge to represent high-order data correlation, in HGAdapter we refer to the high-order data correlation as hyperedge, like the AST family hyperedge, lexical hyperedge, and line hyperedge.

This module includes a parser and a tokenizer. We use tree-sitter[1] as the parser. We adopt the corresponding original tokenizers of different PLMs. When code is input, the parser will parse the code into an AST. We perform a postorder traversal to visit the AST. When visiting a leaf node, the code text of the node is first extracted and fed into the tokenizer to generate tokens. We assign each token a unique token id. If the number of tokens exceeds two, a new hyperedge id is created to map these tokens ids to the hyperedge, while recording the hyperedge type as lexical hyperedge. We use the COO format[2] to record the correspondence between token ids and hyperedge ids, where each entry consists of (token_id, hyperedge_id) pairs representing associations. Similarly, when visiting the leaf node, we can also obtain the corresponding line number for the code text. If we encounter a new line number, we create a new hyperedge id.

[1] https://tree-sitter.github.io/tree-sitter/

[2] https://docs.pytorch.org/docs/stable/sparse.html#sparse-coo-docs

We use a dictionary to record line numbers and their corresponding hyperedge ids. We also record the correspondence between these split token ids and the hyperedge id as the line hyperedge. Upon completing visiting the leaf node, we return token sequences, token ids, hyperedge ids, and hyperedge types to its parent node.

When visiting a parent node, obtain all tokens returned by its leaf nodes. If the number of tokens exceeds two, create a new hyperedge id, record these token ids as belonging to this hyperedge, and label it as an AST family hyperedge. Return all tokens, token ids, hyperedge ids, and hyperedge types, including those collected from its child nodes to the higher-level parent node.

After completing the AST traversal, we obtain the token sequence of the code, token ids, and hyperedge ids that record hyperedge information, along with all hyperedge types.

### 3.2 Hypergraph-based Adapters

The HGAdapter is inserted between the PLM layers. The structure is illustrated in Figure 4. We improved the framework of HGNNs, incorporating a simplified attention mechanism, introducing heterogeneous linear transformations, and integrating it with adapters. During fine-tuning, all PLM parameters are frozen, and only the adapter parameters are updated.

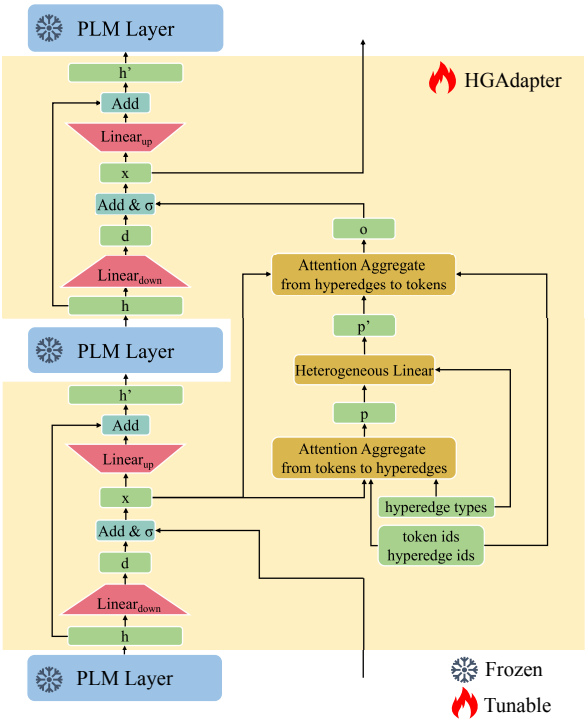



Figure 4: Overall structure of HGAdapter

After input of the token sequence into PLM, the PLM layer outputs the hidden state vectors corresponding to each token, denoted as $h_0^l,\ h_1^l, \ldots, h_{N-1}^l$, where $N$ is the total number of tokens, $l$ denotes the layer number in PLM, $l \in \{1, 2, \ldots, L\}$ with $L$ representing the total number of layers, $h_n^l \in \mathbb{R}^C$, $C$ is the size of the dimension of the hidden state vector , $n = 0,\ 1, \ldots, N-1$ represents the token id in the sequence.

These vectors, along with token ids, hyperedge ids, and hyperedge types, are input into the HGAdapter. Token ids and hyperedge ids record which tokens belong to which hyperedges. Here, a hyperedge is defined as $e$, and the set of tokens it has is denoted as $T(e)$. Given token id $n$, the set of hyperedges to which it belongs is denoted as $S(n)$. The type of hyperedge $e$ is denoted as $\rho(e)$.

First, project the hidden states vectors $h_n^l$ into a lower dimension $C_{down}$ to get $d_n^l$, as shown in Equation 1, where $d_n^l \in \mathbb{R}^{C_{down}}$.

$$d_n^l = W_{down}^l h_n^l + b_{down}^l \tag{1}$$

Afterward, $d_n^l$ and the output $o_n^{l-1}$ of the HGAdapter of the previous layer are summed and passed through an activation function, which yields x, where $\sigma$ is the activation function. We use the ReLU activation function (Glorot et al., 2011).

$$x_n^l = \sigma\left(d_n^l + o_n^{l-1}\right) \tag{2}$$

For the first layer, $d_n^1$ is fed directly into the activation function.

$$x_n^1 = \sigma\left(d_n^1\right) \tag{3}$$

Then, project $x_n^l$ from $C_{down}$ into the normal dimension $C$ and add $h_n^l$, as shown in Equation 4. The vectors $h'^l_0, h'^l_1, \ldots, h'^l_{N-1}$ are then propagated to the next layer of the PLM.

$$h'^l_n = W_{up}^l x_n^l + b_{up}^l + h_n^l \tag{4}$$

For $o_n^l$, its computation process is as follows. First, we aggregate messages from tokens to hyperedges. HGAdapter aggregates the $x^l$ of tokens that belong to the same hyperedge to obtain a vector $p^l$ representing the hyperedge. We employ a relatively simplified attention mechanism to aggregate $x^l$. When a token $n$ belongs to a hyperedge $e$, their attention score $\alpha_{ne}^l$ is calculated as in Equation 5, with the computation $softmax$ occurring across the tokens belonging to the same hyperedge $e$. The

$q_{\rho(e)}$ is a query vector representing the type of hyperedge $e$, which will be updated during training.

$$\alpha_{ne}^{l} = Softmax\left(\frac{\left(q_{\rho(e)}\right)^{T} x_{n}^{l}}{\sqrt{C_{down}}}\right) \tag{5}$$

Aggregate $x^l$ to get $p_e^l$ as shown in Equation 6.

$$p_{e}^{l} = \sum_{n \in T(e)} \alpha_{ne}^{l} x_{n}^{l} \tag{6}$$

Later, $p_e^l$ is subject to a heterogeneous linear transformation that varies depending on the hyperedge type, to integrate the information of the hyperedge type into the vector, as shown in Equation 7.

$$p'^{l}_{e} = W_{\rho(e)}^{l} p_{e}^{l} + b_{\rho(e)}^{l} \tag{7}$$

Lastly, aggregate the hyperedge vectors $p'^l$ to the tokens. The aggregation uses the same attention mechanism. When a hyperedge $e$ has a token $n$, their attention score $\alpha_{en}^l$ is calculated as in Equation 8, with the computation $softmax$ occurring across the hyperedges having the same token $n$.

$$\alpha_{en}^{l} = Softmax\left(\frac{\left(x_{n}^{l}\right)^{T} p'^{l}_{e}}{\sqrt{C_{down}}}\right) \tag{8}$$

Aggregate $p'^l$ to get the new token vector $o_n^l$ as shown in Equation 9.

$$o_{n}^{l} = \sum_{e \in S(n)} \alpha_{en}^{l} p'^{l}_{e} \tag{9}$$

The $o_n^l$ is then added to $d_n^{l+1}$ of the next layer for subsequent operations, as in Equation 2.

# 4 Experiment Settings

## 4.1 Datasets

For code summarization, we use a widely applied public dataset CodeSearchNet(Husain et al., 2019) which contains a million functions collected from open-source code. It has datasets in 6 languages: Ruby, JavaScript, Java, Go, PHP, and Python. The version of the dataset we used is filtered and organized provided by CodeXGLUE(Lu et al., 2021). The statistics is shown in Table 1, where Avg.tokens means the average number of tokens per code snippet, and Avg.hyperedges means the average number of hyperedges per code snippet. Each sample has a code snippet and a segment of natural language description.

For code clone detection, we use the public dataset BigCloneBench(Svajlenko et al., 2014) which is the most widely used Java code clone detection dataset. The version of the dataset we used is also provided by CodeXGLUE(Lu et al., 2021). The statistical information of the dataset is presented in Table 2. Each sample has two code snippets for clone detection.

## 4.2 Baselines

In code summarization, we select several language models that have demonstrated outstanding performance in this generation task as baselines, including RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), Code Llama 7B (Rozière et al., 2023), TinyLlama-Math&Code (Zhang et al., 2024a), Qwen2.5-Coder-0.5B (Hui et al., 2024) as comparisons.

In code clone detection, we select several language models that have demonstrated outstanding performance in this understanding task as baselines, including CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and UniXcoder (Guo et al., 2022).

We used full fine-tuning or adapter tuning (Houlsby et al., 2019) for comparative experiments. We also used the structural adapter (Ribeiro et al., 2021; Montella et al., 2023) for comparison. For the structural adapter, we treat AST family, lexical, and line correlations as pairwise token relationships rather than hyperedges, and we employ the GNN the same as the structural adapter.

## 4.3 Training Settings

We implement the HGAdapter based on the PyTorch[3], transformers[4], and adapters[5] libraries. We conducted the experiments on a machine that has 64GB of RAM and an RTX 3090 GPU with 24GB. We select the tree-sitter[6] to parse the code snippets into ASTs. We use the tokenizers used by the corresponding PLMs. The PLMs and tokenizers that we used are all provided by the official on Hugging Face[7]. The hyperparameter configuration of the PLMs remains unchanged. The dimension size of the vectors in the adapter is 64. We employ the cross-entropy loss function in training.

[3]https://pytorch.org/docs/stable/index.html
[4]https://huggingface.co/docs/transformers/index
[5]https://docs.adapterhub.ml/index.html
[6]https://tree-sitter.github.io/tree-sitter/
[7]https://huggingface.co/

| | Ruby | JavaScript | Java | Go | PHP | Python |
|---|---|---|---|---|---|---|
| Training | 24927 | 58025 | 164923 | 167288 | 241241 | 251820 |
| Validation | 1400 | 3885 | 5183 | 7325 | 12982 | 13914 |
| Testing | 1261 | 3291 | 10955 | 8122 | 14014 | 14918 |
| Avg. tokens | 126.27 | 173.35 | 155.82 | 133.23 | 271.02 | 280.58 |
| Avg. hyperedges | 53.27 | 74.45 | 67.29 | 55.86 | 71.24 | 73.65 |

Table 1: Statistics of CodeSearchNet.

| | BigCloneBench |
|---|---|
| Training | 901028 |
| Validation | 415416 |
| Testing | 415416 |
| Avg. tokens | 401.84 |
| Avg. hyperedges | 182.11 |
| Language | Java |

Table 2: Statistics of BigCloneBench.

In code summarization, we select BLEU-4 (Lin and Och, 2004) as the evaluation metric. We use the BLEU-4 evaluation implemented by Hugging Face[8]. We use Adam optimizer(Kingma and Ba, 2015) with a learning rate of $1 \times 10^{-4}$. The batch size is 64. The number of training epochs is 20. The adapter parameters with the highest BLEU-4 score in the validation set are saved.

For code clone detection, we select F1, precision, and recall as the evaluation metric. We use AdamW optimizer(Loshchilov and Hutter, 2019) with a learning rate of $5 \times 10^{-5}$. The batch size is 4 (including 8 code snippets) and the number of training epochs is 10. The adapter parameters with the highest F1 score in the validation set are saved.

## 5 Results

### 5.1 Code Summarization

The BLEU-4 scores of the models in the testing sets are shown in Table 3. As we can see, the HGAdapter has improved the performance of PLMs to varying degrees in most programming languages. Compared to full fine-tuned PLMs, HGAdapter demonstrates significant overall improvements, specifically showing performance gains of 2.33 over RoBERTa, 1.99 over CodeBERT, 2.01 over GraphCodeBERT, 1.91 over UniXcoder, 1.87 over TinyLlama-Math&Code and 1.76 over Qwen2.5-Coder-0.5B. HGAdapter also achieves notable improvements over general adapter-tuned PLMs, demonstrating performance gains of 2.31 with RoBERTa, 2.08 with CodeBERT, 2.05 with GraphCodeBERT, 2.01 with UniXcoder, 1.45 with Code Llama 7B, 1.92 with TinyLlama-Math&Code, and 1.84 with Qwen2.5-Coder-0.5B. These results validate that the introduction of high-order data correlations can improve the effectiveness of code summarization, and our HGAdapter can encode high-order data correlations within language models and enhance their performance.

HGAdapter shows advantages over the structural adapter that also incorporate structural information, surpassing its performance by 1.68 on RoBERTa, 1.52 on CodeBERT, 1.49 on GraphCodeBERT, 1.58 on UniXcoder, 1.01 on Code Llama 7B, 1.60 on TinyLlama-Math&Code, and 1.63 on Qwen2.5-Coder-0.5B. This validates that for AST family, lexical and line structural information, extracting features by treating tokens as high-order data correlations is better than only as pairwise relationships.

Comparative analysis reveals that HGAdapter shows more substantial improvements over adapter in RoBERTa, CodeBERT, and GraphCodeBERT, while showing relatively smaller performance gains in Code Llama 7B, TinyLlama-Math&Code, and Qwen2.5-Coder-0.5B. This is likely because the former are language models based on the Transformer encoder architecture, while the latter are decoder-based language models. Encoder-based language models excel at extracting richer contextual information, and HGAdapter further extracts features from their hidden vectors based on high-order data correlations, thus achieving greater improvement. We can observe that HGAdapter provides limited improvement for Code Llama 7B, probably because its large parameter size already grants it strong code feature extraction capabilities, leaving less room for HGAdapter to enhance its performance.

We also observed that HGAdapter achieves relatively greater improvements on the Ruby and JavaScript datasets, likely because these datasets are relatively smaller in scale. As a result, full fine-tuning or general adapter tuning may yield insufficient training effectiveness. HGAdapter compensates for this limitation by providing richer feature information.

[8]https://huggingface.co/spaces/evaluate-metric/bleu

| Models | Ruby | JavaScript | Java | Go | PHP | Python | Overall |
|---|---|---|---|---|---|---|---|
| RoBERTa (Full Fine-tuning) | 11.73 | 11.88 | 16.52 | 16.49 | 21.68 | 17.14 | 15.91 |
| RoBERTa (Adapter) | 11.66 | 11.95 | 16.58 | 16.41 | 21.89 | 17.10 | 15.93 |
| RoBERTa (Structural Adapter) | 12.45 | 12.71 | 17.01 | 16.99 | 22.32 | 17.85 | 16.56 |
| RoBERTa (HGAdapter) | **14.27** | **14.60** | **19.06** | **18.55** | **24.03** | **18.94** | **18.24** |
| CodeBERT (Full Fine-tuning) | 12.13 | 13.85 | 17.68 | 16.58 | 22.87 | 18.09 | 16.87 |
| CodeBERT (Adapter) | 12.02 | 13.74 | 17.51 | 16.59 | 22.90 | 17.92 | 16.78 |
| CodeBERT (Structural Adapter) | 12.93 | 14.26 | 18.14 | 17.06 | 23.27 | 18.37 | 17.34 |
| CodeBERT (HGAdapter) | **14.31** | **16.14** | **19.70** | **18.86** | **24.62** | **19.52** | **18.86** |
| GraphCodeBERT (Full Fine-tuning) | 12.42 | 14.80 | 18.98 | 17.86 | 24.02 | 18.05 | 17.69 |
| GraphCodeBERT (Adapter) | 12.51 | 14.75 | 19.00 | 17.63 | 23.94 | 18.06 | 17.65 |
| GraphCodeBERT (Structural Adapter) | 13.06 | 15.33 | 19.47 | 18.34 | 24.63 | 18.45 | 18.21 |
| GraphCodeBERT (HGAdapter) | **14.95** | **16.94** | **21.06** | **19.82** | **25.68** | **19.77** | **19.70** |
| UniXcoder (Full Fine-tuning) | 14.93 | 15.73 | 20.01 | 19.07 | 25.96 | 19.18 | 19.15 |
| UniXcoder (Adapter) | 15.04 | 15.76 | 19.78 | 18.73 | 25.87 | 19.09 | 19.05 |
| UniXcoder (Structural Adapter) | 15.51 | 16.22 | 20.24 | 19.17 | 26.28 | 19.46 | 19.48 |
| UniXcoder (HGAdapter) | **16.92** | **18.07** | **22.04** | **20.95** | **27.81** | **20.57** | **21.06** |
| Code Llama 7B (Adapter) | 15.46 | 17.03 | 21.26 | 19.56 | 27.06 | 20.09 | 20.08 |
| Code Llama 7B (Structural Adapter) | 16.03 | 17.45 | 21.70 | 19.97 | 27.45 | 20.51 | 20.52 |
| Code Llama 7B (HGAdapter) | **17.14** | **18.62** | **22.86** | **21.22** | **28.00** | **21.34** | **21.53** |
| TinyLlama-Math&Code(Full Fine-tuning) | 14.96 | 16.19 | 19.08 | 18.21 | 23.87 | 18.15 | 18.41 |
| TinyLlama-Math&Code(Adapter) | 14.78 | 16.03 | 19.35 | 18.11 | 23.81 | 18.09 | 18.36 |
| TinyLlama-Math&Code(Structural Adapter) | 15.12 | 16.45 | 19.63 | 18.46 | 24.07 | 18.36 | 18.68 |
| TinyLlama-Math&Code(HGAdapter) | **16.89** | **18.04** | **21.11** | **20.19** | **25.54** | **19.92** | **20.28** |
| Qwen2.5-Coder-0.5B (Full Fine-tuning) | 15.10 | 16.32 | 19.61 | 18.76 | 25.97 | 19.43 | 19.20 |
| Qwen2.5-Coder-0.5B (Adapter) | 14.97 | 16.12 | 19.54 | 18.83 | 26.01 | 19.22 | 19.12 |
| Qwen2.5-Coder-0.5B (Structural Adapter) | 15.29 | 16.41 | 19.75 | 19.03 | 26.18 | 19.34 | 19.33 |
| Qwen2.5-Coder-0.5B (HGAdapter) | **17.03** | **18.17** | **21.31** | **20.56** | **27.73** | **20.97** | **20.96** |

Table 3: BLEU-4 results of code summarization on CodeSearchNet (%)

### 5.2 Code Clone Detection

The results of the models on the testing set are shown in Table 4. The results show that HGAdapter improves precision, recall, and F1 scores for most language models. Compared to full fine-tuned PLMs, HGAdapter improves F1 scores by 1.28 for CodeBERT, 1.23 for GraphCodeBERT, and 1.12 for UniXcoder. Compared to adapter tuned PLMs, HGAdapter achieves F1 score improvements of 1.31 for CodeBERT, 1.21 for GraphCodeBERT and 1.17 for UniXcoder. These results validate that introducing high-order data correlations through the HGAdapter can indeed enhance the understanding of code by PLMs, thereby improving the performance on the code clone detection.

HGAdapter outperforms the structural adapter with an F1 score of 1.01 in CodeBERT, 0.80 in GraphCodeBERT, and 0.89 in UniXcoder. This also indicates that for code clone detection task, incorporating AST family, lexical, and line structural information as high-order correlations is better than treating them only as pairwise relationships.

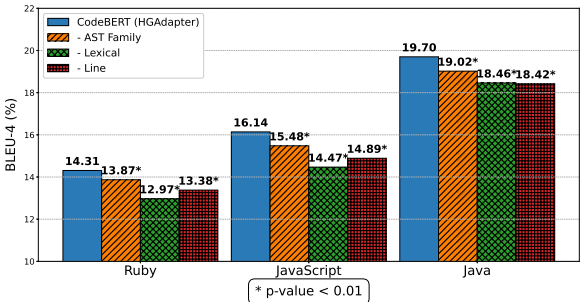
For CodeBERT, HGAdapter achieves improvements of 1.87 in precision and 0.69 in recall compared to full fine-tuning. In GraphCodeBERT, HGAdapter shows performance improvements of 1.76 in precision and 0.71 in recall. For both PLMs, the HGAdapter makes greater improvements in precision. For UniXcoder, HGAdapter achieves a significant 2.35 recall improvement, but does not make precision higher. This observation may come from the UniXcoder inherently high-precision baseline, where HGAdapter recall enhancement comes at the cost of slight precision degradation.

| Models | Precision | Recall | F1 |
|---|---|---|---|
| CodeBERT (Full Fine-tuning) | 94.76 | 94.72 | 94.74 |
| CodeBERT (Adapter) | 94.78 | 94.64 | 94.71 |
| CodeBERT (Structural Adapter) | 95.05 | 94.98 | 95.01 |
| CodeBERT (HGAdapter) | **96.63** | **95.41** | **96.02** |
| GraphCodeBERT (Full Fine-tuning) | 95.39 | 94.68 | 95.03 |
| GraphCodeBERT (Adapter) | 95.44 | 94.65 | 95.05 |
| GraphCodeBERT (Structural Adapter) | 96.01 | 94.92 | 95.46 |
| GraphCodeBERT (HGAdapter) | **97.15** | **95.39** | **96.26** |
| UniXcoder (Full Fine-tuning) | **97.26** | 92.84 | 95.01 |
| UniXcoder (Adapter) | 97.15 | 92.87 | 94.96 |
| UniXcoder (Structural Adapter) | 97.11 | 93.45 | 95.24 |
| UniXcoder (HGAdapter) | 97.08 | **95.19** | **96.13** |

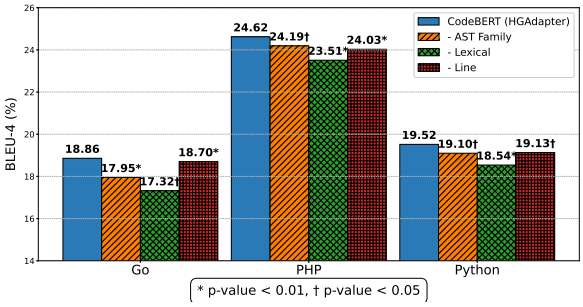
Table 4: Results of code clone detection on BigCloneBench (%)

### 5.3 Ablation Study

In code summarization, we perform ablation experiments in CodeBERT to validate that the introduction of the three types of high-order data correlations can improve performance. They correspond to the AST family hyperedge, lexical hyperedge, and line hyperedge. We separately ablate each type of hyperedge in HGAdapter. The results are shown

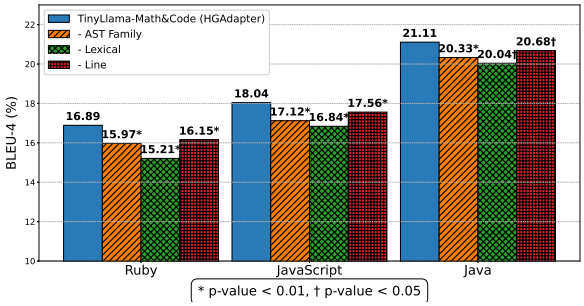



(a) Results of CodeBERT on Ruby, JavaScript and Java

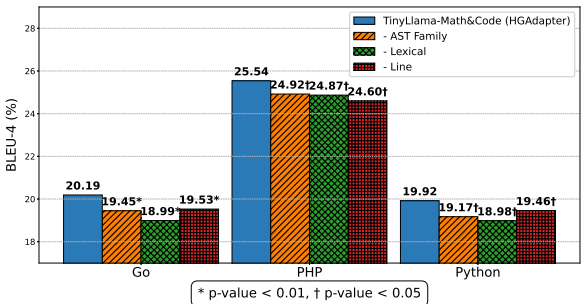



(b) Results of CodeBERT on Go, PHP and Python

Figure 5: Results of ablation study on code summarization in CodeBERT (%)




(a) Results of TinyLlama-Math&Code on Ruby, JavaScript and Java




(b) Results of TinyLlama-Math&Code on Go, PHP and Python

Figure 6: Results of ablation study on code summarization in TinyLlama-Math&Code (%)

in Figure 5 and Figure 6. In CodeBERT, removing AST family hyperedges, lexical hyperedges, and line hyperedges results in an average decrease of 0.59, 1.31, and 0.78 respectively. On TinyLlama-Math&Code, removing AST family hyperedges, lexical hyperedges, and line hyperedges resulted in overall performance drops of 0.79, 1.12, and 0.62 respectively. Through multiple experiments and comparisons with HGAdapter, the resulting p-value was less than 0.05. The experimental results validate that these three types of high-order data correlations can improve the performance of language models in code summarization. We also found that lexical hyperedges have a more significant impact compared to the other two types.

In code clone detection, we also conduct ablation experiments on CodeBERT in BigCloneBench to validate that the introduction of the three types of high-order data correlations can improve performance. We separately ablate each type of hyperedge in HGAdapter. The results are shown in Figure 7. It can be seen that removal of AST family hyperedges, lexical hyperedges, and line hyperedges decreased performance to varying degrees. Removal of AST family hyperedges resulted in a 0.89 decrease in precision, a 0.74 decrease in

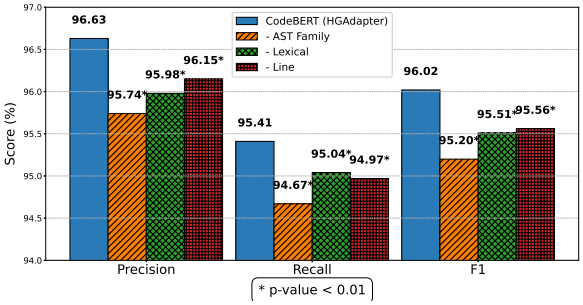



Figure 7: Results of ablation study on code clone detection (%)

recall, and a 0.82 decrease in F1. Eliminating lexical hyperedges led to a 0.65 decrease in precision, a 0.37 decrease in recall, and a 0.51 decrease in F1. Removing line hyperedges reduced precision by 0.48, recall by 0.44, and F1 by 0.46. Multiple experiments demonstrated statistically significant results that the p-value was less than 0.01 compared to the baseline without the removal of the hyperedge. These results validate that by introducing three types of high-order data correlations, the ability of PLMs to understand code can be enhanced from different perspectives. The results also indicate that AST family hyperedges have a more substantial impact on performance compared to other types. This is likely because the AST fam-

| PLM name | Params | Adapter Params | HGAdapter Params |
|---|---|---|---|
| RoBERTa, CodeBERT, GraphCodeBERT | 125M | 1.2M | 1.3M |
| UniXcoder | 126M | 1.2M | 1.3M |
| Code Llama 7B | 6.7B | 16.9M | 17.3M |
| TinyLlama-Math&Code | 1.1B | 5.8M | 6.1M |
| Qwen2.5-Coder-0.5B | 0.5B | 2.8M | 3.1M |

Table 5: Number of parameters for different PLMs and adapters

ily high-order correlations enable language models to better comprehend the code structure, leading to superior performance in program understanding tasks.

### 5.4 Number of Parameters

The number of parameters for different PLMs and their corresponding inserted adapters, as well as the HGAdapter, are shown in Table 5, where 1M represents 1 million and 1B represents 1 billion. The dimension size of the hidden vectors in the adapter is 64. Among them, RoBERTa, CodeBERT and GraphCodeBERT all have the same number of parameters.

Compared to RoBERTa, CodeBERT, GraphCodeBERT and UniXcoder, the parameter count of HGAdapter is only 1% of theirs. Compared to Code Llama 7B, the parameter count of HGAdapter is even as low as 0.3% of its. Compared to TinyLlama-Math&Code, the parameter size of HGAdapter is only 0.5% of its. Similarly, compared to Qwen2.5-Coder-0.5B, the parameter count of the HGAdapter is merely 0.6% of its. We can observe that, compared to PLM, the HGAdapter parameters account for only about 0.3%-1%.

In RoBERTa, CodeBERT, GraphCodeBERT, and UniXcoder, HGAdapter increases the parameter count by approximately 8% compared to the adapter. In Code Llama 7B, the HGAdapter increases the number of parameters by only 2% compared to the adapter. In TinyLlama-Math&Code, the increase is 5%. In Qwen2.5-Coder-0.5B, HGAdapter increases the parameter count by 11%. We find that, compared to the standard adapter, the HGAdapter introduces only around 3%-11% additional parameters. HGAdapter achieves performance improvements and does not significantly increase the number of parameters. This, to some extent, validates the efficiency of HGAdapter.

## 6 Conclusion

In this paper, we propose to introduce high-order data correlations within code tokens into language models. We propose AST family correlation, lexical correlation, and line correlation. We design a tokens and hyperedges generator to capture the three types of high-order data correlation in the code. We improve the architecture of HGNNs and combine it with adapters to propose HGAdapter, it can encode high-order data correlations, and it is allowed to be inserted into various PLMs. We perform experiments on public datasets of code summarization and code clone detection tasks. Experimental results show that our method increases the performance of PLMs, with the introduction of high-order data correlations contributing to an improvement in results. Further ablation studies and parameter comparisons further validate the effectiveness of our method. In the future, we will explore the introduction of more high-order data correlations, the integration of more effective parameter fine-tuning methods with hypergraphs, and the application of our approach to a wider range of code-related tasks.

## Limitations

The impact of HGAdapter on larger-scale PLMs remains to be explored in future work. HGAdapter is a lightweight module that introduces minimal overhead in terms of both parameters and GPU memory. However, due to the additional computational steps involved in hypergraph construction and processing, HGAdapter increased the training time and inference latency. HGAdapter relies on tasks that require complete code input and is therefore not directly applicable to other tasks without code input, such as code generation. More correlations can be mined in code or in natural language. Additionally, other PEFT methods could be explored by investigating how to integrate them either with HGAdapter or with high-order data correlations.

## Acknowledgements

This research was carried out independently. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. Thanks to anonymous reviewers for their insightful suggestions and comments.

## References


Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *The 7th International Conference on Learning Representations, ICLR 2019*.

Claude Berge. 1973. *Graphs and hypergraphs*.

Qiuyuan Chen, Han Hu, and Zhaoyi Liu. 2019. Code summarization with abstract syntax tree. In *Neural Information Processing - 26th International Conference, ICONIP 2019*, volume 1143 of *Communications in Computer and Information Science*, pages 652–660.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 80 others. 2024. Deepseek-v3 technical report. *CoRR*, abs/2412.19437.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*, pages 4171–4186.

Yifan Feng, Haoxuan You, Zizhao Zhang, R. Ji, and Yue Gao. 2019. Hypergraph neural networks. In *The 33rd AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 3558–3565.

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

Dobrik Georgiev, Marc Brockschmidt, and Miltiadis Allamanis. 2022. HEAT: hyperedge attention networks. *Trans. Mach. Learn. Res.*, 2022.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, AISTATS 2011*, volume 15, pages 315–323.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, ACL 2022*, pages 7212–7225.

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 *The 9th International Conference on Learning Representations, ICLR 2021*.

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

Jing Huang and Jie Yang. 2021. Unignn: a unified framework for graph and hypergraph neural networks. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 2563–2569.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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.

Eun-Sol Kim, Woo-Young Kang, Kyoung-Woon On, Yu-Jung Heo, and Byoung-Tak Zhang. 2020. Hypergraph attention networks for multimodal learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020*, pages 14569–14578.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *The 3rd International Conference on Learning Representations, ICLR 2015*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *The 20th International Conference on Computational Linguistics, COLING 2004*.

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.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *The 7th International Conference on Learning Representations, ICLR 2019*.

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, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021*.

Sébastien Montella, Alexis Nasr, Johannes Heinecke, Frédéric Béchet, and Lina Maria Rojas-Barahona. 2023. Investigating the effect of relative positional embeddings on amr-to-text generation with structural adapters. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023*, pages 727–736.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, pages 5546–5555.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2021. Adapterfusion: Non-destructive task composition for transfer learning. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021*, pages 487–503.

Jonas Pfeiffer, Ivan Vulic, Iryna Gurevych, and Sebastian Ruder. 2020. Mad-x: An adapter-based framework for multi-task cross-lingual transfer. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020*, pages 7654–7673.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, NeurIPS 2017*, pages 506–516.

Leonardo F. R. Ribeiro, Yue Zhang, and Iryna Gurevych. 2021. Structural adapters in pretrained language models for amr-to-text generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pages 4269–4282.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

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 *The 30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 476–480.

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 *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, NeurIPS 2017*, pages 5998–6008.

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

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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pages 8696–8708.

Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Pratim Talukdar. 2019. Hypergcn: A new method for training graph convolutional networks on hypergraphs. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, pages 1509–1520.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

Guang Yang, Tiancheng Jin, and Liang Dou. 2023. Heterogeneous directed hypergraph neural network over abstract syntax tree (AST) for code classification. In *The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023*, pages 274–279.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024a. Tinyllama: An open-source small language model. *CoRR*, abs/2401.02385.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024b. Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Trans. Mach. Learn. Res.*, 2024.