Understanding Long Programming Languages with Structure-Aware Sparse Attention

Tingting Liu  
East China Normal University  
Shanghai, China  
ttlu@stu.ecnu.edu.cn

Chengyu Wang  
Alibaba Group  
Hangzhou, Zhejiang, China  
chengyu.wcy@alibaba-inc.com

Cen Chen  
East China Normal University  
Shanghai, China  
cenchen@dase.ecnu.edu.cn

Ming Gao∗  
East China Normal University  
Shanghai, China  
mgao@dase.ecnu.edu.cn

Aoying Zhou  
East China Normal University  
Shanghai, China  
ayzhou@dase.ecnu.edu.cn

ABSTRACT
Programming-based Pre-trained Language Models (PPLMs) such as CodeBERT have achieved great success in many downstream code-related tasks. Since the memory and computational complexity of self-attention in the Transformer grow quadratically with the sequence length, PPLMs typically limit the code length to 512. However, codes in real-world applications are generally long, such as code searches, which cannot be processed efficiently by existing PPLMs. To solve this problem, in this paper, we present SASA, a Structure-Aware Sparse Attention mechanism, which reduces the complexity and improves performance for long code understanding tasks. The key components in SASA are top-k sparse attention and Abstract Syntax Tree (AST)-based structure-aware attention. With top-k sparse attention, the most crucial attention relation can be obtained with a lower computational cost. As the code structure represents the logic of the code statements, which is a complement to the code sequence characteristics, we further introduce AST structures into attention. Extensive experiments on CodeXGLUE tasks show that SASA achieves better performance than the competing baselines.

CCS CONCEPTS
• Computing methodologies → Natural language processing.

KEYWORDS
Programming-based Pre-trained Language Models, Structure-Aware Sparse Attention, Abstract Syntax Tree

1 INTRODUCTION
Transformer-based pre-trained language models such as BERT [6], RoBERTa [15], and ELMo [17] have achieved great success in Natural Language Processing (NLP) tasks. These models are usually pre-trained on a large amount of unlabeled data and fine-tuned on downstream tasks. Similarly, existing Program-based Pre-trained Language Models (PPLMs) such as CodeBERT [7], GraphCodeBERT [9] and PLBART [1] have significantly improved the performance of code-related tasks such as code clone detection [19], code search [10], and code summarization [11].

Despite the success, since the memory and computational complexity of self-attention in the Transformer grow quadratically with the sequence length, PPLMs typically limit the code length to 512. Although 512 is suitable for some tasks, codes in industrial scenarios are generally long and cannot be processed efficiently by existing PPLMs. There are two solutions to this case. One is to truncate the long sequence to 512, which undoubtedly has information loss. The other is to process the full sequence, but this leads to the $O(n^2)$ complexity in both time and memory consumption w.r.t. the sequence length $n$. Recently, several approaches have been developed to process long sequences efficiently in NLP. For example, sparse attention methods [4, 18, 24] sparsify the attention matrix by limiting the attention field to a fixed range. Therefore, these methods can handle longer sequences using similar hardware. However, even though programs and natural languages are both token sequences, we observe that these sparse attention methods do not work well for long codes, which have different syntactical and structural features from natural languages.

In this paper, we present SASA, a Structure-Aware Sparse Attention mechanism, which reduces the complexity and improves performance for long code understanding tasks. SASA consists of two key components, i.e., top-k sparse attention and Abstract Syntax Tree (AST)-based structure-aware attention. Specifically, top-k sparse attention allows each token to attend to only $k$ tokens with the highest attention scores. As in Fig. 1(c), we counted the attention scores for token pairs on the CodeSearchNet [10] dataset and found that the attention scores presented a “long-tail” phenomenon,
public int getLength(ListNode head) {
    int length = 0;
    while (head != null) {
        ++length;
        head = head.next;
    }
    for(…) { … }
    return length; }

Figure 1: Attention patterns in the pre-trained model. The results show that the attention scores among tokens in long codes present a “long-tail” phenomenon.

Figure 2: An example of the code and its corresponding AST. Some nodes have been omitted for simplicity. The distance between two nodes of “length” in AST is not directly related to code length but to the code structure.

i.e. the attention interactions between token pairs are sparse. By means of top-k sparse attention, calculations of self-attention can be simplified, leading to lower computation overhead.

Furthermore, the code structure is an important feature in addition to the sequence characteristics of the code. For example, Abstract Syntax Tree (AST) is a tree that represents the abstract syntactic structure of the source code, leaving out unimportant details such as commas, parentheses. Existing works [21, 25] demonstrate that AST-based models have made significant progress in code-related tasks. Another advantage of AST for long code is that the distance between nodes in AST is not directly related to code length but to the depth of the code structure. As shown in Fig. 2, the distance between the two nodes of “length” is 6 (blue path in AST). When the “For Statement” is inserted after the “While Statement”, the distance between these two nodes in the sequence becomes larger, but remains constant in the AST. Hence, we introduce the AST structures into attention, which explicitly establishes connections to structure-dependent token pairs.

We conducted extensive experiments on four CodeXGLUE [16] tasks to examine the performance of the proposed SASA model. Results show that SASA outperforms the competing methods in long code tasks. Meanwhile, the SASA-based method also has comparable or better results in full datasets containing mostly short codes. Furthermore, SASA can save 23% to 33% memory with sparse self-attention.

2 RELATED WORK

In this section, we summarize the related works on two aspects: PPLMs and sparse transformers.

2.1 Program-based Pre-trained Models

Large PPLMs [7, 9, 12, 13, 22] have significantly improved the performance of code-related downstream tasks. These methods are generally trained on bi-modal data (program languages and natural languages) in a self-supervised manner. SCELMo [13] trains ELMo [17] on corpora of source codes. CuBERT [12] and CodeBERT [7] follow the architecture of BERT [6] and pre-trained by the Masked Language Modeling (MLM) task on CodeSearchNet [10] corpus, learning the context-aware representations of codes. CodeT5 [22] is a unified pre-trained encoder-decoder model for code understanding and generation tasks. All of these models treat source codes as token sequences and pay little attention to the fact that code structures contain crucial code semantics. Specifically, Abstract Syntax Tree (AST) represents the syntax structure of codes as a tree. Data Flow Graph (DFG) is a graph representation of the transfer between variables. Some task-specific methods [21, 23, 25] use these structures to empower code representations. GraphCodeBERT [9] also introduces the edge prediction task to learn representations from the data flow. Most PPLMs are based on the Transformer architecture, and the computational and memory complexity of the self-attention mechanism in the original Transformer is $O(n^2)$ (with $n$ to be the sequence length). Therefore, it is not desirable to directly apply existing PPLMs to long code-related tasks.

2.2 Sparse Transformer

There have been several studies [2, 4, 5, 18, 24] that process long sequences in NLP efficiently. BlockBERT [18] divides the attention matrix into $k$ blocks and defines attention on each block, reducing the computational and memory cost to $O(n^2/k)$. Sparse
Transformers [4] and Longformer [2] employ sliding windows and global tokens to combine local and global information of input sequences. BigBird [24] extends random attention on top of Sparse Transformer. These models achieve time and memory savings without significant performance degradation but are not designed for code-related tasks. In this paper, we study processing long codes efficiently and making full use of code structures.

3 PROPOSED FRAMEWORK

In this section, we elaborate our approach in detail. Fig. 3 presents an overview of the proposed framework. We use CodeBERT as the backbone and improve the self-attention to process long and structured code based on three motivations: (1) Due to the high complexity of the self-attention module, CodeBERT cannot handle long codes well with truncated sequences. (2) According to our experimental observations and conclusions of existing works [3, 14], the calculation of self-attention is redundant. (3) Structure-based representation can complement the sequential semantics of code and improve the performance on downstream tasks [8, 23]. We propose a sparse attention mechanism named SASA, which contains four attention patterns, namely sliding window, global attention, top-k and AST-based attention pattern.

3.1 Sliding Window and Global Attention

As shown in Fig. 1, similar to [14], we observe that the attention matrix is very sparse and has some fixed patterns, such as "vertical" (for [CLS] or [SEP]) and "diagonal" (for local neighbors). Following the BigBird [24] architecture, we introduce global tokens and sliding window, corresponding to the vertical and diagonal pattern respectively. The computation complexity of the sliding window attention is $O(n \times w)$ with $w << n$ where $n$ is the sequence length and $w$ is the window size. Since only a small number of tokens are global tokens, the overall computation complexity of the sliding window and global attention pattern increases linearly with the sequence length. As modern hardware (such as GPU and TPU) is more suitable for parallel computation, discrete attention computation for sliding window, top-k and AST-aware attention cannot make use of their computational advantages. We follow BigBird [24] to divide the attention matrix into blocks and define attention calculations in each block.

Assume that $Q, K \in \mathbb{R}^{n \times d}$ are query and key matrices, where $d$ is the dimension of a token vector. With the block size $b$, we reshape $Q$ and $K$ into $Q'$ and $K'$ with the shape of $\lfloor n/b \rfloor \times b \times d$. The attention matrix of sliding window attention pattern is computed as:

$$A_{ij}^{bst} = \sum_u Q'_{iu} K'_{ju}$$

where $A' \in \mathbb{R}^{\lfloor n/b \rfloor \times \lfloor n/b \rfloor \times b \times b}$, $Q'_{iu}$ is the representation of the $i$-th query block and $K'_{ju}$ is the representation of the $j$-th key block.

For global attention pattern, we specify a set of key blocks as global blocks, denoting as $g$. $A_{ij}^{gst}$ is the attention score of the $i$-th query block and the $j$-th key block. Since every query block attends to global blocks, and global attention is symmetric, the attention calculation is performed when $i$-th query block belongs to $g$ or $j$-th key block belongs to $g$.

3.2 Top-k Sparse Attention

As shown in Fig. 1(a) and Fig. 1(b), except for the attention pattern of local and global, there are still some scattered parts with high attention scores, which is an effective linguistic feature acquired through pre-training of the model. In the trade-off between efficiency and performance, we design a top-$k$ sparse attention pattern that each token attends to the $k$ tokens with the highest attention scores. The computation complexity of this pattern is $O(n \times k)$, which scales linearly with the sequence length.

To speed up the fine-tuning process, we pre-process the CodeSearchNet corpus and obtain an attention frequency matrix $P$ of size $|V| \times |V|$ where $|V|$ is the vocabulary size. If the attention score of a token pair is greater than 0.1 (accounting for more than 10% of the total attention score), the frequency is increased by 1. A larger value of $P_{ij}$ means more attention interactions between the $i$-th token and the $j$-th token. During training, the top-$k$ attention matrix $P \in \mathbb{R}^{n \times n}$ for each input is obtained by looking up the matrix $P$, and then divided into blocks, denoting as $P' \in \mathbb{R}^{\lfloor n/b \rfloor \times \lfloor n/b \rfloor \times b \times b}$. The attention frequency of each block is the sum of all the values in the block, that is, add the matrix $P'$ along with the last 2 dimensions.
Algorithm 1 Structure-Aware Sparse Attention (SASA)

Input: $D = \{D_1, \ldots, D_h\}$ where $D_h = \{c_{t1}, \ldots, c_{tn}\}$, the train data of each task, $c_t$ is the code tokens of the sample $D_t$, $\overline{P} \in \mathbb{R}^{|V| \times |V|}, \overline{T} \in \mathbb{R}^{|V| \times |V|}$, the preprocessed top-$k$ frequency matrix and the AST matrix.

Output: $A \in \mathbb{R}^{[n/b] \times [n/b] \times b \times b}$, the block-based sparse attention matrix.

1. Get the input embedding $X_t$ by the embedding layer of SASA-based model.
2. Reshape the query and key matrix $Q = X_tW_Q, K = X_tW_K \in \mathbb{R}^{n \times d}$ into $Q', K' \in \mathbb{R}^{[n/b] \times b \times d}$.
3. Local pattern $\leftarrow \{(i,j)| i-j \leq \lceil \frac{n}{b} \rceil \}$.
4. Global pattern $\leftarrow \{(i,j)| i \in g \ or \ j \in g\}$, $g$ is the global blocks.
5. Top-$k$ pattern
6. Get top-$k$ attention matrix $P \in \mathbb{R}^{n \times n}$ for the input by looking up the matrix $P$.
7. Divide the matrix $P$ into blocks to get $P' \in \mathbb{R}^{[n/b] \times [n/b]}$.
8. Top-$k$ pattern $\leftarrow \{(i,j) \in \text{topk}(\sum_{s,t} P'_{ijst}k), \ i,j = 0, \ldots, \lceil \frac{n}{b} \rceil - 1\}$.
9. AST pattern $\leftarrow \{(i,j) \in \text{topk}(\sum_{s,t} T'_{ijst}k)\}$.
10. Compute the block-based attention matrix for the input.
11. $A_{ijst} \leftarrow \sum_{u} Q'_{isu}K'_{jut}$, if $(i,j)$ belongs to at least one of the above four patterns, otherwise, $A_{ijst} \leftarrow 0$.

The attention matrix of top-$k$ attention pattern is computed as:

$$A^k_{ijst} = \begin{cases} \sum_{u} Q'_{isu}K'_{jut} & \text{if } (i,j) \in \text{topk}(\sum_{s,t} P'_{ijst}k) \\ 0 & \text{otherwise} \end{cases}$$

where $A^k \in \mathbb{R}^{[n/b] \times [n/b] \times b \times b}$. For each query block, we use topk function to select the $k$ key blocks with the highest attention score for calculation.

3.3 AST-aware Structure Attention

All of the above attention patterns treat codes as sequences. Besides the sequence characteristics, the structural features of codes are equally important. As shown in Fig. 2, The AST distance represents the structural characteristics between nodes.

Specifically, we parse the code to AST by tree-sitter\(^2\). To introduce AST into the attention module, we transform the tree structure into an adjacency matrix. Each non-zero value in the matrix indicates that the corresponding token pair has a structural connection. Similar to top-$k$ attention pattern, we divide the adjacency matrix into blocks, resulting in a matrix $T' \in \mathbb{R}^{[n/b] \times [n/b] \times b \times b}$. The attention matrix of AST-aware structure attention is computed as:

$$A^T_{ijst} = \begin{cases} \sum_{u} Q'_{isu}K'_{jut} & \text{if } (i,j) \in \text{topk}(\sum_{s,t} T'_{ijst}k) \\ 0 & \text{otherwise} \end{cases}$$

where $A^T \in \mathbb{R}^{[n/b] \times [n/b] \times b \times b}$.

3.4 Model Summary

Based on the CodeBERT pre-trained model, we replace self-attention with SASA and fine-tune it on downstream tasks. We have four patterns in SASA, and each pattern has an attention matrix. As shown in Algorithm 1, each query block in SASA attends to $w$ sliding window blocks, $g$ global blocks, $k$ top-$k$ blocks and AST blocks with the total cost of $O(n(w + g + k)b)$.

4 EXPERIMENTS

4.1 Experimental Settings

4.1.1 Tasks. To evaluate the effectiveness of our proposed SASA model, we provide results and ablations on four code understanding tasks as follows: (1) Clone detection is a binary classification task for code pairs, where 1 stands for semantic equivalence and 0 for others. (2) Defect detection (Defect for short) task aims to identify whether a program may attack a software system. (3) Search task (Sum for short) task is to generate natural language comments for a given code. We use the datasets provided by CodeXGLUE \[16\] and select the code with the length greater than 1024 as the long datasets.

4.1.2 Baselines. We compare SASA with several pre-trained models. Roberta \[15\] is a multi-layer bidirectional transformer encoder. CodeBERT \[7\] uses Roberta \[15\] as the backbone and is continuously pre-trained on codes from CodeSearchNet \[10\]. GraphCodeBERT \[9\] follows BERT [6] architecture and is pre-trained on CodeSearchNet corpus with an edge prediction task. Longformer [2] and Bigbird [24] are a sparse attention model with local and global attention patterns that works well with long sequences in NLP.

4.1.3 Model Configuration. We follow CodeBERT \[7\] as the model backbone, the difference being that we set the sequence length to 1024 instead of 512. In addition, we set $w = 3, g = 2, k = 3, b = 32$ for all experiments. The other hyperparameters follow the same setting provided by CodeXGLUE \[16\].

4.2 Overall Performance

Table 1 shows the overall performance on four long code datasets. We can see that: (1) The performance of models that truncate long sequences, such as Roberta-base, CodeBERT, GraphCodeBERT, is greatly degraded. (2) GraphCodeBERT works better than CodeBERT, suggesting that the structural properties of code are important in understanding long codes. (3) Sparse attention models that are effective for long sequences in NLP cannot be directly used for long code tasks, but Longformer and BigBird outperform CodeBERT by loading the pre-training weights of CodeBERT. (4) The SASA model exploits both the advantages of sparse attention mechanism and code structures, and has competitive performance compared with state-of-the-art models.

4.3 Ablation Study

We conduct an ablation study on two tasks to examine the effect of top-$k$ and AST attentions, as shown in Table 2. Without top-$k$
Table 1: Main results for long codes. The best results are in bold font, and the second best are underlined.

| Method          | BigCloneBench | Defect Detection | Code Search | Code Summarization |
|-----------------|---------------|------------------|-------------|--------------------|
|                 | Precision     | Recall           | F1 Score    | Accuracy (%)       | MRR      | BLEU-4 |
| Roberta-base     | 0.613         | 0.831            | 0.706       | 52.59              | 25.60    | 9.17   |
| CodeBERT        | 0.718         | 0.874            | 0.789       | 55.71              | 38.37    | 12.26  |
| GraphCodeBERT   | 0.852         | 0.891            | 0.871       | 55.39              | 43.13    | 12.93  |
| Longformer      | 0.752         | 0.901            | 0.820       | 56.90              | 44.09    | 12.20  |
| BigBird         | 0.892         | 0.898            | 0.895       | 56.79              | 41.89    | 12.42  |
| SASA            | 0.906         | 0.917            | 0.911       | 57.44              | 46.36    | 12.88  |

Table 2: Ablation study of SASA.

| Method       | BigCloneBench | Defect Detection |
|--------------|---------------|------------------|
|              | Pre.          | Rec.             | F1          | Accuracy (%) |
| SASA         | 0.906         | 0.917            | 0.911       | 57.44        |
| w/o. top-k   | 0.753         | 0.919            | 0.828       | 56.84        |
| w/o. AST     | 0.830         | 0.922            | 0.874       | 57.22        |

Figure 4: Memory used by the model. The memory consumption of larger batch sizes for codeBERT is not shown due to out-of-memory (OOM) error.

4.4 Memory Analysis

The sparse attention mechanism reduces computation and memory complexity compared to full self-attention. Fig. 4 shows the memory usage of CodeBERT and SASA. When batch size is 8, SASA saves more than 7GB of memory compared to CodeBERT. When batch size is greater than 10, CodeBERT has an out-of-memory problem on a 32GB V100 machine.

5 CONCLUSION

In this paper, we present SASA, a Structure-Aware Sparse Attention mechanism, which reduces the complexity and improves performance for long code understanding tasks. SASA consists of two novel components, which are top-k sparse attention and Abstract Syntax Tree (AST)-based structure-aware attention. The former simplifies the complexity of self-attention while capturing the rich sequential context. As a complement, AST attention explicitly builds the structural link between tokens. SASA achieves performance improvements and resource-saving in multiple tasks. In industrial scenarios, SASA can be used to generate code comments and retrieve code based on natural language queries.

ACKNOWLEDGMENTS

This work has been supported by the National Natural Science Foundation of China under Grant Nos. U1911203 and 61877018, and Alibaba Group through the Alibaba Innovation Research Program.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021. Association for Computational Linguistics, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Ir Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. CoRR abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[3] Yuchen Bian, Jiaji Huang, Xingyu Cai, Jiahong Yuan, and Kenneth Church. 2021. On Attention Redundancy: A Comprehensive Study. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Ir Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, 930–945. https://doi.org/10.18653/v1/2021.naacl-main.72

[4] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. CoRR abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[5] Zihang Dai, Zhilin Yang, Yiming Yang, Jiae 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. Association for Computational Linguistics, 2978–2988. https://doi.org/10.18653/v1/p19-1285

[6] 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). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423
