Pyramid Attention For Source Code Summarization

Anonymous Author(s)
Affiliation
Address
email

Abstract

In this paper, we present a multi-granularity method for the task of source code summarization, which generates a concise functional description for the given code snippet. We notice that skilled programmers write and read source codes in a hierarchical way and pay close attention to the conceptual entities like statements, tokens, sub-tokens, and the mapping relations between them. The entities have specific emphasis according to their granularities, e.g., statements in coarse-granularity reveal the global logical semantics of code, and the sub-tokens in fine-granularity are more related to the textual semantics. Driven by this observation, we argue that a multi-granularities formulation incorporating the entities in different granularities may benefit the code summarization task. Given a code snippet, we first construct a pyramid-shaped input representation, and a pyramid attention mechanism is proposed for efficient feature aggregation and distribution across different hierarchies. We instantiate our multi-granularity method using the proposed pyramid attention and name it PA-former (Pyramid Attention Transformer), which is evaluated on two source code summarization benchmarks where it surpasses the prior works and achieves new state-of-the-art results.

1 Introduction

Automatic source code summarization is drawing increasing attention as its promising application prospects in software development and maintenance. This task is challenging as the complex syntax structure of programs and the arbitrariness of variable naming. Summarization methods need to exploit textual and grammatical information in programs to learn a meaningful representation for concise description generation. Following the early stage works [12, 29] that treated the code as a sequence of text and directly adopted the well-developed seq2seq [27] models for this task, recent leading methods [26, 24] both exploit the explicit structure features (e.g. abstract syntax tree (AST) or pair-wise relations extracted from it) in programs to boost the performance of source code summarization.

Current structure-aware code summarization methods are either in a hybrid way [26, 1, 13, 3] or in a structure-guided way [24, 34, 9, 18]. Works in hybrid fashion encode the AST-based structure using representation models like GGNN [15] or TreeLSTM [28], and then combine the structural representations with the sequential representation learned using seq2seq models for decoding. While the structure-guided approaches use the pair-wise relations between tokens (or sub-tokens) extracted from code structure as inductive biases to guide the model learning process. Although these methods have achieved leading results, they only view source code as single-granularity, which is inconsistent with the multi-grained nature of source code.

To understand a given code snippet, a skilled programmer hierarchically breaks it down into fine-grained entities (i.e., sub-tokens) and builds up conceptual semantics gradually from the fine-grained entities to coarse-grained ones. We now use an example shown in Figure 1 to illustrate the multi-granularity process for code understanding. Concretely, we first segment the whole code into several

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

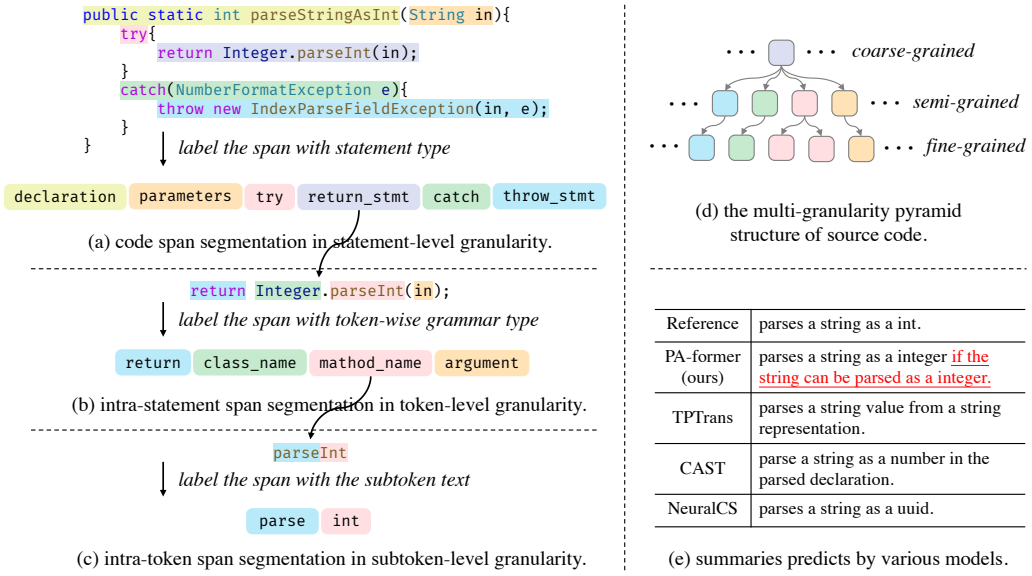

Figure 1: Example of human-like code comprehension. Skilled programmers hierarchically break the source code down to fine-grained entities (i.e., sub-tokens) and gradually build up conceptual semantics understanding for the code from fine-grained entities to coarse-grained ones.

coarse-grained spans according to punctuations and grammars, where a span corresponds to a statement conceptually (Figure 1(a)). Given the coarse-grained sequence, we can roughly summarize the code as *return something if something done*. After that, we dive into each statement and do intra-statement segmentation for it (Figure 1(b)). Now we can detail the *something* as *return something produced by a method invocation*. To further clarify *which method invoked* and *what thing produced*, we go deep into each token and further segment token into sub-tokens. Here, we can figure out that *a int parsing method is invoked* and *a int is produced* (Figure 1(c)). We finally roll back the details from fine-grained entities to coarse-grained ones and obtain the overall understanding of the given code.

Driven by the above analysis, we propose our multi-granularity formulation for source code summarization by mimicking human behavior. We first construct a pyramid input with three granularities including a sub-token sequence, token sequence, and statement sequence at each layer from bottom to top (Figure 1(d)). And a *pyramid attention* mechanism is proposed for efficient feature aggregation and distribution across hierarchies and produces a pyramid representation with rich semantics on all hierarchies. Because both coarse-grained logical patterns and fine-grained textual features are considered simultaneously, our model can predict a more comprehensive summary. Taking the code illustrated in Figure 1(e) as an example, compared with current leading methods, our model notices the *try ... catch ...* structure in the code and uses the additional *if the string can be parsed as an int.* to tell the captured exception handling pattern.

Without bells and whistles, vanilla Transformer [30] equipped with the proposed **P**yramid **A**ttention (PA-former) achieves about 5% performance improvement when evaluated using BLEU [23] on a Java benchmark. And PA-former is evaluated on two source code summarization benchmarks where it surpasses the prior works and reaches new state-of-the-art results. Several ablation studies conducted show the efficiency of our method design.

2 Related Work

Source code representation. Learning-based source code comprehension is drawing increasing attention as its promising application prospects in the field of software development and software maintenance. A meaningful representation can be used for various downstream tasks, such as code clone detection [33, 32], code classification [22, 36, 3], code summarization [10, 29, 3], code search [7], etc. Early learning-based works treated source code as a sequence of text ignoring the structure

features. Mainstream base model like CNN [2], Attention-based LSTM [12] and Transformer [29] were all used directly to capture the textual patterns in programs.

Many follow-ups leveraged both textual and structure information to learn a more meaningful code representation. Tree-based models like TBCNN [22], Code-GRU [16], ASTNN [36] were proposed to represent the source code by considering the tree structure. Some works [1] extended ASTs into graphs by introducing control flows and data flows, then a GGNN was applied to encode code graphs. The set of paths [3, 4] extracted from AST also was used as a representation of the corresponding code. Some works [11, 8, 10] flatten the ASTs into a sequence, and then the off-the-shelf sequential models are applied to learning the textual and syntax features. Currently, most attention is focused on guiding the attention computation in Transformer using the structure bias in programs [24, 9, 37].

Source code summarization. To generate a functional description for the given code snippet, code summarization models should capture both textual and structural information in programs. Some early works [12, 29, 31] just treated the source code as sequence or flatten the AST using a structure-based traversal (SBT) method [10] and then performed a conventional seq2seq learning. The hybrid works represented both textual and structural features separately, then a hybrid mechanism is used to combine the learned representations for summaries. HDeepcom [11] applied the SBT method for structural representation and CAST [26] learning the structure via hierarchical splitting and reconstruction of abstract syntax trees. Many Transformer-based methods [26, 1, 13, 3] use the structure information to guide the attention computation. TPTrans [24] encoded the pairwise path between code tokens and used the path embeddings like relative position encoding. Unlike the above-mentioned methods which view the source code as a single-granularity data, we propose a multi-granularity source code summarization method.

Multi-scale feature learning. The multi-scale feature processing is widely used in the computer vision community. FPN [20] leverages the pyramid-shaped features of ConvNet and creates a feature pyramid that has rich semantics at all scales. Recently, the multi-scale feature hierarchies were also introduced into Vision Transformer (ViT) by a variety of works like MViT [6], Swin [21] and HRFormer [35], which show strong evidence that Transformer could work well with the multi-scale features.

3 Approach

As shown in Figure 2, given a source code, we first construct sequences with three granularities and form them into a pyramid-shaped structure based on the mapping constraints between different granularities. After that, the proposed pyramid attention mechanism (in Figure 3) is applied layer by layer in the encoding stage to produce a feature pyramid which is used by a vanilla Transformer decoder to generate the summary. We introduce the details of the proposed pyramid input constructor, pyramid attention mechanism and implementation details next.

3.1 Pyramid Input Constructor

The constructor builds the pyramid input following the human code comprehension process described in Section 1. Entities in different granularities have their emphasis, telling specific attributes of code from a different viewpoint. And they are complementary to each other. For instance, the coarse-grained statement sequence reveals the global logical semantics of code, the semi-grained token sequence shows the code grammar details, while the fine-grained sub-token split from each token is attached with textual information. We introduce the details of the proposed pyramid input constructor in the following.

Fine-grained sequence. Each leaf node in the AST corresponds to a token (like `boolean`, `value`) in the source code. Since method names and variable names in programs usually are combined by multiple natural language words with *CamelCase* or *snake_case*, we split these combined tokens into sub-tokens. To construct the fine-grained sequence i.e. sub-token sequence, we first obtain the leaf nodes sequence of AST by depth-first pre-order traversal and then we split each token into sub-tokens. Such a sub-token technique is widely used by source code summarization methods [29, 24, 26] and proved to be effective. Formally, let $\mathcal{T} =$

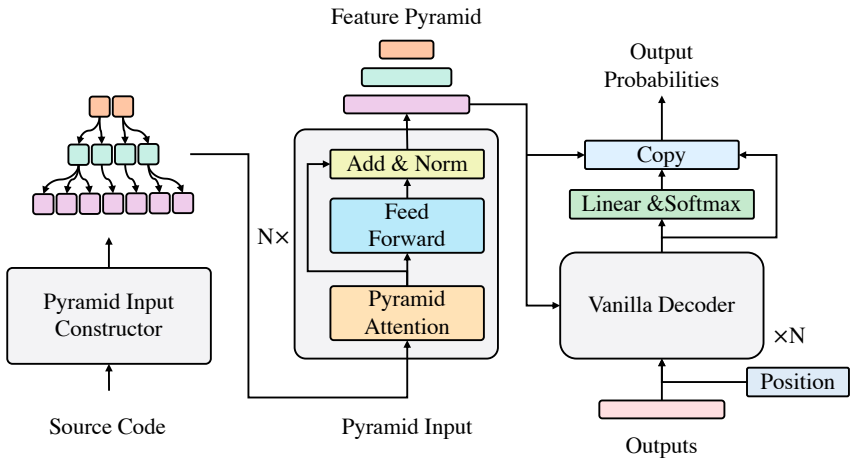

Figure 2: The proposed multi-granularity source code summarization method. We first construct a pyramid input with three granularities. Then our pyramid attention based encoder is applied to the pyramid input and forms a feature pyramid which is used for decoding.

$[t_1, t_2, \ldots, t_N]$ ($\mathcal{T}$ for text) denote the ordered sub-token sequence, that is, the fine-grained input representation situated on the bottom of the pyramid input. For the code in Figure 1, $\mathcal{T}$ is `[public, static, int, parse, String, As, Int, (, ...]`.

Semi-grained sequence. We use the attribute of the parent node of each token (i.e. leaf) node in the parsed AST to represent the token's grammatical property. We construct the semi-grained sequence by replacing the element in order token sequenc with its grammatical property directly. Formally, let $\mathcal{G} = [g_1, g_2, \ldots, g_M]$ ($\mathcal{G}$ for grammar) denotes the ordered token grammatical sequence, that is, the semi-grained input representation situated in middle of the pyramid input. For the code in Figure 1, $\mathcal{G}$ is `[modifier, modifier, type, identifier, (, ...]`. In addition, there is a one-to-many mapping relationship between the token and sub-token. We denote it as $\mathcal{M}_{\mathcal{G}\to\mathcal{T}}$, $\mathcal{M}_{\mathcal{G}\to\mathcal{T}}(g_i, t_j) = 1$ if the sub-token t_j is split from g_i, which will be used to guide the information aggregation process from fine-grained hierarchy to this semi-grained hierarchy.

Coarse-grained sequence. We use the nodes corresponds to statements to form our coarse-grained sequence, which reveals logic side of program. The statement type is limited, we directly identify the AST node as statement-level node if the tag is belongs to the pre-defined statement type set (see Appendix C for more details). Formally, let $\mathcal{L} = [l_1, l_2, \ldots, l_K]$ ($\mathcal{L}$ for logic) denotes the ordered statements sequence, that is, the coarse-grained input representation situated in top of the pyramid input. For the code in Figure 1, $\mathcal{L}$ is `[declaration, parameters, try, return_statement, ...]`. Similar to the semi-grained sequence, we also construct a one-to-many mapping function $\mathcal{M}_{\mathcal{L}\to\mathcal{G}}$ between statement and token, examples are illustrated between $\mathcal{L}$ and $\mathcal{G}$. $\mathcal{M}_{\mathcal{L}\to\mathcal{G}(l_i,g_j)} = 1$ if the statement l_i is the most recent ancestor of token g_j among $\mathcal{L}$ in AST.

In summary, we construct three sequences $\{\mathcal{T}, \mathcal{G}, \mathcal{L}\}$ in three different granularities and the mapping relationships $\{\mathcal{M}_{\mathcal{L}\to\mathcal{G}}, \mathcal{M}_{\mathcal{G}\to\mathcal{T}}\}$ between adjacent granularities are also extracted. Thus, the sequences and the mapping relationships all together form the pyramid input representation of the given code. See Appendix C for more technical details and examples.

3.2 Pyramid Attention

Given the pyramid input $\{\mathcal{T}, \mathcal{G}, \mathcal{L}\}$ and the mapping function $\{\mathcal{M}_{\mathcal{L}\to\mathcal{G}}, \mathcal{M}_{\mathcal{G}\to\mathcal{T}}\}$, the pyramid attention mechanism is proposed to replace the original self-attention module in Transformer for efficient processing of the multi-granularity data. As illustrated in Figure 3, pyramid attention module involves a bottom-to-top aggregation pathway and a cross-granularity information interaction process. We describe the details in the following.

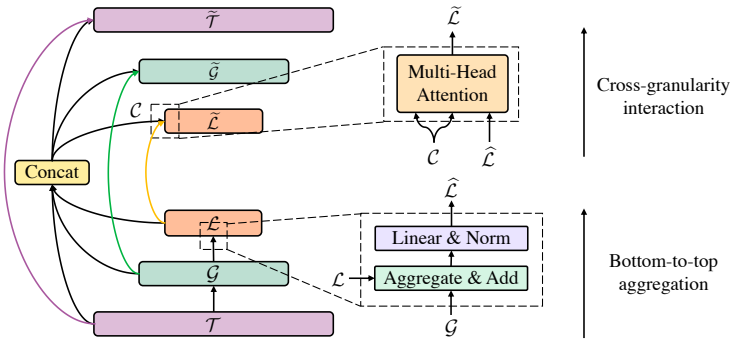

Figure 3: Pyramid attention mechanism. The pyramid attention takes a pyramid-shaped data described in Section 3.1 as input, and outputs a multi-scale representation without changing the original pyramid shape. The whole process consists of a bottom-to-top aggregation pathway and a cross-granularity information interaction process.

Bottom-to-top aggregation. The bottom-to-top pathway does a fine-to-coarse information aggregation, which ensures enriching the higher level representation using the lower level semantics. Given the pyramid input data: $\{\mathcal{T}, \mathcal{G}, \mathcal{L}\}$ and the mapping function $\{\mathcal{M}_{\mathcal{L}\to\mathcal{G}}, \mathcal{M}_{\mathcal{G}\to\mathcal{T}}\}$, we aggregate the representation of $\mathcal{T}$ into $\mathcal{G}$ w.r.t. $\mathcal{M}_{\mathcal{G}\to\mathcal{T}}$. The formulation of the aggregation process is given as:

$$\widehat{\boldsymbol{g}}_i = \text{LayerNorm}\left(\boldsymbol{g}_i + \boldsymbol{W}_{\text{agg}} \frac{\sum_{j=1}^{n} \mathcal{M}_{\mathcal{G}\to\mathcal{T}}(\boldsymbol{g}_i, \boldsymbol{t}_j)\boldsymbol{t}_j}{\sum_{j=1}^{n} \mathcal{M}_{\mathcal{G}\to\mathcal{T}}(\boldsymbol{g}_i, \boldsymbol{t}_j)} + \boldsymbol{b}_{\text{agg}}\right) \tag{1}$$

Where $\boldsymbol{W}_{\text{agg}} \in \mathbb{R}^{D\times D}, b_{\text{agg}} \in \mathbb{R}^D$ are learnable parameters, $\boldsymbol{g}_i, \boldsymbol{t}_i \in \mathbb{R}^D$ are the i-th representation vectors in $\mathcal{G}$ and $\mathcal{T}$ respectively, D represents the dimension of the representations, and $\widehat{\boldsymbol{g}}_i$ is the aggregated representation of $\boldsymbol{g}_i$. The aggregation process from $\mathcal{G}$ into $\mathcal{L}$ w.r.t. $\mathcal{M}_{\mathcal{L}\to\mathcal{G}}$ is the same as above. We further conduct a aggregation directly from $\mathcal{T}$ into $\mathcal{L}$ w.r.t. $\mathcal{M}_{\mathcal{L}\to\mathcal{T}}$, which is obtained by:

$$\mathcal{M}_{\mathcal{L}\to\mathcal{T}}(\boldsymbol{l}_i, \boldsymbol{t}_j) = \vee_{j\in\{1,\ldots,m\}} \left(\mathcal{M}_{\mathcal{L}\to\mathcal{G}}(\boldsymbol{l}_i, \boldsymbol{g}_k) \wedge \mathcal{M}_{\mathcal{G}\to\mathcal{T}}(\boldsymbol{g}_k, \boldsymbol{t}_j)\right) \tag{2}$$

In summary, we update $\{\mathcal{G}, \mathcal{L}\}$ into $\{\widehat{\mathcal{G}}, \widehat{\mathcal{L}}\}$ using $\mathcal{T}$ and $\{\mathcal{T}, \mathcal{G}\}$ respectively. Such an aggregation operation is important in the startup phase. As described in 3.1, the semi-grained sequence and coarse-grained sequence are initialized as the grammatical type of the tokens and statements. They are like torsos with empty skeletons, and we need to fill them in with the textual information to make them complete. And the mapping-based aggregation is reasonable and necessary as the semantics of the element in high levels comes from the combination of its descendants' semantics. There is a ablation experiment demonstrating its effectiveness in Section 5.

Cross-granularity interaction. The aggregated semantics of the elements in semi/coarse-grained sequences tend to be local. To alleviate this dilemma, we further make the elements in a specific granularity to query all elements at three granularities, which guarantees that the model can capture both fine-grained and coarse-grained features.

Formally, we now have $\{\mathcal{T}, \widehat{\mathcal{G}}, \widehat{\mathcal{L}}\}$ after aggregation. Taking sequence $\widehat{\mathcal{G}} \in \mathbb{R}^{M\times D}$ of lenght M in certain granularity as an example, $\mathcal{G}$ will query all of $\{\mathcal{T}, \widehat{\mathcal{G}}, \widehat{\mathcal{L}}\}$ in the cross-granularity interaction process. We first concat the pyramid representation as

$$\mathcal{C} = \text{Concat}([\mathcal{T}, \widehat{\mathcal{G}}, \widehat{\mathcal{L}}]) \in \mathbb{R}^{(K+M+N)\times D} \tag{3}$$

here N, M, K are the sequence lengths of $\mathcal{T}, \widehat{\mathcal{G}}, \widehat{\mathcal{L}}$ respectively. A relative position encoded multi-head attention [25] is applied here and we use the relative position map $\mathcal{R}^{\widehat{\mathcal{G}}} \in \mathbb{Z}^{M\times|\mathcal{C}|}$ to make the

model granular-aware. The multi-head cross-attention (MHCA) process is calculated as:

$$\widetilde{\mathcal{G}} = \widehat{\mathcal{G}} + \Big[\text{MultiHead}(\widehat{\mathcal{G}}, \mathcal{C})\Big] \mathbf{W}_o \in \mathbb{R}^{M \times D} \tag{4}$$

$$\text{MultiHead}(\widehat{\mathcal{G}}, \mathcal{C}) = \text{Concat}([\text{Head}_1(\widehat{\mathcal{G}}, \mathcal{C}), \ldots, \text{Head}_H(\widehat{\mathcal{G}}, \mathcal{C})]) \in \mathbb{R}^{M \times D} \tag{5}$$

$$\text{Head}_h(\widehat{\mathcal{G}}, \mathcal{C}) = [\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_M] \in \mathbb{R}^{M \times \frac{D}{H}} \tag{6}$$

$$\boldsymbol{z}_i = \sum_{j=1}^{|\mathcal{C}|} \frac{\exp \boldsymbol{\alpha}_{ij}}{\sum_{j'=1}^{|\mathcal{C}|} \exp \boldsymbol{\alpha}_{ij'}} \left(\mathcal{C}_j \mathbf{W}_v^h + \widehat{\mathcal{R}}_{ij} \mathbf{W}_{rel_v}^{\widehat{\mathcal{G}}}\right) \in \mathbb{R}^{\frac{D}{H}} \tag{7}$$

$$\boldsymbol{\alpha}_{ij} = \frac{\widehat{\mathcal{G}}_i \mathbf{W}_q^h}{\sqrt{D/H}} \left(\mathcal{C}_j \mathbf{W}_k^h + \widehat{\mathcal{R}}_{ij} \mathbf{W}_{rel_k}^{\mathcal{G}}\right)^\top \in \mathbb{R} \tag{8}$$

$$\widehat{\mathcal{R}} = \text{Embedding}_{rel}^{\widehat{\mathcal{G}}}(\mathcal{R}^{\widehat{\mathcal{G}}}) \in \mathbb{R}^{M \times |\mathcal{C}| \times \frac{D}{H}} \tag{9}$$

where $\mathbf{W}_q^h, \mathbf{W}_k^h, \mathbf{W}_v^h \in \mathbb{R}^{\frac{D}{H} \times \frac{D}{H}}, h \in [1, \ldots, H]$ are projection matrices of Query, Key and Value for h-th head. $\mathbf{W}_o \in \mathbb{R}^{D \times D}$ is projection matrix for attention output. H represents the number of heads and $\widehat{\mathcal{X}}$ represents the output of multi-head self-attention process. $\mathbf{W}_{rel_k}^{\widehat{\mathcal{G}}}, \mathbf{W}_{rel_k}^{\widehat{\mathcal{G}}} \in \mathbb{R}^{\frac{D}{H} \times \frac{D}{H}}$ are projection matrices of Key and Value of relative position embedddings, the $\widehat{\mathcal{G}}$ means that thay are specific to $\widehat{\mathcal{G}}$.

For the construction of relative position map $\mathcal{R}$, we calculate the clipped relative distance with max distance as 32 for the pair within granularity and assign the relative distance as a default value (e.g. max distance + 1 and max distance + 2) for all the pair across the specific granularity. Figure 4 shows a toy example of granular-aware relative position map with max distance as 3 for $\mathcal{G}$ cross-granularity interaction. The cross-granularity interaction for $\mathcal{T}$ and $\mathcal{L}$ is similar to $\widehat{\mathcal{G}}$. We share attention parameters (i.e. $\mathbf{W}_q^h, \mathbf{W}_k^h, \mathbf{W}_v^h$ and $\mathbf{W}_o$) across three granularities,but assign specific the relative path parameters for each granularity. In summary, in this cross-granularity interaction process, semantics of the feature pyramid updated as:

	$\mathcal{T}$									$\mathcal{G}$				$\mathcal{L}$	
	5	5	5	5	5	5	5	5	5	0	1	2	3	4	4
$\mathcal{G}$	5	5	5	5	5	5	5	5	5	1	0	1	2	4	4
	5	5	5	5	5	5	5	5	5	2	1	0	1	4	4
	5	5	5	5	5	5	5	5	5	3	2	1	0	4	4

Figure 4: a toy example of granular-aware relative position map for $\mathcal{G}$ cross-granularity interaction.

$$\widetilde{\mathcal{L}}, \widetilde{\mathcal{G}}, \widetilde{\mathcal{T}} = \text{MHCA}(\widehat{\mathcal{L}}, \mathcal{C}), \text{MHCA}(\widehat{\mathcal{G}}, \mathcal{C}), \text{MHCA}(\mathcal{T}, \mathcal{C}) \tag{10}$$

3.3 Pyramid Attention Transformer

As illustrated in Figure 2, the overall model is similar to the vanilla Transformer, we replace the self-attention module using our proposed pyramid attention mechanism for an efficient process of the multi-scale input. We primally use the fine-grained sequence in the output pyramid of the encoder for decoding, which already contains enough semantics for generation and reduces the complexity of the model design. We empirically show that such a design choice works well without causing performance degradation in Section 5. Following prior works [29], we also apply a copy mechanism in our model which reduces the risk caused by out-of-vocabulary and speedups training convergence.

Although there is overlap between the vocabularies of different granular, the meanings of these common words are quite different when differentiated at different granularities. For instance, `else` maybe just a natural word in the textual sequence, but a keyword for control in a grammatical sequence. Thus, we assign individual embedding layer for each granular to avoid such confusion.

For efficient parameter usage, a parameter sharing strategy is applied in the cross-granularity interaction process as stated in Section 3.2, We find the self-attention mechanism is strong enough to handle this multi-role mission. Thus, our model doesn't add too many parameters compared to Transformer. And such a parameter sharing strategy leads to efficient model implementation, we just need concat the three sequences and the relative position maps, then directly adopt a relative position encoded multi-head self-attention layer to implement the cross-granularity interaction module.

4 Experiments

Dataset and Metrics. To demonstrate the effectiveness of the proposed method, we conduct experiments on two widely-used and well-developed java datasets: EMSE-DeepCom[1] [11] which collected from GitHubs Java repositories and FunCom[2] [14] which have $\sim$2 million java method-comment pairs. We filter out the examples that cannot be parsed properly or have a too large size for both datasets. And deduplication is done on EMSE-DeepCom to avoid data leakage. Table 1 shows the statistics of the datasets.

Table 1: Dataset statistics. #train and #test mean the number of examples for training and test, #sub-token means the max the sequence length when the code is tokenized as sub-token, and the rest are similar to this.

Name	#train	#test	#sub-token	#token	#statement	#summary
EMSE-Deepcom	295,967	12,226	196	160	16	24
FunCom	1,017,964	53,936	256	196	32	32

We use 6 metrics to evaluate the model performance on the datasets. BLEU [23], Rouge-L [19] and Meteor [5] are widely used to evaluate the quality of text similarity in the domain of natural language generation. Another 3 metrics are used to qualify the token-wise prediction performance. We use beam search with a beam size of 3 for all models in the testing phase.

Data preprocessing. All data preprocessing of code is based on the parsed syntax tree. We obtain the AST using the open-sourced tool Tree-sitter[3]. Please refer to 3.1 for details of the proposed pyramid data construction. We re-implement the data preprocessing pipeline for all compared baselines based on the AST parsed by Tree-sitter, please refer to Appendix C for more details.

Baselines. Early works viewed source code as a sequence of text and use the sequential model like LSTM or Transformer, we mainly compare with CODE-NN [12] and NeuralCodeSum [29] for this type of methods. A few works only learned the structure features for this generation task, we compare with TreeLSTM [28] here. Most works both exploit the textual and structural information for this task. One group of works learn to represent these two types of information separately, then a hybrid mechanism is used to combine them in the decoding phase. We do comparison with HDeepCom [11], ASTAttnGRU [13] and CAST [26] that in the hybrid fashion. Another line of methods uses structure relations as an inductive bias to guide the information interactions in the model. Here, we compare with SiT [34], GREAT [9] and TPTrans [24]. For clarity, the backbone and taxonomy of each method are listed in Table 2

And we add a copy mechanism for all transformer-based models. We directly adept the model implementation script if the code is publicly available, and re-implement the method according to the description in the corresponding paper if the code is not provided. What's more, all baselines are integrated under a unified PyTorch-based code summarization framework developed by ourselves for fair comparisons. We provide implement details for baselines in Appendix A.2.

Hyperparameters and setup. For a fair comparison, all the Transformer-based models use the default Transformer configurations with embedding dimension as 512, feedforward dimension as 2048, head number as 8, and layer number for encoder/decoder as 6 and all RNN-based models use the hidden dimension with 512. For embeddings, we use the embedding dimension with 512 for all approaches. The size of vocabularies is limited by 50000 over code sequences and 30000 over summary sequences in EMSE-DeepCom, and the corresponding configuration is 35000 and 30000 in FunCom. All model is trained using NVIDIA Tesla A100 GPUs with a batch size of 64. We train all baselines including our models using AdamW optimizer with a multi_step learning rate scheduler, and set the initial learning rate to 0.0002 and 0.003 for Transformer-based and RNN-based models, respectively. And warmup strategy is used for stable training in the Transformer-based model. The training epochs for each method are listed in Table 2.

[1] `https://github.com/xing-hu/EMSE-DeepCom`

[2] `http://leclair.tech/data/funcom/`

[3] `https://tree-sitter.github.io/tree-sitter/`

Table 2: Comparisons with other code summarization methods on a middle-size dataset and a large-size dataset.

Methods	Backbone	Hybrid	Structure	Epochs	BLEU	Rouge-L	Meteor	Precision	Recall	F1
RMSE-Deepcom (middle-size)										
CODE-NN [12]	LSTM			24	28.448	43.506	17.886	47.266	45.165	44.774
TreeLSTM [28]	LSTM		✓	24	28.992	43.985	18.179	48.486	45.115	45.287
HDeepCom [11]	GRU	✓	✓	24	32.179	49.029	21.528	54.273	50.584	50.754
ASTAttnGRU [13]	GRU	✓	✓	24	33.041	49.761	22.205	54.539	51.754	51.467
SiT [34]	Transformer		✓	30	35.689	53.750	24.196	60.784	54.243	55.717
GREAT [9]	Transformer		✓	30	36.382	53.606	24.181	60.008	54.230	55.460
NeuralCodeSum [29]	Transformer			30	37.133	54.800	25.051	61.356	55.413	56.682
CAST [26]	Transformer	✓	✓	30	37.195	54.868	25.069	61.601	55.368	56.747
TPTrans [24]	Transformer		✓	30	37.248	54.996	25.022	62.022	55.354	56.884
PA-former (ours)	Transformer			30	**38.837**	**55.978**	**25.923**	**62.278**	**56.617**	**57.816**
Funcom (large-size)										
CODE-NN [12]	LSTM			24	31.862	48.896	19.108	54.018	49.241	49.924
TreeLSTM [28]	LSTM		✓	24	31.462	48.293	18.869	53.237	48.902	49.332
HDeepCom [11]	GRU	✓	✓	24	35.063	53.350	22.645	59.460	53.693	54.808
ASTAttnGRU [13]	GRU	✓	✓	24	37.001	55.034	23.753	61.244	55.315	56.523
SiT [34]	Transformer		✓	36	42.121	59.330	26.819	65.562	59.427	60.839
GREAT [9]	Transformer		✓	36	43.286	60.364	27.439	66.186	60.580	61.831
NeuralCodeSum [29]	Transformer			36	43.355	60.405	27.540	66.488	60.459	61.860
TPTrans [24]	Transformer		✓	36	43.450	60.566	27.607	66.765	60.568	62.030
CAST [26]	Transformer	✓	✓	36	43.580	60.524	27.665	66.458	60.657	61.976
PA-former (ours)	Transformer			36	**44.649**	**61.450**	**28.274**	**67.210**	**61.593**	**62.863**

Comparisons with baselines. The comparison with baselines is listed in Table 2. Without bells and whistles, we find that our proposed PA-former substantially surpasses all baselines and achieves new state-of-the-art on both middle-size and large-size datasets. Compared with NeuralCodeSum which is a relative positional encoded Transformer, our method has a 5% improvement in Bleu on the middle-size dataset. For the current leading methods CAST and TPTrans, which both take all the nodes in ASTs and the sub-token sequence as inputs, our model only takes a subset of AST nodes and sub-tokens surpass both of them over 1.0+ Bleu scores on two datasets. And benefited from the efficient implementation and parameter sharing strategy, our model needs far less training time for converging, while TPTrans needs a very long time to converge as the large size pair-wise paths and the sparse scatter algorithm. The leading results demonstrate that our multi-scale formulation for the task of source code summarization is reasonable. To further evaluate the effectiveness of our method, we provide a case study in Appendix D.

5 Ablation Study

We now analyze the importance of designs in our proposed PA-former using a series of ablation studies on EMSE-DeepCom dataset. Unless otherwise specified, the training setup is consistent with the above.

Multi-scale *v.s.* single-scale. To demonstrate the effectiveness of multi-granularity formulation, we degenerate the pyramid input into a single large sequence by concating the three granularity sequences and directly feed the concated sequence into a relative positional encoded Transformer with a copy mechanism. As shown in Tabel 3, more data in the input sequence obtains performance improvement compared to NeuralCodeSum which only takes a sub-token sequence as input. However, processing the same data using our proposed multi-scale way achieves higher performance.

Table 3: Evaluation results of multi-scale *v.s.* single-scale.

Methods	BLEU	Rouge-L	Meteor	Precision	Recall	F1
single-scale	37.329	55.032	25.114	61.586	55.750	56.946
multi-scale	**38.848**	**56.095**	**25.895**	**62.498**	**56.642**	**57.903**

Effectiveness of pyramid attention. The core components of pyramid attention are the bottom-to-top aggregation and cross-granularity interaction. We show the necessity for both of these two components here. To demonstrate the effectiveness of bottom-to-top aggregation, we evaluate a variant of PA-former whose bottom-to-top pathway is disabled. As shown in Table 4, the model performance drops by about 1.1 BLEU scores without such a component. To demonstrate the effectiveness of cross-granularity interaction, we evaluate a variant of PA-former whose cross-granularity interaction is disabled which directly degenerates into NeuralCodeSum as we only use the fine-grained encoder output for decoding. As shown in Table 4, the model performance drops by about 1.7 BLEU scores without such a component. Such results reveal the importance of cross-granularities interaction. We further disable the granular-aware relative embedding to investigate its importance, and we observe a big drop in performance shown in Table 4, which shows its effectiveness.

Table 4: Evaluation results of pyramid attention ablation.

Methods	BLEU	Rouge-L	Meteor	Precision	Recall	F1
PA-former	**38.848**	**56.095**	**25.895**	**62.498**	**56.642**	**57.903**
w/o bottom-to-top	37.532	55.153	25.131	61.815	55.590	56.977
w/o cross-granularity	37.133	54.800	25.051	61.356	55.413	56.682
w/o granular-aware rel.	36.905	54.690	24.896	61.443	55.224	56.560

Decoding strategy. The outputs of our PA-former encoder are a pyramid representation, thus there are various ways for decoding. Our PA-former directly uses the fine-grained sequence just for simplicity. Here, We study other implementations that utilize all three granularity of representation. The most intuitive method is concating the three granularity representations, and another way is using them in a serial strategy [17]. As Table 5 shows, using the fine-grained representation is good enough in this task to benefit from the design of information interaction across granularities.

Table 5: Evaluation results of decoding strategy.

Decoding strategy	BLEU	Rouge-L	Meteor	Precision	Recall	F1
fine-grained	**38.848**	**56.095**	**25.895**	**62.498**	**56.642**	**57.903**
serial	38.729	55.894	25.803	62.223	56.425	57.669
concat	38.556	55.819	25.797	62.174	56.456	57.638

Grammar input. When constructing the pyramid input, we intentionally enforce the bias between different granularities by assigning different types of content. For example, we bias the semi-grained sequence into grammar features by assigning token grammar types to it. To evaluate the rationality of assigning different types of content, we compare the proposed pyramid input without grammar information, which set the initial content of semi-grained and coarse-grained sequences to all empty. The results in Table 6 prove the rationality of pyramid input design.

Table 6: Evaluation results of grammar input.

pyramid input content	BLEU	Rouge-L	Meteor	Precision	Recall	F1
PA-former	**38.848**	**56.095**	**25.895**	**62.498**	**56.642**	**57.903**
w/o grammar	38.394	55.744	25.706	62.198	56.458	57.637

6 Conclusion

In this paper, we present a multi-scale formulation for source code summarization. Given a code snippet, we construct a pyramid input with three granularities. And a pyramid attention mechanism is proposed for efficient information interaction across and within granularities. Without bells and whistles, the proposed methods obtain 5% improvement over The strong NeuralCodeSum on a middle-size dataset and achieve new state-of-the-art results. Additional ablation studies show the effectiveness of proposed modules. Furthermore, our multi-scale formulation for the source code can be easily extended to other code understanding tasks.

References

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[2] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

[3] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages (POPL)*, 2019.

[5] S. Banerjee and A. Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 2005.

[6] H. Fan, B. Xiong, K. Mangalam, Y. Li, Z. Yan, J. Malik, and C. Feichtenhofer. Multiscale vision transformers. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2021.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. In *International Conference on Learning Representations (ICLR)*, 2020.

[8] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2021.

[9] V. J. Hellendoorn, P. Maniatis, R. Singh, C. Sutton, and D. Bieber. Global relational models of source code. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.

[10] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2018.

[11] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation with hybrid lexical and syntactical information. In *Empirical Software Engineering*, 2019.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

[13] A. LeClair, S. Haque, L. Wu, and C. McMillan. Improved code summarization via a graph neural network. In *Proceedings of International Conference on Program Comprehension (ICPC)*, 2020.

[14] A. LeClair and C. McMillan. Recommendations for datasets for source code summarization. In *Proceedings of NAACL-HLT*, 2019.

[15] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[16] Y. Liang and K. Q. Zhu. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[17] J. Libovický, J. Helcl, and D. Marecek. Input combination strategies for multi-source transformer decoder. In *Proceedings of the Third Conference on Machine Translation (WMT)*, 2018.

[18] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu. Improving code summarization with block-wise abstract syntax tree splitting. In *Proceedings of 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2021.

[19] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the 42th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2004.

[20] T.-Y. Lin, P. Dollt'ar1, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[21] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2021.

[22] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

[23] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2002.

[24] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin. Integrating tree path in transformer for code representation. In *Advances of 35th Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[25] P. Shaw, J. Uszkoreit, and A. Vaswani. Self-attention with relative position representations. In *In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.

[26] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of International Conference on Program Comprehension (ICPC)*, 2021.

[27] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances of the Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

[28] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

[29] A. W. Uddin, C. Saikat, R. Baishakhi, and C. Kai-Wei. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, and ukasz Kaiser. Attention is all you need. In *Advances of the 31st Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[31] B. Wei, G. Li, X. Xia, Z. Fu, , and Z. Jin. Code generation as a dual task of code summarization. In *Advances of 33th Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[32] H.-H. Wei and M. Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[33] M. White, M. Tufano, C. Vendome, , and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[34] H. Wu, H. Zhao, and M. Zhang. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP*, 2021.

[35] Y. Yuan, R. Fu, L. Huang, W. Lin, C. Zhang, X. Chen, and J. Wang. Hrformer: High-resolution transformer for dense prediction. In *Advances of 35th Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[36] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[37] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann. Language-agnostic representation learning of source code from structure and context. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.