LongCoder: A Long-Range Pre-trained Language Model for Code Completion

Daya Guo [*1] **Canwen Xu** [*2] **Nan Duan** [3] **Jian Yin** [1] **Julian McAuley** [2]

Abstract

In this paper, we introduce a new task for code completion that focuses on handling long code input and propose a sparse Transformer model, called LongCoder, to address this task. LongCoder employs a sliding window mechanism for self-attention and introduces two types of globally accessible tokens — *bridge tokens* and *memory tokens* — to improve performance and efficiency. *Bridge tokens* are inserted throughout the input sequence to aggregate local information and facilitate global interaction, while *memory tokens* are included to highlight important statements that may be invoked later and need to be memorized, such as package imports and definitions of classes, functions, or structures. We conduct experiments on a newly constructed dataset that contains longer code context and the publicly available CodeXGLUE benchmark. Experimental results demonstrate that LongCoder achieves superior performance on code completion tasks compared to previous models while maintaining comparable efficiency in terms of computational resources during inference.

1. Introduction

Code completion is a crucial task in software development that helps developers save time and effort by suggesting and auto-completing code based on context. With the advancement of large language models, Transformer-based models (Vaswani et al., 2017) have demonstrated impressive results in code completion (Chen et al., 2021). However, the computational cost of these models grows quadratically with the length of input, making them less suitable for modeling long code context. On the other hand, modeling long code can potentially improve the accuracy of code completion and enable applications on a file and even project level. An efficient model that can scale to such long input can be suitable for code completion that contains long context.

In this paper, we propose a new pre-trained language model, named LongCoder, for long code modeling. As shown in Figure 1, LongCoder features a sparse attention mechanism that reduces the computational complexity (to linear). LongCoder exploits a sliding window mechanism for self-attention that attends only to local context. To allow LongCoder to maintain an understanding of the entire code file, we introduce *bridge attention* and *global attention*, with the corresponding two types of globally accessible tokens, *bridge tokens* and *memory tokens*. Bridge attention aggregates the information of a code snippet and allows it to be accessed from a long distance. Bridge tokens are inserted throughout the input sequence and can attend to a fixed length of context. Memory tokens provide global attention to statements that include a package import, definitions of classes, functions, or structures. The scope of these statements is often global and invoked later, which means they have a longer impact than other statements, making them worth memorizing. By referring to these statements, the model can exploit long context while maintaining linear complexity.

To evaluate the effectiveness of LongCoder and encourage future research on **L**ong **C**ode **C**ompletion, we construct a new dataset called LCC by filtering code from GitHub based on length, with the goal of focusing on longer code examples. On average, the examples in LCC are 5× longer than those in existing datasets (Lu et al., 2021). We benchmark several baselines, LongCoder and OpenAI Codex (Chen et al., 2021) on LCC. Our experimental results demonstrate that code completion can benefit from taking longer context into consideration, and our LongCoder achieves superior performance compared to existing models with comparable computational costs.

Overall, our contributions are as follows:

- We construct a new dataset (LCC) for code completion tasks that requires long code modeling to encourage more research in such scenarios.
- We propose two types of sparse attention, motivated by observations on attention patterns of existing models and how human programmers write code.

[*]Equal contribution [1]Sun Yat-sen University [2]University of California, San Diego [3]Microsoft Research Asia. Correspondence to: Daya Guo <guody5@mail2.sysu.edu.cn>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

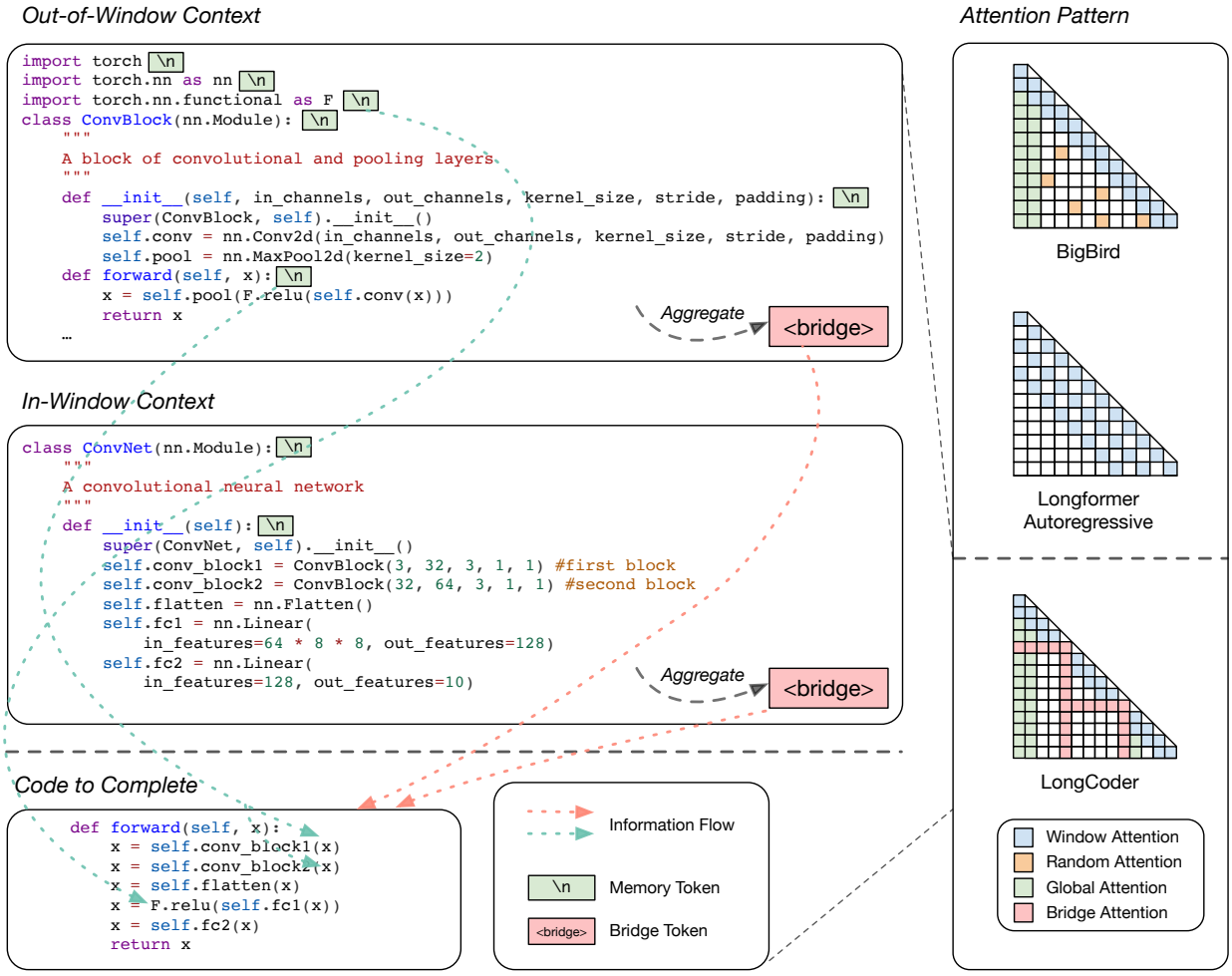

Figure 1. ***(Left)*** An example of how LongCoder facilitates completion with longer context. The memory tokens save potentially useful information (including package imports, class and function definitions) for global access despite whether they are within the sliding window. The bridge tokens aggregate local information by attending to a fixed length of tokens. The information flow within the window is omitted for clarity. ***(Right)*** Attention patterns used in BigBird (Zaheer et al., 2020), Longformer (Beltagy et al., 2020) and LongCoder. Best viewed in color.

- We train and release LongCoder, a sparse and efficient pre-trained Transformer model for long code modeling, which achieves superior performance on both long and regular code completion with comparable computational resources.[1]

2. Related Work

Code Completion Code completion is an essential task that helps programmers improve their efficiency by suggesting and automatically completing code based on context and previous inputs. Prior works have explored the use of statistical learning for the code completion task, such as the use of n-gram techniques (Tu et al., 2014; Hindle et al., 2016) and probabilistic grammar-based methods (Allamanis & Sutton, 2014; Bielik et al., 2016; Raychev et al., 2016; Hellendoorn & Devanbu, 2017). With the success of pre-training in natural language processing (Devlin et al., 2019; Radford et al., 2019; Brown et al., 2020), decoder-only pre-trained models based on Transformer have been proposed to promote the development of code completion. Svyatkovskiy et al. (2020) and Lu et al. (2021) respectively propose GPT-C and CodeGPT, which are pre-trained by generating code from left to right in an auto-regressive manner on large amounts of code. Liu et al. (2020) and Guo et al. (2022) pre-train similar models CugLM and UniXcoder with multi-task learning by leveraging code structure for code completion. Codex (Chen et al., 2021), PolyCoder (Xu et al., 2022), CodeGen

[1]All the codes and data are available at https://github.com/microsoft/CodeBERT.

(Nijkamp et al., 2022), InCoder (Fried et al., 2022), and AlphaCode (Li et al., 2022) build large language models with billions of parameters and achieve impressive performance on code generation by training on a large-scale and high-quality code corpus. For these pre-trained models, it is impractical to simply expand the context window to model long-range sequences, due to computational complexity of the attention mechanism increasing quadratically with the input length. Therefore, Clement et al. (2021) propose to extract the most important code fragments and integrate them into a fixed-length context window. However, due to the constraint of fixed window length, some high-priority code, such as class and function definitions, may be omitted. Additionally, increasing the window length would also introduce additional computational overhead. Different from these works, LongCoder is a sparse Transformer that can take advantage of the entire file-level code context while maintaining comparable efficiency in terms of computational resources during inference.

Long-Range Transformer Models The original Transformer (Vaswani et al., 2017) is inefficient for modeling long sequences since its time and space complexity is $O(n^2)$, where n is the length of the sequence. Prior studies focus on optimizing the complexity to enable processing of longer sequences. To name a few, Sparse Transformer (Child et al., 2019) reduces the quadratic complexity of standard self-attention by computing attention on sparse query-key pairs. Sparse Transformer uses a dilated sliding window to capture local context. Reformer (Kitaev et al., 2020) proposes locality sensitive hashing (LSH) attention to reduce the complexity and memory footprint. Longformer (Beltagy et al., 2020) uses dilated sliding windows to model longer sequences and adds global memory tokens to allow interaction with all tokens. Performer (Choromanski et al., 2021) generalizes attention calculation by introducing kernel functions. They then propose a random kernel function, namely orthogonal random features (ORF) to approximate the standard self-attention. Linformer (Wang et al., 2020) applies low-rank projection to the length dimension to reduce the complexity of self-attention. Linear Transformers (Katharopoulos et al., 2020) uses a kernel function that exploits the associativity property of matrix products to reduce complexity. BigBird (Zaheer et al., 2020) has an attention pattern comprised of random attention, window attention and global attention. CosFormer (Qin et al., 2022) proposes a linear operator and a cosine-based distance re-weighting mechanism as the substitute for softmax attention. We recommend Tay et al. (2022) as a more comprehensive survey on long-range efficient Transformer models. Different from these works, our LongCoder introduces code heuristics into the dynamic construction of global attention to imitate how human programmers code.

3. Long Code Completion

Code completion is a fundamental and important task for code models, which can help programmers improve their efficiency while coding. Previous public benchmarks primarily focused on completion with short code context. For instance, CodeXGLUE (Lu et al., 2021) offers two code completion datasets from *PY150* (Raychev et al., 2016) in Python and *Github Java Corpus* Allamanis & Sutton (2013) in Java, and also builds two test datasets to evaluate next-line prediction. The average length of the code context in the two test datasets is 478 tokens and 365 tokens, respectively. However, according to our statistics, the average length of a Python source file on GitHub is 1,305 tokens. After tokenization, the average length becomes 2,090 tokens while 41%/24% of the files have a length longer than 1,024/2,048 tokens, which highlights the need for models that can handle longer code sequences in order to be more practical and useful in the real-world. Meanwhile, longer code sequences contain more complex structures and require models to consider more context and dependencies. This can be challenging for previously proposed code completion models that focus on short code and do not take into account the long context of the code. By evaluating models on longer code sequences, we can better understand their ability to handle more complex and realistic scenarios. Meanwhile, long code completion poses new challenges for efficiency of code models, as in vanilla Transformers (Vaswani et al., 2017), the computational resources grow quadratically with the input length.

In this paper, we introduce the **L**ong **C**ode **C**ompletion Benchmark (**LCC**), a new benchmark that focuses on code completion with long code context for three programming languages: Python, Java, and C#. Specifically, we construct our datasets from the *github-code*[2] dataset, which contains a vast number of code files sourced from GitHub with an open-source license that permits research use. The steps to construct the datasets are as follows:

- We first follow Allamanis (2019) to deduplicate examples with high similarity (Jacobi similarity ≥ 0.9) in order to eliminate forked files, and then remove code files that can't be parsed into an abstract syntax tree using a standard compiler tool called *tree-sitter*.[3]
- Since the benchmark primarily focuses on the code completion task with long code context, we remove code files whose length of code tokens after tokenization is shorter than 512. Additionally, we also eliminate excessively long code files with a length greater than

[2] https://huggingface.co/datasets/codeparrot/github-code

[3] https://github.com/tree-sitter/tree-sitter

Table 1. Data statistics of the code context length in LCC test set. 25%/50%/75% refer to the first/second/third quartile.

Language	Average	25%	50%	75%
Python	1993.3	1056	1438	2211
Java	1841.4	1058	1307	2003
C#	1970.5	1023	1396	2143

10,000 tokens.

- For each programming language, we sample 100k examples for training, and 10k examples for development and 10k for testing. For each sample on development and test sets, we randomly sample an uncommented line of code not shorted than 3 tokens and ensure that there is sufficient context, i.e., a context larger than 512 code tokens. The data statistics of the context length in the LCC test sets are listed in Table 1.

We follow Lu et al. (2021) to evaluate the performance of the models in terms of Exact Match (EM) and Edit Similarity (Edit Sim) on a per-line basis (Svyatkovskiy et al., 2020).

4. LongCoder

LongCoder is an attempt to tackle the efficiency problem of modeling longer code. It applies sparse attention to reduce quadratic time and space complexity of self-attention to linear. There are three types of attention in LongCoder — window attention, bridge attention, and global attention. Each type is motivated by observations on previous models and focuses on one important aspect in modeling long code. The three types of attention are illustrated in Figure 1 and we will describe them individually.

4.1. Window Attention

Code completion largely relies on local context while only a few instances of long-distance dependencies are present. For example, in Figure 1 (bottom left), generating assignment operators and parentheses only depend on the current statement, whereas to generate variables such as `x` and `conv_block`, the model needs to look at neighboring statements. Intuitively, we can exploit such locality to sparsify the attention to achieve better efficiency. We further verify this observation by counting the distribution of average attention scores between two tokens within different distances. As shown in Figure 2, a large portion of attention is concentrated within a narrow window. Notably, a fixed window of 256 covers more than 90% of the attention weights. This sparsity enables us to apply a sliding window attention mechanism (Beltagy et al., 2020; Zaheer et al., 2020).

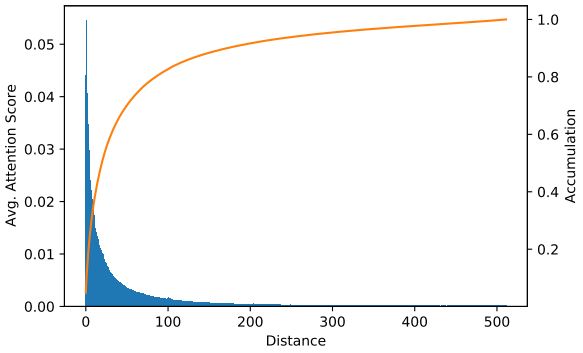

Figure 2. Distribution of average attention scores between two tokens within different distances in CodeGPT (Lu et al., 2021). The attention score is an average of 100 Python code examples across all Transformer layers.

Formally, given the linear projections Q, K, V, the self-attention scores in Transformer are calculated as follows:

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + M)V \quad (1)$$

where M is a mask matrix (to be completed in Equation 5) to control the context a token can attend to when computing its contextual representation. If the i-th token is allowed to attend to the j-th token, then M_{ij} is set to 0, otherwise $-\infty$.

For window attention, the mask attention matrix M^{window} is calculated as follows:

$$M_{ij}^{window} = \begin{cases} 0 & \text{if } i - j \leq w \\ -\infty & \text{otherwise} \end{cases} \quad (2)$$

where w is the window size. This window attention pattern reduces the complexity of the self-attention mechanism by limiting the receptive field size of each token to a small window of size w at each layer. The computation complexity of this pattern is $O(n \times w)$, which scales linearly with input sequence length n. After applying N transformer layers of such sliding window attention, the receptive field size increases to $N \times w$ at the top layer. Since each token only attends to w tokens to its left rather than the entire preceding sequence, the model can achieve faster inference speed.

4.2. Bridge Attention

Window attention is good at handling local dependencies and it also has a wide receptive field as discussed above. However, if a token needs to access tokens from a distance of L tokens away, it would require $\lceil \frac{L}{w} \rceil$ hops through window attention. This makes it challenging to access information from distant context as the attention score between them will be greatly reduced due to accumulation through multiplication. Thus, we introduce a new type of special

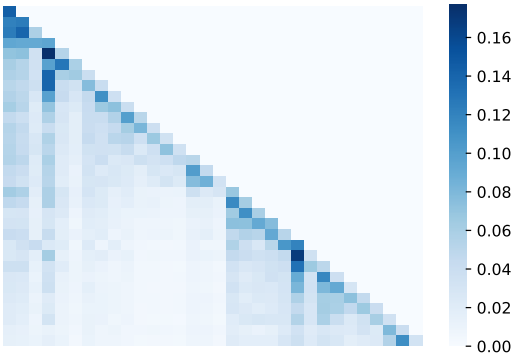

Figure 3. Visualization of the attention matrix (partially shown for clarity) in CodeGPT (Lu et al., 2021). The attention matrix is averaged across all Transformer layers.

token, namely *bridge tokens*, to aggregate local information for global access. Bridge tokens can attend to a fixed length of tokens and be attended from all subsequent tokens. From the perspective of representation learning, a bridge token can be seen as a learned representation for the corresponding slice of code.

Specifically, we insert m bridge tokens S_b every $\lceil \frac{n}{m} \rceil$ tokens and use a separate set of projections, Q_b , K_b , V_b to compute attention scores for the bridge attention. The bridge tokens do not involve next token prediction but they are used to aggregate information from the preceding $\lceil \frac{n}{m} \rceil$ tokens. The use of additional projections allows for the ability to model different types of attention. Finally, the mask matrix for bridge attention is calculated as follows:

$$M_{ij}^{bridge} = \begin{cases} 0 & \text{if } j \in S_b \text{ and } i \geq j \\ 0 & \text{if } i \in S_b \text{ and } i - j \leq \lceil \frac{n}{m} \rceil \\ -\infty & \text{otherwise} \end{cases} \quad (3)$$

The complexity of bridge attention is $O(m \times n) \approx O(n)$ where $m \ll n$. Compared to stacked window attention, bridge attention allows each token to attend to any preceding token with at most 2 hops, which enables the model to effectively access long-range context.

4.3. Global Attention

Identifiers with the global scope, for example, package imports, global functions, classes and their member functions (i.e., methods), can be called from any location within a file. For long code, a local sliding window cannot capture such information that should be globally accessible. This is especially outstanding for package imports, which are usually located at the beginning of a file. For example, in Figure 1, without knowing the user has imported `torch.nn.functional` as a new identifier `F`, the model cannot be sure whether to use `F` or the full package name.[4] Also, to call the forward function of `conv_block1` and `conv_block2`, the model needs access to the original definition of the class `ConvBlock`, which also falls outside the current sliding window. Directly accessing these tokens is similar to how human programmers quickly refer to definitions in the code. This global effect can also be observed in the visualization of the attention matrix in CodeGPT (Lu et al., 2021). As shown in Figure 3, some tokens seem to have a global impact while others only matter locally.

Therefore, in addition to bridge tokens, where the model automatically learns to aggregate globally useful information, we add another type of global token, namely *memory tokens*, to inject code heuristics to the attention. Specifically, we leverage the structure of code with *tree-sitter* to parse the code into an abstract syntax tree (AST). Then, we find all statements that include a package import, class or function definition and grant the line feeds (LF, `\n`) of those statements global access. We denote the set of positions of these line feeds as G, where $k = |G|$. The mask matrix M^{global} of global attention is calculated as follows:

$$M_{ij}^{global} = \begin{cases} 0 & \text{if } j \in G \text{ and } i \geq j \\ -\infty & \text{otherwise} \end{cases} \quad (4)$$

The complexity of the global attention is $O(kn) \approx O(n)$, as we have $k \ll n$, where n is the length of the sequence.

Unlike previous work (Clement et al., 2021) which extracts the most significant statements and encode them in a fixed-length context window, our global attention requires less memory and can reuse previously encoded hidden states.

Finally, considering all three types of attention together, M in Equation 1 becomes:

$$M = \max(M^{window}, M^{bridge}, M^{global}) \quad (5)$$

where $\max$ is the element-wise maximum function.

5. Experiments

5.1. Experimental Settings

Baselines We evaluate LongCoder against several publicly available pre-trained code generation models, including GPT-2 (Radford et al., 2019), CodeGPT (Lu et al., 2021), and UniXcoder (Guo et al., 2022). GPT-2 is pre-trained on a text corpus and CodeGPT is pre-trained on the CodeSearchNet dataset (Husain et al., 2019) using next token prediction as the objective. UniXcoder based on UniLM (Dong et al., 2019) is pre-trained on a cross-modal dataset that includes code, text, and abstract syntax trees. Additionally, we also compare LongCoder with sparse Transformer models, such

[4] Both are common code styles in PyTorch.

Table 2. Experimental results on the Long Code Completion (LCC) dataset.

Model	#Param.	Memory	Runtime	Python		Java		C#	
				EM	Edit Sim	EM	Edit Sim	EM	Edit Sim
OpenAI Codex	12B	-	-	39.65	68.97	43.15	72.05	53.89	77.93
Transformer	124M	191M	750ms	10.64	43.64	15.32	47.52	19.16	48.87
GPT-2	124M	191M	750ms	11.20	42.62	17.09	47.18	20.27	48.27
CodeGPT	124M	191M	750ms	12.24	43.81	19.20	49.50	22.58	51.03
UniXcoder	126M	191M	750ms	16.55	50.22	23.93	55.38	27.97	57.29
LongFormer	150M	381M	781ms	16.79	51.07	24.80	56.03	29.75	58.23
BigBird	128M	205M	804ms	17.03	51.14	25.19	56.91	30.27	58.66
LongCoder	150M	211M	812ms	**17.88**	**55.07**	**26.42**	**61.21**	**31.34**	**64.37**
- w/o pretrain	150M	211M	812ms	17.61	54.82	25.96	60.81	31.22	64.18

Table 3. Data statistics about the context length of CodeXGLUE test dataset. 25%/50%/75% refer to the first/second/third quartile.

Language	Average	25%	50%	75%
Python	477.8	83	197	502
Java	365.0	74	171	397

as LongFormer (Beltagy et al., 2020) and BigBird (Zaheer et al., 2020). LongFormer uses a dilated sliding window to model long sequences in the generation task, while BigBird has an attention pattern that includes random, window, and global attention. In addition to these comparable baselines, we also report the performance of OpenAI Codex on LCC for reference. Note that Codex is 100× larger than other models and is likely to have seen the test set of LCC in its pretraining thus is not directly comparable.

Benchmarks We evaluate the performance of LongCoder and the baselines on two benchmarks: LCC (introduced in Section 3), and the code completion task benchmark in CodeXGLUE (Lu et al., 2021). CodeXGLUE provides PY150 (Raychev et al., 2016) and JavaCorpus (Allamanis & Sutton, 2013) datasets in Python and Java for line-level code completion. The statistics for the context length of the CodeXGLUE test datasets are listed in Table 3. We can see that the context length of the input sequence is 5 times shorter than LCC, and only a small portion of the samples require modeling for long code sequences. The objective of evaluating the performance of sparse models on the CodeXGLUE dataset is to examine their effectiveness in scenarios where the code context is relatively short.

Moreover, longer context can benefit applications including cross-file code completion (Liu et al., 2023). We test the performance of LongCoder on the cross-file-random (XF-R) setting of RepoBench (Liu et al., 2023). The task is to predict the next line of code based on a given in-file context, consisting of import statements and preceding lines before the target line, as well as a cross-file context, comprising snippets from other files in the code repository, parsed by import statements.

Table 4. Results on CodeXGLUE code completion benchmark.

Model	PY150		JavaCorpus	
	EM	Edit Sim	EM	Edit Sim
Transformer	38.51	69.01	17.00	50.23
GPT-2	41.73	70.60	27.50	60.36
CodeGPT	42.37	71.59	30.60	63.45
UniXcoder	43.12	72.00	32.90	65.78
LongCoder	**43.77**	**73.37**	**33.13**	**67.32**

Table 5. Cross-file code completion results on RepoBench XF-R (Liu et al., 2023).

Model	Python		Java	
	EM	Edit Sim	EM	Edit Sim
Transformer	7.0	38.3	5.8	34.4
GPT-2	15.5	48.2	11.6	41.4
CodeGPT	16.6	49.1	13.3	44.2
UniXcoder	18.0	53.4	16.5	50.0
LongCoder	**21.4**	**59.7**	**19.7**	**60.1**

Evaluation Metrics We report the number of parameters, inference memory consumption, runtime, Exact Match (EM) and Edit Similarity (Edit Sim) of the baselines. The inference memory consumption and runtime per example are calculated using a beam search with beam size of 5 and maximum generation length of 64 on a single V100 GPU.

Table 6. Ablation study on LongCoder without pre-training.

Model	Python		Java		C#	
	EM	Edit Sim	EM	Edit Sim	EM	Edit Sim
LongCoder	17.61	54.82	25.96	60.81	31.22	64.18
w/o memory tokens	16.80	53.90	24.87	60.35	30.16	63.61
w/o bridge tokens	17.54	51.92	25.88	57.48	30.80	59.09
w/o out-of-window context	16.66	50.63	24.17	55.97	28.81	57.79
w/ equidistant memory tokens	17.16	54.25	24.96	60.42	30.31	63.65

5.2. Training Details

We set the maximum length of code context to 512 and 4096 for non-sparse and sparse models, respectively. In order to make a fair comparison between sparse models and non-sparse models, we set the window size w to 512 so that both types of models maintain the same local context length during inference. Note that this setting is different from the original setting of RepoBench (Liu et al., 2023), thus the results are not directly comparable to those reported in Liu et al. (2023). For sparse models, we use the parameters of UniXcoder released by Guo et al. (2022) to initialize the models. For LongCoder, we set the maximum size of bridge tokens n and global tokens k as 16 and 64, respectively. To ensure fair comparison with other models, we pre-train LongCoder on the CodeSearchNet dataset using the same next token prediction objective and pre-training setting as baselines (Lu et al., 2021; Guo et al., 2022). During fine-tuning, we use the Adam optimizer with a batch size of 16 and a learning rate of 2e-4. We fine-tune the model for 10 epochs and perform early stopping on the development set. Note that although the maximum context sequence length is 4096, during inference, we only retain a cache of at most 592 tokens for past key and value hidden states to maintain efficiency in terms of computational resources.

5.3. Experimental Results

Table 2 illustrates the comparison results of LongCoder with other models on the LCC dataset. The results reveal that the sparse models (i.e., the last two groups) have superior performance compared to the non-sparse models (i.e., the second group) on both EM and Edit Sim metrics, and they also maintain a similar inference speed. LongFormer is initialized using the parameters of UniXcoder, with the sole difference being the use of a sliding window attention mechanism. This mechanism allows the model to maintain a consistent inference speed while having a larger receptive field, resulting in improved performance. This demonstrates the effectiveness of the sliding window attention mechanism in code completion tasks. Compared to other sparse models, LongCoder achieves an improvement of 0.8%–1.3% in Exact Match score and 4.0%–6.0% in Edit Similarity, which reveal the effectiveness of our proposed bridge and global attention mechanisms. Table 4 shows the result of LongCoder on CodeXGLUE code completion benchmarks. It can be observed that LongCoder achieves state-of-the-art performance, which illustrates its effectiveness in scenarios where the code context is short. As shown in Table 5, LongCoder has an even larger advantage compared to UniXcoder, indicating its potential in more complex scenarios.

5.4. Ablation Study

To better understand the impact of different components on overall performance, we conduct an ablation study on LongCoder, and the results are shown in Table 6. We can see that the average score of Exact Match drops by approximately 1% when memory tokens are removed (**w/o memory tokens**), which demonstrates the importance of these tokens. On the other hand, when bridge tokens are removed (**w/o bridge tokens**), the average score drops by about 3% in terms of Edit Similarity. This is likely because bridge tokens assist LongCoder in understanding the semantics of the code context and generating more accurate patterns, while memory tokens enable it to access concrete identifiers with global scope, thus improving accuracy on libraries, classes, and functions invoked. Additionally, we observe that selecting one as a memory token every 64 tokens (**equidistant memory tokens**) results in worse performance than LongCoder, indicating that the advantage of the memory tokens is not solely due to increased context length. We also evaluate the performance of LongCoder by only using code context within the window size during inference to verify if the improvement is solely attributed to the use of long code context or whether other factors such as fine-tuning settings also contribute. By only keeping the last 512 tokens as code context (**w/o out-of-window context**), we can see that the performance is nearly the same as UniXcoder in Table 2, which shows the importance of modeling long code context.

5.5. Case Study

We also conduct a case study to demonstrate the effectiveness of LongCoder, as shown in Figure 4. We provide two examples in the Python and Java programming languages

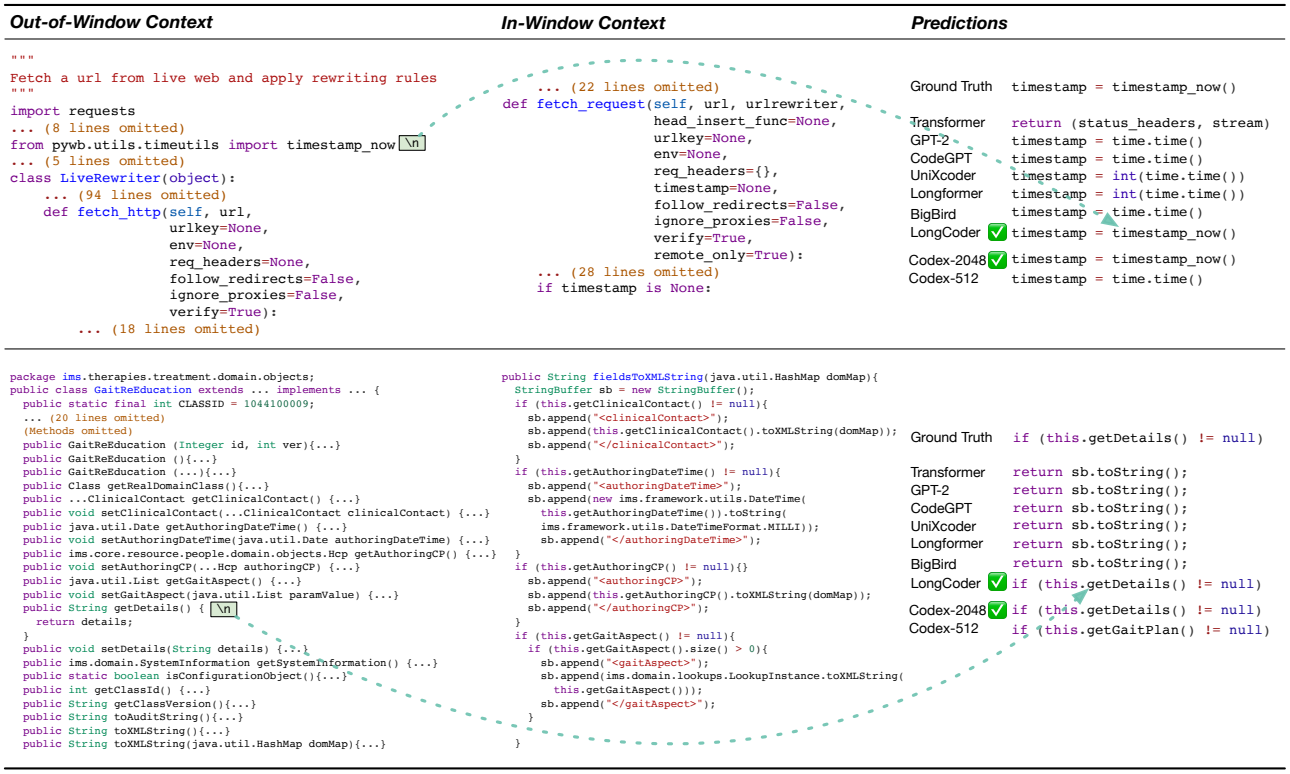

Figure 4. Two LCC examples of Python ***(top)*** and Java ***(bottom)*** code and predictions of different models. Codex-2048 refers to the original Codex model with the maximum context length of 2,048 while Codex-512 is the same model with a maximum context length set to 512. Key information is highlighted with arrows.

and output predictions from different models. (1) From the Python example, we can see that all models infer the correct intended outcome, which is to assign the current timestamp to the `timestamp` variable. However, only LongCoder and Codex-2048 produce the correct result. This is primarily because these two models are able to refer to the import statement at the beginning of the file, which imports the `timestamp_now` function. Codex-2048 uses a long context window to cover the entire file, but this approach increases memory consumption and decreases inference speed as discussed above. Additionally, as the failure of Codex-512 shows, even a large powerful model can struggle to identify the correct function from other candidates if the required information exceeds the window size. In contrast, LongCoder utilizes a more efficient memory attention mechanism, storing information based on the scope of different statements. This method is more effective, allowing access to statements from the global scope while remaining efficient. (2) In the Java example, the function to be completed aims to convert a `HashMap` variable into an XML string. The function sequentially calls the getter functions of the `GaitReEducation` class and has already completed calling the `getGaitAspect` function. From the out-window context, it is clear that the next call should be made to the `getDetails` function. In order to correctly complete the function, it is essential to keep track of all function definitions. As seen in the output results, only LongCoder and Codex-2048, which both make use of long code context, can predict the correct results. Additionally, it can be observed that Codex-512, due to its limited context, can only make a guess for a member function. We can see that LongCoder leverages the structure of the code to analyze the scope of statements and stores those that have potential long-term dependencies. This not only improves performance but also achieves comparable efficiency in terms of computational resources during inference.

6. Discussion

Limitations One limitation of LongCoder is its small size. Due to resource constraints, we are not able to train a large model that is comparable to Codex. Besides, to compare fairly with other baselines, we only pre-train LongCoder on a small-scale corpus (CodeSearchNet). It would be interesting to see how the idea of sparse attention scales with more data.

Another limitation of our work is the evaluation datasets. Many existing code datasets and LCC share the same source of data from GitHub. The same data can appear in the

pretraining data, making the evaluation less reliable. Additionally, models with larger-scale pretraining are even more likely to have seen the test data before. For example, OpenAI Codex is trained on all GitHub repositories, that undoubtedly, include most (if not all) test data in PY150, JavaCorpus, and LCC. As the code completion models have seen wider adoption in software development, future evaluation can be "self-fulfilling". For example, GitHub CoPilot[5] is a popular commercial code completion tool powered by OpenAI Codex. A lot of code generated by Codex may have already been submitted to GitHub. This could give widely-used models like Codex an advantage if it is evaluated on a dataset with a data source of the latest GitHub repositories, as we could be evaluating Codex on its own input. To address these problems, we need the community to contribute new, clean, and high-quality datasets with code from private projects to support future research.

Future Work LongCoder opens up new research opportunities in code generation not only within a large file, but also across multiple files. For example, we could allow the model to look at other files in the project for even more accurate code completion. It could enable new applications including automatically extracting package requirements, generating build files, refactoring the project, etc.

ACKNOWLEDGMENTS

Jian Yin is the corresponding author. Daya Guo and Jian Yin are supported by the National Natural Science Foundation of China (U1911203, U2001211, U22B2060), Guangdong Basic and Applied Basic Research Foundation (2019B1515130001), Key-Area Research and Development Program of Guangdong Province (2020B0101100001). We would like to thank the anonymous reviewers and the meta-reviewer for their insightful comments.

References

Allamanis, M. The adverse effects of code duplication in machine learning models of code. In *Onward!*, pp. 143–153. ACM, 2019.

Allamanis, M. and Sutton, C. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 207–216. IEEE, 2013.

Allamanis, M. and Sutton, C. Mining idioms from source code. In *SIGSOFT FSE*, pp. 472–483. ACM, 2014.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bielik, P., Raychev, V., and Vechev, M. T. PHOG: probabilistic model for code. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 2933–2942. JMLR.org, 2016.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *NeurIPS*, 2020.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Choromanski, K. M., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlós, T., Hawkins, P., Davis, J. Q., Mohiuddin, A., Kaiser, L., Belanger, D. B., Colwell, L. J., and Weller, A. Rethinking attention with performers. In *ICLR*. OpenReview.net, 2021.

Clement, C. B., Lu, S., Liu, X., Tufano, M., Drain, D., Duan, N., Sundaresan, N., and Svyatkovskiy, A. Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy. In *EMNLP*, pp. 4713–4722. Association for Computational Linguistics, 2021.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pp. 4171–4186. Association for Computational Linguistics, 2019.

Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H. Unified language model pre-training for natural language understanding and generation. In *NeurIPS*, pp. 13042–13054, 2019.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. In *ACL*, pp. 7212–7225. Association for Computational Linguistics, 2022.

Hellendoorn, V. J. and Devanbu, P. T. Are deep neural networks the best choice for modeling source code? In *ESEC/SIGSOFT FSE*, pp. 763–773. ACM, 2017.

[5] https://github.com/features/copilot

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. T. On the naturalness of software. *Commun. ACM*, 59(5): 122–131, 2016.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are rnns: Fast autoregressive transformers with linear attention. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165. PMLR, 2020.

Kitaev, N., Kaiser, L., and Levskaya, A. Reformer: The efficient transformer. In *ICLR*. OpenReview.net, 2020.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Liu, F., Li, G., Zhao, Y., and Jin, Z. Multi-task learning based pre-trained language model for code completion. In *ASE*, pp. 473–485. IEEE, 2020.

Liu, T., Xu, C., and McAuley, J. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint*, 2023.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Qin, Z., Sun, W., Deng, H., Li, D., Wei, Y., Lv, B., Yan, J., Kong, L., and Zhong, Y. cosformer: Rethinking softmax in attention. In *ICLR*. OpenReview.net, 2022.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Raychev, V., Bielik, P., and Vechev, M. T. Probabilistic model for code with decision trees. In *OOPSLA*, pp. 731–747. ACM, 2016.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: code generation using transformer. In *ESEC/SIGSOFT FSE*, pp. 1433–1443. ACM, 2020.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6): 1–28, 2022.

Tu, Z., Su, Z., and Devanbu, P. T. On the localness of software. In *SIGSOFT FSE*, pp. 269–280. ACM, 2014.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NIPS*, pp. 5998–6008, 2017.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *MAPS@PLDI*, pp. 1–10. ACM, 2022.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontañón, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.