Beyond Sequences: Two-dimensional Representation and Dependency Encoding for Code Generation

Xiangyu Zhang[1], Yu Zhou[1,*], Guang Yang[1], Wei Cheng[1], Taolue Chen[2,*]
[1]Nanjing University of Aeronautics and Astronautics, [2]Birkbeck, University of London
{zhangx1angyu, zhouyu, yang.guang, chengweii}@nuaa.edu.cn
t.chen@bbk.ac.uk

Abstract

The advent of large language models has significantly advanced automatic code generation, transforming the way programmers writing code. Inspired by natural language processing, mainstream code generation approaches represent code as a linear sequence of tokens. In this paper, we propose to represent code snippets as two-dimensional entities, where both code lines and tokens within lines are explicitly modeled. This representation allows us to capture the hierarchical and spatial structure of code, especially the dependencies between code lines. Our method CoDE introduces a dependency encoding approach that leverages dictionary learning to perform semantic matching between code lines. As such, it avoids the reliance on strict position indices, leading to better generalization to code with diverse context and lengths. We thoroughly evaluate CoDE based on four categories of tasks. The experimental results showcase its generalizability, context understanding and retrieval, as well as interpretability in code generation.

1 Introduction

Fueled by the rapid advancement of natural language processing (NLP), code language models (CLMs, (Chen et al., 2021; Li et al., 2022; Roziere et al., 2023)) have made impressive progress recently. These models exhibit strong programming capabilities, markedly enhancing developers' productivity (Xu et al., 2022).

As the backbone of most SOTA language models (LMs), the Transformer architecture (Vaswani, 2017) relies on the attention mechanism (Bahdanau, 2014) which, by design, cannot perceive positional information within sequences, but treats tokens as unordered sets. Clearly, the order between tokens, to a certain extent, carries important information. As such, positional encodings (Sukhbaatar et al., 2015) were thus introduced and incorporated in the Transformer architecture.

Despite their ubiquity in LMs (Touvron et al., 2023; Bai et al., 2023a; Bi et al., 2024), positional encodings may compromise generalization. First, explicit positional encodings inherently sacrifice permutation invariance, as it imposes a strict ordering on the input sequence. While this design choice is well-suited for some NLP tasks in which token order carries semantic significance, it does not generalize well for tasks involving more structured data, where the semantics is often invariant to the input order. Second, positional encodings are hard to generalize to longer sequences (Zhao et al., 2023; Li et al., 2023b; Kazemnejad et al., 2024), as the typically restrict position indices to a fixed range during training. As a result, when encountering sequences exceeding the length of those observed during training, the learned positional encodings often fail to generalize effectively (Huang et al., 2023).

These two issues are more pronounced as far as code generation is concerned. Many code fragments, e.g., functions, classes or independent modules, exhibit semantic permutation invariance, i.e., reordering these elements within a codebase largely leaves their underlying semantics unchanged. However, CLMs relying on fixed positional encodings struggle to capture such invariances (Chen et al., 2024a), making them ill-suited to accurately understand the code semantics. That being said, tokens within source code also exhibit sequential relationships (e.g., variables must be defined before being used), indicating that discarding positional encoding entirely is not an ideal solution either. Rather, it is necessary to take a more flexible, structured, coarse-grained perspective on token relationship, which can balance global invariances and local sequential constraints.

Moreover, positional information that lacks semantic grounding is inherently non-robust. For

[*] Corresponding author.

example, source code typically includes non-functional elements such as comments and line breaks, which enhance readability, but which in most cases do not directly contribute to program logic. Consequently, such elements may interfere with the position indices, making positional encoding more challenging and less reliable.

Another challenge arises from the high variability in code lengths, particularly in repository-level code generation (Zhang et al., 2023a; Pan et al., 2024). In large-scale codebases, crucial context is often fragmented and distributed across multiple files, residing not only within the current file but also in external files such as imported modules or API documentation. Traditional positional encoding methods, which primarily focus on local information within a sequence, are insufficient for fusing long-range dependencies effectively.

Two-dimensional (2D) encoding for code. Current CLMs predominately treat code snippets as plain text (Jiang et al., 2024; Zhang et al., 2024a). While this approach facilitates a more straightforward adaptation of techniques from NLP, it overlooks the hierarchical and modular structure in source code which is, arguably, "unnatural." In programming practice, when writing or understanding code, developers focus more on the dependencies between lines of code rather than the specific position of individual tokens. From this perspective, representing code in two dimensions carries far more significance than the one-dimensional positioning of tokens.

In this paper, we propose to represent code as a two-dimensional structure, organized into lines of code (the vertical dimension) and tokens within each line (the horizontal dimension). Conceptually, the vertical dimension encodes the logical flow of the program across lines, such as control structures, function declarations, and inter-line dependencies, while the horizontal dimension captures fine-grained token-level operations and data manipulations. This representation aligns closely with how developers naturally read, comprehend and write code, reflecting its structural organization.

From positional encoding to CoDE. The transition from a linear to a two-dimensional code representation necessitates a rethinking of positional encoding. While the explicit two-dimensional positional information can enhance structural awareness, it does not address the generalization problem rooted in its reliance on fixed positional indices. To address this limitation, we prioritize the modeling of line-wise dependencies in code snippets. More concretely, we decompose a (multi-layer) Transformer model into two functional components. In the shallower layers of the model, we extract semantic representations for each code line. In the deeper layers, these representations are integrated by unifying the token embeddings within each line based on their semantics. The unified embeddings are then leveraged to model and capture the dependencies between lines of code.

We draw inspiration from the recent advances in mechanistic interpretability, e.g., techniques that utilize Sparse Autoencoders (SAEs) (Ng et al., 2011) to extract polysemantic and interpretable features from the hidden states of LMs (Cunningham et al., 2023; Gao et al., 2024; Lieberum et al., 2024). These extracted features collectively form a dictionary, which serves as a structured representation. In our approach, we harness SAE to identify dependencies between code lines, which can capture a combination of distinct relational features for programming, such as sequential dependencies, conditional relationships and function calls.

Benefits. The lens of 2D structured code and the associated SAE-based semantics matching gives rise to a novel neural code generation method Code Dependency Encoding (CoDE). By emphasizing dependencies rather than explicit positional information, CoDE mitigates the overly reliance on actual positions within the current CLMs. First, it achieves a form of "soft" permutation invariance within code lines through in-line modeling, and aligns line-wise dependencies based on semantic matching. This enables the model to effectively capture semantic relationships across code lines, even when their positions differ from those encountered during training (e.g., being swapped). Second, by incorporating dependency encoding between code lines, CoDE avoids the use of fixed positional indices, thereby facilitating extrapolation. Third, CoDE demonstrates the capability to effectively handle long-range dependencies in code generation tasks. Unlike traditional positional encodings, which are primarily designed for NLP tasks and tend to focus on local information, CoDE explicitly models inter-line relationships across the entire context, hence can accurately capture dependencies between code lines, even when they span diverse and distributed parts of the codebase. This is crucial for emerging repository-level code generation (Zhang et al., 2023a; Wang et al., 2024a).

Experiments. We evaluate CoDE across four categories of tasks, viz. code modeling, long-context understanding, functional correctness and context retrieval. The experimental results have confirmed the superiority of CoDE in various code-related tasks. For instance, when trained with sequence length 512 and tested on longer code snippets (1,024–4,096 tokens) CoDE achieves lower perplexity and higher accuracy, highlighting its extrapolation capability. For functional correctness of the generated code, CoDE consistently show the highest pass@1 and pass@10 rate on widely adopted HumanEval/HumanEvalPlus benchmarks. Furthermore, a context retrieval task designed to assess long-context dependency utilization shows that CoDE significantly outperforms baseline methods, indicating its effectiveness in code-related tasks.

Structure. Section 2 presents the background. Section 3 describes the proposed approach. Section 4 presents the experimental design and Section 5 reports the results. Section 6 reviews the related work. Section 7 concludes the paper.

2 Background

2.1 Attention and Positional Encodings

The Transformer model relies on self-attention mechanisms, where positional encoding serves as a crucial auxiliary component, enabling the model to differentiate between tokens at various positions.

For a sequence $T = t_1, \ldots, t_m$ of tokens with corresponding (column) vector embeddings $X = x_1, \ldots, x_m$ where each $x_i \in \mathbb{R}^d$. For any x_i, x_j with $1 \leq i, j \leq m$, the raw attention score is computed as

$$a_{i,j} = q_i^T k_j + M_{i,j},$$

where $q_i = \mathbf{W_Q} x_i$ and $k_j = \mathbf{W_K} x_j$ are commonly referred to as the query and key projections of x_i and x_j, respectively. Here, $\mathbf{W_Q}, \mathbf{W_K} \in \mathbb{R}^{H \times d}$ are learnable weight matrices, and $M_{i,j}$ represents the entry of the attention mask $M \in \mathbb{R}^{m \times m}$, which captures the constraints on attention computation by specifying which tokens the model is allowed to attend to.

To encode the sequential structure of the input, the positional encoding p_i is usually added to the token embeddings $x_i \leftarrow x_i + p_i$, for $1 \leq i \leq m$.

2.2 Sparse Autoencoder

The Sparse Autoencoder (SAE) (Ng et al., 2011) is an unsupervised learning algorithm designed to extract explainable features from data, based on

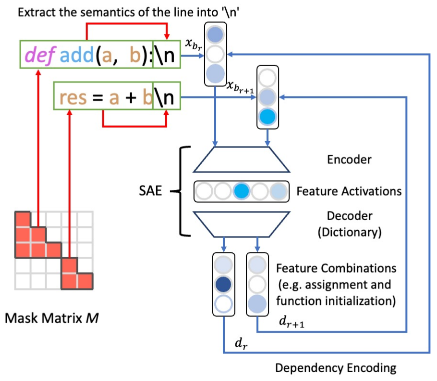

Figure 1: Illustration of CoDE. The red arrows indicate operations in the shallow layers, designed to independently extract the semantic of individual lines of code. The blue arrows in the deeper layers, use these semantic to model the dependency between lines of code.

the principles of dictionary learning (Cunningham et al., 2023; Gao et al., 2024). SAE comprises an encoder that computes feature activation and an overcomplete decoder, often referred to as a dictionary.By imposing a sparsity constraint on the feature activation, SAE learns interpretable, overcomplete and polysemantic feature representations. Formally, given an input vector x, SAE generates a reconstruction x' via

$$c = \mathrm{ReLU}(\mathbf{E}x + b), \quad x' = \mathbf{D}c,$$

where c represents the feature activation, $\mathbf{E} \in \mathbb{R}^{F \times H}$ is the encoder and $\mathbf{D} \in \mathbb{R}^{H \times F}$ is the decoder (F and H denote the dimensions of features and hidden states, respectively). Training of the SAE involves minimizing the following loss function $\mathscr{L} = \|x - x'\|_2^2 + \alpha \|c\|_1$, where α is a hyperparameter that regulates the sparsity level, and the ℓ_1 penalty term $\|c\|_1$ promotes sparsity in the feature coefficients.

The effectiveness of SAE in extracting semantic features from hidden states underscores its potential for capturing dependencies between code lines. Moreover, by leveraging a finite dictionary to model the dependencies, SAE avoids the reliance on unbounded positional indices, which gives a more context-dependent, finite encoding with superior extrapolation capabilities.

3 Our Approach

CoDE is based on the Transformer architecture, but it introduces two key modifications to mitigate the issues discussed in Section 1.

Figure 1 provides an overview of CoDE. Intuitively, we decompose a multi-layer Transformer into two functional stages, which focus on capturing intra-line semantic and line-wise relationships, respectively. In the shallow layers (e.g., layers 0–1 in our setup), the primary focus is on in-line tokens modeling, which is designed to separately extract the (semantic) representation of each line of code. In contrast, the deeper layers (e.g., layers 3–7) model contextual dependencies by encoding relationships between lines of code.[1]

To represent source code in two dimensions, a naive approach would be based on grid similar to an image. However, as source code lines normally exhibit a high variation of lengths, this approach would introduce a substantial number of padding tokens, reducing computational efficiency. We instead adopt a one-dimensional sequence representation to process source code, but highlight its line structure. Namely, source code of length m is separated by the line break token '\n' into different lines. Let the positions of '\n' are $1 = b_1 < b_2 < \cdots < b_k = m$. (Note that we assume boundary conditions $b_1 = 1$ and $b_k = m$ for convenience.) Tokens t_i and t_j are in the same line if there exists an index $1 < r < k$ such that $b_{r-1} < i, j \leq b_r$.

3.1 In-line Tokens Modeling

The shallow layers of the model (cf. Figure 1) are designed to extract the semantic representation of each individual line. To this end, we constrain the attention to operate within individual lines, for which purpose we use mask matrix M where $M_{i,j}$ $(1 \leq i, j \leq m)$ is defined as

$$M_{i,j} = \begin{cases} 0, & \text{if } b_{r-1} < i, j \leq b_r, \\ -\infty, & \text{otherwise.} \end{cases}$$

By decoupling intra-line dynamics from inter-line relationships, the model achieves a more structured understanding of every individual line of code, thus achieve "soft" permutation invariance.

3.2 Line-wise Dependency Encoding

As mentioned before, we introduce line-wise dependency encoding to capture the hierarchical relationships between lines of code.

SAE for Semantic Matching. As illustrated in Figure 1, we employ the line break token '\n' as an "anchor" for extracting the dependency encoding of the subsequent line of code. Prior research on CLMs has highlighted the role of '\n' functioning as a semantic anchor that compresses rich contextual information (Ge et al., 2023; Zhang et al., 2024b; Pang et al., 2024). Consequently, it is particularly well-suited for the extraction of dependencies for the following line of code. Specifically, for the tokens $t_{b_{r-1}+1}, \ldots, t_{b_r}$ in code line r, we use $x_{b_{r-1}}$ (i.e., the embedding of $t_{b_{r-1}}$) as the semantic anchor representing the contextual information required for dependency encoding.

Consequently, the dependency encoding d_r for tokens in line r can be derived using SAE as

$$c_r = \mathrm{ReLU}(\mathbf{E}x_{b_{r-1}} + b), \quad d_r = \mathbf{D}c_r,$$

where $\mathbf{E} \in \mathbb{R}^{F \times H}$, $\mathbf{D} \in \mathbb{R}^{H \times F}$ and $b \in \mathbb{R}^F$ are learnable parameters in SAE. Specifically, $\mathbf{E}$ encodes $x_{b_{r-1}}$ into feature activation c_r. $\mathbf{D}$ serves as a feature dictionary, with each column representing a feature. The resulting d_r serves as the line-wise relation embedding for tokens in line r.

The learned dependency encoding d_r is integrated into the model by adding the token embeddings of each line with their corresponding dependency encoding. Formally, for embeddings in $x_{b_{r-1}+1}, \ldots, x_{b_r}$, their dependency-encoded embeddings are defined as

$$x_i \leftarrow x_i + d_r, \quad \forall i \in [b_{r-1} + 1, b_r]$$

During training we use the loss function $\mathscr{L} = \mathscr{L}_{\text{model}} + \alpha \|c\|_1$, where $\mathscr{L}_{\text{model}}$ denotes the cross-entropy loss function utilized for training LM, and α controls the sparsity of the activation in SAE. This not only improves the model's generalization capacity but also contributes to the interpretability of the learned features, as will be analyzed in detail in Section 5.4.

Concentrating Attention. In the context of long-context tasks such as repository-level code generation, attention may become distracted in CoDE, leading to a decline in model performance (Wang et al., 2024b; Ding et al., 2024). This issue manifests in two primary ways. **(i)** CoDE applies a unified dependency encoding for tokens within the same code line, which increases the number of attended tokens as dependencies grow. **(ii)** CoDE lacks so called *position-based attention decay* which is adopted in most position encoding strategies (Su et al., 2024; Press et al., 2021). As the context length increases, the model's attention

[1]Layer 2 is not subject to additional modifications and employs full attention. Its primary purpose is to provide contextual information for each line of code for better dependency modeling.

is distributed across more tokens, leading to a performance decline (Wang et al., 2024b; Zhang et al., 2024c).

To address **issue (i)**, consider two embeddings $x_i, x_j \in X$. Let p_i and p_j represent their corresponding dependency encoding derived from SAE. As in the vanilla Transformer, the pre-softmax attention score between x_i and x_j is given by $a_{i,j} = q_i^T k_j$, where

$$q_i = \mathbf{W_Q}(x_i + d_i), \quad k_j = \mathbf{W_K}(x_j + d_j).$$

Note that

$$q_i^T k_j = \underbrace{x_i^T \mathbf{W_Q}^T \mathbf{W_K} x_j}_{\text{(a)}} + \underbrace{x_i^T \mathbf{W_Q}^T \mathbf{W_K} d_j}_{\text{(b)}} + \underbrace{d_i^T \mathbf{W_Q}^T \mathbf{W_K} x_j}_{\text{(c)}} + \underbrace{d_i^T \mathbf{W_Q}^T \mathbf{W_K} d_j}_{\text{(d)}}.$$

Intuitively, term (a) refers to semantic attention, capturing the interaction between the semantics of x_i and x_j, terms (b) and (c) refer to cross-attention between semantic content and encodings, and term (d) refers to dependency attention, representing the direct interaction of line-wise dependencies. Prior studies highlighted the significance of term (d) in explicit modeling of positional interactions (Raffel et al., 2020). However, in our setup tokens within the same code line share identical dependency encoding, which may result in uniformly elevated attention scores between tokens across lines with strong dependency, inadvertently increasing model's attention on a larger set of tokens.

We suppress the direct positional attention (i.e., term (d)) and instead treat the attention scores corresponding to it as distinct features. Specifically, we flatten the attention scores along the head dimension and concatenate the scores from the heads for (a), (b), (c) and (d). These concatenated features are then passed through an MLP, which alleviates the attention distraction caused by (d). The final attention score is computed as

$$q_i^T k_j = \text{(a)} + \text{(b)} + \text{(c)} + \text{MLP}(\text{concat}[\text{(a)}, \text{(b)}, \text{(c)}, \text{(d)}]).$$

To address **issue (ii)**, we introduce a position-aware activation enhancement (PACE) mechanism. Compared to the *position-based attention decay* which restricts the model to focus more on local information, this mechanism dynamically scales the activation of features based on the length of the context, thereby enabling the model to focus attention on critical dependencies in longer contexts. Formally, PACE is defined as

$$\hat{c_r} = \log_m \big(\text{clip}(l, m)\big) c_r,$$

where m refers to the training length of the model, l is the position of the current token, and clip(l, m) ensures that the position value is capped at a maximum of m.

4 Experiment Setup

Implementation. We train CoDE (and baseline models) using the Llama architecture (Touvron et al., 2023) with next token prediction, a training sequence length of 512, and reinitialized weights. can be found in Appendix A.1. **Datasets and metrics.** We conduct experiments across four distinct task categories.

Code Modeling. It is akin to language modeling in NLP (Golovneva et al., 2024; Chen et al., 2024b), which refers to predicting the next token in a programming-related context.For the experiment, we extract 1,000 samples, each with a sequence length of 4,096, from the StarCoder dataset (Li et al., 2023a). We measure model's performance using language modeling perplexity which reflects how well the model predicts the likelihood of sequences, and accuracy which evaluates the proportion of correctly predicted tokens.

Long Context Understanding. This task is to predict the next line of code given the context, which is to evaluate model's generation capability in handling long context. Our experiment utilizes two code-related datasets, LCC (where the context consists code lines only) and RepoBench (where the context is at the repository level), from the LongBench benchmark (Bai et al., 2023b). We use BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020) metrics to quantify the similarity between generated code and reference code.

Functional Correctness. We use HumanEval (Chen et al., 2021) and its improved version HumanEvalPlus (Liu et al., 2023) to evaluate model's ability to generate functionally correct code. We consider the pass@1 and pass@10 metrics, which refer to the probability that the model produces a correct solution within its top 1 and top 10 generated code, respectively.

Context Retrieval. To evaluate model's ability to retrieve and utilize contextual information, we design two experimental subtasks, namely Copy (Gu and Dao, 2023; Golovneva et al., 2024; Lv et al., 2024) and Counting (Golovneva et al., 2024). In contrast to the Long Context Understanding task, which primarily emphasizes generating code, the Context Retrieval task shifts the focus to measuring the model's ability to accurately identify and utilize dependencies within the given context.

Further details regarding the dataset implementation are provided in Appendix A.2.

Baselines. We compare CoDE against six baseline methods **RoPE** (Su et al., 2024), **NoPE** (Haviv et al., 2022), **NoPE**$_\lambda$ (Wang et al., 2024b), **FIRE** (Li et al., 2023b), **HoPE** (Chen et al., 2024b) and **ALiBi** (Press et al., 2021). A brief introduction is given in Appendix A.3.

5 Experiment Results

5.1 Generalization

To evaluate the generalization capabilities of the model, we consider the in-distribution and extrapolation scenarios. The in-distribution scenario examines model's ability to predict within the training length (512 in our setup), while the extrapolation scenario assesses its performance on sequences exceeding the training length.

In-distribution scenario. We evaluate CoDE on the Code Modeling task under the in-distribution setting, as highlighted in Table 1 marked blue. The experimental results reveal that RoPE achieve the lowest perplexity within the training length, but it is fair to say that all methods exhibit comparable performance in terms of both perplexity and accuracy (usually with marginal differences). Notably, even the NoPE method, which entirely omits positional encoding, achieves competitive results. These results suggest that the evaluated encoding methods are sufficiently effective for in-distribution Code Modeling task.

Extrapolation scenario. The results are presented in the remaining columns of Table 1. In the Code Modeling task, CoDE achieves better performance at sequence lengths of 1,024 and 2,048. However, at sequence length 4,096, ALiBi achieves a lower perplexity, outperforming CoDE. This is due to ALiBi's attention bias mechanism, which reduces the model's reliance on long-range dependencies by progressively decreasing attention weights for distant tokens. This mechanism stabilizes attention entropy, aligning with the distribution seen during training and ensuring consistent perplexity even for longer sequences (Wang et al., 2024b; Zhang et al., 2024c).

As highlighted in prior work (Chen et al., 2024b), relying solely on perplexity to evaluate model's extrapolation ability can be misleading. For instance, ALiBi, which disregards long-range dependencies to maintain low perplexity, is unreliable in code generation task, where long-range dependencies are critical. Therefore, we further evaluate CoDE and baseline methods on the Long Context Understanding task using two code generation datasets (i.e., LCC and Repobench), where long-range contextual dependencies play a significant role. Experimental results demonstrate that CoDE achieves CodeBLEU scores of 27.86 and 17.35 on the LCC and Repobench datasets, respectively, surpassing baseline methods, highlighting its superior ability to leverage long-range dependencies in extrapolation tasks.

5.2 Functional Correctness

For code generation, perplexity/accuracy can be considered as indirect measures of the model performance. Functional correctness of the generated code is arguable a more direct reflection of the quality of the model. We use two datasets, HumanEval and HumanEvalPlus, which contain test cases for assessing whether the generated code passes unit tests. Experimental results in Table 3 demonstrate that CoDE consistently outperforms baseline methods in both the pass@1 and pass@10 metrics. This improvement is primarily attributed to its ability to exploit permutation invariance of code, thereby enhancing the model's generalization capabilities. Additionally, CoDE achieves superior performance on the pass@10 metric, underscoring its capacity to produce diverse and functionally correct code solutions.

5.3 Context Retrieval

The Copy subtask aims to evaluate model's potential ability to accurately utilize predefined parameters, functions and imported APIs. Notably, reordering the functions in the input does not affect the output. Experimental results (Table 4) show the accuracy with which the methods correctly copy the specified context. These results reveal that baseline models incorporating positional encodings struggle to account for permutation invariance. For instance, even the best-performing baseline, ALiBi, achieves an average accuracy of only 0.25 across various test sets. These models overly rely on positional information without semantic grounding,

Task	Code Modeling									
	Perplexity					Accuracy				
Length	512	1,024	2,048	4,096	Avg.	512	1,024	2,048	4,096	Avg.
RoPE	**4.37**	25.21	>100	>100	91.31	**0.69**	0.39	0.17	0.13	0.35
NoPE	4.71	12.11	>100	>100	>100	0.68	0.51	0.16	0.04	0.35
NoPE$_\lambda$	5.98	5.38	6.12	14.20	7.92	0.63	0.64	0.61	0.46	0.59
FIRE	4.39	3.46	4.78	17.63	7.57	**0.69**	0.72	0.67	0.47	0.64
HoPE	4.38	29.06	92.67	>100	64.46	**0.69**	0.37	0.21	0.18	0.36
ALiBi	4.68	3.81	3.84	**3.85**	4.15	0.67	0.71	0.71	**0.71**	0.70
CoDE	4.46	**3.28**	**3.30**	4.08	**3.79**	**0.69**	**0.74**	**0.73**	0.69	**0.71**

Table 1: **Code Modeling task**. We report in-distribution (marked blue) and extrapolation perplexity and accuracy.

Task	Long Context Understanding			
	LCC		Repobench	
	BLEU	CB	BLEU	CB
RoPE	0.03	0.78	0.02	1.66
NoPE	0.13	2.48	0.00	0.63
NoPE$_\lambda$	14.78	18.31	8.48	12.73
FIRE	24.89	26.64	**14.45**	17.02
HoPE	0.41	2.35	0.07	2.58
ALiBi	22.10	24.22	12.10	16.90
CoDE	**25.62**	**27.86**	13.59	**17.35**

Table 2: **Long Context Understanding task.** CB stands for the CodeBLEU metric.

Task	Functional Correctness			
	HumanEval		HumanEvalPlus	
	pass@1	pass@10	pass@1	pass@10
RoPE	2.50	3.05	2.38	3.05
NoPE	1.65	3.05	1.65	3.05
NoPE$_\lambda$	0.91	1.22	0.91	1.22
FIRE	1.89	3.05	1.89	3.05
HoPE	2.07	2.44	2.07	2.44
ALiBi	0.73	1.22	0.73	1.22
CoDE	**2.69**	**4.27**	**2.56**	**4.27**

Table 3: **Functional Correctness task**.

resulting in poor convergence in this task.

In our experiments, even after providing an additional 3 million training samples, most baseline models (except for ALiBi) struggled to converge effectively (cf. Table 6). This conforms to the previous study which report models require extensive training data to stimulate the model's copy capability. (Lv et al., 2024). In contrast, CoDE achieved an average accuracy of 0.99 on the Copy dataset using only 300K training samples, significantly outperforming the baselines. This superior performance can be attributed to CoDE's ability to naturally account for the permutation invariance of code by contextual semantic matching. These properties enable the model to generalize effectively, overcoming the limitations of baseline methods that rely heavily on positional information.

The Counting subtask reflects real-world applications where programs must track values such as loop iterations or the depth of nested function calls, which demand precise positional awareness to ensure correctness. In this task, the model is required to count the number of specific operations that occur after a given starting point. Therefore, these operations are inherently permutation-variant, as the order of operations affects the outputs. Experimental results show that almost all methods can accurately count in the in-distribution scenario. However, in extrapolation scenarios, baseline models experience a significant decline in performance, while CoDE continues to accurately complete the task with an average accuracy of 0.99. These results highlight that CoDE has overcome the model's reliance on positional information, allowing it to accurately count even over long spans.

These two datasets collectively validate our hypothesis that while decaying attention over long-distance tokens can improve metrics such as perplexity, such an approach often neglects critical contextual information. In contrast, CoDE effectively balances extrapolation and contextual retrieval through its semantic matching, enabling it to deliver consistent and contextually accurate results across diverse tasks.

5.4 Interpretability

To investigate the interpretability of the features learned by CoDE, we employ automated interpretability methods introduced by Bills et al. (2023). These methods aim to determine the activation conditions of each latent feature and leverage an LLM to generate automated explanations for the features. Additionally, the LLM produces simulated activation values, which quantify the confidence in these explanations.

We conduct a detailed analysis of the interpreted features extracted by CoDE, leading to two key findings. First, features with semantically similar interpretations tend to be represented closely within the high-dimensional space, forming distinct clusters. For example, features associated with code initialization snippets naturally group into a cohesive

Task	Context Retrieval									
	Copy					Counting				
Sequence	50	100	150	200	Avg.	100	200	300	400	Avg.
RoPE	0.13	0.18	0.10	0.14	0.14	**1.00**	0.31	0.13	0.17	0.40
NoPE	0.11	0.19	0.11	0.06	0.12	**1.00**	0.36	0.22	0.18	0.44
NoPE$_\lambda$	0.14	0.16	0.08	0.14	0.13	0.30	0.82	0.68	0.42	0.56
FIRE	0.13	0.19	0.11	0.16	0.15	**1.00**	0.44	0.23	0.21	0.47
HoPE	0.15	0.21	0.10	0.11	0.14	**1.00**	0.35	0.20	0.19	0.44
ALiBi	0.33	0.22	0.24	0.20	0.25	**1.00**	0.57	0.39	0.36	0.58
CoDE	**1.00**	**0.98**	**1.00**	**0.97**	**0.99**	**1.00**	**1.00**	**1.00**	**0.97**	**0.99**

Table 4: **Context Retrieval task**. We report the prediction accuracy for both in-distribution (marked in blue) and extrapolation scenarios.

cluster, reflecting their shared semantic properties.

Second, features that are functionally related exhibit higher (d) scores. This suggests that attention scores between the corresponding code lines are also elevated. For instance, features related to the import statements show high (d) score with features related to function calls. This aligns with human intuition, as these features are often functionally and semantically interdependent.

These findings demonstrate that the learned feature representations align well with human-interpretable patterns, providing evidence of the interpretability of CoDE. Further details are presented in Appendix B.

6 Related Work

Code Generation. Early code generation models (Jia and Liang, 2016; Ling et al., 2016) primarily treated code as natural languages, applying standard NLP techniques without explicitly accounting for the unique structural properties of code. Subsequently, a number of approaches sought to incorporate code-specific features, such as Abstract Syntax Trees (ASTs) (Yin and Neubig, 2018; Sun et al., 2020; Zhang et al., 2023b), to enhance model's understanding of code structure. With the advent of LLMs, the exponential growth in the number of model parameters has led to, as in other task, a remarkable improvement of performance in code generation (Chen et al., 2021; Li et al., 2022; Fried et al., 2022; Roziere et al., 2023; Le et al., 2022; Nijkamp et al., 2022). However, LLMs avoids the adoption of approaches that explicitly integrate code-specific features due to compatibility challenges and inefficiencies in training such models.

In contrast, our approach does not alter the overall architecture or incorporate explicit code-specific representations. This design not only ensures compatibility with existing LLM architectures but also demonstrates superior performance in capturing the dependencies of code.

Positional Encoding. The vanilla Transformer model (Vaswani, 2017) initially employed fixed absolute positional encodings. However, subsequent research demonstrated that relative positional encodings are more effective for modeling natural language (Shaw et al., 2018; Dai, 2019). This discovery has spurred significant interest in the study of positional encoding. For instance, T5's bias (Raffel et al., 2020) and its derivatives, ALiBi (Press et al., 2021), Kerple (Chi et al., 2022) and FIRE (Li et al., 2023b), introduced attention bias scalars directly into the attention scores, providing an efficient mechanism for position encoding. Furthermore, RoPE (Su et al., 2024) encodes relative position information by rotating the query and key vectors, and it has since become the most widely adopted method in SOTA LLMs. Further advancements such as YaRN (Peng et al., 2023), NTK-RoPE (Chen et al., 2023) and HoPE (Chen et al., 2024b) extend RoPE by focusing on improving its capability for length extrapolation, enabling these models to generalize to significantly longer contexts beyond their training sequences. In addition, HiRoPE (Zhang et al., 2024a), which is a positional encoding designed for code, enhances RoPE into a hierarchical format based on the hierarchical structure of the source code.

Our approach replaces positional encoding with dependency-based encoding, which focuses on the relationships between code lines. This avoids explicit positional indices and demonstrates better generalization across various contexts and lengths.

7 Conclusion

In this paper, we have introduced CoDE, a novel method for neural source code generation. The crux of our method is a two-dimensional perspective of the source code, naturally structured as code lines, and line-wise dependency encoding in lieu of explicit positional encoding in transformer models. Experiments confirm its better generalizability, contextual understanding and interpretability. These experiments further corroborate the robustness and effectiveness of CoDE.

Limitations

Due to the constraints of computational resources, we were unable to scale CoDE to LLMs. As a result, its effectiveness in scenarios involving significantly larger parameters and extensive datasets remains unexplored. Future work should address this limitation to assess the scalability and applicability of CoDE in more demanding settings.

Another critical aspect lies in the sensitivity of CoDE to two hyperparameters, namely F and α, both of which significantly influence its performance. Specifically, F governs the number of features that the model can learn, while α determines the number of features activated during inference. We provide a detailed analysis of hyperparameter selection in Appendix A.5. Increasing F allows the model to capture finer-grained relationships, which can potentially improve performance. However, this improvement comes at the cost of higher computational and memory demands due to the increased number of parameters. Conversely, α controls the sparsity of feature activation within the SAE. If α is set too low, the model may fail to learn distinct and interpretable dependency features. On the other hand, excessively high values of α can result in no features being activated, which severely compromises the model's representational capacity. Thus, the careful tuning of F and α is critical to balancing performance and computational efficiency. Identifying optimal configurations for these hyperparameters requires additional experimental effort, which we leave for future work.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62372232), the High Performance Computing Platform of Nanjing University of Aeronautics and Astronautics, and the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by an overseas grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04).

References

Dzmitry Bahdanau. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023a. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023b. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Steven Bills, Nick Cammarata, Dan Mossing, Henk Tillman, Leo Gao, Gabriel Goh, Ilya Sutskever, Jan Leike, Jeff Wu, and William Saunders. 2023. Language models can explain neurons in language models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Pei Chen, Soumajyoti Sarkar, Leonard Lausen, Balasubramaniam Srinivasan, Sheng Zha, Ruihong Huang, and George Karypis. 2024a. Hytrel: Hypergraph-enhanced tabular data representation learning. *Advances in Neural Information Processing Systems*, 36.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Yuhan Chen, Ang Lv, Jian Luan, Bin Wang, and Wei Liu. 2024b. Hope: A novel positional encoding without long-term decay for enhanced context awareness and extrapolation. *arXiv preprint arXiv:2410.21216*.

Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. 2022. Kerple: Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems*, 35:8386–8399.

Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. 2023. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*.

Zihang Dai. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. 2024. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2406.04093*.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*.

Olga Golovneva, Tianlu Wang, Jason Weston, and Sainbayar Sukhbaatar. 2024. Contextual position encoding: Learning to count what's important. *arXiv preprint arXiv:2405.18719*.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Adi Haviv, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. 2022. Transformer language models without positional encodings still learn positional information. *arXiv preprint arXiv:2203.16634*.

Yunpeng Huang, Jingwei Xu, Zixu Jiang, Junyu Lai, Zenan Li, Yuan Yao, Taolue Chen, Lijuan Yang, Zhou Xin, and Xiaoxing Ma. 2023. Advancing transformer architecture in long-context large language models: A comprehensive survey. *CoRR*, abs/2311.12351.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*.

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*.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. 2024. The impact of positional encoding on length generalization in transformers. *Advances in Neural Information Processing Systems*, 36.

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, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. 2023b. Functional interpolation for relative positions improves long context transformers. *arXiv preprint arXiv:2310.04418*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. 2024. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2. *arXiv preprint arXiv:2408.05147*.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Ang Lv, Ruobing Xie, Xingwu Sun, Zhanhui Kang, and Rui Yan. 2024. Language models" grok" to copy. *arXiv preprint arXiv:2409.09281*.

Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

Andrew Ng et al. 2011. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Zhiyuan Pan, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. Enhancing repository-level code generation with integrated contextual information. *arXiv preprint arXiv:2406.03283*.

Jianhui Pang, Fanghua Ye, Derek Fai Wong, Xin He, Wanshun Chen, and Longyue Wang. 2024. Anchor-based large language models. *arXiv preprint arXiv:2402.07616*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*.

Ofir Press, Noah A Smith, and Mike Lewis. 2021. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

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

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. *Advances in neural information processing systems*, 28.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8984–8991.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024a. Teaching code llms to use autocompletion tools in repository-level code generation. *arXiv preprint arXiv:2401.06391*.

Jie Wang, Tao Ji, Yuanbin Wu, Hang Yan, Tao Gui, Qi Zhang, Xuanjing Huang, and Xiaoling Wang. 2024b. Length generalization of causal transformers without position encoding. *arXiv preprint arXiv:2404.12224*.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–47.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Kechi Zhang, Ge Li, Huangzhao Zhang, and Zhi Jin. 2024a. Hirope: Length extrapolation for code models using hierarchical position. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13615–13627.

Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023b. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1291–1302.

Xiangyu Zhang, Yu Zhou, Guang Yang, Harald C Gall, and Taolue Chen. 2024b. Anchor attention, small cache: Code generation with large language models. *arXiv preprint arXiv:2411.06680*.

Zhisong Zhang, Yan Wang, Xinting Huang, Tianqing Fang, Hongming Zhang, Chenlong Deng, Shuaiyi Li, and Dong Yu. 2024c. Attention entropy is a key factor: An analysis of parallel context encoding with full-attention-based pre-trained language models. *arXiv preprint arXiv:2412.16545*.

Liang Zhao, Xiaocheng Feng, Xiachong Feng, Bin Qin, and Ting Liu. 2023. Length extrapolation of transformers: A survey from the perspective of position encoding. *arXiv preprint arXiv:2312.17044*.

A Appendix

A.1 Model Configuration

The model configurations for the various tasks are detailed in Table 5. The 71M model is evaluated on three tasks including Code Modeling, Functional Correctness, and Long Context Understanding. This model is trained on the StarCoder's Python dataset with a total of 5 billion tokens, utilizing 8 NVIDIA RTX A6000 GPUs. In contrast, the smaller 10M parameter model is assessed on two subtasks within the Context Retrieval task. Training for this model is conducted on a single NVIDIA RTX 4090 GPU.

	71M	10M
Training length	512 tokens	(50) 100 lines
Batch size	32×8	16×1
Training data size	5B tokens	300K samples
Learning rate	5e-4	5e-5
Hidden size	512	256
Intermediate dimension	2,048	1,024
Number of layer	8	4
Number of head	8	4
In-line modeling layers	2	1
Line-wise embedding layers	5	3
Number of feature	512	32
Sparsity coefficient α	5e-5	2e-6
Precision	bfloat16	bfloat16

Table 5: Model configurations. For the Context Retrieval task, the training setup varies across specific sub-tasks. In the Copy sub-task, the training sequence length is set to 50 lines of code, whereas in the Counting sub-task, the training sequence length is configured to 100 lines of operation.

A.2 Dataset Construction

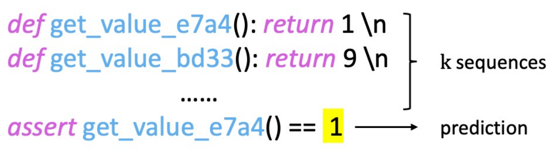
```
def get_value_e7a4(): return 1 \n
def get_value_bd33(): return 9 \n
......
assert get_value_e7a4() == 1
```

Figure 2: Specific example of Copy task.

Task	Context Retrieval				
	Copy				
Sequence	50	100	150	200	Avg.
RoPE	0.23	0.15	0.15	0.17	0.18
NoPE	0.15	0.18	0.11	0.09	0.13
NoPE$_\lambda$	0.13	0.16	0.11	0.11	0.13
FIRE	0.16	0.20	0.11	0.11	0.15
HoPE	0.23	0.11	0.09	0.09	0.13
ALiBi	**1.00**	**1.00**	0.98	0.55	0.88
CoDE	**1.00**	0.98	**1.00**	**0.97**	**0.99**

Table 6: **Copy task.** Models except CoDE are trained on 3M addtional samples. We report in-distribution (marked blue) and extrapolation accuracy.

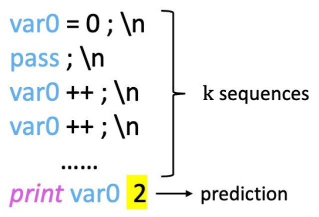
```
var0 = 0 ; \n
pass ; \n
var0 ++ ; \n
var0 ++ ; \n
......
print var0 2
```

Figure 3: Specific example of Counting task.

Copy Subtask. Copying is a fundamental capability of CLMs, as it determines whether a model can accurately invoke APIs or predefined functions. Figure 2 presents an illustrative example from the copy dataset. In this dataset, each sample consists several functions, where each function name is appended with a unique 4-character suffix, and each function returns an integer within the range of 0 to 9. The task requires the model to predict the return value of a randomly selected function. Importantly, since the order of functions is irrelevant to the output, the code snippets in this subtask exhibits permutation invariance. To evaluate the model's performance on this subtask, we generated a dataset of 300K training samples, each containing 50 functions. Additionally, we constructed four test sets, each consisting of 100 samples, with the number of functions per test set set to [50, 100, 150, 200], respectively. These test sets were specifically designed to assess model performance in both in-distribution and extrapolation settings. Initial experiments revealed that the baseline methods, trained on 300K samples, failed to predict the correct return values consistently. Prior studies have indicated that the copying capability of LMs with traditional positional encodings emerges only when exposed to substantially large amounts of training data (Lv et al., 2024). Motivated by this observation, we augmented the training set by providing the baseline methods with additional 3M samples. The results of this extended training are summarized in Table 6. Experimental results demonstrate that even with the expanded training data, all baselines except ALiBi methods failed to make accurate predictions. ALiBi is capable of correctly predicting return values within the training sequence length (achieving 100% accuracy on test samples with 50 functions). However, ALiBi still exhibited significant performance degradation as the sequence length in the test set increased, highlighting its limitations in extrapolation scenarios.

Counting Subtask, in contrast, require a more uniform distribution of attention over a specific span. For instance, in code-related tasks, counting involves a model's ability to comprehend and track multiple elements across a sequence, such as variables or operations, thereby demanding a more global understanding. Figure 3 illustrates a specific example from the counting dataset. The counting dataset is composed of sequences of operations, including: *set* (initialize a variable to zero), *increment* (increment the variable's value by 1), and *pass* (perform no operation). These operations are randomly generated based on predefined weights: $w_{set} = 1, w_{incr} = 5$, and $w_{pass} = 100$. Unlike the copy subtask, reordering operations in the counting subtask directly affects the final result, making this task sensitive to the order of code snippets. For this subtask, we generated 300K training samples, with each containing 100 operations. Similarly, we created four test datasets, each consisting of 100 samples, where the number of operations per test set is set to [100, 200, 300, 400]. To further evaluate the model's ability to handle long-range dependencies, we varied the weights of the pass operation (w_{pass}) in the test datasets, setting them to [100, 200, 300, 400], respectively.

A.3 Baselines

We compare CoDE against six baseline methods.

RoPE (Su et al., 2024) encodes positional information by applying rotational transformations to input embeddings, enabling better handling of sequence order and relative positions.

NoPE (Haviv et al., 2022) has shown that LLMs can achieve non-trivial performance *without* positional encodings, and exhibit stronger extrapolation capabilities.

NoPE$_\lambda$ (Wang et al., 2024b) extends NoPE by introducing a temperature scaling mechanism, which improves model's extrapolation ability.

FIRE (Li et al., 2023b) introduces a novel positional interpolation mechanism, which leverages an MLP to map relative positional information into a continuous space, thereby improving extrapolation.

HoPE (Chen et al., 2024b) replaces specific components of RoPE with position-independent elements, retaining only high-frequency signals which enhances model's context awareness and extrapolation ability.

ALiBi (Press et al., 2021) introduces an attention bias designed to emphasize tokens with closer relative distances while systematically down-weighting contributions from tokens at greater distances. This approach effectively prioritizes local dependencies in the input sequence, thereby reducing the model's reliance on longer-range interactions.

Since CoDE uses additional parameters (related to SAE), in order to ensure fairness in comparison, we add an additional Transformer layer to the baseline models to ensure the consistency of total model parameters.

A.4 Ablation Study

We conduct ablation studies to evaluate the contribution of individual components in CoDE to its performance. As shown in Table 7 8 9 10, we first examine the impact of removing the in-line modeling, which involves omitting additional attention constraints at the shallow layers. This modification leads to an increase in perplexity and a corresponding decline in performance across other tasks, underscoring the importance of capturing the semantic information of independent code lines. By capturing the permutation invariance of code, this component proves essential for enhancing generalization.

To mitigate the issue of attention distraction during extrapolation, we incorporate several key components, including PACE, an MLP layer and removing term (d). These modules are specifically designed to enable the model to focus attention on critical contextual information. Ablation experiments on these components reveal a significant increase in perplexity during extrapolation when any of them is removed. This finding highlights their roles in maintaining focused attention, which is crucial for effective extrapolation.

A.5 Hyperparameter

The selection of the two hyperparameters, F and α, in CoDE is critical as they determine the number of learned features and the sparsity of activated features, respectively. The hyperparameter F controls the total number of features that the model can learn. As F increases, the model has the capacity to capture a greater number of features. However, studies in the field of mechanical interpretability have demonstrated that increasing the number of SAE features does not necessarily enable the model to attend to a broader set of features; instead, it leads to the learning of finer-grained features. As shown in Table 11, our experiments indicate that increasing F can improve the model's performance to some extent. However, this improvement comes at the cost of increased model parameters and com-

Task	Code Modeling							
	Perplexity				Accuracy			
Length	512	1,024	2,048	4,096	512	1,024	2,048	4,096
CoDE	4.46	**3.28**	**3.30**	**4.08**	**0.69**	**0.74**	**0.73**	**0.69**
w/o In-line Modeling	4.52	3.87	4.08	5.21	0.67	0.71	0.70	0.66
w/o PACE	4.46	3.31	3.47	4.33	**0.69**	0.73	0.71	0.67
w/o MLP	4.49	3.42	3.78	4.71	0.68	0.73	0.70	0.67
w (d)	**4.45**	3.58	3.72	4.87	**0.69**	0.72	0.70	0.67

Table 7: **Code Modeling task for ablation study**. We report in-distribution (marked blue) and extrapolation perplexity and accuracy.

Task	Long Context Understanding			
	LCC		Repobench	
	BLEU	CB	BLEU	CB
CoDE	**25.62**	**27.86**	**13.59**	**17.35**
w/o In-line Modeling	22.87	24.68	12.01	14.80
w/o PACE	24.82	26.88	13.18	16.42
w/o MLP	23.61	23.86	12.88	16.07
w (d)	23.23	25.87	13.06	16.37

Table 8: **Long Context Understanding task for ablation study**. CB stands for the CodeBLEU metric.

Task	Functional Correctness			
	HumanEval		HumanEvalPlus	
	pass@1	pass@10	pass@1	pass@10
CoDE	**2.69**	**4.27**	**2.56**	**4.27**
w/o In-line Modeling	2.50	3.66	2.38	3.66
w/o PACE	**2.69**	**4.27**	**2.56**	**4.27**
w/o MLP	2.50	3.66	2.07	3.66
w (d)	**2.69**	**4.27**	**2.56**	**4.27**

Table 9: **Functional Correctness task for ablation study**.

putational overhead. To balance performance and efficiency, we set the number of features to 512, aligning it with the dimensionality of the model.

The hyperparameter α governs the sparsity of feature activation within the SAE. If α is set too low, an excessive number of features will be activated, making it difficult for the model to focus on key features. Conversely, if α is set too high, it may result in no features being activated, thereby impairing the model's ability to understand long-range contexts. Thus, careful tuning of α is essential to strike a balance between sparsity and the model's contextual understanding capability.

B Details on Interpretability

In this section, we provide a detailed explanation of the dependency features learned by CoDE, aiming to analyze its underlying operational mechanisms. We first employ the autointerpretation protocol (Bills et al., 2023) to interpret the features within the dictionary. The autointerpretation process is structured as follows:

1. We begin by sampling 10,000 samples from the StarCoder dataset and segmenting them into code fragments of length 512. For each line within these fragments, we measure the activation of the features and rescale them to integer values ranging from 0 to 10.

2. We identify the top 20 lines with the highest activation scores for each feature. Along with these lines, their rescaled activation, contexts and the prompt we create are sent to GPT-4o to generate an explanation for when the feature activates, yielding an interpretation.

3. Using GPT-4o, we simulate each feature's activation for the activated lines, conditional on the proposed explanations from step 2. We prompt GPT-4o to output an integer from 0-10 as the simulated activation.

4. To score an explanation, we compare two lists of values: the simulated activation values, and the actual activation values to quantify the alignment between the feature's real behavior and its simulated interpretation.

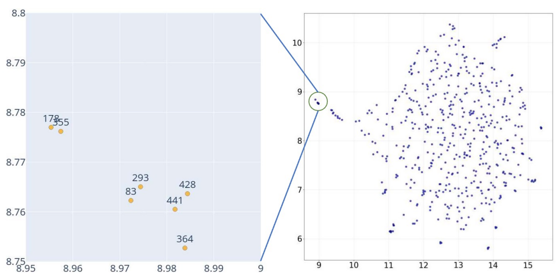

Figure 4: The right part of presents a UMAP visualization of the features in the SAE from the model's 4th layer, while the left side highlights a specific cluster within this visualization, detailing the features it contains along with their corresponding indices.

Next, we apply UMAP visualization (McInnes et al., 2018) to visualize the features in two dimensions. Notably, we observe that features with semantically similar explanations tend to form distinct clusters. For instance, Figure 4 shows a cluster of features from the SAE in 4th layer of the model,

Task	Context Retrieval							
	Copy				Counting			
Sequence	50	100	150	200	100	200	300	400
CoDE	**1.00**	0.98	**1.00**	**0.97**	**1.00**	**1.00**	**1.00**	**0.97**
w/o In-line Modeling	0.98	0.96	0.97	0.95	**1.00**	0.99	0.97	**0.97**
w/o PACE	**1.00**	0.98	**1.00**	**0.97**	**1.00**	**1.00**	**1.00**	0.96
w/o MLP	**1.00**	0.98	0.99	**0.97**	**1.00**	0.98	0.97	**0.97**
w (d)	**1.00**	**0.99**	0.99	0.96	**1.00**	**1.00**	0.98	0.96

Table 10: **Context Retrieval task for ablation study**. We report the prediction accuracy for both in-distribution (marked in blue) and extrapolation scenarios.

Task	Hyperparameter Selection			
	Perplexity			
Length	512	1,024	2,048	4,096
$F = 256$	4.59	3.57	3.35	4.56
$\mathbf{F = 512}$	4.46	**3.28**	3.30	4.08
$F = 1024$	**4.39**	3.29	**3.26**	**4.06**
$\alpha = 5e-4$	4.49	3.68	3.70	4.65
$\boldsymbol{\alpha = 5e-5}$	**4.46**	**3.28**	**3.30**	**4.08**
$\alpha = 5e-6$	4.51	3.58	3.74	4.57

Table 11: **Code Modeling task for Hyperparameter Selection**. We conducted experiments on hyperparameter selection for both F and α.

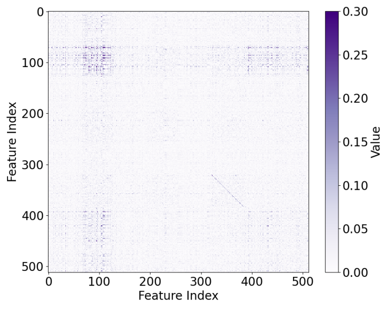

Figure 5: To provide a clearer representation of the relationships between features, we present a heatmap of the (d) scores between features. For clarity, the values are constrained to the range [0, 0.3].

with corresponding explanations provided in Table 12. These features primarily activate in the context of code related to initialization, configuration, and setup processes. Combining the insights from the figure and table, we find that features that cluster together share similar functionalities, which aligns with our intuition. Furthermore, this finding supports the effectiveness of the autointerpretability method in explaining these features. Additionally, although these features serve similar purposes, their usage scenarios differ. For example, feature 4-355 focuses on initialization in method definitions, while features 4-441 and 4-428 are activated in class-based structures and scripts, respectively. This suggests that these features have more granular activation contexts.

Furthermore, we computed the (d) score between features, which directly reflects the relationships between code lines through dependency attention, and visualized the results with a heatmap, as shown in Figure 5[2]. Our findings reveal that semantically related features exhibit higher (d) scores, suggesting that the attention scores between the code lines activating these features are strengthened. For example, as shown in Table 13, the (d) score between Feature 4-72 and Feature 4-110 is particularly high, as they are frequently activated together to represent the traversal between tuples and lists.

Similarly, Feature 4-73 and Feature 4-81 show strong semantic correlations. Feature 4-73 focuses on detecting import statements, while Feature 4-233 is activated by function calls. The correlation between these two features ensures that the model can associate imported packages with corresponding function calls, maintaining contextual consistency during code generation.

Lastly, Feature 4-87 and Feature 4-94 demonstrate notable semantic alignment. Feature 4-87 is sensitive to function definitions, whereas Feature 4-94 is activated by lines of code involving return statements. This alignment highlights the model's ability to focus on function inputs and type annotations when generating return values, ensuring that the returned values are both semantically and type-accurately consistent.

We further analyzed the dictionary learned in the Counting task and observed that each operation corresponds to specific features within the model, as illustrated in Table 14. This observation indicates that the model is capable of accurately identifying the position of each operation based on semantic matching. Consequently, it can compute the results of various combinations of operations over longer contexts with high precision.

[2]Since we observed the feature correlations in isolation, without context, the activation values for these features cannot be obtained. For simplicity, we set these activation values to 1.

Feature id	Score	Description (Generated by GPT-4o)
4-355	0.30	Feature 355 activates for lines related to initializing, resetting, or configuring elements or systems, with a specific focus on method definitions, initialization processes, and setting values or constraints.
4-364	0.28	Feature 364 looks for lines of code related to either method definitions or inline documentation within classes, particularly focusing on initialization and reset procedures.
4-441	0.27	Feature 441 is looking for code documentation comments, particularly those describing the initialization and setup of class-based structures or functions.
4-293	0.14	Feature 293 is primarily looking for the initialization or setup processes in code, particularly focusing on function or method comments detailing initial states or configurations.
4-178	0.25	Feature 174 is looking for code definitions or class initializations that involve setting or using attributes within a defined structure or method.
4-428	0.06	Feature 428 is looking for code lines that involve initializing or assigning configuration from a dictionary structure, often within context related to setup or configuration handling in Python scripts.
4-83	0.46	Feature 83 is looking for code lines related to the initialization or definition of functions or objects, particularly around quantile functions, data insertions, and password conditions.

Table 12: We offer an autointerpretation of the clusters illustrated in Figure 4. In the column titled "Feature ID", the first number represents the model layer, while the following numbers indicate the feature index. The "Score" column displays the autointerpretation score assigned, and the "Description" column contains the explanation generated by GPT-4o for the corresponding feature.

Feature id	(d)	Score	Description (Generated by GPT-4o)
4-72	0.68	0.29	Feature 72 identifies code lines related to iteration settings or constants, often involving numerical values.
4-110		0.19	Feature 110 is looking for structured data representations, specifically tuple and list elements, often containing numeric values or identifiers.
4-73	0.38	0.72	Feature 73 is most strongly looking for import statements.
4-81		0.42	Feature 81 is looking for lines that involve function definitions or calls related to processing or manipulating data, often involving operations or method invocations.
4-87	0.30	0.31	Feature 87 is looking for patterns associated with function definitions or initializations within a code context.
4-94		0.12	Feature 94 activates for lines involving environment setup, return statements, and comments in code.

Table 13: We present several sample feature pairs, where "(d)" denotes the (d) score in attention computation.

Feature id	Score	Description (Generated by GPT-4o)
2-9	0.54	Feature 9 is looking for the presence of the specific code line "pass ."
2-22	0.47	Feature 22 is looking for code lines that contain the pattern "var0 = 0 ."
2-26	0.69	Feature 26 is looking for code lines where the variable "var0" is being printed.
2-28	0.40	Feature 28 is looking for lines of code where a variable is incremented using the "++" operator.

Table 14: We provide four activated features associated with the Counting task.