Tackling Long Code Search with Splitting, Encoding, and Aggregating

Fan Hu[1*], **Yanlin Wang**[2†‡], **Lun Du**[3],
Hongyu Zhang[4], **Shi Han**[3], **Dongmei Zhang**[3], **Xirong Li**[1]
[1] Renmin University of China
[2] School of Software Engineering, Sun Yat-sen University
[3] Microsoft
[4] Chongqing University

Abstract

Code search with natural language helps us reuse existing code snippets. Thanks to the Transformer-based pretraining models, the performance of code search has been improved significantly. However, due to the quadratic complexity of multi-head self-attention, there is a limit on the input token length. For efficient training on standard GPUs like V100, existing pretrained code models, including GraphCodeBERT, CodeBERT, RoBERTa (code), take the first 256 tokens by default, which makes them unable to represent the complete information of *long code* that is greater than 256 tokens. To tackle the long code problem, we propose a new baseline SEA (Split, Encode and Aggregate), which splits long code into code blocks, encodes these blocks into embeddings, and aggregates them to obtain a comprehensive long code representation. With SEA, we could directly use Transformer-based pretraining models to model long code without changing their internal structure and re-pretraining. We also compare SEA with sparse Trasnformer methods. With GraphCodeBERT as the encoder, SEA achieves an overall mean reciprocal ranking score of 0.785, which is 10.1% higher than GraphCodeBERT on the CodeSearchNet benchmark, justifying SEA as a strong baseline for long code search.

Keywords: code search, long code understanding, code representation

1. Introduction

A good code search technique helps developers to boost software development by searching for code snippets using natural language. Recent advancements have demonstrated the effectiveness of Transformer-based code pre-training methods, including CodeBERT (Feng et al., 2020), CoCLR (Huang et al., 2021), and GraphCodeBERT (Guo et al., 2021), which have significantly improved code search performance through self-supervised pre-training on large-scale code corpus.

However, these approaches face an inherent limitation. The computational and memory complexity of self-attention in the original Transformer grows quadratically with the input length, imposing a constraint on the input length of approximately 512 tokens. For efficient training on standard GPUs like V100, GraphCodeBERT and CodeBERT consider only the first 256 tokens of code snippets and discard any tokens beyond this limit. Nonetheless, this length restriction can lead to accuracy issues, especially for long code snippets. For instance, when examining the challenging cases of GraphCodeBERT, we found that GraphCodeBERT has low performance for some long code snippets where crucial information resides towards the end. As illustrated in Figure 1, the keywords "Tensor" and "patches" appear after the 256-token cutoff set by GraphCodeBERT, resulting in their exclusion from consideration. Consequently, the corresponding code snippet is ranked at position 21,148.

We further conducted empirical studies on GraphCodeBERT in publicly used CodeSearchNet dataset (Husain et al., 2019), and observed a gradual decrease in search performance as the length of the ground-truth code in the query increased (refer to Table 1). This issue is similar to the long text problem in natural language processing, for which various approaches have been proposed, including hierarchical processing (Zhang et al., 2019b), sparse attention (Child et al., 2019; Beltagy et al., 2020), and segment-level recurrence (Dai et al., 2019). However, directly applying these methods to long code presents two challenges. Firstly, these techniques modify the internal structure of the Transformer model, potentially rendering the existing pre-training parameters invalid. Secondly, long code differs from long text in that it is a highly structured language. Unlike a long text document that can be treated as a cohesive whole with complete semantics, the semantics of code are discontinuous, and different functions are distributed across various locations. The comparison experiments conducted in Section 6.2 provide evidence

* Work done during internship at Microsoft Research Asia.
† The corresponding author. Contact: Yanlin Wang (wangylin36@mail.sysu.edu.cn).
‡ Work done at Microsoft Research Asia.

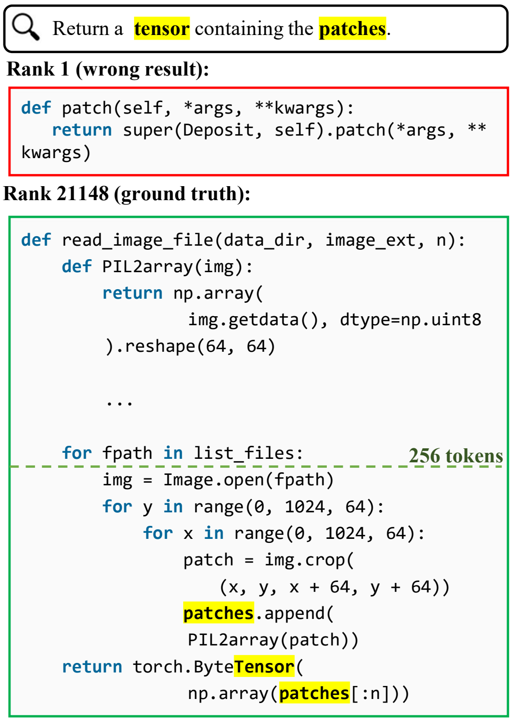


Figure 1: Example case of GraphCodeBERT. GraphCodeBERT truncates tokens beyond 256 tokens. Key tokens are highlighted in yellow.

supporting these concerns.

Therefore, our goal is to divide long code while preserving its semantic information. We aim to achieve this without altering the internal structure of Transformer-based pretraining models or requiring re-pretraining. To address this, we propose SEA (Split, Encode, and Aggregate) to handle long code and obtain improved code representations.

As depicted in Figure 2, the process involves splitting the long code into a set of code pieces, followed by utilizing the sliding window method to generate a partially overlapping code block set. Existing code encoders are then used to obtain embeddings for each code block. Finally, these embeddings are aggregated to generate representations for the entire long code. Through extensive experiments, we have found that the proposed AST-based splitting method and attention-based aggregation method outperform other techniques for splitting and aggregation. Due to the varying numbers of code blocks obtained from different code snippets, parallel operation becomes challenging. To address this problem, we have designed a combine-divide module for acceleration. It is important to note that SEA is encoder-agnostic, meaning it can be used with different Transformer-based encoders. When compared to various Transformer-based encoder baselines, SEA achieves a significant improvement in mean reciprocal ranking (MRR) performance, ranging from 7% to 10%.

Table 1: The code search performance (MRR) of different ground-truth code token lengths. We set the code truncation length from 50 to 512. The highest results in each column are highlighted. Dataset: CodeSearchNet python. Model: GraphCodeBERT.

Token length	Code truncation length				
	50	100	256	400	512
[0, 256)	**0.6274**	0.6856	0.6909	0.6897	0.6906
[256, 512)	0.6239	**0.7027**	**0.7237**	0.7258	0.7265
[512, 768)	0.6004	0.6467	0.7168	0.7180	0.7181
[768, 1024)	0.6038	0.6315	0.7111	**0.7375**	**0.7276**
[1024, 1943)	0.6202	0.6573	0.6589	0.6835	0.6825

The contributions can be summarized as:

- Empirical finding and verification of the difficulty for modeling long code in existing Transformer-based code search models.
- We propose a new baseline SEA and explore an optimal splitting and aggregation setting. We also design a combine-divide module for acceleration.
- Through extensive experiments, we show the effectiveness of the proposed SEA with different encoder baselines in six programming languages, resulting in a strong baseline for code search. Our source code and experimental data are available at: https://github.com/fly-dragon211/SEA.

2. Related Work

2.1. Code Search Methods

Early studies (Nie et al., 2016; Yang and Huang, 2017; Rosario, 2000; Hill et al., 2011; Satter and Sakib, 2016; Lv et al., 2015; Van Nguyen et al., 2017) in code search mainly applied information retrieval (IR) techniques directly, treating code search as a text matching task. Both queries and code snippets were considered plain text, and traditional text matching algorithms such as bag-of-words (BOW) (Schütze et al., 2008), Jaccard (Jaccard, 1901), term frequency-inverse document frequency (TF-IDF) (Robertson and Jones, 1976), BM25 (an improved version of TF-IDF) (Robertson and Zaragoza, 2009), and the extended boolean model (Lv et al., 2015) were employed. Since code length has minimal impact on modeling complexity, these methods could encode long code without truncation.

Following the introduction of the large-scale pre-training model BERT (Devlin et al., 2019), CodeBERT was proposed by Feng et al. (2020). CodeBERT is a model pre-trained on unlabeled source code and comments, which achieved impressive performance in text-based code search through fine-tuning on text-code paired datasets. Huang et al. (2021) introduced CoCLR, a contrastive learning method that enhances query-code matching. Sun et al. (2022) developed a context-aware code translation technique that translates code snippets into natural language descriptions. Gu et al. (2022) utilized deep hashing and code classification to accelerate code search, while Chai et al. (2022) adapted few-shot meta-learning to code search. Guo et al. (2021) proposed GraphCodeBERT, incorporating structure-aware pre-training tasks to improve code understanding and performance. Recently, Hu et al. (2023) utilized a two-stage fusion code search framework that combines bi-encoders and cross-encoders to enhance performance. However, the computational complexity of Transformers and limited GPU memory often lead to the truncation of long code snippets.

2.2. Neural Code Representation with Code Structure

Recently, there have been notable advancements in neural code representation methods that leverage code structure, particularly Abstract Syntax Trees (AST), yielding impressive performance (Alon et al., 2020; Sun et al., 2020; Bui et al., 2021; Kim et al., 2021; Peng et al., 2021; Hellendoorn et al., 2019; Allamanis et al., 2021; Georgiev et al., 2022; Ma et al., 2023; Du and Yu, 2023). MMAN (Wan et al., 2019) incorporates a multi-modal attention fusion layer to combine AST and Control Flow Graph (CFG) representations. ASTNN (Zhang et al., 2019a) and CAST (Shi et al., 2021) segment large ASTs into sequences of smaller statement trees, encoding them into vectors by capturing the lexical and syntactical information of each statement. TBCAA (Chen et al., 2019) employs a tree-based convolution network over API-enhanced ASTs. UniXcoder (Guo et al., 2022) leverages both AST and code comments to enrich code representation. GraphCodeBERT (Guo et al., 2021) incorporates variable relations extracted from ASTs in its pre-training tasks. In our work, we specifically aim to capture and model the structural information present in long code snippets.

2.3. Transformer for Long Text

The application of Transformer models for long text can be broadly divided into two categories: scaling up attention and enhancing the original Transformer model, and aggregation methods. The first category includes four main approaches: sparse attention (Child et al., 2019; Correia et al., 2019; Beltagy et al., 2020; Kitaev et al., 2019; Roy et al., 2021; Ainslie et al., 2020; Jiang et al., 2020; Günther et al., 2023), recurrence (Dai et al., 2019), hierarchical mechanisms (Zhang et al., 2019b; Gao and Callan, 2022), and compressed attention (Ye et al., 2019; Guo et al., 2019). Sparse attention restricts each token to attend to only a subset of other tokens. Recurrence integrates recurrent neural network elements into Transformer models to extend their attention span. Hierarchical mechanisms model long input text hierarchically, from sentences to paragraphs. Compressed attention selectively compresses specific parts of the input.

Table 2: The code token length statistic of CodeSearchNet evaluation set.

Length	Ruby	JS	Go	Py	Java	Php	Overall
[0, 256)	16%	10%	22%	14%	13%	13%	14%
[256, 512)	44%	29%	38%	30%	27%	26%	32%
[512, $+\infty$)	41%	62%	40%	56%	60%	61%	54%

The second category, aggregation methods, involves aggregating multiple passage scores or representations for a long document. For instance, Wang et al. (2019) proposed a multi-passage BERT model to globally normalize answer scores across all passages in the question answer task. In the context of document ranking, SMITH (Yang et al., 2020) learns a document representation through hierarchical sentence representation aggregation. PARADE (Li et al., 2020) employs Max, CNN, Attention, and Transformer to aggregate the passage representations. Tsujimura et al. (2023) uses a sliding window method to manage long input sequences in the context of medical Named Entity Recognition tasks.

However, these methods may not be entirely suitable for highly structured code. In well-designed programs, code within the same module, such as a function, is closely interconnected, while interactions between different modules are loosely coupled, adhering to the principle of high cohesion and low coupling. Conversely, long text in natural language tends to exhibit coherence. In this paper, we investigate the applicability of long text methods in the field of code search and propose a new baseline SEA for long code search.

3. Motivation: Long Code Problem

3.1. Preliminaries

Code search aims to find the most relevant code snippet C from a given codebase that matches a query Q. For a current deep-learning model, we first transform query Q and the code snippets C to query and code tokens with the $\mathrm{tokenizer}$ such as BPE (Sennrich et al., 2016). Then we transform the token ids of the query Q and the code snippets C to vector representations $\mathbf{e_q}$ and $\mathbf{e_c}$ by neural network encoders, and calculate the similarity (or distance) measures in Euclidean space such as Cosine similarity or Euclidean distance to obtain the cross-modal similarity score s. The calculation can be formalized as follows:

$$\begin{cases} \mathbf{e_q} = \Gamma(\mathbf{tokenizer}(Q)) \\ \mathbf{e_c} = \Gamma'(\mathbf{tokenizer}(C)), C \in Codebase \\ s = sim(\mathbf{e_q}, \mathbf{e_c}) \end{cases} \tag{1}$$

where Γ and Γ' are two well-trained neural network encoders learned from labeled paired data.

3.2. The Long Code Problem

To control memory and computation costs in training stage, it is common practice to truncate long code. For example, GraphCodeBERT typically takes the first 256 code tokens by default. To investigate whether this truncation method results in information loss, we conducted token length statistics on CodeSearchNet. As shown in Table 2, we found that snippets with a token length less than 256 accounted for only 14.1%, while 53.5% of code snippets exceeded the maximum encoding length of 512 tokens for Transformers. This indicates that truncation leads to information loss for snippets with a token length greater than 256.

To examine the search performance difference of GraphCodeBERT across query subsets with varying ground truth (GT) code lengths, we divided the python test subset of CodeSearchNet (CSN) into 5 distinct query sets based on different GT code token lengths. We calculated the Mean Reciprocal Rank (MRR) of GraphCodeBERT for various code truncation lengths, as shown in Table 1. Notably, we observed a downward trend in search performance as the ground-truth code token length increased (from top to bottom) for code token lengths surpassing 256 tokens, indicating that long code snippets pose challenges for GraphCodeBERT. Moreover, as the code truncation length extended from left to right, we observed a relatively consistent search performance when the truncation length exceeded the token length. And there emerged an upward trend in the search performance for code snippets with the token length surpassing the truncation length. This suggests that simply truncating long code may result in the loss of valuable information.

4. SEA

In this section, we present a comprehensive overview of SEA, encompassing the model architecture, splitting methods, aggregation techniques, and the combine-divide method designed to accelerate inference.

4.1. Model Architecture

We introduce our SEA in this section. The overall pipeline is illustrated in Figure 2. Given a code snippet C, our objective is to derive a code representation e_c. To achieve this, we employ a multi-step approach. We first split the code snippet into a code piece set:

$$P = \mathbf{Split}(C) = \{p_1, p_2, \ldots, p_n\}. \tag{2}$$

Then we use the sliding window method to obtain a partially overlapping code block set:

$$B = \mathbf{SlidingWindow}(P) = \{b_1, b_2, \ldots, b_k\}. \tag{3}$$

Assuming the window size is w and the step is s, then the code block number is $k = \lfloor \frac{n-w}{s} + 1 \rfloor$, where $\lfloor \cdot \rfloor$ refers to round down. Next, we utilize a code encoder, such as GraphCodeBERT, to obtain embeddings for each of the k code blocks:

$$e_B = \{e_{b_1}, e_{b_2}, \ldots, e_{b_k}\}. \tag{4}$$

Finally, an aggregation method is applied to combine the k embeddings into the code representation e_c:

$$e_c = \mathbf{Aggregation}(e_B) \tag{5}$$

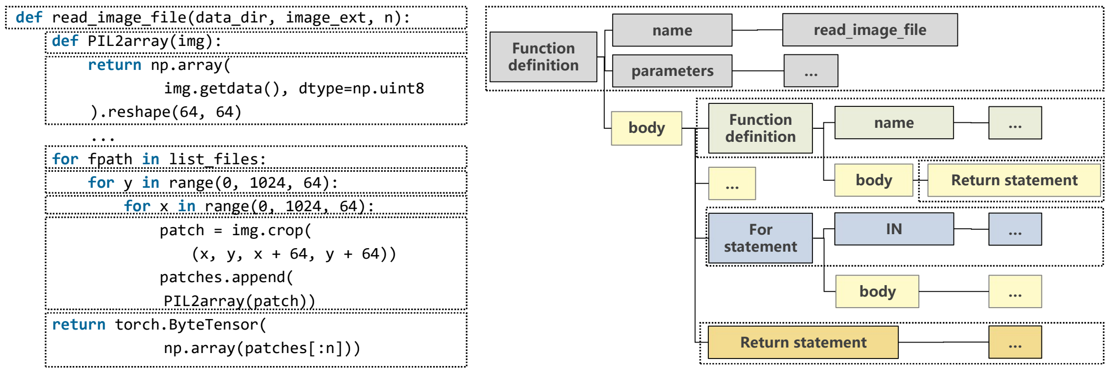
4.2. Splitting Methods

To obtain the code piece set, we explore four splitting methods, namely space-based splitting, token-based splitting, line-based splitting, and AST-based splitting. Space-based splitting is simply splitting by space, resulting in splitting a string like "def read_image_file" is divided into {'def', 'read_image_file'}. Similarly, token-based splitting and line-based splitting entail splitting based on tokens and lines, respectively.

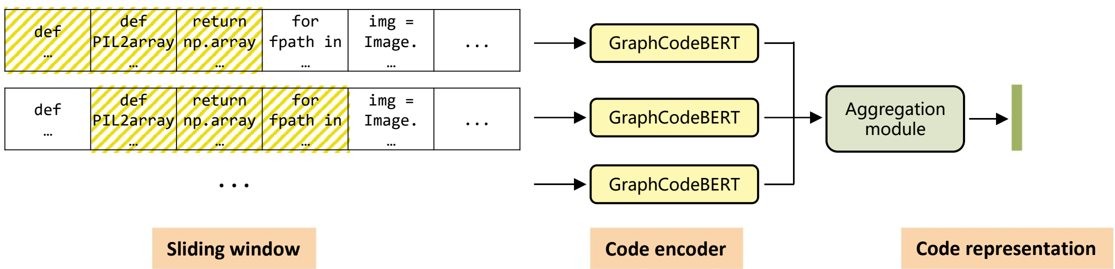
An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of source code written in a programming language. Each node in the AST corresponds to a specific construct in the code, such as expressions, statements, or declarations. The hierarchical structure of ASTs reflects the syntax of programming languages, abstracting away certain syntactic details to focus on the core structure.

For AST-based splitting, our goal is to devise a method that is both straightforward and applicable to various programming languages. Inspired by CAST (Shi et al., 2021), we parse a source code into an Abstract Syntax Tree with tree_sitter[1], and visit this AST by preorder traversal. In the case of composite structures (*i.e.* for, if, def, etc.), as depicted in Figure 2(a), we define the set of AST nodes {head_block, body}, where head_block

[1] https://github.com/tree-sitter/py-tree-sitter



(a) AST-based code splitting.



(b) Slidding window and aggregation.

Figure 2: The pipeline of our proposed SEA (split, encode and aggregate) architecture.

is responsible for splitting the header and body of nested statements such as if and While statements, while body corresponds the method declarations. When encountering a composite structure, we insert a splitting mark before and after the head_block, effectively dividing a large AST into a sequence of non-overlapping subtrees. Subsequently, based on the AST splitting, we construct the code piece set P by splitting the original code accordingly.

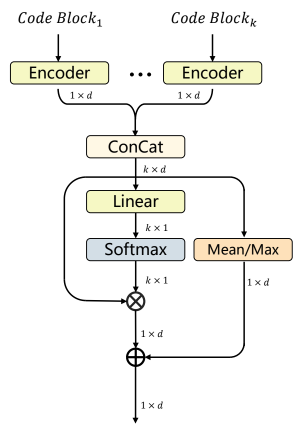
4.3. Aggregation Methods

Meanpooling / Maxpooling. A straightforward approach to aggregate the embeddings of k code blocks is to calculate the mean or maximum of their embeddings:

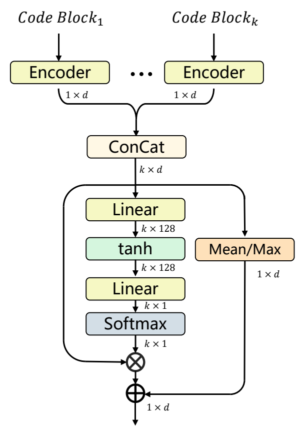
$$e_c = \mathbf{Mean/Max}(\{e_{b_1}, e_{b_2}, \dots, e_{b_k}\}). \tag{6}$$

However, a limitation of meanpooling is that each code block contributes equally to the final representation, regardless of their individual qualities. Similarly, maxpooling gives prominence to the block with the highest value. To address these limitations and enhance the aggregation process, we propose the incorporation of weighted embedding methods.

Attention-based aggregation. Recognizing that not all code blocks hold equal importance in representing long code snippets, we introduce self-adaptive weights α for each block embedding dur-



(a) One layer attention with mean / max.



(b) Two layer attention with mean / max.

Figure 3: The attention-based aggregation methods.

ing aggregation:

$$e_c = \sum_{i}^{k} \alpha_i e_{b_i}. \tag{7}$$

Inspired by attention-based Multi-Instance Learning (Li et al., 2021) and Lightweight Attentional Feature Fusion (Hu et al., 2022), we compute the weights $\{\alpha_1, \dots, \alpha_k\}$ as follows:

$$\{a_1, \dots, a_k\} = softmax(Linear(\{e_{b_1}, \dots, e_{b_k}\})). \tag{8}$$

Table 3: Computation cost analysis. n is the sequence length, d is the representation dimension, k is the code block number, l is the layer number. Note that we use one layer attention for SEA.

Method	Parameters	Complexity
GraphCodeBERT	$5d^2 \cdot l$	$O(n^2 \cdot d \cdot l)$
SEA	$5d^2 \cdot l + d$	$O(\frac{n^2}{k} \cdot d \cdot l)$

For one layer attention, $Linear$ refers to a fully connected layer that transforms the dimension to 1. For two layer attention, $Linear$ refers to two fully connected layers that first transform the dimension to 128 and then transform the dimension to 1. Furthermore, as illustrated in Figure 3(a) and Figure 3(b), we explore the combination of attention with meanpooling / maxpooling methods:

$$e_c = \sum_{i}^{k}(\alpha_i e_{b_i}) + \mathbf{Mean/Max}(\{e_{b_1}, e_{b_2}, \ldots, e_{b_k}\}). \tag{9}$$

For computation cost analysis, SEA employs the sliding window method to significantly reduce complexity to $1/k$. The original complexity of GraphCodeBERT is given by $O(n^2 \cdot d \cdot l)$, where n, d, l represent sequence length, representation dimension, and layer number, respectively. By using the sliding window method, the complexity for each window becomes $O(w^2 \cdot d \cdot l)$, where w denotes the window size. Setting the step $s = w$, the total number of code blocks becomes $k = \frac{n}{w}$, leading to the window size $w = \frac{n}{k}$. Consequently, the overall complexity is simplified to:

$$O(k \cdot w^2 \cdot d \cdot l) = O(k \cdot (\frac{n}{k})^2 \cdot d \cdot l) = O(\frac{n^2}{k} \cdot d \cdot l). \tag{10}$$

This remarkable reduction in complexity to $\frac{1}{k}$ allows SEA to encode *long code* with less memory and computation costs.

Furthermore, as shown in Table 3, we observe that compared to GraphCodeBERT, SEA incorporating one-layer attention Aggregation introduces only d additional learnable parameters. Despite this modest increase in parameter count, it plays a pivotal role in enhancing the effectiveness of the aggregation stage, as our experiments will provide the evidence in Section 6.1.

4.4. Batch Processing

To enhance inference efficiency on large datasets, it is necessary to devise a batch processing method capable of encoding multiple long code snippets simultaneously. As outlined in Section 4.1, we obtain multiple code blocks from each long code snippet. However, due to the varying number of corresponding code blocks for different long code snippets, devising a general batch processing approach poses a challenge.

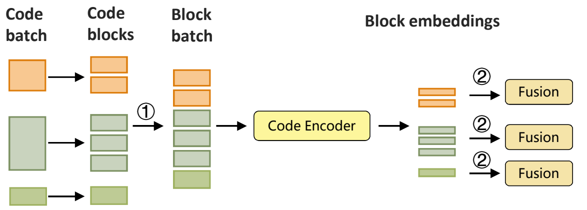


Figure 4: The batch processing combine-divide method. ① and ② refer to combination and division methods.

To address this issue, we introduce the **combine-divide** method. As illustrated in Figure 4, assuming a batch size of 3 (comprising three code snippets), the corresponding number of code blocks for each snippet is 2, 3, and 1, respectively. We begin by combining these six code blocks into a *block batch* and establish a mapping M that links the code index to the block index. Subsequently, we input this block batch into the code encoder in parallel to obtain block embeddings. Finally, leveraging the information from mapping M, we segregate the embeddings into three groups and input them into the aggregation module to obtain distinct code representations.

5. Experimental Design

5.1. Datasets

We conduct experiments on the widely used CodeSearchNet (Husain et al., 2019) dataset, comprising six programming languages, *i.e.* , Ruby, JavaScript, Go, Python, Java, and PHP. Following the approach in (Guo et al., 2021), we apply filtering to eliminate low-quality queries and expand the retrieval set to encompass the entire code corpus.

5.2. Evaluation Metrics

In our evaluation, we use two popular automatic criteria: MRR (Mean Reciprocal Ranking) and R@k (top-k accuracy, k=1, 5, 10, 100). They are commonly used for in previous code search studies (Lv et al., 2015; Gu et al., 2018; Sachdev et al., 2018; Husain et al., 2019; Feng et al., 2020; Huang et al., 2021; Guo et al., 2021). In addition, we report the number of parameter and inference time as the efficiency measure.

5.3. Experimental Settings

Our baseline is GraphCodeBERT. The parameters of code and natural language encoders are

Table 4: The search performance of different SEA variants. Dataset: CodeSearchNet Ruby.

	Window	Step	Splitting	Aggregation	MRR	R@1	R@5	R@10	R@100
GraphCodeBERT	–	–	–	–	0.6948	59.3	82.1	87.3	96.5
SEA-SpaceSplitting	256	128	Space	Maxpooling	0.6919	58.5	82.0	87.2	95.2
	256	128	Space	Meanpooling	0.6929	58.3	83.0	87.4	95.6
	256	128	Space	Attention (two layers)	0.6940	58.7	83.4	87.1	94.8
	256	128	Space	Attention (two layers) + Mean	0.7490	66.3	85.2	88.9	94.4
	256	128	Space	Attention (one layer)	0.6989	59.6	82.2	86.8	95.0
	256	128	Space	Attention (one layer) + Mean	0.7495	66.1	86.3	89.0	94.3
	128	64	Space	Attention (one layer) + Mean	0.7545	66.2	87.5	90.2	95.2
	64	32	Space	Attention (one layer) + Mean	0.7431	65.1	85.6	88.7	94.0
SEA-TokenSplitting	256	128	Token	Attention (one layer) + Mean	0.7752	68.4	89.1	91.9	96.0
	128	64	Token	Attention (one layer) + Mean	0.7606	67.2	87.5	91.3	95.6
	64	32	Token	Attention (one layer) + Mean	0.7352	62.8	87.2	90.6	95.0
SEA-LineSplitting	64	32	Line	Attention (one layer) + Mean	0.7635	67.3	88.2	91.3	95.6
	32	16	Line	Attention (one layer) + Mean	0.7537	66.1	87.2	90.3	95.2
	16	8	Line	Attention (one layer) + Mean	0.7498	65.5	86.9	90.3	95.0
SEA-ASTSplitting	64	32	AST	Attention (one layer) + Mean	0.7539	65.7	**91.4**	**95.0**	**97.6**
	32	16	AST	Attention (one layer) + Mean	**0.7762**	**68.8**	89.1	92.0	96.4
	16	8	AST	Attention (one layer) + Mean	0.7744	68.8	88.7	91.4	96.3

initialized by GraphCodeBERT. For training, we randomly select 6 code blocks from the divided code blocks of one long code. The training batch size is 32. For evaluation, we use all divided code blocks of one long code. The evaluated batch size is 256. All experiments are conducted on a machine with Intel Xeon E5-2698v4 2.2Ghz 20-Core CPU and two Tesla V100 32GB GPUs.

6. Experimental Results

6.1. The Optimal SEA Configuration

To identify the optimal configuration for SEA, we conducted experiments by varying our architecture using different code splitting methods and aggregation methods, while measuring the resulting changes in search performance. Given that the CodeSearchNet Ruby dataset is relatively small, we focused on conducting experiments on the ruby subset, and we present the results in Table 4.

In Table 4 rows SpaceSplitting, we experimented with various aggregation methods as described in Section 4.3. Our findings showed that using any single aggregation method in isolation did not yield significant performance improvements compared to the GraphCodeBERT Baseline. However, upon fusing the attention method with meanpooling, we observed substantial performance enhancement. Specifically, the Attention (one layer) + Mean aggregation method improved MRR and R@1 by 7.9% and 11.5%, respectively. Consequently, for subsequent experiments, we opted to use the Attention (one layer) + Mean aggregation method.

In Table 4 rows SpaceSplitting, TokenSplitting, LineSplitting, ASTSplitting, we explored different code split methods, as detailed in Section 4.2. For space and token-based splitting methods, we set the window size from 64 to 256 due to the finer granularity of division. Conversely, for line and AST-based split methods, we set the window size from 16 to 64. Notably, we observed that the AST-based split method displayed outstanding performance, achieving the highest MRR and R@1 with a window size of 32. As a result, in subsequent experiments, SEA refers to SEA-ASTSplitting with a window size of 32, step size of 16 and the Attention (one layer) + Mean aggregation method.

6.2. Comparison with Three Sparse Transformers

In this section, we conduct a comparison between SEA and three sparse Transformers, BIGBIRD (Zaheer et al., 2020), Longformer (Beltagy et al., 2020), and LongCoder (Guo et al., 2023). BIGBIRD and Longformer are two well-known long document-oriented Transformers. LongCoder employs a sliding window mechanism to handle long code input for code completion. Specifically, we leverage the bigbird-roberta-base[2], longformer-base-4096[3] and longcoder-base[4] models, with a token length of 1024. Due to BIGBIRD and Longformer not being pretrained on the code dataset, we also conducted experiments to initialize BIGBIRD and Longformer with the parameters of GraphCodeBERT. The results are presented in Table 5. Comparing the results before and after initializing BIGBIRD and Longformer with the parameters of GraphCodeBERT, we found that MRR results improved from 0.2952 and 0.5016 to 0.6121 and 0.6595, respec-

[2]https://huggingface.co/google/bigbird-roberta-base

[3]https://huggingface.co/allenai/longformer-base-4096

[4]https://huggingface.co/microsoft/longcoder-base

Table 5: Comparison with sparse Transformers. The notation (G) indicates that the model is initialized with GraphCodeBERT parameters. The code inference time is determined by randomly selecting 1000 codes and calculating the average inference time. We repeat each time calculating experiment three times and report the mean and standard deviation. Dataset: CodeSearchNet Ruby. SEA outperforms other models significantly ($p < 0.01$).

Model	#Param.	Token Length	Inference Time	MRR	R@1	R@5	R@10
GraphCodeBERT	124.6M	256	6.3 $\pm$ 0.3ms	0.6948	59.3	82.1	87.3
BIGBIRD	127.5M	1024	20.1 $\pm$ 0.2ms	0.2952	19.2	39.8	51.1
BIGBIRD (G)	127.5M	1024	19.8 $\pm$ 0.0ms	0.6121	50.8	74.2	80.7
Longformer	148.7M	1024	33.7 $\pm$ 0.2ms	0.5128	39.9	65.3	72.4
Longformer (G)	148.7M	1024	33.7 $\pm$ 0.1ms	0.6595	55.1	79.4	84.0
LongCoder	149.6M	1024	68.6 $\pm$ 0.2ms	0.4718	35.8	61.1	67.8
SEA	124.6M	-	7.2 $\pm$ 0.5ms	**0.7762**	**68.8**	**89.1**	**92.0**
- w/o combine-divide	124.6M	-	24.3 $\pm$ 2.4ms	**0.7762**	**68.8**	**89.1**	**92.0**

tively. We attribute this performance gap to the need for re-pretraining models that were originally pretrained on natural language datasets. We observed that LongCoder's MRR was 0.4718, which represents a significant decrease compared to GraphCodeBERT, suggesting that LongCoder may be primarily suited for Code Completion tasks. We also conducted t-tests between our SEA and other baselines, and the results demonstrate that SEA significantly outperforms all sparse Transformer baselines ($p < 0.01$), highlighting its superior performance in the domain of code search.

In terms of model parameters and search efficiency, SEA stands out as it boasts a lower parameter count and shorter inference time compared to BIGBIRD, Longformer and LongCoder. It's worth noting that SEA's parameter count is closely aligned with that of GraphCodeBERT, differing only by the addition of a single attention layer. However, this minor change results in a significant boost in search performance. We also present experimental results without employing the combine-divide method in Table 5. We observed that while the search performance stays stable, the inference time increases by more than threefold. It highlights the considerable improvement in inference time brought about by the combine-divide method, thereby confirming its effectiveness in accelerating the model's inference process.

6.3. SEA Performance on Varied Code Lengths

To explore the improvement of the proposed SEA for code snippets with varying lengths, we present the search performance comparison between the baseline method GraphCodeBERT and SEA under different ground-truth code token lengths. The results are depicted in Figure 5.

Notably, the retrieval performance of each query subset exhibits noticeable enhancements, particularly for long code retrieval results. We attribute this improvement to two crucial factors. Firstly, the aggregation module of SEA adaptively captures and incorporates information from diverse segments of the long code, leading to a more comprehensive and informative code representation. Secondly, the code splitting method employed by SEA can be viewed as a form of data augmentation, providing additional context and variation that aids in strengthening the code representation. In summary, SEA yields a more robust code representation, significantly enhancing the overall retrieval performance.

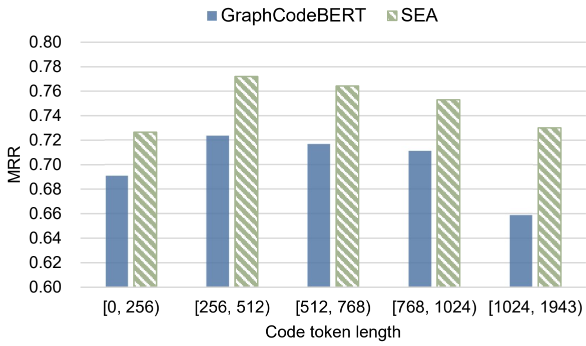


Figure 5: The performance comparison between GraphCodeBERT and SEA in different ground-truth code token lengths. Compare to GraphCodeBERT, SEA achieves significantly ($p < 0.01$) better performance for different code token lengths.

6.4. Baseline Comparison Across Multiple Programming Languages

To ensure a fair and reproducible comparison, we carefully selected pretraining-based baselines that meet the following three criteria: 1) The source code is publicly available; 2) The overall model is adaptable to all the six programming languages on the CodeSearchNet dataset; 3) The paper is peer-reviewed if it is published as a research pa-

Table 6: The MRR on six languages of the CodeSearchNet dataset. SEA here refers to SEA-ASTSplitting with window size 32 and step 16. SEA +RoBERTa refers to SEA with RoBERTa as the code encoder. SEA outperforms baselines significantly ($p < 0.01$).

Model / Method	Ruby	Javascript	Go	Python	Java	Php	Overall
RoBERTa	0.587	0.517	0.850	0.587	0.599	0.560	0.617
UniXcoder	0.586	0.603	0.881	0.695	0.687	0.644	0.683
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.628	0.693
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	0.649	0.713
SEA +RoBERTa	0.651 (10.9%↑)	0.593 (14.6%↑)	0.879 (3.5%↑)	0.633 (7.9%↑)	0.666 (11.1%↑)	0.647 (15.6%↑)	0.678 (10.0%↑)
SEA +UniXcoder	0.648 (10.7%↑)	0.692 (14.8%↑)	0.896 (1.8%↑)	0.707 (1.7%↑)	0.739 (7.5%↑)	0.712 (10.5%↑)	0.732 (7.3%↑)
SEA +CodeBERT	0.742 (9.3%↑)	0.696 (12.3%↑)	0.905 (2.6%↑)	0.714 (6.2%↑)	0.732 (8.3%↑)	0.711 (13.2%↑)	0.750 (8.3%↑)
SEA +GraphCodeBERT	**0.776 (10.4%↑)**	**0.742 (15.2%↑)**	**0.921 (2.7%↑)**	**0.754 (8.9%↑)**	**0.768 (11.1%↑)**	**0.748 (15.3%↑)**	**0.785 (10.1%↑)**

per. Consequently, we select four deep end-to-end approaches: **RoBERTa** (Liu et al., 2019), **UniXcoder** (Guo et al., 2022), **CodeBERT** (Feng et al., 2020), and **GraphCodeBERT** (Guo et al., 2021).

In Table 6, we present the MRR results, demonstrating that SEA outperforms all methods across all six programming languages. Notably, this conclusion remains consistent for the recall metric and another variant of SEA, the results of which can be found in our replication package. These findings reinforce the superiority of SEA as compared to the pretraining-based baselines across diverse programming languages.

7. Conclusion

In this paper, we address the challenge of effectively modeling *long code* for code search. We introduce SEA, an effective approach that yields improved code representations for long code snippets. Despite its simplicity, our experimental results show the remarkable effectiveness and efficiency of SEA. We believe this work opens up new possibilities for code search.

8. Ethical Statement

Future extensions and applications arising from our work should be mindful of the environmental impact of training large-scale models. They should actively avoid its potential misuse by searching malicious intent. However, it is unlikely that the model in its current form would lead to such an impact in the near future. Our model also has the potential for making a positive impact in areas such as code search, long code understanding and code representation.

9. Acknowledgments

The work described in this paper is partially supported by CCF-Huawei Populus Grove Fund CCF-HuaweiSE202301.

10. Bibliographical References

Joshua Ainslie, Santiago Ontañón, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. 2020. ETC: encoding long and structured inputs in transformers. In EMNLP.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. In NeurIPS.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In ICML.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. arXiv.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Treecaps: Tree-based capsule networks for source code processing. In AAAI.

Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with few-shot meta learning. arXiv.

Long Chen, Wei Ye, and Shikun Zhang. 2019. Capturing source code semantics via tree-based convolution over api-enhanced ast. In CF.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. arXiv.

Gonçalo M Correia, Vlad Niculae, and André FT Martins. 2019. Adaptively sparse transformers. In EMNLP-IJCNLP.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In ACL.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In NAACL-HLT.

Yali Du and Zhongxing Yu. 2023. Pre-training code representation with semantic flow graph for effective bug localization. In FSE/ESEC.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In EMNLP.

Luyu Gao and Jamie Callan. 2022. Long document re-ranking with modular re-ranker. In SIGIR.

Dobrik Georgiev, Marc Brockschmidt, and Miltiadis Allamanis. 2022. Heat: Hyperedge attention networks. arXiv.

Wenchao Gu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Michael Lyu. 2022. Accelerating code search with deep hashing and code classification. In ACL.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In ICSE.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, et al. 2023. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. arXiv.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In ACL.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In ICLR.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian J. McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In ICML, Proceedings of Machine Learning Research. PMLR.

Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. 2019. Star-transformer. In NAACL-HLT. Association for Computational Linguistics.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In ICLR.

Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In ASE. IEEE.

Fan Hu, Aozhu Chen, Ziyue Wang, Fangming Zhou, Jianfeng Dong, and Xirong Li. 2022. Lightweight attentional feature fusion: A new baseline for text-to-video retrieval. In ECCV. Springer.

Fan Hu, Yanlin Wang, Lun Du, Xirong Li, Hongyu Zhang, Shi Han, and Dongmei Zhang. 2023. Revisiting code search in a two-stage paradigm. In WSDM.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. In ACL.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv.

Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des alpes et des jura. Bull Soc Vaudoise Sci Nat, pages 547–579.

Jyun-Yu Jiang, Chenyan Xiong, Chia-Jung Lee, and Wei Wang. 2020. Long document ranking with query-directed sparse transformer. In EMNLP Findings.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In ICSE.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2019. Reformer: The efficient transformer. In ICLR.

Canjia Li, Andrew Yates, Sean MacAvaney, Ben He, and Yingfei Sun. 2020. Parade: Passage representation aggregation for document reranking. ACM Transactions on Information Systems.

Xirong Li, Yang Zhou, Jie Wang, Hailan Lin, Jianchun Zhao, Dayong Ding, Weihong Yu, and Youxin Chen. 2021. Multi-modal multi-instance learning for retinal disease recognition. In ACMMM.

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 pre-training approach. arXiv.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In ASE.

Y Ma, Yali Du, and Ming Li. 2023. Capturing the long-distance dependency in the control flow graph via structural-guided attention for bug localization. In IJCAI.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. IEEE Transactions on Services Computing, pages 771–783.

Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating tree path in transformer for code representation. In NeurIPS.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. Now Publishers Inc.

Stephen E Robertson and K Sparck Jones. 1976. Relevance weighting of search terms. Journal of the American Society for Information science, pages 129–146.

Barbara Rosario. 2000. Latent semantic indexing: An overview. Techn. rep. INFOSYS, pages 1–16.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2021. Efficient content-based sparse attention with routing transformers. Transactions of the Association for Computational Linguistics, 9:53–68.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In MAPL.

Abdus Satter and Kazi Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. In ICCIT, pages 586–591. IEEE.

Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. Introduction to information retrieval, volume 39. Cambridge University Press Cambridge.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In ACL.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In EMNLP.

Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code search based on context-aware code translation. arXiv.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In AAAI.

Tomoki Tsujimura, Koshi Yamada, Ryuki Ida, Makoto Miwa, and Yutaka Sasaki. 2023. Contextualized medication event extraction with striding ner and multi-turn qa. Journal of Biomedical Informatics, page 104416.

Thanh Van Nguyen, Anh Tuan Nguyen, Hung Dang Phan, Trong Duc Nguyen, and Tien N Nguyen. 2017. Combining word2vec with revised vector space model for better code retrieval. In ICSE-C. IEEE.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In ASE.

Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. 2019. Multi-passage bert: A globally normalized bert model for open-domain question answering. In EMNLP.

Liu Yang, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. 2020. Beyond 512 tokens: Siamese multi-depth transformer-based hierarchical encoder for long-form document matching. In CIKM.

Yangrui Yang and Qing Huang. 2017. Iecs: Intent-enforced code search via extended boolean model. Journal of Intelligent & Fuzzy Systems, pages 2565–2576.

Zihao Ye, Qipeng Guo, Quan Gan, Xipeng Qiu, and Zheng Zhang. 2019. Bp-transformer: Modelling long-range context via binary partitioning. arXiv.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: Transformers for longer sequences. In NeurIPS.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019a. A novel neural source code representation based on abstract syntax tree. In ICSE.

Xingxing Zhang, Furu Wei, and Ming Zhou. 2019b. Hibert: Document level pre-training of hierarchical bidirectional transformers for document summarization. In ACL.