Overlapping Context with Variable-Length Stride Increases Diversity when Training Large Language Model for Code

Geonmo Gu[*†], **Jaeho Kwak**[*†], **Haksoo Moon**[§†], **Hyun Seung Shim**[†]
Yu Jin Kim[‡], **Byoungjip Kim**[§‡], **Moontae Lee**[‡], **Hyejeong Jeon**[¶†]
[†]LG Electronics [‡]LG AI Research
{geonmo.gu, jaeho95.kwak, hs.shim, hyejeong.jeon}@lge.com
haksoo.moon@gmail.com, {yujin.kim, moontae.lee}@lgresearch.ai, byoungjip.kim@gmail.com

Abstract

The pretraining of code LLMs typically begins with general data and progresses to domain-specific data through sequential stages. In the latter stages, a challenging issue is that the data of a target domain can be limited in size, and conventional approach of increasing the number of epochs does not lead to a performance gain. In this paper, we propose a novel packing method, which is extracting overlapping contexts from the training data using variable-length stride. Our method can mitigate the data-scarcity issue by providing more diverse and abundant examples of next token prediction than non-overlapping contexts. While the training time of our approach is increased proportionally to the amount of augmented examples, we present space-efficient implementations to store overlapping contexts. Extensive experiments with real datasets show that our approach outperforms the conventional approach of controlling the number of epochs in terms of the pass@k rate.

1 Introduction

Large language models for code (code LLMs) are gaining more and more attention nowadays due to their wide applicability. After Codex (Chen et al., 2021) successfully demonstrated that LLMs are capable of generating Python codes, extensive research has been conducted to broaden their capabilities such as handling multiple programming languages (Nijkamp et al., 2023), understanding natural language instructions (Luo et al., 2024), and dealing with the infilling task (Fried et al., 2023). Code LLMs can be applied to repairing faulty code, explaining the functionality of existing code, and generating code given natural language instructions (Muennighoff et al., 2024), which together lead

[*]Equal contribution
[§]Work was done while Haksoo and Byoungjip were affiliated with LG Electronics and LG AI Research, respectively
[¶]Corresponding author

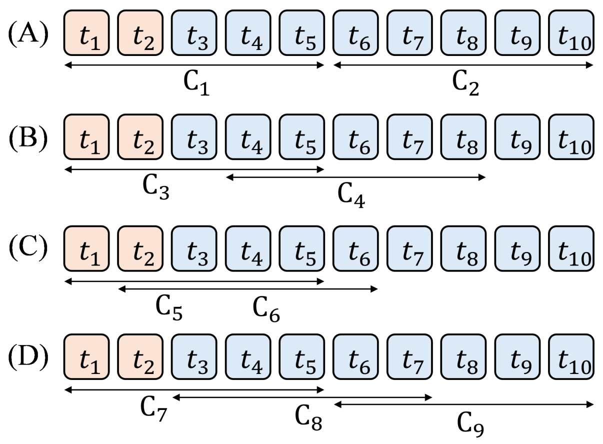
(A)
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
C_1 C_2
(B)
C_3 C_4
(C)
C_5 C_6
(D)
C_7 C_8 C_9

Figure 1: Tokens with the same color represent that they are from an identical file. (A) Non-overlapping contexts with fixed-length stride. (B) Overlapping contexts with fixed-length stride of 3. (C) Overlapping contexts with fixed-length stride of 1. Even with stride of 1, examples of next token prediction in two adjacent contexts can be different (see Example 3.1). (D) Overlapping contexts with variable-length stride, where the variable-length is determined by the end token of a file and the default value.

to increased productivity in software development (Solohubov et al., 2024; Peng et al., 2023b).

Code LLMs are continually pretrained through multiple stages from general datasets to domain-specific datasets (Nijkamp et al., 2023; Li et al., 2023; Rozière et al., 2023). In the early stages, they are trained on huge datasets (over trillion tokens) that cover diverse domains, such as code, natural language, and math. In the latter stages, they are trained on relatively small datasets from much narrower target domains, such as Python code, code review, and in-house code of a commercial company. Especially, for commercial companies that utilize open code LLMs, continually pretraining the model on their internal (private) codes is crucial for the prediction accuracy, since open code LLMs are pretrained on public source codes only.

A challenging issue in the latter stages of pretraining is that the data of a target domain is often limited and difficult to collect. For instance, it is not possible to collect more data from public sources if we aim to adapt the model to in-house data of a commercial company. A conventional way to train a model on such scarce data is increasing the number of epochs with the data at hand. However, it is empirically shown that training LLMs for more than 4 epochs with repeated data gives diminishing returns (Muennighoff et al., 2023). Thus, we need a new way to increase the prediction accuracy of code LLMs when the pretraining data is scarce.

In this paper, to address the data-scarcity issue when continually pretraining code LLMs, we propose to extract *overlapping contexts* with variable-length stride from the training data, where overlapping contexts are token sequences that can overlap. Figure 1 shows examples of non-overlapping contexts, overlapping contexts, fixed-length stride, and variable-length stride. Overlapping contexts provide more diverse and abundant examples of next token prediction, while the variable-length stride filters out less effective overlapping contexts. Combining overlapping contexts with variable-length stride leads to a higher prediction accuracy when continually pretraining code LLMs. Our contributions are summarized as follows.

- We propose a novel packing method, which is the combination of overlapping contexts and variable-length stride. We present three different implementations for overlapping contexts in terms of space complexity.
- We conduct extensive experiments to show the effectiveness of our method in the code generation task. The experiments are two fold: (1) training billion-scale code LLMs on inhouse dataset for deployment and (2) training million-scale code LLMs on public dataset for reproducibility. The experiments include different models in terms of size and structure, training datasets, benchmarks, and training settings, which together show the generalizability of our method.
- Experimental results show that utilizing overlapping contexts with variable-length stride outperforms the conventional approach of controlling the number of epochs with non-overlapping contexts in terms of the pass@k rate.

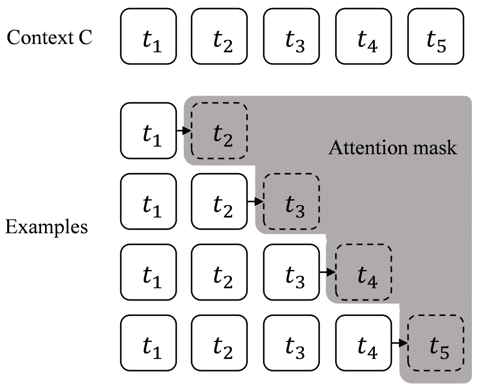

Figure 2: Examples of next token prediction in a context. The answer tokens are hidden by the attention mask during training.

2 Preliminaries

2.1 Notation

A *token* is a positive integer that represents one or more characters. A *tokenizer* maps a token to (possibly one) characters, and the mapping in the tokenizer is called the *vocabulary*. A *token sequence* is a sequence of tokens drawn from the domain of a vocabulary. For a token sequence T, $T[i:j]$ represents the continuous subsequence of T starting from the i-th token and ending at the j-th token. For a token sequence T and a positive integer l, a *context* C^l of T is a continuous subsequence of T whose length is l. For the simplicity of notation, we will omit the superscript l if the length is clear from context.

2.2 Next Token Prediction

Next token prediction is the task of predicting the next token that can appear after a given token sequence. An *example* of next token prediction is denoted by $(t_1, t_2, \dots, t_l) \rightarrow t_{l+1}$, where $(t_1, t_2, \dots, t_l)$ is the input token sequence and t_{l+1} is the answer.

LLMs are trained for the next token prediction task by minimizing the cross entropy loss of the predicted probability of next token with respect to the ground truth next token (Radford et al., 2018). For what follows, let T be the token sequence corresponding to an entire code corpus, on which we want the model to train. The probability of a context C^l in T can be expressed as the product of $l-1$ conditional probabilities as follows:

$$P(C) = \prod_{i=2}^{l} P(t_i | t_1, t_2, \dots, t_{i-1}), \qquad (1)$$

where $P(t_i | t_1, t_2, ..., t_{i-1})$ represents the probability of t_i (i.e., the next token) given the subsequence $(t_1, t_2, \ldots, t_{i-1})$ of C^l.

Notably, decoder-only transformers can process the $l-1$ examples of the next token prediction in parallel for a context C composed of l tokens (Vaswani et al., 2017). For example, consider a context $C = (t_1, t_2, t_3, t_4, t_5)$ of length five in Figure 2. There are four examples of next token prediction; $(t_1) \to t_2$, $(t_1, t_2) \to t_3$, $(t_1, t_2, t_3) \to t_4$, and $(t_1, t_2, t_3, t_4) \to t_5$. In the remainder of the paper, all examples of next token prediction in a context are considered in the same manner as in Figure 2.

2.3 Related Work

In this paper, we focus on continual pretraining code LLMs in data-scarce scenario. Extensive research has been conducted on pretraining code LLMs with an unlabeled code corpus for next token prediction (see **Code LLMs** and **Continual Pretraining Code LLMs** in Appendix A), and many effective data augmentation techniques for natural language processing have been proposed in recent years (see **Data Augmentation for NLP** in Appendix A). However, to the best of our knowledge, this paper is the first to study pretraining code LLMs in data-scarce scenario.

3 Overlapping Context

3.1 Packing

In pretraining, the training dataset is composed of contexts that do not contain padding for efficiency. A general approach of constructing such contexts from multiple code files consists of three phases: In the first phase, we convert each file into a sequence of tokens using a model-specific tokenizer; In the second phase, we concatenate all token sequences to form one long token sequence. During the second phase, special tokens indicating the beginning or the end of the file can be added between token sequences; In the third phase, we cut the long token sequence into contexts of equal length. This process is called *packing*.

3.2 Non-Overlapping Context with Fixed-Length Stride

Suppose that we are extracting contexts of length l from token sequence $T = (t_1, t_2, \cdots, t_n)$ in the third phase. A conventional approach is extracting non-overlapping contexts by adding *stride* $s = l$ to the start position of the previously extracted context. That is, the first context is $T[1:l]$, the second context is $T[l+1:2l]$, the third context is $T[2l+1:3l]$, and so on. The last chunk of T is dropped if its length is less than l. With stride l, the number of extracted contexts is $\lfloor \frac{n}{l} \rfloor$.

3.3 Overlapping Context with Fixed-Length Stride

A simple way to extract more contexts in the third phase of packing is to set the stride s to be a smaller number than the context length. *Overlapping contexts* are contexts that are extracted with stride s such that $1 \le s < l$, where l is the context length. Note that by setting $s < l$, two adjacent contexts overlap $l - s$ positions in T.

Conjecture 3.1. *Overlapping contexts with a moderate stride provide more diversity when training large language model for code.*

Here is an intuitive example of Conjecture 3.1 in the domain of coding. Suppose that a token sequence T of length 8K contains eight functions $f_1, f_2, \ldots, f_8$, and that the length of each function is 1K. T can be partitioned into four non-overlapping contexts, each with a length of 2K, capturing the four relationships between pairs $(f_1, f_2), (f_3, f_4), (f_5, f_6), (f_7, f_8)$. On the other hand, if we extract overlapping contexts from T with a stride of 1K, they can capture all four relationships above plus additional relationships between pairs $(f_2, f_3), (f_4, f_5), (f_6, f_7)$.

Lemma 3.1. *Given stride s, context length l, and a token sequence T of length n such that $1 \le s < l \le n$, we can extract $\lceil \frac{n-l+1}{s} \rceil$ overlapping contexts such that any two contexts share at most $l - s$ positions in T.*

Proof. See Appendix B. □

Lemma 3.1 means that we can extract about l times more contexts if we set $s = 1$ compared to the number of non-overlapping contexts with $s = l$.

Although two overlapping contexts share positions in T, their examples of next token prediction can be different.

Example 3.1. *Consider two overlapping contexts $C_5 = (t_1, t_2, t_3, t_4, t_5)$ and $C_6 = (t_2, t_3, t_4, t_5, t_6)$ with four tokens overlap in Figure 1. Examples of next token prediction in C_5 are $(t_1) \to t_2$, $(t_1, t_2) \to t_3$, $(t_1, t_2, t_3) \to t_4$, and $(t_1, t_2, t_3, t_4) \to t_5$. Examples in C_6 are $(t_2) \to t_3$, $(t_2, t_3) \to t_4$, $(t_2, t_3, t_4) \to t_5$, and $(t_2, t_3, t_4, t_5) \to t_6$. If*

Stride	Unique	Total	%
2048	1,263,146,560	1,265,784,967	99.79
1024	2,524,323,164	2,530,849,390	99.74
512	5,044,421,251	5,060,978,236	99.67
256	10,078,484,177	10,121,235,928	99.57

Table 1: The number of unique examples of next token prediction in the in-house dataset for varying fixed-length stride. The context length is 2048. Overlapping contexts (stride < 2048) have almost the same proportion of unique examples as that in non-overlapping contexts (stride $= 2048$).

$t_1 \neq t_2$, C_5 and C_6 do not share examples of next token prediction.

Lemma 3.2. *Consider the set of overlapping contexts in Lemma 3.1 with stride s and context length l, extracted from a token sequence T of length n. Assume that $T[i] \neq T[i+s]$ for any $1 \leq i \leq n-s$. Then no two adjacent overlapping contexts share examples of next token prediction.*

Proof. See Appendix C. □

Lemma 3.2 means that even if two adjacent overlapping contexts share tokens, they do not share the identical examples of next token prediction if the first tokens of them are different. Table 1 shows that the proportion of unique examples is over 99% even if we set stride to 256 for context length of 2048 (overlapping 1792 tokens) in real dataset.

Theorem 3.3. *Given stride s, context length l, and a token sequence T of length n such that $1 \leq s < l \leq n$, assume that $T[i] \neq T[i+s]$ for any $1 \leq i \leq n-s$. We can extract $\lceil \frac{n-l+1}{s} \rceil$ overlapping contexts such that no two adjacent contexts share examples of next token prediction.*

Proof. The proof is direct from Lemmas 3.1 and 3.2. □

Implementation Details. There are multiple implementation choices for storing overlapping-contexts. We describe three methods and compare their space usages.

- The first method is extracting overlapping contexts by sliding window and simply storing all of them in memory. This method requires memory proportional to the extracted contexts.
- The second method is extracting overlapping contexts by sliding window and storing only the start indices of them in memory. The actual contexts corresponding to the start indices are extracted during training. Since only the start indices are stored and they are not duplicated, this method requires additional memory at most twice as large as the original dataset.
- The third method is randomly sampling the start indices of the contexts during training. The random sampling method does not require additional memory, but it does not guarantee that the start indices are unique. Also, this method cannot use the variable-length stride, which will be described in the next subsection.

3.4 Overlapping Context with Variable-Length Stride

An extracted overlapping context can be less effective when the context contains tokens from multiple unrelated files (e.g., C_6 in Figure 1). To avoid extracting less effective contexts, we propose a way to set variable-length stride.

Suppose that we are extracting overlapping contexts by sliding window. Given a token sequence T, context length l, and the default stride s, let the current window is $T[i : i+l-1]$. The *variable-length stride s'* is defined as follows:

- If the current window contains at least one end token of a file, we set $s' = j - i + 1$, where j is the rightmost end-token's position.
- If the current window does not contain the end token of a file, we set $s' = s$.

This approach sets a long stride when there are multiple files in the current window, and sets a short stride when the current window is a part of a long file. Hence, we can filter out less effective overlapping contexts utilizing the variable-length stride.

4 Experimental Results

In this section we present experimental results to show the effectiveness of overlapping context in continual pretraining code LLMs in data-scarce scenario. The experiments are twofold: (1) training billion-scale LLMs on in-house dataset, which are deployed in our company, and (2) training small LLMs on public dataset for reproducibility.

Training Data. We collected an in-house code dataset, which contains hundreds of private repositories. After applying the filtering techniques introduced in previous works (Chen et al., 2021; Nijkamp et al., 2023; Kocetkov et al., 2023; Li et al., 2023), we obtained 3.67GiB (1.49 billion tokens) C/C++ codes.

For the public dataset, we use Swift codes of TheStack (Kocetkov et al., 2023). We first applied deduplication and filtering, and then extracted 10% of the remaining files. The resulting dataset contains 598MiB (180 million tokens) Swift codes.

Evaluation Data. To evaluate a model with respect to the in-house dataset, we created a benchmark dataset similar to HumanEval (Chen et al., 2021). HumanEval consists of 164 hand-written Python problems, each of which is a task of generating a function's body given the function's header and comments about the function. The generated code is considered to solve the problem if it passes all predefined unit tests. We extracted 100 functions from the in-house dataset and created tasks of generating a function's body given the function's header, together with corresponding unit tests. The resulting benchmark dataset is called U100.

To evaluate a model trained on the public dataset, we use the MultiPL-E (Cassano et al., 2023) benchmark, which is a multilingual version of HumanEval. Specifically, we use the Swift version of HumanEval.

Metric. We report the pass@k rate (Chen et al., 2021; Kulal et al., 2019), which represents the rate of solved problems when a model can speculate the answer k times for each problem.

Baseline. As mentioned in Section 2.3, our paper focuses on continual pretraining code LLMs in data-scarce scenario. In this setup, increasing the number of epochs has been the only way to achieve the performance gain so far. Therefore, our primary baseline is increasing the number of epochs with non-overlapping contexts.

Models. For the in-house dataset, we continue pretraining on EXACODE-8.8B-BASE, which is a code version of EXAONE-2.0 (Research, 2024). EXACODE-8.8B-BASE is a pretrained language model with 8.8 billion parameters trained on ThePile (Gao et al., 2020), TheStack (Kocetkov et al., 2023), and extra natural language dataset (459 billion training tokens in total). We compare the following models.

- EXACODE-8.8B-OC: it is initialized with the weights of EXACODE-8.8B-BASE, and continually pretrained full-parameter on the in-house code dataset. It uses overlapping contexts with a fixed-length stride.
- EXACODE-8.8B-NOC: it has the same training setting as that of EXACODE-8.8B-OC except that it uses non-overlapping contexts.

For the public dataset, we continue pretraining on CodeGen-350M-Multi (Nijkamp et al., 2023). We chose this model because its size is suitable for conducting ablation studies. Also, for efficient training, we use LoRA (Hu et al., 2022) in such a way that the percentage of the trainable parameters becomes 8%.

The detailed hyperparameter settings will be presented in Appendix D. We also apply our method to CodeLlama (Rozière et al., 2023) in Appendices E and F, where the effect of overlapping contexts on CodeLlama is similar to that on EXACODE. A number of additional experiments are also performed: effect of varying fixed-length stride and batch size (Appendix F), applying the mix-review strategy to alleviate the forgetting problem (Appendix G), counting the number of unique examples with different context length (Appendix H), and counting the number of unique examples with the random sample method (Appendix I).

4.1 Unique Examples in the In-House Dataset

Table 1 shows the number of unique examples of next token prediction in the in-house dataset for varying values of fixed-length stride. Here, the context length is 2048, and thus stride $= 2048$ means that there is no overlap between any two contexts. Although there is no overlap, the number of unique examples is slightly smaller (99.79%) than the total number of examples because short examples tend to have duplicates throughout the dataset.

Even when two adjacent contexts overlap, the proportion of unique examples remains high (99.57%) until stride $= 256$. That is, we can have almost 8 times more unique examples than non-overlapping contexts.

4.2 Evaluation on U100

To see the effect of overlapping contexts in terms of prediction accuracy on the in-house dataset, we compare the two approaches: (1) training 2 epochs utilizing the overlapping contexts with fixed-length stride of 256, and (2) training 8 epochs utilizing the non-overlapping contexts. We report the pass@1 rate, where the beam search with 2 beams is used for the decoding strategy.

Figure 3 shows the pass@1 rate of the two approaches. Utilizing the overlapping contexts, EXACODE-8.8B-OC outperforms EXACODE-8.8B-NOC in terms of pass@1. Specifically, the best pass@1 rate of EXACODE-8.8B-OC is 29% and the

Context Length	Stride	Stride Type	Learning Rate	Batch Size	Epoch	Training Step	Pass@1
1024	128	Variable-Length	2e-4	512	2	3302	7.9
1024	128	Fixed-Length	2e-4	512	2	5506	6.5
1024	1024	Fixed-Length	2e-4	512	10	3440	6.2
1024	1024	Fixed-Length	2e-4	512	20	6880	6.2
1024	1024	Fixed-Length	1e-3	512	2	688	5.0

Table 2: The pass@1 rate of `MultiPL-E` Swift for the `CodeGen-350M` models trained on the public dataset.

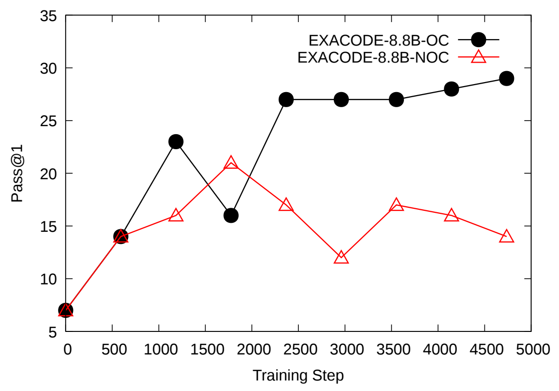

Figure 3: Training billion-scale LLMs on the in-house dataset. `EXACODE-8.8B-OC` (overlapping contexts) outperforms `EXACODE-8.8B-NOC` (non-overlapping contexts) in terms of the pass@1 rate.

best one of `EXACODE-8.8B-NOC` is 21%. The improved performance supports that overlapping contexts can increase diversity when training code LLMs (Conjecture 3.1). The declining performance of `EXACODE-8.8B-NOC` after the 3rd epoch represents a sign of overfitting. In contrast, the performance of `EXACODE-8.8B-OC` increases until 4736 steps (corresponding to 8 epochs of `EXACODE-8.8B-NOC`) without signs of overfitting. This is because the overlapping contexts provide an enlarged training datasets, in which duplicate examples are less than 1%.

4.3 Evaluation on MultiPL-E Swift

For the reproducible work, we evaluate our techniques on the public dataset (described in **Training Data**). This experiment also shows the generalizability of our approach because we use different models, datasets, benchmarks, and train settings.

Table 2 shows various versions of `CodeGen-350M` trained on the Swift codes of `TheStack`. For each version, we saved checkpoint for every 10% training step, and report the average pass@1 rate of top-3 checkpoints.

The version that utilizes overlapping contexts with variable-length stride outperforms all non-overlapping versions in terms of the pass@1 rate. The performance of the version that utilizes only the overlapping contexts is marginally better than the non-overlapping versions, which implies that there are less effective contexts in the overlapping contexts if we do not utilize the variable-length stride.

To see if a higher learning rate or a large number of epochs can mitigate the data-scarcity issue, we compare the non-overlapping version with 20 epochs and the version with learning rate of 1e-3. Nevertheless, the performances of these versions are similar or worse than that with less number of epochs and smaller learning rate.

5 Discussion

In this paper we defined the variable-length stride by the end token of a file. However, there can be different definitions of the variable-length stride. For instance, one can apply *dependency parsing* (Guo et al., 2024) to group and to order files in a repository, and define the variable-length stride by the end token of a code repository. One can also define the variable-length stride by the end token of a paragraph in natural language dataset.

Regarding natural language dataset, although overlapping contexts are shown to be effective for code datasets, it is not guaranteed that overlapping contexts will be equally effective for natural language datasets because their characteristics are different. For example, code corpora are more repetitive and predictable (Casalnuovo et al., 2019), and they have longer context than natural language corpora, which can make overlapping contexts more beneficial for code than for natural language.

6 Conclusion

In this paper we have introduced a new packing method utilizing overlapping contexts with variable-length stride. Our method is useful for continual pretraining code LLMs when the amount of training dataset is insufficient. Extensive experiments on the in-house dataset and the public dataset have demonstrated the effectiveness of our method in terms of the pass@k rate. Applying overlapping contexts to natural language dataset is an interesting future work.

Limitations

As more contexts are extracted in overlapping contexts compared to non-overlapping contexts, the overlapping context method increases training time proportionally to the amount of additional contexts.

Ethics Statement

In the experiments, we use `CodeGen` (Nijkamp et al., 2023) in Section 4 and `CodeLlama` (Rozière et al., 2023) in Appendix E, which are open-source models. Our use of `CodeGen` and `CodeLlama` is consistent with their licenses and acceptable use policies. We do not see any potential risks derived from our work.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, and 22 others. 2023. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Nitay Calderon, Eyal Ben-David, Amir Feder, and Roi Reichart. 2022. DoCoGen: Domain counterfactual generation for low resource domain adaptation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*.

Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2019. Studying the difference between natural and programming language corpora. *Empirical Software Engineering*, 24:1823–1868.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. In *International Conference on Learning Representations*.

Yanbing Chen, Ruilin Wang, Zihao Yang, Lavender Yao Jiang, and Eric Karl Oermann. 2024. Refining packing and shuffling strategies for enhanced performance in generative language models. *arXiv preprint arXiv:2408.09621*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Sreyan Ghosh, Chandra Kiran Evuru, Sonal Kumar, S Ramaneswaran, S Sakshi, Utkarsh Tyagi, and Dinesh Manocha. 2023. DALE: Generative data augmentation for low-resource legal nlp. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. 2024. Better & faster large language models via multi-token prediction. In *Forty-first International Conference on Machine Learning*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the large language model meets programming - The rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Tianxing He, Jun Liu, Kyunghyun Cho, Myle Ott, Bing Liu, James Glass, and Fuchun Peng. 2021. Analyzing the forgetting problem in pretrain-finetuning of open-domain dialogue response models. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The Stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*, volume 32.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, and Jenny Chim, et al. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. StarCoder 2 and The Stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering code large language models with evol-instruct. In *International Conference on Learning Representations*.

Michael McCloskey and Neal J. Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of Learning and Motivation*, volume 24, pages 109–165.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction tuning code large language models. In *International Conference on Learning Representations*.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. 2023. Scaling data-constrained language models. In *Advances in Neural Information Processing Systems*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023a. YaRN: Efficient context window extension of large language models. In *International Conference on Learning Representations*.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023b. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

LG AI Research. 2024. EXAONE 3.5: Series of large language models for real-world use cases. *arXiv preprint arXiv:2412.04862*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2023. Are emergent abilities of large language models a mirage? In *Advances in Neural Information Processing Systems*.

Illia Solohubov, Artur Moroz, Mariia Yu Tiahunova, Halyna H Kyrychek, and Stepan Skrupsky. 2024. Accelerating software development with AI: exploring the impact of ChatGPT and GitHub Copilot. In *CEUR Workshop Proceedings (2024, in press)*, pages 76–86.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C.H. Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Jason Wei and Kai Zou. 2019. EDA: Easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Chenxi Whitehouse, Monojit Choudhury, and Alham Fikri Aji. 2023. LLM-powered data augmentation for enhanced crosslingual performance. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860.

Yu Zhao, Yuanbin Qu, Konrad Staniszewski, Szymon Tworkowski, Wei Liu, Piotr Miłoś, Yuxiang Wu, and Pasquale Minervini. 2024. Analysing the impact of sequence composition on language model pretraining. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023a. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X. pages 5673–5684.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Dawei Zhu, Nan Yang, Liang Wang, Yifan Song, Wenhao Wu, Furu Wei, and Sujian Li. 2023. PoSE: Efficient context window extension of LLMs via positional skip-wise training. In *International Conference on Learning Representations*.

A Extended Related Work

Code LLMs. Large language models for code (Chen et al., 2021; Nijkamp et al., 2023; Zheng et al., 2023a; Li et al., 2023; Wang et al., 2023; Allal et al., 2023; Guo et al., 2024) are based on transformer-decoder architecture that generates next tokens autoregressively from given prompt tokens. Code LLMs are pretrained on large unlabeled code corpus for the next token prediction objective. Notably, `Codex` (Chen et al., 2021) is a decoder-only model for Python code generation, which is released with an evaluation set called `HumanEval`. `StarCoder` (Li et al., 2023) and `CodeLlama` (Rozière et al., 2023) are decoder-only models that are trained on permissively licensed code datasets, allowing companies to use them without concerns about licensing issues.

Some models are based on encoder-decoder transformer and are trained with different pretraining objectives. `CodeT5+` (Wang et al., 2023) is an encoder-decoder model that is trained with mixture of pretraining objectives, including span denoising, contrastive learning, text-code matching, and next token prediction. `CodeRL` (Le et al., 2022) is a framework of training code LLMs by reinforcement learning that utilizes compilation results and unit test results. Recently, a multi-token prediction model architecture (Gloeckle et al., 2024) is introduced, which offers faster inference than next-token prediction architecture and also offers higher accuracy on coding evaluation benchmarks as the model size increases.

We refer the reader to (Zheng et al., 2023b; Jiang et al., 2024) for comprehensive surveys of code LLMs.

Continual Pretraining Code LLMs. Code LLMs are pretrained through multiple stages (Nijkamp et al., 2023; Rozière et al., 2023; Gururangan et al., 2020). Initially, they are pretrained on a large-scale general corpus. Then, they are continually pretrained on a subset of the corpus seen in the previous stage or a specific target corpus (such as a private in-house dataset) that has not been previously seen. For instance, `CodeLlama-Python` (Rozière et al., 2023) is first trained on 2 trillion tokens from natural language, code, and math datasets (Touvron et al., 2023). It is then trained on 500 billion tokens from code-heavy dataset that covers multiple programming languages, and lastly, it is pretrained on 100 billion tokens of a Python-heavy dataset, followed by long context fine-tuning on 20 billion tokens.

In the domain of code, the prompt can include related code files and detailed instructions, and the model can output an entire function or a class definition (Guo et al., 2024; Lozhkov et al., 2024). Therefore, code LLMs must handle a relatively long context length. In general, code LLMs are first pretrained on a large-scale corpus with contexts of moderate length, and then they are fine-tuned for long contexts (Zhu et al., 2023; Peng et al., 2023a; Su et al., 2024; Chen et al., 2023).

Data Augmentation for NLP. Data scarcity is common in natural language processing (NLP) both for pretraining and for fine-tuning. Muennighoff et al. (Muennighoff et al., 2023) conducted a study on scaling LLMs for NLP in data-constrained regimes. They quantify the impact of multiple epochs in LLM training and empirically validate that training for more than 4 epochs with repeated data gives diminishing returns (i.e., the loss does not decrease as much as having unique data). To mitigate data scarcity, they propose code augmentation for natural language tasks. They observed that filling up to 50% of data with code shows no deterioration, but beyond that, performance decreases quickly on natural language tasks.

For downstream NLP tasks (e.g., summarization, translation), models are typically trained on labeled dataset. Collecting labeled datasets can be costly, especially when human annotators are involved. Extensive research has been conducted to collect or augment labeled dataset using techniques such as word insertion, deletion, substitution, and leveraging pretrained language models to generate new examples or paraphrasing existing ones (Wei and Zou, 2019; Calderon et al., 2022; Whitehouse et al., 2023; Ghosh et al., 2023). However, applying these techniques to pretraining code LLMs can be challenging because they are specifically designed for natural languages, have different training objectives than next token prediction, and are tailored to transformer-encoder models.

Packing. Recently, the impact of packing strategy on pretraining LLMs has been explored (Zhao et al., 2024; Chen et al., 2024). If the lengths of files are shorter than the context length, the context may be composed of several irrelevant files, and the inclusion of distracting information can degrade the performance of the models. In this case, all `UniChunk`, `BM25Chunk`, and `Intra-Document Causal Masking` methods (Zhao et al., 2024) can improve in-context learning, knowledge memorization, and context utilization abilities of language models.

On the other hand, if the lengths of files are longer than the context length, a file may be divided into several contexts. These correlated contexts are separated while shuffling and are put into different batches if the unit of data shuffled is one. The impact of various unit sizes is also explored (Chen et al., 2024).

B Proof of Lemma 3.1

Proof. Let $\mathcal{C}$ be the set of contexts $T[i : i + l - 1]$ for $1 \leq i \leq n - l + 1$ such that $i - 1$ is a multiple of s. Any two contexts in $\mathcal{C}$ overlap at most $l - s$ positions in T because $i - 1$ is a multiple of s. The number of contexts in $\mathcal{C}$ is $\lceil \frac{n-l+1}{s} \rceil$, which is the number of positions i in T such that $i - 1$ is divisible by s. Therefore, $\mathcal{C}$ contains the overlapping contexts of the lemma. □

C Proof of Lemma 3.2

Proof. We prove by contradiction. Assume that there is an identical example of next token prediction between two adjacent overlapping contexts $C_i = T[i : i + l - 1]$ and $C_{i+s} = T[i + s : i + s + l - 1]$.

By the assumption of the lemma, $T[i]$ and $T[i + s]$ are different, and thus C_i and C_{i+s} do not have a common prefix. However, in order for C_i and C_{i+s} to have an identical example, there must be a common prefix between C_i and C_{i+s}, which is a contradiction. □

D Hyperparameter Settings

For training the in-house dataset, we use the AdamW (Loshchilov and Hutter, 2019) optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = $ 1e-8. We use the cosine decay learning rate scheduler that gradually decreases the learning rate to 10% of its maximum value after 175 wamup steps. The maximum learning rate is 1e-4 for the `CodeLlama-7B` models and 1.6e-5 for the `EXACODE-8.8B` models. The default context length is 2048. We use different combinations of stride and batch size for diverse comparison. For training the public dataset, we use the constant learning rate scheduler with learning rate of 2e-4 and set context length to 1024 by default.

For all models, we applied mixed precision training using bfloat16 to speed up the training. For `EXACODE-8.8B` and `CodeLlama-7B` models, we conducted full-parameter training on 64 A100-40GB GPUs using FSDP (Zhao et al., 2023) with

Model	Stride	Batch Size	#Train Tokens	Training Step 0	592	1184	1776	2368	2960	3552	4144	4736
				Pass@1								
EXACODE-8.8B-NOC	2048	1024	2.98B	7.00	14.00	**16.00**	-	-	-	-	-	-
EXACODE-8.8B-OC	1024	1024	5.96B	7.00	**16.00**	14.00	14.00	15.00	-	-	-	-
EXACODE-8.8B-OC	512	2048	11.92B	7.00	15.00	14.00	**19.00**	**19.00**	-	-	-	-
EXACODE-8.8B-OC	256	2048	23.84B	7.00	14.00	23.00	16.00	27.00	27.00	27.00	28.00	**29.00**
				ROUGE								
EXACODE-8.8B-NOC	2048	1024	2.98B	35.47	37.67	**39.98**	-	-	-	-	-	-
EXACODE-8.8B-OC	1024	1024	5.96B	35.47	38.45	38.02	**38.92**	38.88	-	-	-	-
EXACODE-8.8B-OC	512	2048	11.92B	35.47	38.13	38.51	40.30	**41.82**	-	-	-	-
EXACODE-8.8B-OC	256	2048	23.84B	35.47	37.65	43.07	40.47	44.45	45.65	**47.04**	46.46	46.67

Table 3: The `U100` pass@1 rate and ROUGE score of the `EXACODE-8.8B` models for varying numbers of training steps. The context length is fixed to 2048. All models are trained for 2 epochs. The bold fonts indicate the highest score among checkpoints for each model.

Model	Stride	Batch Size	#Train Tokens	Epoch 0.0	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
				Pass@1										
CodeLlama-7B-NOC	2048	8192	2.86B	2.00	2.00	2.00	3.00	6.00	**7.00**	5.00	6.00	6.00	5.00	5.00
CodeLlama-7B-OC	1024	8192	5.72B	2.00	2.00	7.00	6.00	10.00	5.00	5.00	**13.00**	9.00	8.00	8.00
CodeLlama-7B-OC	512	8192	11.44B	2.00	4.00	7.00	14.00	22.00	21.00	**31.00**	20.00	18.00	19.00	18.00
				ROUGE										
CodeLlama-7B-NOC	2048	8192	2.86B	20.24	20.80	27.89	29.68	32.85	32.02	**34.37**	33.80	34.14	33.54	33.71
CodeLlama-7B-OC	1024	8192	5.72B	20.24	24.07	29.64	33.41	35.21	39.54	36.60	**40.80**	40.70	40.21	40.23
CodeLlama-7B-OC	512	8192	11.44B	20.24	29.75	36.38	39.92	42.76	46.01	43.36	46.15	44.61	46.47	**47.09**

Table 4: The `U100` pass@1 rate and ROUGE score of the `CodeLlama-7B` models for varying numbers of epochs. The context length is fixed to 2048. The bold fonts indicate the highest score among checkpoints for each model.

the full sharding strategy. The `CodeGen-350M` models are trained on one A100-80GB GPU using LoRA (Hu et al., 2022). We set $r = 256$, $\alpha = 512$, and adapted the query, key, value, and out projection matrices.

E Applying Overlapping Context to CodeLlama

In addition to the `EXACODE-8.8B` models described in Section 4, we compare the following models based on `CodeLlama-7B-BASE`[1] (Rozière et al., 2023):

- `CodeLlama-7B-OC`: it is initialized with the weights of `CodeLlama-7B-BASE`, and trained on the mixed dataset of the in-house dataset and `TheStack`. It uses overlapping contexts with a fixed-length stride.
- `CodeLlama-7B-NOC`: it has the same training setting as that of `CodeLlama-7B-OC` except that it uses non-overlapping contexts.

[1]https://huggingface.co/codellama/CodeLlama-7b-hf

In order to alleviate the forgetting problem that can occur in sequential pretraining, we apply data mixing similar to the *mix-review* strategy (He et al., 2021). Specifically, we mix the in-house dataset with a random subset of C/C++ codes of `TheStack` so that `TheStack` constitutes 10% of the resulting training dataset.

F Varying Stride and Batch Size

We use different combinations of fixed-length stride and batch size for two model structures, `CodeLlama` and `EXACODE`. Overlapping context offers an enlarged dataset that contains diverse examples, which allow us to increase the batch size while maintaining the number of training steps.

In this subsection we use the ROUGE score for an additional metric. While pass@k is a good metric for evaluating the functionality of the generated code, it is discontinuous metric and thus it makes the performance of the model to appear sharp and unpredictable (Schaeffer et al., 2023). Thus, we also report the ROUGE-1 F1 score (Lin, 2004) between ground truth function body and the generated

Model	Context Length	Stride	Batch Size	Epoch or Step	Pass@1	Pass@10
CodeLlama-7B-BASE	-	-	-	-	29.23	57.39
CodeLlama-7B-NOC	2048	2048	8192	1.0	29.67	57.45
CodeLlama-7B-OC	2048	1024	8192	1.4	30.25	58.29
CodeLlama-7B-OC	2048	512	8192	1.2	31.46	58.30
EXACODE-8.8B-BASE	-	-	-	-	17.98	31.84
EXACODE-8.8B-NOC	2048	2048	1024	1184	18.37	33.65
EXACODE-8.8B-OC	2048	1024	1024	592	18.88	34.00
EXACODE-8.8B-OC	2048	512	2048	2368	18.39	33.20
EXACODE-8.8B-OC	2048	256	2048	4736	17.85	32.37

Table 5: Accessing the degree of forgetting with `HumanEval-X` C++. The pass@k rates of the `CodeLlma-7B` models consistently increase as the stride decreases due to the mix-review strategy.

function body.

Table 4 shows the pass@1 rate and the ROUGE score of the `CodeLlama-7B` models for `U100`. The general trend is that both pass@1 rates and ROUGE scores increase as we decrease the stride. Specifically, the highest pass@1 rates of the `CodeLlama-7B` models are 7%, 13%, 31% for strides of 2048, 1024, 512, respectively. The highest ROUGE scores are 34.37, 40.80, 47.09 for strides of 2048, 1024, 512, respectively. `CodeLlama-7B-OC` with stride=512 outperforms `CodeLlama-7B-NOC` by an absolute 24% pass@1 rate and by an absolute 12.72 ROUGE score.

Tables 3 shows the pass@1 rate and the ROUGE score of the `EXACODE-8B` models for `U100`. The trend that the model performs better with a lower stride is similar to that shown in the `CodeLlama-7B` models. Comparing the combinations of $(\text{stride, batch size}) \in \{(1024, 1024), (512, 2048)\}$, we can see that the increased batch size leads to better performances in terms of the pass@1 rate and the ROUGE score.

G Applying Mix-Review Strategy to Alleviate Forgetting

Sequential training of language models can cause the forgetting problem (McCloskey and Cohen, 1989). To assess the degree of forgetting, we evaluate the `EXACODE-8.8B` and `CodeLlma-7B` models on `HumanEval-X` (Zheng et al., 2023a), which is a multilingual version of `HumanEval`. Since our training dataset contains only C/C++ codes, we measure the performances for the C++ language of `HumanEval-X`. We generate 200 samples for each problem using the top-p sampling (Holtzman et al., 2020) with $p = 0.95$, and report the pass@1 and pass@10 rates. We use two sampling temperatures, 0.2 and 0.6, and report the highest pass@k rate among the results.

Context Length	Stride	Unique / Total (%)
1024	1024	99.61
1024	512	99.50
1024	256	99.35
512	512	99.12
512	256	98.88
512	128	98.54

Table 6: The number of unique examples of next token prediction in the in-house dataset for varying context length and stride.

Recall that we applied the mix-review strategy when training the `CodeLlama-7B` models by mixing the random subset of C/C++ codes of `TheStack` (i.e., general source codes), whereas we used only the in-house dataset when training the `EXACODE-8.8B` models. Thus, we can see the effect of the mix-review strategy on the forgetting problem by comparing the `CodeLlama-7B` models against the `EXACODE-8.8B` models.

For each model in Tables 3 and 4, we select the best checkpoint whose `U100` pass@1 rate is the highest (i.e., the most optimized models to the in-house dataset). Table 5 shows the pass@1 and pass@10 rates of the best checkpoints. For the `CodeLlama-7B` models, the pass@k rates consistently increase as we decrease the stride. However, for the `EXACODE-8.8B` models, the pass@k rates reach the peak at stride of 1024 and then decline as we decrease the stride. Therefore, mixing in general source codes is beneficial to alleviate the forgetting problem when continual pretraining code LLMs on a domain-specific dataset.

H Unique Examples with Different Context Length

When reducing the context length from 2048 to 1024 and 512 on the in-house dataset, it results in a marginally lower unique ratio as shown in Table 6.

Context Length	Stride	Unique / Total (%)
2048	2048	99.81
2048	1024	99.73
2048	512	99.61

Table 7: The number of unique examples of next token prediction in the in-house dataset with the random sampling method.

Nevertheless, the unique ratio remains above 98%.

However, it is difficult to predict whether a lower context length will affect the final outcome because not only the number of unique examples but also the context length itself can affect the final outcome. For example, reducing the context length to 512 results in failures of some problems in HumanEval because the context length must be longer than 600 in order to solve all problems in HumanEval.

I Unique Examples with Random Sampling Method

Table 7 shows the unique ratio on the in-house dataset with the random sampling method presented in Section 3. Although in theory the randomly extracted indices are not guaranteed to be unique, empirically the unique ratio of the random sampling method is similar to that of the deterministic method (see Table 1).