Code Summarization with Structure-induced Transformer

Hongqiu Wu[1,2,3], **Hai Zhao**[1,2,3,*], **Min Zhang**[4]
[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2] Key Laboratory of Shanghai Education Commission for Intelligent Interaction and Cognitive Engineering, Shanghai Jiao Tong University, Shanghai, China
[3] MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China
[4] Institute of Artificial Intelligence, School of Computer Science and Technology, Soochow University, Suzhou, China
wuhongqiu@sjtu.edu.cn, zhaohai@cs.sjtu.edu.cn, minzhang@suda.edu.cn

Abstract

Code summarization (CS) is becoming a promising area in recent language understanding, which aims to generate sensible human language automatically for programming language in the format of source code, serving in the most convenience of programmer developing. It is well known that programming languages are highly structured. Thus previous works attempt to apply structure-based traversal (SBT) or non-sequential models like Tree-LSTM and graph neural network (GNN) to learn structural program semantics. However, it is surprising that incorporating SBT into advanced encoder like Transformer instead of LSTM has been shown no performance gain, which lets GNN become the only rest means modeling such necessary structural clue in source code. To release such inconvenience, we propose structure-induced Transformer, which encodes sequential code inputs with multi-view structural clues in terms of a newly-proposed structure-induced self-attention mechanism. Extensive experiments show that our proposed structure-induced Transformer helps achieve new state-of-the-art results on benchmarks.

1 Introduction

By 2020, software development and maintenance become an indispensable part of human work and life. Various assistant technical measures have been taken to facilitate more enjoyable software development, among which it is especially welcomed by programmers when there is a code summarization task generating sensible human language annotations automatically.

Code (Java)	private void attachPlot (SVGPlot newplot) {
	this.plot = newplot;
	if (newplot == null) {
	super.setSVGDocument(null);
	return;
	}
	newplot.synchronizeWith(synchronizer);
	super.setSVGDocument(
	newplot.getDocument());
	super.setDisableInteractions(
	newplot.getDisableInteractions());
	}

Summ.	Attach to a new plot and display.

Code (Python)	def get_change_lines_in_file_for_tag(tag, change_dict):
	cleaned_lines = []
	data_list = change_dict.get('data', [])
	for data_dict in data_list:
	block = data_dict.get('block', '')
	lines = block.split('\\n')
	for line in lines:
	index = line.find(tag)
	if (index >(-1)):
	line = line[index:]
	cleaned_lines.append(line)
	return cleaned_lines

Summ.	The received change_dict is the jsonified version of the changes to a file in a changeset being pushed to the Tool Shed from the command line. This method cleans and returns appropriate lines for inspection.

Table 1: Task samples of code summarization, where summ. refers to the output summary.

In early days, code summarization was a derivative problem of information retrieval (Haiduc et al., 2010; Eddy et al., 2013; Wong et al., 2013, 2015) by matching the most similar code snippets which are labeled with summaries. Such method lacks generalization and performs unsatisfactorily. Thus in recent years, researchers treated code summarization as a task of language generation (Iyer et al., 2016; Liang and Zhu, 2018), which usually depends on RNN-based Seq2Seq models (Cho et al., 2014; Bahdanau et al., 2015).

It is already known that RNN-based models may encounter bottleneck when modeling long

*Corresponding author. This paper was partially supported by National Key Research and Development Program of China (No. 2017YFB0304100), Key Projects of National Natural Science Foundation of China (U1836222 and 61733011), Huawei-SJTU long term AI project, Cutting-edge Machine Reading Comprehension and Language Model. This work was supported by Huawei Noah's Ark Lab.

Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pages 1078–1090

	Structure-sensitive	Long-term dependency	Feat-model match
LSTM			
Tree-LSTM	✓		
Transformer		✓	
LSTM + SBT	✓		✓
Transformer + SBT	✓	✓	
SiT	✓	✓	✓

Table 2: Comparison of the previous models with proposed SiT model. The last column refers to whether input features match with the corresponding model.

sequences due to its poor long-term dependency. For instance, a normal snippet of Java as shown in Table 1 usually has hundreds of tokens. More recently, Ahmad et al. (2020) used an enhanced Transformer-based model to capture long-term and non-sequential information of source code, which outperformed previous RNN-based models by a large margin.

On the other hand, in the light of the structural nature of programming languages, structure clues are supposed to greatly enhance programming language processing task like code summarization (Fernandes et al., 2019). Indeed, substantial empirical studies showed that Abstract Syntax Tree may help models better comprehend code snippets and achieve more sensible generation results. Previous approaches could be divided into two categories. The first is to employ non-sequential encoders (e.g., TBCNN (Mou et al., 2016), Tree-LSTM (Shido et al., 2019), Tree-Transformer (Harer et al., 2019), Graph Neural Network (Allamanis et al., 2018; Liu et al., 2020; Alex et al., 2020; Wang et al., 2021)) to directly model structural inputs. The other is to pre-process structural inputs to apply sequential models on them. Uri et al. (2019) used LSTM to encode code structure by sampling possible paths of AST. Another similar work is structure-based traversal (SBT) (Hu et al., 2018a), which manages to flatten ASTs into linear sequences.

Though existing studies achieve success on the concerned code summarization task more or less, there is still room in improving both of the above modeling approaches. It is well known RNN encoders like LSTM only have limited capabilities in capturing long-range dependencies in sequence, and GNN-like models may be too sensitive to local information, which casts a natural solution, what if incorporating SBT into the Transformer? However, it is surprising that SBT only works effectively with LSTM but not the Transformer according to Ahmad et al. (2020). We attribute this to the linear and nonlinear inconsistence between SBT and encoder forms. SBT enables sequential encoders to learn non-sequential relationship (such as syntax) still in a certain elaborate linear forms. RNN may be effectively enhanced by SBT right because of its sequential architecture through attention mechanism. Transformer learns features through self-attention network (SAN), nevertheless which acts more like a non-sequential process. Consequently, such sequential features are unsuitable for a non-sequential architecture to extract implicit structural information. We boldly call it Feature-Model Match problem in Table 2. In this paper, we thus design an improved Transformer variants, structure-induced Transformer (SiT) to alleviate such difficulty in terms of a structure-induced self-attention mechanism, so that the resulted model may enjoy both merits, capturing long-range dependencies and more global information. The proposed model design has been applied to benchmark datasets and helps achieve new state-of-the-art performance.

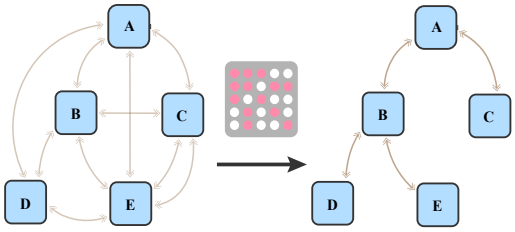


Figure 1: Use of adjacency matrix to transform original self-attention, left-hand complete graph, into structure-induced self-attention, right-hand graph which looks clear-cut. Note that we omit self-circles for concision.

2 Structure-based Code Summarization

The following sections present our code summarization method with two parts, in which the first is about structure representation of code, and the second is our proposed structure-induced Transformer.

2.1 Structure Representation of Code

Note that programming language like source code is subtle that certain different formats may result in different compilations. Thus pre-processing could be an great impact in code summarization.

We adopt Abstract Syntax Tree (AST) for representing the language grammar of source code as usual. Figure 2 depicts a typical AST, which is composed of terminal nodes and non-terminal nodes. A

non-terminal node represents certain construction like *If* and *BinaryOp*, while terminal nodes represent its semantic components, such as identifiers and numbers.

In model implementation, we adopt adjacency matrix A to represent the AST instead of structure based traversal method as in Hu et al. (2018a), which represents tree structure in a sequential format. Such choice is well compatible with Transformer, which calculates attention weights by performing a dot-product of key-query pairs and results in an attention matrix of $l \times l$. We let l equal to number of AST nodes, then code summarization with Transformer becomes possible through applying a position-wise multiplication of A and original attention matrix.

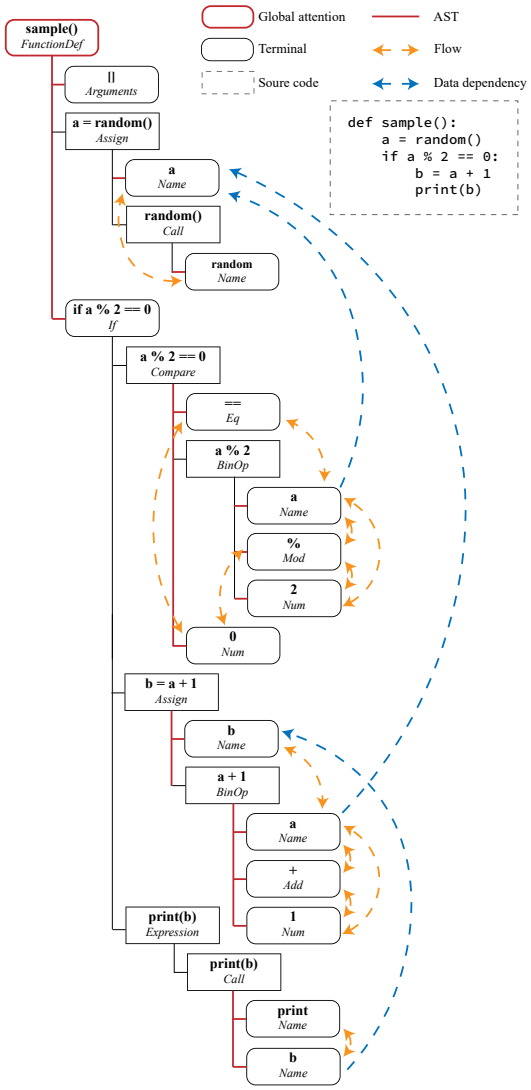


Figure 2: A Python code sample of multi-view graph used in Si-SAN. The code snippet is referred from Liu et al. (2020), which is original in Java.

Inspired by Code Property Graph (CPG) (Yamaguchi et al., 2014; Liu et al., 2020), we further expand AST into a multi-view network (MVN or multi-view graph) (Sindhwani et al., 2005; Zhou and Burges, 2007; Kumar et al., 2011). An MVN is composed of multiple views, each view corresponding to a type of structural relationships while all views sharing the same set of vertexes (Shi et al., 2018). In this paper, we construct a three-view graph based on different code semantics, which are abstract syntax, control flow and data dependency. We show an example in Figure 2, where we use colorful strokes to describe different compositions in the graph. Note that we only utilize terminal nodes which are marked as rounded rectangles.

Specifically, we first generate an AST, on the basis of which we add additional edges to further represent the flow of control and data. For control flow, since Transformer is order-sensitive with position encoding, we only need to focus on each statement node. For instance, nodes *b*, =, *a*, +, *1* make a complete statement *b=a+1*. We connect each of them since they are in the same execution order. For data dependencies, we connect relevant data across the whole program, as the variable *b* in expression *print(b)* and assignment *b=a+1* respectively, where the former is defined and loaded from the latter.

Now we may obtain three adjacency matrices of syntax, flow and dependency respectively, which are colored in red, yellow and blue in Figure 2. We combine them together and finally obtain a multi-view graph. Additionally, we add global attention on the root, which is allowed to attend to all tokens in the code, and all tokens in the code can attend to it. With aggregated structure, our structure-based code summarization is expected to capture various semantics of programs.

Note that our multi-view graph is different from CPG. which is original for C/C++ only and we do not find an appropriate analysis platform for other languages.

2.2 Structure-induced Transformer

Followed by appropriate structure representation and graph construction, we now propose our structure-induced Transformer (SiT) for code summarization, which is a structure-sensitive transformer (Zhang et al., 2020b; Narayan et al., 2020; Xu et al., 2020) model and is able to comprehend code snippets both semantically and syntac-

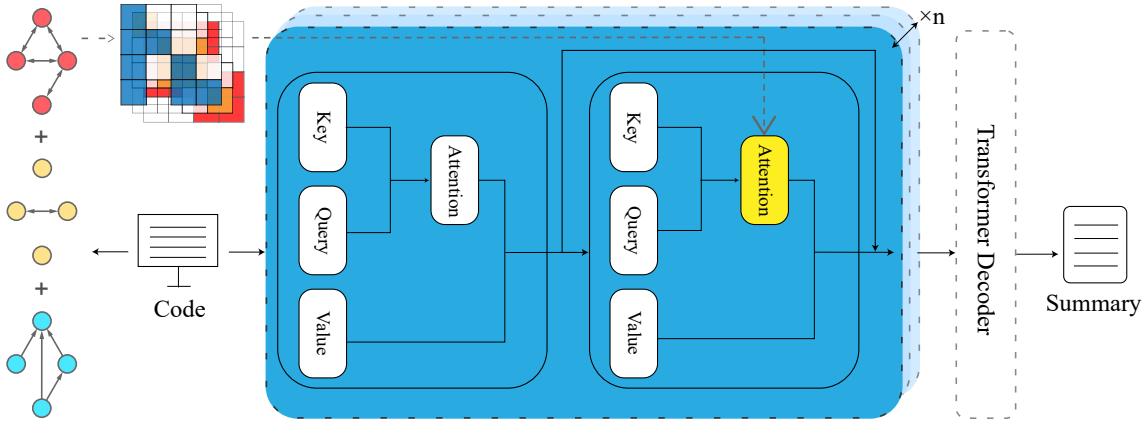


Figure 3: Overall architecture of Structure-induced Transformer (SiT).

tically. Meanwhile, we do not introduce extra parameters in SiT so that guarantee the training efficiency. In this section, we first review the self-attention network (SAN) of Transformer in terms of attention graph. Then we correspondingly propose structure-induced self-attention to build the structure-induced Transformer.

Vanilla Self-Attention Transformer is composed of stacks of identical layers for both encoder and decoder (Vaswani et al., 2017). Each layer emphasizes on self-attention mechanism, which is denoted as:

$$SAN(X) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

where $X = (x_1, \ldots, x_l)$ denotes the input sequence of sub-words, l denotes the sequence length and d_k denotes the hidden size per head. Now we view each sub-word as a vertex n and inner product of each key-value pair as a directed edge e, the SAN can be described as a directed cyclic graph. Equation 1 can be rewritten as follow:

$$SAN(X) = E \cdot N \quad (2)$$

The attention scores $E = \{e_{ij}\}$ refers to a weight matrix of edges where e_{ij} represents how significant node n_i attend to node n_j, while value matrix $N = \{n_i\}$ refers to each node representation. Figure 1 depicts the process of calculating attention scores.

Note that SAN actually generates a fully connected cyclic graph without consideration of the very needed structure-aware representation for our concerned task.

Structure-induced Self-Attention To represent the needed structure information, we propose structure-induced self-attention network (Si-SAN).

Specifically, we introduce multi-view network into Equation 1, that is, multiply the adjacency matrix by key-query pairs:

$$SiSAN(X) = Softmax\left(\frac{A_{mv} \cdot QK^T}{\sqrt{d_k}}\right)V \quad (3)$$

where A_{mv} refers to the multi-view representation of code.

Note that Si-SAN does not change the input code but appropriately incorporate code structure into SAN by changing its attention pattern. As shown in Figure 1, when $a_{ij} = 0$ in A_{mv}, the attention between n_i and n_j will be dropped out (Wu et al., 2021). We consequently obtain a more explicit attention graph. Different from calculating global information onto the whole sentence in original SAN, Si-SAN is expected to calculate structural information more accurately.

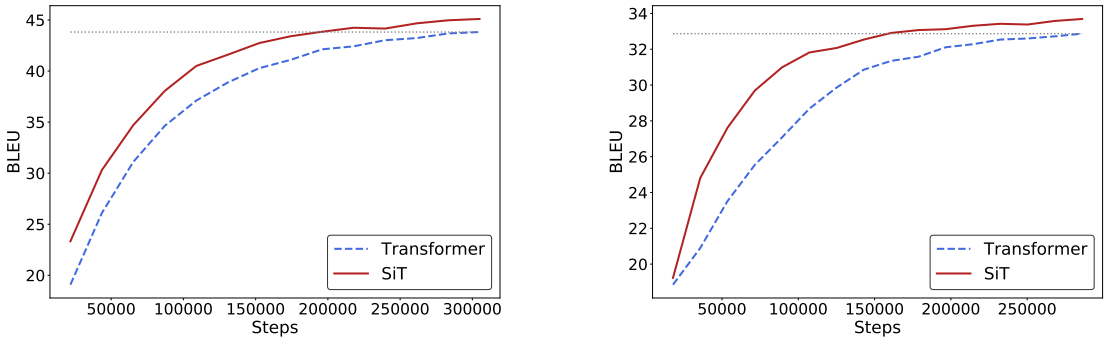
Structure-induced Module To enhance robustness and avoid over-pruning, we introduce structure-induced module, which is a stack of two layers, SAN and Si-SAN. In each module, SAN is followed by Si-SAN and the output is the combination of both layers. Specifically, given input sequence $X = (x_1, \ldots, x_l)$, where l denotes sequence length, we first pass it through an SAN layer to obtain hidden representation denoted as $H = (h_1, \ldots, h_l)$:

$$H = Concat(SAN_1(X), \ldots, SAN_h(X)) \quad (4)$$

where h refers to number of heads of multi-head attention while SAN_i refers to self-attention of

Model	Java			Python		
	BLEU	**ROUGE-L**	**METEOR**	**BLEU**	**ROUGE-L**	**METEOR**
CODE-NN (Iyer et al., 2016)	27.60	41.10	12.61	17.36	37.81	09.29
Tree2Seq (Eriguchi et al., 2016)	37.88	51.50	22.55	20.07	35.64	08.96
Hybrid2Seq (Wan et al., 2018)	38.22	51.91	22.75	19.28	39.34	09.75
DeepCom (Hu et al., 2018a)	39.75	52.67	23.06	20.78	37.35	09.98
API + Code (Hu et al., 2018b)	41.31	52.25	23.73	15.36	33.65	08.57
Dual Model (Wei et al., 2019)	42.39	53.61	25.77	21.80	39.45	11.14
Transformer (Ahmad et al., 2020)	44.58	54.76	26.43	32.52	46.73	19.77
Transformer* (Ahmad et al., 2020)	44.87	54.95	26.58	32.85	46.93	19.86
SiT	**45.76**(↑1.18)	**55.58**(↑0.82)	**27.58**(↑1.15)	**34.11**(↑1.59)	**48.35**(↑1.62)	**21.11**(↑1.34)
CodeBERT*† (Feng et al., 2020)	43.33	54.64	26.20	33.47	49.35	21.69
SiT on CodeBERT†	**45.19**(↑0.61)	**55.87**(↑1.11)	**27.52**(↑1.09)	**34.31**(↑1.79)	**49.71**(↑2.98)	**22.09**(↑2.32)

Table 3: BLEU, ROUGE-L and METEOR for our approach compared with other baselines. † refers to pre-trained models while * refers to models we rerun. The results of upper part are directly reported from Ahmad et al. (2020). Note that we only rerun Transformer and CodeBERT since they are much stronger than the other baselines. However, our results are even stronger. We show the ranges compared to the Transformer in Ahmad et al. (2020).



(a) BLEU score with training steps on Java

(b) BLEU score with training steps on Python

Figure 4: Convergence between Transformer and SiT.

head i. Subsequently, we pass H through a Si-SAN layer to obtain $H' = (h'_1, \ldots, h'_l)$:

$$H' = Concat(SiSAN_1(H), \ldots, SiSAN_h(H)) \tag{5}$$

Finally, we use an aggregation to fuse H and H' to obtain final representation $\bar{H} = (\bar{h_1}, \ldots, \bar{h_l})$:

$$\bar{H} = Aggr(H, H') \tag{6}$$

where the aggregation we use is simple position-wise sum. We explore that the structure-induced module is more robust and leads to a better performance. In each stack, model begins to learn global information with SAN, where all connections are available. Subsequently, through Si-SAN, model is told which of the connections are useful and which should be shut down and thus avoiding over-pruning. Note that SiT with 3 stacks of structure-induced modules still consists of 6 encoder layers and 6 decoder layers, but only changes the architecture between modules of Transformer, not introducing any extra parameters.

Figure 3 depicts the overall architecture of SiT. Compared to original Transformer, our SiT with Si-SAN encodes a more accurate relative representation of code through pruning redundant connections.

2.3 SiT-based Code Summarization

Based on our structure-induced Transformer (SiT), now we specify our code summarization process.

We first transform the input code into adjacency matrices of multiple views and combine them through a weighted sum:

$$A_{mv} = \alpha A_{ast} + \beta A_{fl} + \gamma A_{dp} \tag{7}$$

where α, β, γ refer to the corresponding weight for each view. Then we pass code sequences and corresponding adjacency matrices into SiT encoder, which contains 3 Si-SAN layers. For decoder, we

apply original Transformer decoder with cross attention. Finally, the summarization of the input code is generated through autoregressive decoding.

3 Experiments

3.1 Datasets and Pre-processing

Datasets Our experiments are conducted on two benchmarks of Java (Hu et al., 2018a) and Python (Wan et al., 2018), and for both we follow their training, test and development divisions.

Graph Construction For Java code, we refer to the method provided in (Hu et al., 2018a). They use *javalang* module of Python to compile Java and fetch AST in a dictionary form. For Python code, we generate trees by ourselves based on *ast* and *asttokens* modules. Finally, we write a script to resolve ASTs into multi-view adjacency matrices[1], where we let $\alpha = \beta = \gamma = 1$ for all experiments[2].

Out-Of-Vocabulary Code corpus in programming language may have a much bigger vocabulary than natural language, including vast operators and identifiers. We have to introduce vast out-of-vocabulary (OOV) tokens (usually replaced by $\langle$UNK$\rangle$) (Hu et al., 2018a) to keep it in a regular size. To avoid OVV problem, we apply *CamelCase* and *snake_case* tokenizers (Ahmad et al., 2020) to reduce code vocabulary and remove all extra nodes which do not correspond to specific tokens.

3.2 Baselines

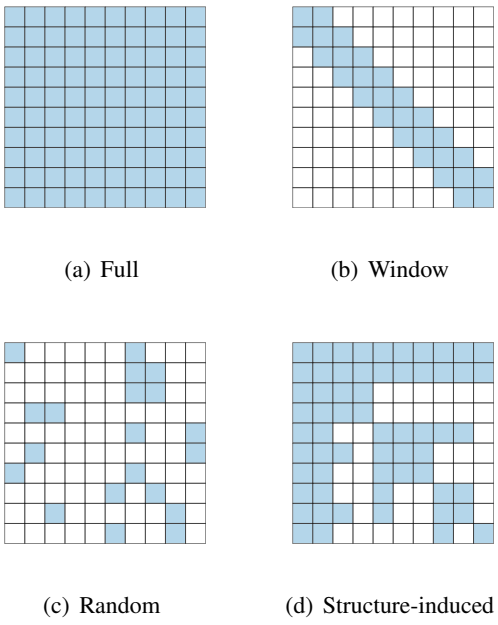
We take all three categories of state-of-the-art models as our baselines for comparison.

Transformer We refer to the enhanced Transformer in (Ahmad et al., 2020) which equipped with copy attention (See et al., 2017) and relative position encoding (RPE) (Shaw et al., 2018). For fair enough comparison, we run their model on our machine under the same environment with SiT. Note that we also utilize RPE in SiT because of its better capability in capturing long sequences, while we do not utilize copy attention.

LSTM This group includes all relevant LSTM models with sequential and non-sequential inputs (Iyer et al., 2016; Eriguchi et al., 2016; Wan et al., 2018; Hu et al., 2018a,b; Wei et al., 2019).

[1]https://github.com/gingasan/astruc

[2]We try to adjust the weights of three views, showing little performance variant, which suggests that self-attention network itself may balance the relative significance between the three.



(a) Full (b) Window

(c) Random (d) Structure-induced

Figure 5: Comparison of different types of self-attention pattern. (b) Window attention with $w = 2$. (c) Random attention with $r = 2$.

Pre-trained Language Model We also compare our model with CodeBERT (Feng et al., 2020), a pre-trained language model on both natural and programming languages. It is pre-trained over six programming languages with MLM (Devlin et al., 2019) and RTD (Clark et al., 2020).

3.3 Training Details

We train our model on a single nVidia Titan RTX with batch size in {32, 64}. The learning rate is in {3e-5, 5e-5} with warm-up rate of 0.06 and L2 weight decay of 0.01. The maximum number of epochs is set to 150 for Transformer and 30 for CodeBERT. For validation, we simply use greedy search, while for evaluation, we use beam search with beam size in {4, 5, 8} and choose the best result[3].

3.4 Main Results

Scores Table 3 shows the overall results on Java and Python benchmarks. The Transformer baseline is strong enough as it outperforms all the previous works by a significant margin. However, our model is more powerful, further boosting Transformer with more than 1 BLEU points on Java and Python respectively and achieves new state-of-the-art results. Specifically, SiT achieves higher scores

[3]https://github.com/gingasan/sit3

on Python, increasing by 1.59, 1.62 and 1.34 points on BLEU, ROUGE-L and METEOR respectively. According to dataset statistics, Python contains 5 times more unique code tokens than Java, which makes it much more challenging. Thus the superiority of SiT on Python tends to be notable. Even so, SiT still boosts Transformer by 1.18, 0.82 and 1.15 points on BLEU, ROUGE-L and METEOR respectively on Java.

Convergence Moreover, Figure 4 shows the trend of BLEU scores on development set over training steps. SiT achieves a much faster convergence rate than Transformer. For instance on Python dataset, SiT arrives the best performance of Transformer in about 100 epochs, while the latter one still needs 50 more to finally achieve the optimal. Note that the running time of each epoch for both models is the same. Such high convergence rate helps showcase the necessity of Si-SAN.

Pre-training On the other hand, we can see that CodeBERT also achieves competitive results on both Java and Python. However, SiT is still more powerful on most metrics, which outperforms CoderBERT by 2.15, 0.95 and 1.15 points on BLEU, ROUGE-L and METEOR respectively on Java. However, CodeBERT performs much better on Python, which outperforms SiT by 1.00 and 0.58 points on ROUGE-L and METEOR. Note that CodeBERT is much bigger in size than Transformer and SiT (see Appendix A).

For further verification, we follow CodeBERT and conduct a RoBERTa-based (Liu et al., 2019) SiT to further fine-tune on both Java and Python. As shown in Table 3, pre-trained SiT obtains attractive results, further improving CodeBERT on all the metrics, which implies that our elaborate encoder design is still effective even under powerful pre-training assistance.

4 Ablation Study and Analysis

This section reports our ablation studies to valid our model on the dataset of Python-V2[4] (Barone and Sennrich, 2017), in which we conduct standard and unified pre-processing for strict fair comparison.

4.1 Si-SAN vs. SAN

To valid the effectiveness of Si-SAN, we gradually replace SAN layers in original Transformer with Si-SAN. Take Transformer model with Si-SAN proportion of 50% as an instance, we replace the second, fourth and last three encoder layers with Si-SAN and do not apply structure-induced module.

The results of variant models with incremental proportions of Si-SAN layers are shown in Table 4. Intuitively, all of the Transformers obtain improvements when equipped with Si-SAN layers. We can also see that SiT outperforms Transformer with similar proportion of Si-SAN, which proves the effectiveness of structure-induced module. However, it is surprising that Transformer with all 6 layers of Si-SAN still outperforms original Transformer even if it may be over-pruned.

Model	Prop.	BLEU	ROUGE-L	METEOR
Transformer	0	47.42	57.28	29.62
Transformer	50%	49.64	59.39	31.16
Transformer	100%	49.80	59.38	31.30
SiT	50%	**50.04**	**59.56**	**31.46**

Table 4: BLEU, ROUGE-L and METEOR for variant models with incremental proportions of Si-SAN.

Model	Attn.	BLEU	ROUGE-L	METEOR
Transformer	Full	47.42	57.28	29.62
Transformer	Window	49.28	58.80	30.90
Transformer	Random	38.06	57.28	22.76
Transformer	Struc.	**49.80**	**59.38**	**31.30**

Table 5: BLEU, ROUGE-L and METEOR for variant models with different attention patterns.

4.2 Si-SAN vs. Sparse SAN

To further valid our structure-based approach, we compare the performance of structure-induced attention with other sparse attention patterns, window attention in Longformer, ETC (Beltagy et al., 2020; Ainslie et al., 2020) and random attention in BigBird (Zaheer et al., 2020). We depict different attention patterns in Figure 5. The default sequence length in SiT is 400, and then we set both w and r to 64 in window and random attention respectively.

As shown in Table 5, Transformer with arbitrary sparse attention can not bring improvement as Si-

Model	BLEU	RE.-L	MTR.	SPEED
Transformer	44.87	54.95	26.58	1.0x
Transformer + SBT	43.34	53.97	25.02	1.5x
SiT-AST only	45.43	55.30	27.21	1.0x
SiT	**45.76**	**55.58**	**27.58**	**1.0x**

Table 6: Comparison of Si-SAN and SBT methods. Both methods only leverage AST information.

[4]https://github.com/EdinburghNLP/code-docstring-corpus/tree/master/V2

Model	Para.	BLEU	RE.L	MTR.
Transformer-8	140M	47.42	57.28	29.62
SiT-8	139M	50.04	59.56	31.46
Transformer-12	244M	50.11	59.47	31.44
SiT-12	242M	50.53	60.08	31.96
Transformer-16	370M	50.43	59.80	31.75
SiT-16	367M	**50.97**	**60.51**	**32.35**
Transformer-ALBERT enc.	124M	44.83	55.34	27.73
SiT-ALBERT enc.	124M	**49.31**	**58.46**	**30.83**

Table 7: BLEU, ROUGE-L and METEOR for variant models with different sizes, where RE.L and MTR. refer to ROUGE-L and METEOR respectively. Models like SiT-12 refers to SiT with 12 heads.

SAN, which refutes that SiT learns better through denoising. Specifically, random attention seriously deteriorates Transformer. It is surprising that window attention achieves a better result than Vanilla Transformer. Intuitively, tree structures like AST are highly localized. That is why window attention may show good performance. Nevertheless, Transformer with Si-SAN still outperforms window attention by 0.52 BLEU point.

4.3 Si-SAN vs. SBT

We reproduce SBT method on Java (Hu et al., 2018a) and apply it on our Transformer. For fair enough comparison, we let $\beta = \gamma = 0$ and conduct single-view SiT which only leverages AST information. As depicted in Figure 6, flattening ASTs into linear sequences does not result in improvement, which is consistent with Ahmad et al. (2020). However, we achieve substantial improvement while incorporating AST into Transformer using Si-SAN, which indicates our improved model design is indeed effective.

In addition, the average length of the input code will be much longer with SBT, which may introduce additional training cost. As shown in Figure 6, SiT is 1.5 times faster than Transformer with SBT.

4.4 Large Model

It is known that for nearly all deep models, increasing model size may cover quite much of model structure design improvement. Thus, it is possible that the improvement on base-size model may not work on large-size one. To valid this, we compare SiTs with Transformers under larger scale. As we can see pictorially in Table 7, with increasing parameter scale, SiTs with 12 heads and 16 heads both outperform the corresponding Transformers by 0.42 and 0.54 BLEU point respectively.

4.5 Parameter Sharing

Recently, parameter sharing on BERT (Devlin et al., 2019) has achieved promising results (Lan et al., 2020). Similar as ALBERT, we introduce cross-layer parameter sharing in both Transformer and SiT, sharing all parameters in all encoder layers. Note that we train our models from scratch and keep the decoder fixed.

As shown in Table 7, SiT performs much better on parameter sharing than Transformer does. We believe that code summarization task highly depends on structural information, and this is why SiT can still achieve good results with simply one group of encoder parameters while Transformer encounters a serious decline. On the other hand, it makes possible for lite model, which may balance high efficiency and performance.

5 Related Work

RNN-based Approaches While numbers of works (Haiduc et al., 2010; Eddy et al., 2013; Wong et al., 2013, 2015; Zhang et al., 2020a) on code summarization usually depended on information retrieval, most of the recent works tend to treat it as a machine translation problem. Meanwhile attention mechanism is broadly used for better performance on capturing long-range features. Allamanis et al. (2016) proposed a Convolution Neural Network (CNN) with copy attention, and more commonly, Iyer et al. (2016); Liang and Zhu (2018) proposed to use Recurrent Neural Network (RNN) with attention mechanism to summarize code snippets into natural language. Hu et al. (2018b) introduced API knowledge from related tasks while Cai et al. (2020) introduced type information to assist training, which also gained promising results. Additionally, reinforce learning (Wan et al., 2018) and dual learning (Wei et al., 2019; Ye et al., 2020) are also shown effective to boost model performance.

Transformer-based Approaches It is known that RNN-based models may encounter bottleneck when modeling long code sequences. Ahmad et al. (2020) proposed an enhanced Transformer with copy attention and relative position encoding while Gupta (2020); Dowdell and Zhang (2020) proposed to use Transformer (Vaswani et al., 2017) and Transformer-XL (Dai et al., 2019), all of which outperformed previous RNN-based models by a large margin.

Structure-based Approaches Recent works on code summarization pay more and more attention on structural information, which usually treats the source code in form of its Abstract Syntax Tree (AST). Hu et al. (2018a); LeClair et al. (2019); Uri et al. (2019) leveraged flattened ASTs as inputs and trained with LSTMs. Mou et al. (2016); Bui et al. (2021a); Shido et al. (2019); Harer et al. (2019) proposed TBCNN, TreeCaps, Tree-LSTM and Tree-Transformer to directly encode tree-style inputs. Differ from modeling code with sequential models, Allamanis et al. (2018); Liu et al. (2020); Alex et al. (2020) treated AST as graph and applied graph neural network, while Wang et al. (2021) applied heterogeneous graph neural network to model different types of nodes.

Pre-training Approaches Apart from training from scratch, CodeBERT (Feng et al., 2020) is pre-trained on vast bimodal corpora with masked language model (Devlin et al., 2019) and replaced token detection (Clark et al., 2020), and achieves powerful performances on downstream tasks. Nie et al. (2020) intensified contextualized code representation through masked code fragment predictions while Bui et al. (2021b) incorporated structural information using TBCNN. However, all of them do not include generation-related objectives. It is worth further exploration and practice on pre-training approaches for out concerned tasks.

6 Conclusion

This paper presents a novel structured-induced Transformer model on code summarization task. By well-designed architecture, the proposed model may effectively incorporate multi-view structure into attention mechanism without tricky implementation. We further adopt a new module architecture to aggregate both global self-attention and structure-induced self-attention representations. Experiments on two challenging benchmarks including Java and Python show that the proposed model yields new state-of-the-art results.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Joshua Ainslie, Santiago Ontañón, Chris Alberti, Philip Pham, Anirudh Ravula, and Sumit Sanghai. 2020. ETC: encoding long and structured data in transformers. *CoRR*, abs/2004.08483.

LeClair Alex, Haque Sakib, Wu Lingfei, and McMillan Collin. 2020. Improved code summarization via a graph neural network. In *2020 IEEE/ACM International Conference on Program Comprehension*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2091–2100, New York, New York, USA. PMLR.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017, Volume 2: Short Papers*, pages 314–319. Asian Federation of Natural Language Processing.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *CoRR*, abs/2004.05150.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021a. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021b. Infercode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*.

Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG : Type auxiliary guiding for code comment generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 291–301. Association for Computational Linguistics.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning

phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: pre-training text encoders as discriminators rather than generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Thomas Dowdell and Hongyu Zhang. 2020. Language modelling for source code with transformer-xl. *CoRR*, abs/2007.15813.

Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 13–22. IEEE.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Vivek Gupta. 2020. Deepsumm - deep code summaries using neural transformer architecture. *CoRR*, abs/2004.00998.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Abhishek Kumar, Piyush Rai, and Hal Daumé III. 2011. Co-regularized multi-view spectral clustering. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 1413–1421.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A lite BERT for self-supervised learning of language representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 795–806. IEEE / ACM.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second*

AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pages 5229–5236. AAAI Press.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Automatic code summarization via multi-dimensional semantic fusing in gnn. *arXiv preprint arXiv:2006.05405*.

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.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Shashi Narayan, Joshua Maynez, Jakub Adámek, Daniele Pighin, Blaz Bratanic, and Ryan T. McDonald. 2020. Stepwise extractive summarization and planning with structured transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 4143–4159. Association for Computational Linguistics.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2020. Contextualized code representation learning for commit message generation. *CoRR*, abs/2007.06934.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1073–1083. Association for Computational Linguistics.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Yu Shi, Fangqiu Han, Xinran He, Carl Yang, Jie Luo, and Jiawei Han. 2018. mvn2vec: Preservation and collaboration in multi-view network embedding. *CoRR*, abs/1801.06597.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Vikas Sindhwani, Partha Niyogi, and Mikhail Belkin. 2005. A co-regularization approach to semi-supervised learning with multiple views. In *Proceedings of ICML workshop on learning with multiple views*, volume 2005, pages 74–79. Citeseer.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2021. Learning to represent programs with heterogeneous graphs.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems*, pages 6563–6573.

Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389. IEEE.

Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Not all attention is all you need. *CoRR*, abs/2104.04692.

Jiacheng Xu, Zhe Gan, Yu Cheng, and Jingjing Liu. 2020. Discourse-aware neural extractive text summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5021–5031, Online. Association for Computational Linguistics.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 590–604. IEEE Computer Society.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *WWW '20: The Web*

Conference 2020, Taipei, Taiwan, April 20-24, 2020, pages 2309–2319. ACM / IW3C2.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020a. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1385–1397, New York, NY, USA. Association for Computing Machinery.

Zhuosheng Zhang, Yuwei Wu, Junru Zhou, Sufeng Duan, Hai Zhao, and Rui Wang. 2020b. Sg-net: Syntax-guided machine reading comprehension. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 9636–9643. AAAI Press.

Dengyong Zhou and Christopher J. C. Burges. 2007. Spectral clustering and transductive learning with multiple views. In *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*, volume 227 of *ACM International Conference Proceeding Series*, pages 1159–1166. ACM.

A Model Parameters

Model	d_h	d_{ff}	h	l
Transformer	64	2048	8	12
SiT	64	2048	8	12
Transformer-Window	64	2048	8	12
Transformer-Random	64	2048	8	12
Transformer-Struc.	64	2048	8	12
CodeBERT	64	3072	12	12
SiT on CodeBERT	64	3072	12	12
Transformer-ALBERT enc.	64	2048	8	12
SiT-ALBERT enc.	64	2048	8	12

Table 8: Model parameters in our experiments.

B Qualitative Samples

For qualitative analysis, we give some samples of code summarization with different models. We can see that SiT performs most precisely, while CodeBERT performs better than Transformer does.

Code	private float computeOverscrollPercent () {
	if (mOverScrollOffset >= _NUM) {return mOverScrollOffset / mMaxOverScroll;}
	else {return mOverScrollOffset / mMaxUnderScroll;}
	}

Summary	Gold: determine the current amount of overscroll . if the value is 0 , there is no overscroll . if the value is < 0 , tabs are overscrolling towards the top or or left . if the value is > 0 , tabs are overscrolling towards the bottom or right . SiT: determine the current amount of overscroll . if the value is 0 , there is no overscroll . if the value is < 0 , tabs are overscrolling towards the top or or left . if the value is > 0 , tabs are overscrolling towards the bottom or right . Transformer: determine the current amount of overscroll . if the value is 0 , there is no overscroll . if the value is < 0 , tabs are overscrolling towards the top or or left . if the value is > 0 , tabs are overscrolling towards the top or right . CodeBERT: determine the current amount of overscroll . if the value is 0 , there is no overscroll . if the value is < 0 , tabs are overscrolling towards the top or or left . if the value is > 0 , tabs are overscrolling towards the bottom or right .

Code	public String peek () {
	String result = null;
	if (isEmpty()) {return null;}
	else {
	int cachedCurrentIndex = currentIndex;
	if (isEatingBlocksOfDelimiters) {trimStartingDelimiters();}
	int nearestDelimeter = -_NUM ;
	for (int i = _NUM; i <delimiters.length(); i++) {
	int delimiter = source.indexOf(delimiters.charAt(i), currentIndex);
	if (nearestDelimeter == -_NUM || delimiter != -_NUM && delimiter <nearestDelimeter) {
	nearestDelimeter = delimiter;
	}
	}
	if (nearestDelimeter == -_NUM) {result = source.substring(currentIndex);}
	else {result = source.substring(currentIndex, nearestDelimeter);}
	currentIndex = cachedCurrentIndex;
	}
	return result;
	}

Summary	Gold: returns null if there is nothing left . SiT: returns null if there is nothing left . Transformer: finds the next unique identifier . CodeBERT: returns the index of the first delimited string removing from the current position .

Table 9: Qualitative samples of Java code summarization.

Code	def _asFilesystemBytes(path, encoding=None):
	if (type(path) == bytes): return path
	else:
	if (encoding is None):
	encoding = sys.getfilesystemencoding()
	return path.encode(encoding)

Summ.	Gold: return cpath as a string of lbytes suitable for use on this systems filesystem . SiT: return cpath as a string of lunicode suitable for use on this systems filesystem . Transformer: convert a filesystem path of a byte string . CodeBERT: return a byte string suitable for use in cpath as a byte string .

Code	def absent(name, DomainName, region=None, key=None, keyid=None, profile=None):
	ret = {'name': DomainName, 'result': True, 'comment': '', 'changes': {}}
	r = __salt__['boto_elasticsearch_domain.exists'](DomainName, region=region, key=key, keyid=keyid, profile=profile)
	if ('error' in r):
	ret['result'] = False
	ret['comment'] = 'Failed to delete domain: {0}.'.format(r['error']['message'])
	return ret
	if (r and (not r['exists'])):
	ret['comment'] = 'Domain {0} does not exist.'.format(DomainName)
	return ret
	if __opts__['test']:
	ret['comment'] = 'Domain {0} is set to be removed.'.format(DomainName)
	ret['result'] = None
	return ret
	r = __salt__['boto_elasticsearch_domain.delete'](DomainName, region=region, key=key, keyid=keyid, profile=profile)
	if (not r['deleted']):
	ret['result'] = False
	ret['comment'] = 'Failed to delete domain: {0}.'.format(r['error']['message'])
	return ret
	ret['changes']['old'] = {'domain': DomainName}
	ret['changes']['new'] = {'domain': None}
	ret['comment'] = 'Domain {0} deleted.'.format(DomainName)
	return ret

Summ.	Gold: ensure domain with passed properties is absent . SiT: ensure domain with passed properties is absent . Transformer: ensure the iam role exists . CodeBERT: ensure the named domain is absent .

Table 10: Qualitative samples of Python code summarization.