Towards Understanding What State Space Models Learn About Code

Anonymous ACL submission

Abstract

State Space Models (SSMs) have emerged as an efficient alternative to the Transformer architecture. Prior work shows that, when trained under comparable conditions, SSMs can match or surpass Transformers on code understanding tasks. However, their internal mechanisms remain a black box. We present the first systematic analysis of what SSM-based code models learn along with the direct comparison between SSM and Transformer models in this domain. Our analysis shows that SSMs capture syntactic and semantic structure more effectively than Transformers during pretraining but forgets certain relations during fine-tuning on some tasks. To investigate this behavior, we introduce *SSM-Interpret*, a frequency-domain framework that exposes a "spectral shift" toward short-range dependencies during fine-tuning. Guided by these findings, we propose architectural modifications that significantly improve the performance of SSM-based code model by upto +6 MRR on NLCodeSearch. This demonstrates that our analysis not only explains model behavior but also leads directly to better designs.

1 Introduction

Transformers are the dominant architecture for sequence modeling across domains, including the code understanding domain, which is in the focus of this work. However, they suffer from inherent limitations, including quadratic complexity, large data requirement, and positional biases.

State Space Models (SSMs) (Gu et al., 2021) have emerged as a computationally efficient alternative, particularly promising for long-context tasks. While SSM-based architectures have historically struggled to match the empirical performance of transformers (Zuo et al., 2024; Ren et al., 2025), Verma et al. (2025) showed that SSMs can outperform Transformers on code understanding tasks such as retrieval and classification while achieving better compute and sample efficiency. However, the internal mechanisms enabling the performance gains and the extent to which state space models capture syntactic or semantic code structure remain unexplored. Preliminary interpretability work focuses exclusively on selective SSMs[1] and is limited to synthetic tasks. For SSMs no interpretability studies exist. This gap limits our ability to diagnose failures, guide architectural improvements, and predict when SSMs will succeed or struggle on code understanding and generation tasks.

The interpretability gap stands in stark contrast to the Transformer ecosystem, where attention maps and learned representations have been rigorously analyzed to understand how code properties are captured (Anand et al., 2024; Wan et al., 2022; López et al., 2022; Karmakar and Robbes, 2021). In this work, we turn our attention to SSMs and real-world data. Specifically, we (a) conduct the first systematic comparative analysis of SSM and Transformer representations in the code domain and (b) propose the first framework for analyzing the convolution kernel of SSM blocks in a multi-layer model.

Our focus on SSM, despite their current limited real-world application, is motivated by recent theoretical works, such as Nishikawa and Suzuki (2025), which showed that, when combined with non-nonlinearities, SSMs can indeed perform dynamic token selection comparable to self-attention. At the same time, multiple works have shown the failure of selective-SSMs on tasks that require dynamic token selection, such as input copying and state tracking (Chen et al., 2025; Jafari et al., 2024; Jelassi et al., 2024). SSMs are also significantly more efficient compared to both transformers and selective-SSMs. Also, SSMs can extrapolate to 8x

[1]We make a distinction between SSMs which rely on RNN mode and hardware optimized parallel scan algorithms, like Mamba, and SSMs that rely on convolution during training, like S4D. For simplicity, we refer to the former as selective-SSMs and the latter as SSMs.

the pretraining context (Verma et al., 2025) while Mamba cannot extrapolate (Azizi et al., 2025).

While SSMs have outperformed transformers at similar parameter and dataset scale, they have not been scaled to billions of parameters and trillions of tokens and have not been applied on generative tasks. One of the key reasons for this has been the failure of SSMs on certain tasks such as type inference, which are similar to generative tasks (large class size; long and short range dependency). While scaling is an important aspect to improve performance and generalizability of the models, naive scaling will lead to limited gains. Thus, in this work we focus on understanding why SSM fails on certain tasks, instead of scaling.

For our study, we use an existing SSM-based model, CodeSSM (Verma et al., 2025), which has shown superior performance to Transformer on numerous tasks but lag behind on type inference. We compare the internal representation of CodeSSM with RoCoder (Verma et al., 2025) as both the models has been trained at the same parameter and data scale, removing these factors as confounders.

Our comparative investigation shows that while pretrained CodeSSM captures code properties more effectively than RoCoder, this advantage collapses during finetuning. Specifically, when finetuned on type inference task, CodeSSM forgets critical syntactic relations that RoCoder retains. To diagnose this failure, we develop a novel framework for analyzing the convolution kernels of multi-layer SSMs. The kernel analysis reveals a "spectral shift": during fine-tuning, the kernels in early layers bias heavily towards short-range dependencies, effectively discarding the long-range context required for type inference. Guided by these insights, we introduce architectural changes that prevent this degeneration, improving the model's ability to reason about complex code structures. These changes can form the basis for scaling and generative application of SSMs in future works.

In summary, our main contributions are:

- *Comparative Hidden Representation Analysis*: We provide the first direct comparison of hidden states in SSM and Transformer code models, where code structure (AST, DFG) enable rigorous evaluation of syntactic and semantic understanding. We empirically demonstrate that while CodeSSM captures better syntactic and semantic relations compared to RoCoder during pretraining, it forgets certain relations during fine-tuning.
- *SSM Kernel Analysis Framework:* We propose the first framework for analyzing convolution kernels in multi-layer SSMs. Using this method, we identify a strong correlation between the high-frequency spectral shift in CodeSSM's kernel and its failure on type inference.
- *Interpretability-Driven Improvements:* Leveraging our analytical findings, we propose two enhanced variants of CodeSSM. These architectures mitigate the spectral shift, yielding significant performance gains of upto +6 MRR on NLCodeSearch, +3.8 MRR on Long Context Retrieval and +2 F1 type inference, validating that insights from our analysis translate directly into performance improvements.

2 Related Works

Hidden Representation Analysis. Various methodologies have been proposed to analyze the internal representations of LLMs. The most prevalent approach employs "probes" – trained classifiers or learned transformations – to evaluate hidden states (Ahmed et al., 2023; Karmakar and Robbes, 2021; López et al., 2022; Yang et al., 2023). However, classifier-based probes suffer from several significant limitations (Maudslay et al., 2020; Hewitt and Liang, 2019; Belinkov, 2022). To address these shortcomings, Anand et al. (2024) proposed applying DirectProbe (Zhou and Srikumar, 2021), a classifier-free methodology, to analyze the hidden representations of pretrained code models directly.

Interpretability of SSMs. Recent work has begun analyzing SSM architectures through various lenses. Chen et al. (2025) identify limitations of Mamba on synthetic copying tasks but do not examine real-world domain adaptation or convolution-based SSMs. Paulo et al. (2024) analyzed if the interpretability methods developed for transformer can be used for architectures such as Mamba but do not evaluate if they apply to general convolution based SSMs. Ali et al. (2025) reformulate Mamba as an implicit attention mechanism, offering theoretical insights and attention based analysis.

Our work fills critical gaps by comparing the capabilities of SSMs and transformers on real-world tasks, and we are the first to develop a framework for layer-wise frequency-domain analysis of the learned convolutional kernels, resulting in concrete architectural improvements.

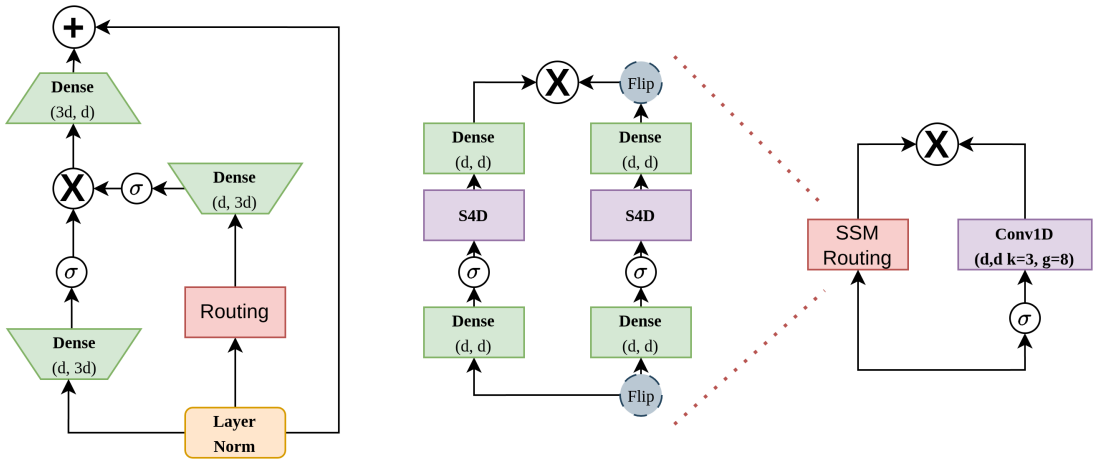


Figure 1: The CodeSSM layer architecture (left) showing the original routing mechanism (center) and the proposed routing (right).

We discuss additional related works on analysis using spectral methods and control theory in Appendix A.

3 Comparative Hidden Representation Analysis

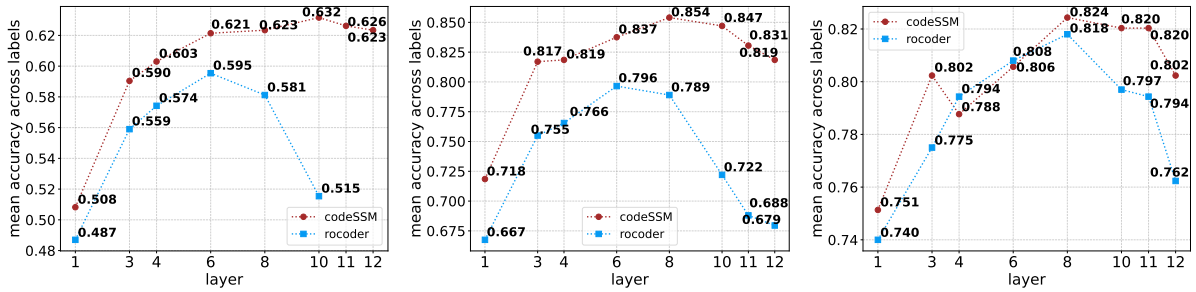
In this section, we describe the comparative hidden representation analysis of SSM and transformers.

3.1 Analysis Setup

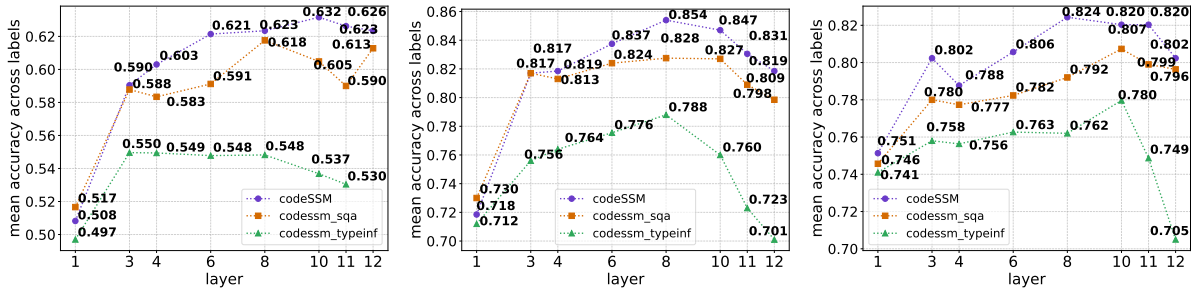
To evaluate the efficacy of SSM-based models in capturing code properties, we perform a comparative analysis between a SSM and a Transformer model. We utilize the CodeSSM and RoCoder models (Verma et al., 2025) (architecture details in Appendix C.2), both having similar size and trained on same amount of data, thus removing these factors as confounders. In addition to the pre-trained versions, we evaluate models fine-tuned on StackOverflow question-answer retrieval (SQA) and type inference tasks (denoted with suffixes `-sqa` and `-typeinf`). These specific tasks were selected to investigate a critical performance anomaly: CodeSSM significantly outperforms RoCoder on SQA but lags behind on type inference (Verma et al., 2025). The performance anomaly suggests that CodeSSM might not be a good architecture for tasks with a large class size and long-and-short dependencies such as type inference. Generative tasks also have similar modeling requirements as type inference and hence, improving the architecture, and performance, on type inference might also help improve SSM's performance on generative modeling.

The evaluation is performed using the classifier-free framework established by Anand et al. (2024), which leverages the *DirectProbe* algorithm (Zhou and Srikumar, 2021) to measure the quality of hidden representation. By using a classifier-free interpretability method we remove any impact of classifier training on conclusions drawn. Direct probe uses convex optimization to create disjoint clusters of hidden representation where a single class label may be represented by multiple distinct clusters (see Appendix C for details). To quantify representational quality, these generated clusters are used for nearest-neighbor classification of held-out data. High classification accuracy indicates that the the model has effectively captured the code property.

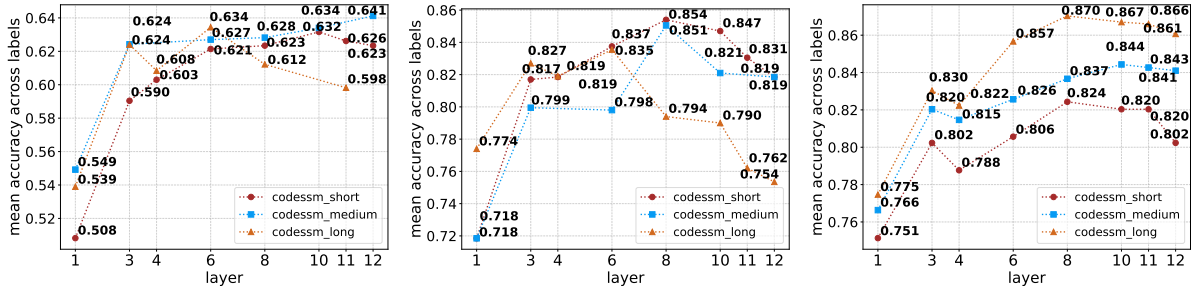
We evaluate the hidden representations using three probing tasks established by Anand et al. (2024), designed to isolate specific syntactic and semantic capabilities. Syntactic structure is modeled via the Abstract Syntax Tree (AST), while semantic relationships are captured using the Data Flow Graph (DFG), where edges denote dependencies between variables. The tasks are: 1) **AST Sibling Prediction**: Determines whether two tokens share a parent node in the Abstract Syntax Tree (AST). This task measures the model's grasp of local syntactic relationships; 2)**AST Distance Prediction**: Estimates the shortest path length between two tokens in the AST. This serves as a proxy for short and long-range syntactic understanding and the model's understanding of program flow (Anand et al., 2024); and 3)**DFG Edge Prediction**: Identifies whether a data flow edge exists between two variable occurrences in the Data Flow Graph (DFG). This task evaluates the model's encoding



(a) Comparison of hidden representation of CodeSSM and RoCoder.



(b) Comparison of hidden representation of CodeSSM and its finetuned variants



(c) Comparison of hidden representation of CodeSSM on varying context lengths

Figure 2: Result of hidden representation analysis in terms of mean accuracy across task labels on distance (left), siblings (center) and edge (right) prediction tasks.

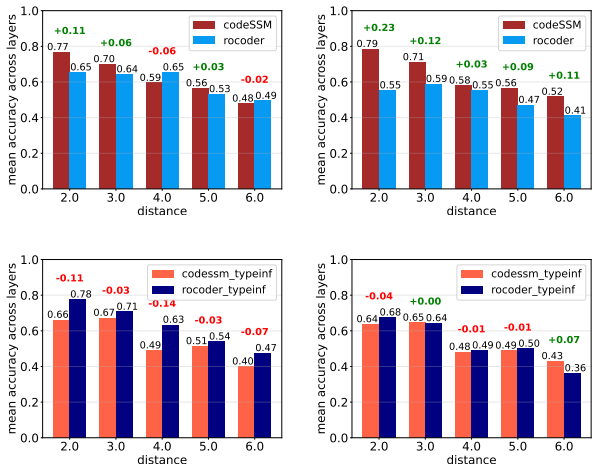


Figure 3: Accuracy of CodeSSM, CodeSSM-typeinf, Rocoder and Rocoder-typeinf on distance prediction tasks for layers 6 and 10.

of semantic dependencies.

3.2 Analysis Results

We performed direct probe analysis of CodeSSM and RoCoder on varying lengths: Short (up to 500), Medium (1k-2k), Long (2k-8k). Figure 2 presents the probing accuracy across tasks.

Pre-training. CodeSSM exhibits superior syntactic capture compared to RoCoder (Figure 2a). While semantic performance is comparable in early layers, CodeSSM significantly outperforms RoCoder in deeper layers. Notably, RoCoder suffers from representational degradation in the final layers, a phenomenon consistent with other Transformer architectures (Anand et al., 2024); the inclusion of Rotary Positional Embeddings (RoPE) appears insufficient to mitigate this limitation.

Length Extrapolation. In Figure 2c, we observe that CodeSSM shows the best accuracy in capturing semantic properties (DFG task) for long lengths, showing remarkable length extrapolation

up to 8k context length (32x the pretraining context of 256). The superior performance on the DFG task implies that CodeSSM captures long-range dependencies very well especially for long lengths. However, CodeSSM shows representational degradation in deeper layers for local dependencies (i.e., siblings tasks) with long inputs. Nevertheless, CodeSSM retains both syntactic and semantic information upto 8x pretraining context, while RoCoder does not (see Appendix D).

Fine-tuning reveals distinct adaptation dynamics between the architectures. CodeSSM forgets certain code relations learned during pretraining (Figure 2b). While this degradation is negligible for SQA, with CodeSSM_sqa performing better than RoCoder_sqa, it is significant in case of type inference. To diagnose this, we analyze performance by token distance in layers 6 and 10 (Figure 3). We observe that fine-tuning on type inference causes CodeSSM to lose syntactic information, particularly at short ranges. Conversely, RoCoder significantly improves its short-range syntactic modeling after fine-tuning on type inference.

This divergence is consistent with the downstream performance. Type inference demands both local (short-range) and global (long-range) context understanding. While RoCoder successfully adapts to these requirements, improving its short-range syntactic understanding during fine-tuning, CodeSSM fails to retain its pre-trained syntactic and semantic knowledge. Consequently, when a task requires both short and long range understanding, CodeSSM's representations fail.

While previous work, such as Zuo et al. (2024), has argued that SSMs are poor at capturing short-range dependencies, our work is the first to provide evidence for this in a multilayer SSM-based model. This complementarity – SSMs for global context and Transformers for local dependencies – provides potential explanation for the success of hybrid models (Ren et al., 2025; Dao and Gu, 2024). However, the underlying mechanics remain opaque, prompting two critical questions: 1) Why do some layers capture more code properties? 2) Why does code understanding decrease on the type inference task? To answer these questions we designed the kernel analysis.

4 SSM Kernel Analysis

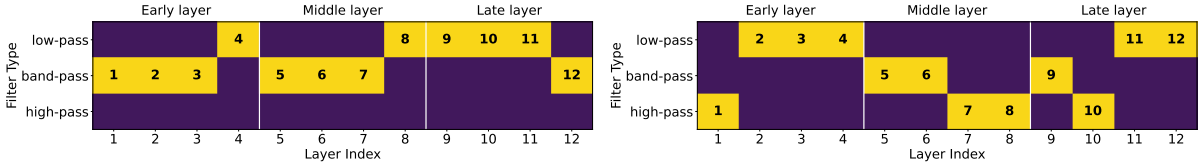
To systematically analyze the token dependencies captured by state-space models, we propose *SSM-Interpret*, a novel framework that categorizes SSM kernels based on their spectral characteristics. By mapping kernels to the frequency domain, this framework allows us to infer whether a model prioritizes local (high-frequency) or global (low-frequency) token interactions (Figure 6). We use the framework to analyze CodeSSM which uses S4D (Gu et al., 2022a) with a single kernel shared across all channels. The size of the kernel equals the context length. We set the context length to 4096, however, the framework is context length agnostic (see Appendix B.2). Each layer has two SSM kernels (one forward and one backward as shown in Figure 1). The *SSM-Interpret* is applicable to complex NPLR and DPLR S4 implementations (Gu et al., 2022b) as well.

4.1 The SSM-Interpret Kernel Analysis

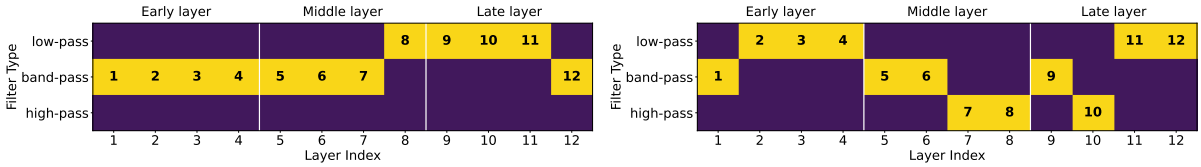
The core objective of our kernel analysis is to characterize the behavior of SSM kernels in terms of the range of dependency captured. To achieve this, we extract the forward and backward kernels from all 12 layers of CodeSSM and analyze their spectral properties via the Fourier transform. By examining which frequency bands are amplified or attenuated, we classify kernels into low-pass (long-range), high-pass (short-range), or band-pass categories, following established links between spectral behavior and dependency range (Ravikumar et al., 2026).

Prior approaches rely on the dominant frequency (frequency with the highest magnitude) for categorization (Ravikumar et al., 2026). However, a kernel may exhibit a sharp peak in one spectral region while the majority of its energy resides elsewhere, leading to misclassifications. To mitigate this, we propose a robust classification strategy based on total spectral energy distribution. We employ two complementary metrics: Spectral Centroid (SC) (Blackledge, 2006) and Low-to-High Frequency Energy Ratio (LHFR) (Constantinescu and Brad, 2023). Reliance on the spectral centroid alone risks mirroring the pitfalls of dominant frequency analysis by ignoring secondary high-energy regions (Massar et al., 2011). However, our empirical evaluation demonstrates that combining the spectral centroid with LHFR yields a robust and perceptually accurate classification of kernel behavior.

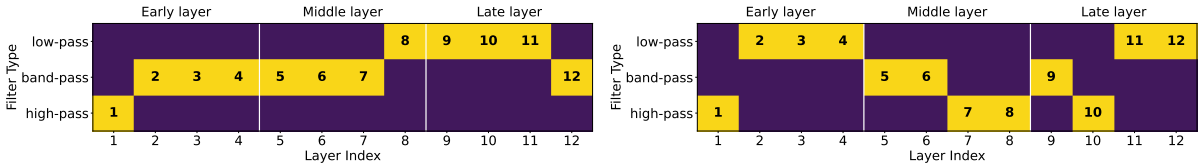
SC is the weighted mean of all the frequencies in the frequency domain representation of a signal, where the weights are the magnitude of each frequency component. Often described as the spec-



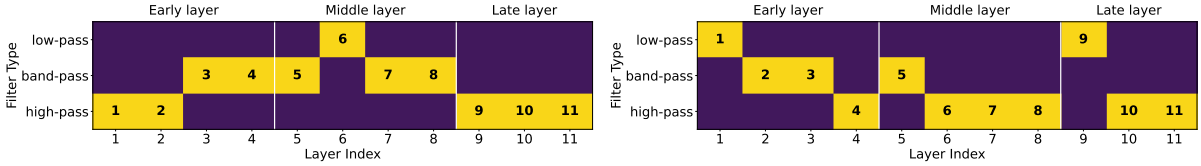
(a) CodeSSM



(b) CodeSSM_sqa



(c) CodeSSM_typeinf



(d) CodeSSM-HF

Figure 4: Layer wise filter classification of forward (left) and backward kernel (right) of CodeSSM and its variants.

trum's "center of gravity," the SC identifies the point around which the spectral energy is concentrated. A higher centroid value indicates a concentration of energy towards the high-frequency region. Formally, we calculate the SC as:

$$\text{SC} = \frac{\sum_{n=0}^{N-1} f(n)\, X(f(n))}{\sum_{n=0}^{N-1} X(f(n)} \tag{1}$$

where n denotes the index of the frequency component, $f(n)$ is the frequency at index n, and $X(f(n))$ is the magnitude of the Fourier transform at $f(n)$. Based on the computed SC, we classify kernel behavior using the following thresholds, derived from the observed frequency distribution of CodeSSM kernels:

$$\begin{aligned} &\textbf{Low-pass:} && \text{Centroid} < \frac{1}{3} \cdot 0.5 \approx 0.16 \\ &\textbf{High-pass:} && \text{Centroid} > \frac{2}{3} \cdot 0.5 \approx 0.33 \\ &\textbf{Band-pass:} && \text{otherwise} \end{aligned} \tag{2}$$

Here, 0.5 represents the normalized highest representable frequency (in cycles per sample).

LHFR measures the balance of spectral power by comparing the energy concentrated in the lower versus upper frequency bands, i.e., $\text{LHFR} = \frac{E_{\text{low}}}{E_{\text{high}}}$, where E_{low} is the sum of magnitudes within the bottom 10% of the frequency spectrum, and E_{high} is the sum within the top 40%. This ratio serves as an indicator of whether the kernel focuses on global structure (low-frequency dominance) or local ones (high-frequency dominance). We classify the kernels based on the LHFR as follows:

$$\begin{aligned} &\textbf{Low-pass:} && \text{LHFR} \gg 10 \\ &\textbf{High-pass:} && \text{LHFR} \ll 1 \\ &\textbf{Band-pass:} && \text{otherwise} \end{aligned} \tag{3}$$

Kernel Categorization. We combine the SC and LHFR metrics to categorize the SSM kernels into low-pass, high-pass, or band-pass filters. Figure 5 illustrates the resulting classification for CodeSSM. We established the specific thresholds for both metrics through a rigorous qualitative analysis of the kernel spectral profiles of all the models and our chosen values minimizes the outliers. Detailed justifications for these criteria are provided in Appendix B.1.

4.2 Kernel Analysis Results

Why do some layers capture more code properties? The Forward and backward kernels of certain CodeSSM layers exhibit complementary frequency responses. For instance, in layer 8, the forward kernel acts as a high-pass filter and the backward kernel as a low-pass filter, indicating that most of the forward kernel's energy lies in the high-frequency range, whereas the backward kernel concentrates in the low-frequency range.

This complementary filter behavior suggests that using two directionally distinct kernels enables the model to capture richer contextual information from both past and future tokens as well as the far away and nearby tokens when making predictions. Notably, layers displaying such complementary forward–backward kernel patterns tend to capture the most code-related properties. For example, layer 8 achieves the best performance on the siblings and edge prediction tasks, while layer 10 performs best on the distance task (see Figure 2a). Both layers exhibit complementary filter characteristics (layer 8: forward–high-pass, backward–low-pass; layer 10: forward–low-pass, backward–high-pass; see Figure 4a).

In contrast, layers where both the forward and backward kernels emphasize the same frequency range tend to perform poorly. For instance, layer 11 of CodeSSM-TypeInf, which exhibits low-pass behavior in both kernels, shows reduced performance on the sibling and DFG tasks.

Why does code understanding decrease on the type inference task? In the fine-tuned models, the forward kernel behavior in the initial layers shifts toward higher frequencies compared to the pretrained models. For instance, in Figure 4c, the forward kernels of layers 1 and 4 transition to high-pass and band-pass filters, respectively. Additionally, the frequency response of both forward and backward kernel of layer 1 lies in the same region of the spectrum (high-pass) due to which the kernel misses a wide range of code properties which lie in the other region of the spectrum. The layers with complementary filter behavior (layer 8 and layer 10 of CodeSSM_typeinf) still capture more code properties (Figure 2b) but the proportion of code properties captured reduces significantly due to early high frequency shift.

Because the outputs of the forward and backward kernels are multiplied in the time domain (see Figure 1), they become convolved in the frequency domain. When both kernels exhibit high-pass characteristics in the initial layers (as in Figure 4c), this convolution amplifies the high-frequency response, potentially further biasing the model toward short-range dependencies. Moreover, emphasizing short-range dependencies too strongly in the early layers limits the model's ability to learn long-range dependencies, as reflected in the drop in accuracy at distances 5 and 6 shown in Figure 3.

As discussed in Section 3.2, the type inference task depends on both short- and long-range dependencies. Transformers can flexibly adapt to short-range syntactic patterns, but CodeSSM lacks this flexibility due to its use of only two SSM kernels per layer. In Figure 4c, we observe that CodeSSM shifts toward learning short-range dependencies, yet without sufficient capacity to do so effectively. Verma et al. (2025) hypothesized that CodeSSM's decreased performance on the type inference task stems from its inability to capture short-range dependencies. Our filter behavior and hidden-representation (distance 2 and 3 in Figure 3) analyses are consistent with this hypothesis.

5 Analysis-Driven Improvements

Our analysis of hidden representations and SSM kernels has uncovered the strengths and critical limitations in the CodeSSM architecture. Leveraging these insights, we propose targeted architectural modifications which yield significant performance gains across code understanding tasks.

5.1 Proposed Modifications

Our analyses reveal that CodeSSM struggles with tasks that require understanding of short- and long-range dependencies due to lack of encoding high-frequency. While hybrid architectures interleaving Transformer layers offer a solution (Zuo et al., 2024; Ren et al., 2025), they also have the quadratic complexity of self-attention.

CodeSSM-HF. Given that the limitation is due to the handling of high-frequency information, we instead propose an efficient alternative which we refer as `CodeSSM-HF`: introducing a dedicated high-frequency path in parallel to the low-frequency SSM path (see Figure 1). This path employs a 1D Convolutional Neural Network (CNN) with a kernel size of 3 (Abello et al., 2021), providing a strong inductive bias for short-range dependencies (attending to the current, preceding, and succeeding tokens) analogous to local attention heads in Trans-

formers. To offset the parameter increase from the CNN, we employ grouped convolutions (group size 8) and reduce the depth to 11 layers. These constraints ensure CodeSSM-HF, has slightly fewer parameters than the original CodeSSM. As shown in Table 1, incorporating a high-frequency path yields consistent performance gains on the SQA and type inference tasks, confirming the critical role of high frequency information across tasks. The performance gain is also observed on the adversarial NLCodeSearch task (Lu et al., 2021).

Multi-kernel. In Section 4.2, we hypothesized that CodeSSM's performance on type inference might be hindered by having just two SSM kernels per layer. To evaluate the hypothesis, we train a model with 1024 kernels (i.e., equal to the input dimensions)(referred as `CodeSSM-1024k`). This is similar to previous works on SSMs (Gu et al., 2022a,b). While having 1024 kernels improves the performance on NLCodeSearch and type inference, the performance remains same on SQA and the performance on NLCodeSearch is lower than CodeSSM-HF (Table 1).

To understand the trade-off between increasing capacity and performance, we pretrained CodeSSM with kernels 1, 4, 8, ..., 512, 1024 on a small dataset. The pretraining performance was best for 8 kernels. Subsequently, we pretrained a model with 8 kernels (`CodeSSM-8k`). This variant achieves the best performance on all tasks. In this model, each kernel is shared across 128 dimensions. These results suggest that increasing capacity while maintaining shared transformation across some dimensions, is optimal for better code understanding.

5.2 Kernel Analysis of CodeSSM-HF

Since CodeSSM-HF retains the single-kernel SSM block structure, we directly compare its kernels against the baseline CodeSSM using our *SSM-Interpret* analysis. Figure 4d shows the kernel classification of CodeSSM-HF (see Appendix B.4 for analysis of CodeSSM-8k). We observe two distinct spectral shifts in the CodeSSM-HF kernels.

First, the HF model exhibits a greater prevalence of high-pass filters, particularly in the early layers, unlike CodeSSM. In these layers, we observe a complementary spectral behavior where high-pass filters are paired with low- or band-pass filters. Moreover, unlike CodeSSM kernels, which typically display a single dominant frequency which decays on each side, CodeSSM-HF kernels often possess multiple crests across frequency bands (see Figure 14). This spectral heterogeneity allows early layers to simultaneously capture both long- and short-range dependencies, correcting the "blind spots" identified in 4.2. Consequently, this simple architectural modification enhances the SSM's ability to model rich token-level dependencies, leading to improved overall performance.

Table 1: Results on NLCodeSearch, SQA and Type Inference tasks as mean over 3 different seeds (Standard Deviation in subscript). The best performance is in bold.

Model	NLCodeSearch (MRR)	SQA (MRR)	Type Inference (F1)
CodeSSM	$25.31_{0.51}$	$76.02_{0.08}$	$59.24_{0.45}$
CodeSSM-HF	$30.26_{0.37}$	$78.33_{0.10}$	$60.04_{0.36}$
CodeSSM-8K	$\mathbf{31.30}_{0.25}$	$\mathbf{79.80}_{0.25}$	$\mathbf{61.22}_{0.22}$
CodeSSM-1024K	$28.19_{0.41}$	$76.01_{0.12}$	$60.38_{0.39}$

Second, the final two layers of CodeSSM-HF shift predominantly toward high-pass behavior. This contrasts with CodeSSM, where later layers show low- or band-pass characteristics. This shift toward high-pass behavior in the later layers (reflecting a focus on short-range dependencies) is intuitively consistent with effective hierarchical representation learning: after the early layers perform extensive global mixing through the SSM convolution kernel (low-pass), long-range context is effectively "localized" into the hidden states. Consequently, the deeper layers only need to model short-range dependencies.

6 Conclusion

In this work, we presented the first systematic analysis of hidden representations and convolution kernels in multi-layer SSMs. Our empirical results reveal a nuanced landscape: while SSMs fundamentally outperform Transformers in capturing global code properties, they exhibit weaknesses in modeling short-range dependencies, particularly in tasks like type inference. To diagnose this failure of SSMs, we proposed a novel interpretability framework to analyze the convolutional kernel of SSMs, which showed a shift towards short range dependency understanding during fine-tuning. Guided by these insights, we introduced architectural refinements that significantly narrow this performance gap.

We anticipate that the introduced framework will serve as a foundation for future interpretability research. Additionally, with the architectural improvements SSMs can be scaled and adapted for generative tasks in future work.

Limitations

Broadly, our work has the following limitations.

First, the SSM-interpret framework considers several threshold values for classifying the kernel as low-pass, high-pass, or band-pass, and for calculating LHFR. Although these threshold values are empirically validated, they may not be optimal for some kernels or other models. Thus, future research can refine classification thresholds for the spectral centroid and the low-to-high frequency energy ratio by accounting for the kernel's frequency response, thereby improving the robustness of frequency-based kernel classification.

Second, we did not perform the analysis of generative models. Although there are no S4D-based generative models, our findings show that this absence may not be accidental. Similar to type inference, generative tasks involve a large vocabulary and require modeling both short- and long-range dependencies simultaneously. Thus, the architectural modifications we proposed can guide future research in developing S4/S4D-based generative models.

Third, we performed the hidden representation analysis for lengths up to 8k. Currently, models are also being evaluated for lengths up to 32k. However, we limited our analysis to 8k due to resource constraints and the small pre-training context of CodeSSM and RoCoder (256). Our analysis still covers a 32x length extrapolation. Based on our hypothesis and observations, CodeSSM and its variants will show significantly better code understanding with length extrapolation than transformer models.

Ethical Considerations

The aim of our work is to analyze and understand the success and failure of SSMs with respect to code properties. The analysis does not raise any ethical risks. In our work, we have used models whose weights are publicly available and are trained on publicly available datasets. We have also trained models on publicly available data. It is not clear whether the models can leak personally identifiable information if present in the pretraining dataset, but we acknowledge that such potential risk exists.

References

Antonio A. Abello, Roberto Hirata, and Zhangyang Wang. 2021. Dissecting the high-frequency bias in convolutional neural networks. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 863–871.

Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, Prem Devanbu, and Kenji Sagae. 2023. Towards understanding what code language models learned. *CoRR*, abs/2306.11943.

Ameen Ali Ali, Itamar Zimerman, and Lior Wolf. 2025. The hidden attention of mamba models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1516–1534, Vienna, Austria. Association for Computational Linguistics.

Abhinav Anand, Shweta Verma, Krishna Narasimhan, and Mira Mezini. 2024. A critical study of what code-LLMs (do not) learn. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 15869–15889, Bangkok, Thailand. Association for Computational Linguistics.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. 2023. Zoology: Measuring and improving recall in efficient language models. *Preprint*, arXiv:2312.04927.

Seyedarmin Azizi, Souvik Kundu, Mohammad Erfan Sadeghi, and Massoud Pedram. 2025. Mambaextend: A training-free approach to improve long context extension of mamba. In *The Thirteenth International Conference on Learning Representations*.

Yonatan Belinkov. 2022. Probing classifiers: Promises, shortcomings, and advances. *Comput. Linguistics*, 48(1):207–219.

J. M. Blackledge. 2006. *Digital Signal Processing: Mathematical And Computational Methods, Software Development And Applications (Second Edition)*. Horwood Publishing Limited.

Tianyi Chen, Pengxiao Lin, Zhiwei Wang, and Zhi-Qin John Xu. 2025. Achilles' heel of mamba: Essential difficulties of the mamba architecture demonstrated by synthetic data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Constantin Constantinescu and Remus Brad. 2023. An overview on sound features in time and frequency domain. *International Journal of Advanced Statistics and ITC for Economics and Life Sciences*, 13:45–58.

Tri Dao and Albert Gu. 2024. Transformers are ssms: generalized models and efficient algorithms through structured state space duality. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. 2023. Hungry hungry hippos: Towards language modeling with state space models. In *The Eleventh International Conference on Learning Representations*.

Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. 2022a. On the parameterization and initialization of diagonal state space models. In *Advances in Neural Information Processing Systems*, volume 35, pages 35971–35983. Curran Associates, Inc.

Albert Gu, Karan Goel, and Christopher Re. 2022b. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*.

Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. 2021. Combining recurrent, convolutional, and continuous-time models with linear state space layers. In *Advances in Neural Information Processing Systems*, volume 34, pages 572–585.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control 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 2019, Hong Kong, China, November 3-7, 2019*, pages 2733–2743. Association for Computational Linguistics.

Ermo Hua, Che Jiang, Xingtai Lv, Kaiyan Zhang, Youbang Sun, Yuchen Fan, Xuekai Zhu, Biqing Qi, Ning Ding, and Bowen Zhou. 2025. Fourier position embedding: Enhancing attention's periodic extension for length generalization. In *Forty-second International Conference on Machine Learning*.

Farnoush Rezaei Jafari, Grégoire Montavon, Klaus Robert Muller, and Oliver Eberle. 2024. MambaLRP: Explaining selective state space sequence models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. 2024. Repeat after me: Transformers are better than state space models at copying. *Preprint*, arXiv:2402.01032.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1332–1336. IEEE.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, and 39 others. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari A. Sahraoui. 2022. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 11:1–11:11. ACM.

Jiahao Lu. 2025. Frequency regularization: Unveiling the spectral inductive bias of deep neural networks. *Preprint*, arXiv:2512.22192.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Melody L. Massar, Matthew Fickus, Erik Bryan, Douglas T. Petkie, and Andrew J. Terzuoli. 2011. Fast computation of spectral centroids. *Adv. Comput. Math.*, 35(1):83–97.

Rowan Hall Maudslay, Josef Valvoda, Tiago Pimentel, Adina Williams, and Ryan Cotterell. 2020. A tale of a probe and a parser. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7389–7395. Association for Computational Linguistics.

William Merrill, Jackson Petty, and Ashish Sabharwal. 2024. The illusion of state in state-space models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Eric Nguyen, Karan Goel, Albert Gu, Gordon W. Downs, Preey Shah, Tri Dao, Stephen A. Baccus, and Christopher Ré. 2022. S4nd: modeling images and videos as multidimensional signals using state spaces. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Naoki Nishikawa and Taiji Suzuki. 2025. State space models are provably comparable to transformers in dynamic token selection. In *The Thirteenth International Conference on Learning Representations*.

Gonçalo Paulo, Thomas Marshall, and Nora Belrose. 2024. Does transformer interpretability transfer to rnns? *Preprint*, arXiv:2404.05971.

Biqing Qi, Junqi Gao, Kaiyan Zhang, Dong Li, Jianxing Liu, Ligang Wu, and Bowen Zhou. 2024. SMR: State memory replay for long sequence modeling. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 8102–8116, Bangkok, Thailand. Association for Computational Linguistics.

Srividya Ravikumar, Abhinav Anand, Shweta Verma, and Mira Mezini. 2026. Analysis of long range dependency understanding in state space models. *Preprint*, arXiv:2601.13048.

Liliang Ren, Yang Liu, Yadong Lu, yelong shen, Chen Liang, and Weizhu Chen. 2025. Samba: Simple hybrid state space models for efficient unlimited context language modeling. In *The Thirteenth International Conference on Learning Representations*.

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

Sora Togawa and Kenya Jin'no. 2025. Understanding convolutional neural networks through z-transform: Frequency domain analysis of kernel feature extraction. *Nonlinear Theory and Its Applications, IEICE*, 16(4):878–895.

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. 2021. Mlp-mixer: An all-mlp architecture for vision. *Preprint*, arXiv:2105.01601.

Asher Trockman and J. Zico Kolter. 2022. Patches are all you need? *Preprint*, arXiv:2201.09792.

Shweta Verma, Abhinav Anand, and Mira Mezini. 2025. CodeSSM: Towards state space models for code understanding. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 34219–34235, Suzhou, China. Association for Computational Linguistics.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2377–2388, New York, NY, USA. Association for Computing Machinery.

Junxiong Wang, Jing Nathan Yan, Albert Gu, and Alexander Rush. 2023. Pretraining without attention. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 58–69, Singapore. Association for Computational Linguistics.

Shida Wang and Qianxiao Li. 2024. Stablessm: alleviating the curse of memory in state-space models through stable reparameterization. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Kang Yang, Xinjun Mao, Shangwen Wang, Yihao Qin, Tanghaoran Zhang, Yao Lu, and Kamal Al-Sabahi. 2023. An extensive study of the structure features in transformer-based code semantic summarization. In *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*, pages 89–100. IEEE.

Yichu Zhou and Vivek Srikumar. 2021. Directprobe: Studying representations without classifiers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 5070–5083. Association for Computational Linguistics.

Itamar Zimerman, Ameen Ali, and Lior Wolf. 2024. Explaining modern gated-linear rnns via a unified implicit attention formulation. *Preprint*, arXiv:2405.16504.

Simiao Zuo, Xiaodong Liu, Jian Jiao, Denis X Charles, Eren Manavoglu, Tuo Zhao, and Jianfeng Gao. 2024. Efficient hybrid long sequence modeling with state space augmented transformers. In *First Conference on Language Modeling*.

A Additional Related Works

Frequency Domain Analysis.

Prior research on frequency-domain analysis has primarily focused on feature extraction within Convolutional Neural Networks (CNNs) or input-dependent filter analysis. For instance, Togawa and Jin'no (2025) evaluated CNN kernels via the Z-transform to classify them based on their amplitude spectrum medians. Additionally, a more recent work (Ravikumar et al., 2026) focuses on the analysis of CNNs and single layer SSMs using time and frequency domain analysis. The classification criteria used by Ravikumar et al. (2026) is based on dominant frequency. The work also shows how long-range dependency understanding varies across different architectures of SSMs. However, we have talked about the disadvantages of relying on dominant frequency in Section 4.1.

In the context of generalization, Lu (2025) demonstrated that regularization enforces a strong spectral bias towards low frequencies. Similarly, regarding architectural constraints, Nguyen et al. (2022) filtered out high-frequency components to prevent aliasing in multi-dimensional state space models, while Abello et al. (2021) found that CNNs exhibit a tendency to learn high- and mid-frequency patterns over low-frequency ones. Frequency-domain analysis has also been applied to Transformers to understand length generalization (Hua et al., 2025), though this work was restricted to positional embeddings. In contrast to these approaches, ours is the first to develop a framework specifically for the frequency-domain analysis of SSM kernels, a framework that remains applicable to CNNs as well.

Explainability in SSMs. Numerous studies have conducted both theoretical and empirical analyses of selective state space models (SSM), particularly Mamba. These investigations have identified

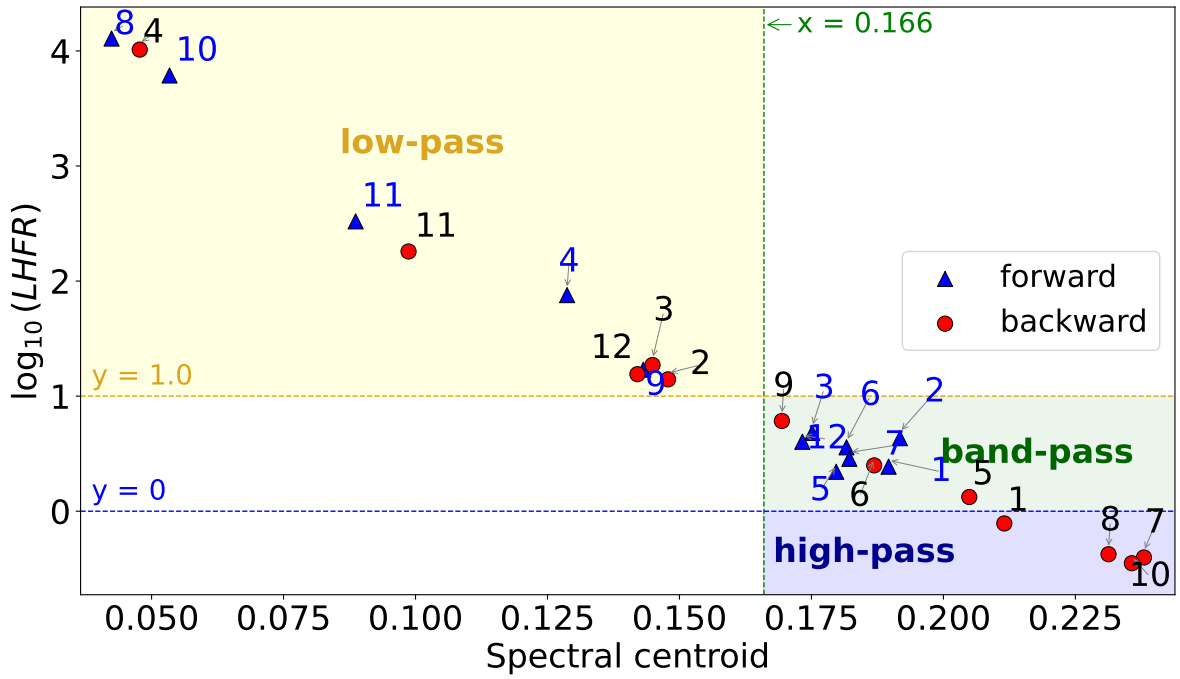


Figure 5: Filter classification of each layer of the CodeSSM on the basis of spectral centroid and LHFR.

several notable limitations of Mamba. For example, Chen et al. (2025) demonstrate that Mamba relies on non-linear convolution to retrieve relevant information, and that this non-linearity introduces an asymmetry bias, which impedes the model's ability to recognize symmetrical patterns and relationships. The authors propose straightforward mitigation strategies, such as incorporating residual paths or gating around the convolution, and report promising improvements in SSM models. However, their analysis is restricted to synthetic tasks. Jafari et al. (2024) apply Layer-wise Relevance Propagation (LRP) to Mamba and identify specific architectural components responsible for generating unfaithful explanations. They further introduce more interpretable variants of Mamba, termed MambaLRP. Additional studies have employed attention-based explainability techniques for Mamba (Zimerman et al., 2024). Ali et al. (2025) argue that Mamba can be interpreted as an attention-based model and provide a theoretical comparison between the mechanisms underlying Mamba and those of attention. In their analysis, they reformulate Mamba layers as self-attention and conduct attention-based evaluations. Furthermore, Jelassi et al. (2024) and Arora et al. (2023) report that Mamba exhibits difficulties with input copying tasks.

Qi et al. (2024) employed event-triggered control to conduct a theoretical analysis of the stability of SSMs. Wang and Li (2024) demonstrated that SSMs lacking reparameterization are subject to the "curse of memory," a limitation also observed in Recurrent Neural Networks. They introduced a novel reparameterization that enhances model performance compared to the approach in Gu et al. (2022b). Merrill et al. (2024) established that SSMs, like Transformers, are incapable of representing complex state-tracking problems. The authors further showed that the expressive power of SSMs can be increased by utilizing input-dependent transition matrices. Additionally, Nishikawa and Suzuki (2025) found that SSMs, when combined with specific non-linearities, exhibit dynamic token selection abilities comparable to those of Transformers.

B Kernel Analysis

B.1 Kernel Classification

We analyze both the forward and backward kernels of CodeSSM across all 12 layers, classifying their behavior based on the frequency ranges they amplify or attenuate. Kernels are categorized as low-pass, high-pass, or band-pass, reflecting whether they primarily capture long-range dependencies or focus on local interactions (Ravikumar et al., 2026). Low-pass kernels correspond to long-range dependency capture, whereas high-pass kernels empha-

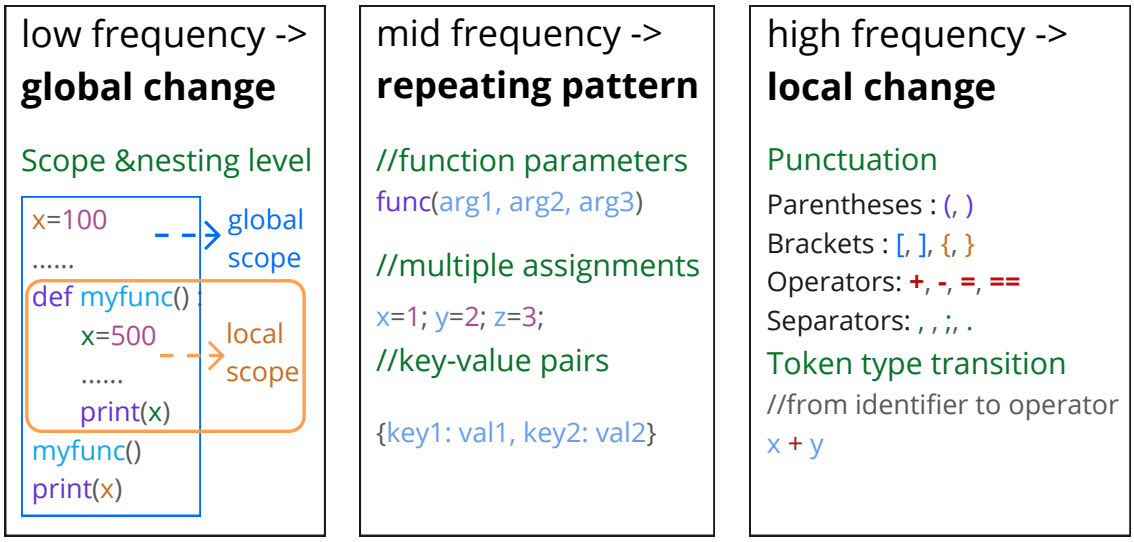


Figure 6: Relationship between frequency ranges and code patterns.

size local dependencies. The relationship between frequency and code token representation is illustrated in Figure 6.

In the kernel analysis, we characterize the frequency response of the filters based on the concept that originates from signal processing. The kernel weights represent the contribution of the token at each time step in the sequence to the current token prediction. Since these weights are learned parameters rather than data samples, we compute the Fourier Transform of the kernel weights using a sample rate fs = 1.0, meaning one sample per unit time. Because frequency components are normalized, the resulting frequencies are relative rather than absolute. We therefore measure the relative energy distribution across frequencies to classify filters.

In traditional signal analysis, the cutoff frequency is used for non-normalized responses where frequency is measured in cycles per second. However, since our kernel frequencies are normalized, a specific cutoff frequency is not meaningful. Instead, we use the spectral centroid and the Low-to-High Frequency Ratio (LHFR) as proxy measures for cutoff frequency. While LHFR can depend on the choice of cutoff or median frequency, we found that using the median often introduced numerous outliers. To address this, we qualitatively analyzed the kernels and experimented with multiple threshold values. The most stable results—yielding no outliers across all CodeSSM model kernels—were achieved by defining low- and high-frequency energy thresholds at 10% and 40%, respectively.

We also performed multiple ablation studies varying the spectral centroid and LHFR thresholds to refine the classification criteria. For the CodeSSM model and its variants, the threshold values reported in the paper provided the most consistent and robust classification, minimizing outliers across all experiments. The final kernel classification results using the selected criteria are shown in Figure 5.

B.2 Effect of Kernel Length

The Spectral Centroid (SC) and LHFR are computed over normalized frequencies and are invariant to absolute kernel length. We observe a change of less than 0.8% on SC when the kernel length is increased from 256 to 4096. The filter classifications are therefore stable across sequence lengths.

Importantly, the kernel classification remains the same for kernels of size 4096 and 256. Thus, the same conclusions hold for smaller or larger kernels under a given training setup. This means that SSM-Interpret generalizes to longer-context scenarios: as sequence length scales, the metrics remain meaningful and the framework remains applicable without modification.

The filter classification in paper is on kernel length 4096.

B.3 Spectral Behaviour of SSM Kernels

In addition to the filter classification of the kernels, we also show the spectral power of the kernels as a heatmap in Figure 13. The layers with warm colors (high spectral power) at lower frequencies and cool colors (low spectral power) at higher frequencies show low pass behavior. A constant cool color (low spectral power) across low to high frequencies shows high pass behavior. A slight change in color (small change in spectral power) from low to high frequencies show band pass behavior. We can also see a lot of warm colors in CodeSSM-8k at lower frequencies which shows a higher spectral power towards lower frequencies. Additionally, we observe a lot of variation in spectral power at different frequencies, which shows that each kernel focus on different characteristics of code.

B.4 Kernels of CodeSSM-8k

CodeSSM-8k is the best performing CodeSSM variant and has 8 kernel per layer. We studied the kernels of this model. The kernels for layer 1 are shown in Figure 15 and mean of all 8 kernels of each layer is shown in Figure 13. Some characteristics of these kernels stand out.

First, the kernels model complex relations across tokens, similar to CodeSSM-HF model. Second, we observe redundancies and range shifts. For example, the third and fifth forward kernels capture similar relations. But the third kernel encode it at a shorter range (higher spectral centroid) while the fifth kernel encode similar relations among tokens at longer distances. Similarly, the third and fifth backward kernels encode similar relations. But the third one allocates higher magnitude to lower frequency.

With just 8 kernels, the SSM block is able to model complex token relations. Increasing the number of kernels can result in more redundancies and does not contribute to the performance. Additionally, with significantly more kernels (for example, 1024), there is a higher chance of out of phase kernels in forward and backward paths which can cancel out the encoded token relations when the hidden representation of forward and backward paths are multiplied (equivalent to convolution in frequency domain). We believe this is the reason for lower performance of CodeSSM-1024kernel.

B.5 Regularization in CodeSSM

The low- and high-pass filter behaviors of the SSM kernel also help explain other observations about CodeSSM (Verma et al., 2025). Prior studies have reported a high-frequency bias in CNNs (Abello et al., 2021), but our analysis finds no such bias in CodeSSM. Regularization techniques such as dropout and L2 have been shown to mitigate high-frequency bias (Lu, 2025). However, since the SSM in CodeSSM is not inherently biased toward high frequencies, such regularization is unnecessary. Indeed, Verma et al. (2025) report that incorporating dropout in CodeSSM actually decreases performance.

C Hidden Representation Analysis

C.1 DirectProbe

DirectProbe takes as input a dataset of token representation pairs, each annotated with a relational label (e.g., the existence of an edge in a Data Flow Graph). The algorithm employs an agglomerative clustering approach: initializing each pair as a distinct cluster, it iteratively merges the nearest cl2usters, subject to the constraint that the convex hull of the merged cluster must not intersect with the convex hulls of clusters having different labels. We provide a visual illustration of DirectProbe in Figure 7. Each data point in the input data consists of hidden representation H_i^l and H_j^l for tokens i and j at the output of layer l and a label. As shown in Figure 7, DirectProbe starts with each hidden representation point as a separate cluster. Then nearest clusters with the same labels are merged provided the convex hull of the merged cluster does not overlap with any other cluster. The process results in non-overlapping clusters with each cluster having a single label.

We perform the analysis with three tasks:

Distance prediction. The labels are distance in the AST and range from 2 (minimum possible distance in the tree) to 6. We ignore any token pairs with a distance longer than 6. The clustering is done over difference of the hidden representation.

Siblings prediction. The labels are $siblings$ and $notsiblings$. The clustering is done over the concatenated hidden representations of the two tokens.

Edge prediction. The labels are $noedge$, $comesfrom$ and $computedfrom$. The clustering is done over the concatenated hidden representation of the two tokens.

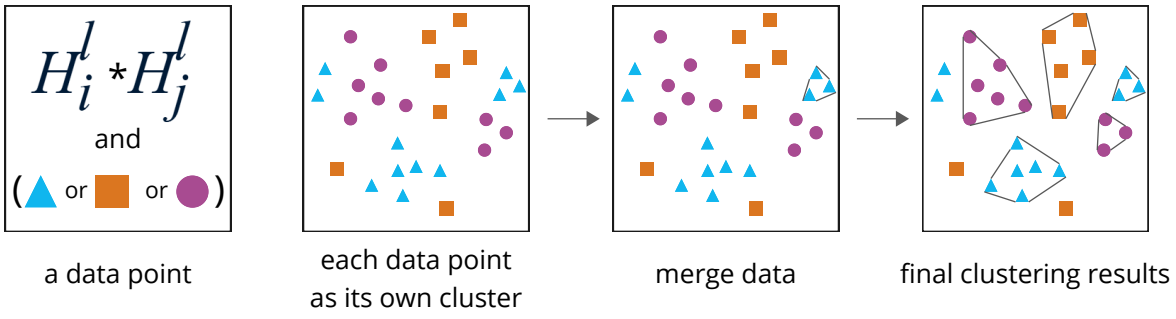


Figure 7: Visual representation of DirectProbe.

On each task, we run the analysis once for each layer of the model.

C.2 Models

We focus on encoder-only architectures as they have been shown to capture code syntax and semantics more effectively than significantly larger decoder-only counterparts (Anand et al., 2024). RoCoder adapts the BERT architecture by replacing absolute positional embeddings with Rotary Positional Embeddings (RoPE) (Su et al., 2024) to enable length generalization. CodeSSM is built upon the BiGS architecture (Wang et al., 2023), with its layer structure detailed in Fig. 1.

D Length Extrapolation

We performed direct probe analysis of CodeSSM and RoCoder on varying lengths: 1) Short (up to 500), 2) Medium (1k-2k), 3) Long (2k-8k). The results on short lengths are presented in Figure 2a, which shows that CodeSSM captures more syntactic and semantic properties of code than RoCoder. The results for medium lengths for CodeSSM and RoCoder are shown in Figure 8. The results for long length are only available for CodeSSM (shown in Figure 9) because RoCoder consumes significantly more memory, and it was not possible to do the analysis with the resources we have. We observe that the gap between CodeSSM and RoCoder increases with increasing context length. CodeSSM captures better syntactic and semantic properties at longer lengths.

CodeSSM shows remarkable length extrapolation up to 8k context length, which is 32 times the pretraining length, while maintaining both syntactic and semantic performance. In Figure 10, we observe that CodeSSM shows the best accuracy in capturing semantic properties (i.e., on the DFG task) for long lengths. The superior performance on the DFG task implies that CodeSSM captures long-range dependencies very well (especially for long lengths). While we observe that CodeSSM shows significantly better understanding of long-range dependencies at long lengths, it shows representational degradation in later layers for local dependencies (i.e., siblings tasks) at long lengths.

The exact mechanism that enables CodeSSM to extrapolate to such a large context is unclear. We hypothesize that it is due to the Discrete Fourier Transform (DFT) in the convolution operation.

Numerous variants of Positional Embeddings have been proposed for the Transformer architecture. RoPE (Su et al., 2024) is the one most commonly used. However, RoPE provides about 2x extrapolation over pretraining context length. Hua et al. (2025) showed that RoPE implicitly performs non-uniform DFT. However, RoPE suffers from spectral leakage due to a single frequency for each dimension. Hua et al. (2025) proposed FoPE which uses a mixture of frequency across each dimension to avoid spectral leakage. FoPE provides upto 4x extrapolation over pretraining context size.

S4D performs explicit DFT in the convolution operation. Additionally, each dimension comprises of a mixture of frequency, just like FoPE. However, unlike FoPE, the basis of the frequency mixture is defined by the A matrix, which has a special initialization – HiPPO (Fu et al., 2023). The HiPPO matrix is designed to improve the long-range memory and thus this special initialization might be the reason for the extreme length extrapolation. Positional Embedding derived from the HiPPO matrix might also help improve length extrapolation of Transformers. However, additional experiments is required to validate this claim robustly. Besides, the HiPPO initialization, the learnable parameters might also help SSM adapt the token relations to the data.

However, SSM alone is not sufficient for length extrapolation (Ravikumar et al., 2026) and has not achieved success on practical real-world applications. In CodeSSM, the SSM routing comprises of two feed forward networks, one preceding and one following the SSM block. This architectural setup is similar to the previously proposed MLP-Mixer (Tolstikhin et al., 2021) and Conv-Mixer (Trockman and Kolter, 2022) architectures. Similar to these architectures, the CodeSSM model separates the channel-wise mixing (using feed forward layer) and the length-wise mixing (using S4D). CodeSSM differs in that the length-wise mixing is performed globally over the entire sequence length.

E Long Range Probing Tasks

We would like to explain why our probing tasks capture long-range dependencies in the sense most relevant to code. Long-range dependency discussions in the literature are typically framed in terms of token distance. However, probing tasks for code are performed over structural representations — AST and DFG — because these capture the semantically meaningful relationships in programs, which do not necessarily align with token distance. A distance of 5–6 in the AST, or an edge in the DFG between two variable occurrences, can span a substantial number of tokens depending on nesting depth, loop structure, and branch complexity. Our distance prediction task explicitly evaluates performance at AST distances 2–6 (Figure 4), with distances 5–6 corresponding to structurally distant token pairs. Similarly, DFG edge prediction captures data-flow dependencies between variables that may be arbitrarily far apart in token space. We also present some examples from our dataset which show that our tasks require long-range dependency understanding. We present some examples for DFG in Figure 16 and Figure 17. There are 1099 and 1486 tokens between two $positions$ variables and the two $start\_time$ variables respectively.

F Additional Evaluation Results

In this section, we present additional evaluation results. Figure 12 shows the comparison between the two models and their fine-tuned version. We can observe the forgetting of code properties after finetuning on type inference. We also show the comparison of RoCoder with its finetuned variants in Figure 11a.

Additional forward and backward kernels have been shown in Figure 14. The figure shows kernel for first layer, middle layer (layer 5) and last layer for both the models. The visualization of these kernels shows that CodeSSM-HF kernels learns more complex relations between tokens compared to CodeSSM across all layers.

Table 2: Comparison of CodeSSM-HF with convolution group sizes of 4 and 8.

Model	Type Inference (F1)
CodeSSM-hf-grp-4	58.76
CodeSSM-hf-grp-8	60.04

G Training Details

All variants of CodeSSM compared in Table 1 were trained under same conditions. The pre-training was done on 4 A100 80GB GPUs. The models were first trained on Wikipedia data with a sequence length of 128 and and a batch size of 256 for 3 days. The models were then trained on 1.8 million git issues samples from StarCoder dataset (Li et al., 2023) for 10 epochs with a per GPU batch size of 64 and 256 sequence length. Finally, the models were trained on 1.8 million code samples from StarCoder dataset with a per GPU batch size of 64 and 256 sequence length. For all pretraining, the learning rate is $5e-5$ with a cosine scheduler and 300 warm up steps.

The training on git issues and code follows Verma et al. (2025) but the training on Wikipedia data is significantly less than that of BiGS Wang et al. (2023). This results in a slightly lower performance of CodeSSM trained by us compared to Verma et al. (2025). Similar to Verma et al. (2025), we used the CodeT5plus-220m tokenizer.

H CodeSSM-HF Group Size

We experiment with group sizes of 4 and 8. We found that having a group size of 8, despite having fewer parameters, performed better than group size of 4. The results of both the configuration is shown in Table 2.

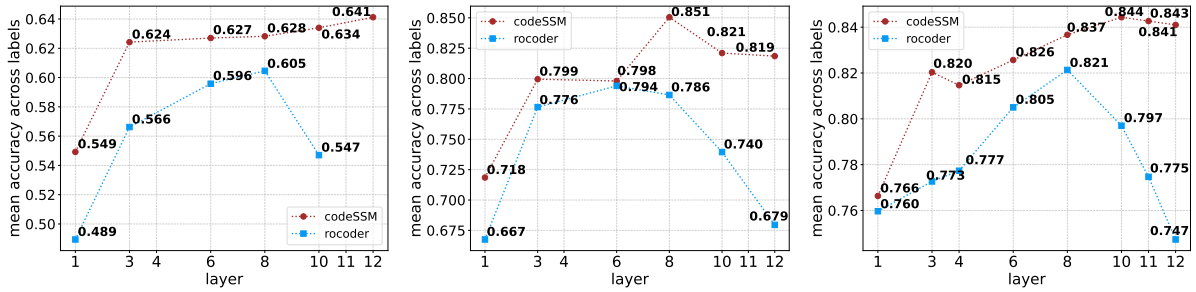


Figure 8: Comparison of hidden representation of CodeSSM and RoCoder on distance (left), siblings (center) and edge (right) prediction tasks for medium lengths.

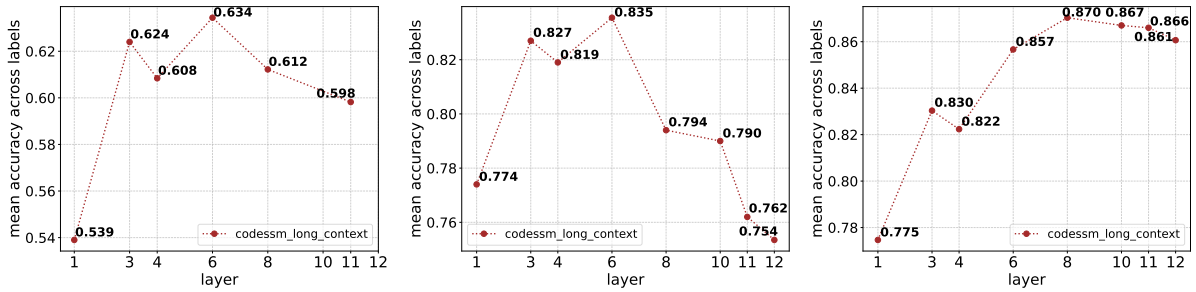


Figure 9: Comparison of hidden representation of CodeSSM and RoCoder on distance (left), siblings (center) and edge (right) prediction tasks for long lengths.

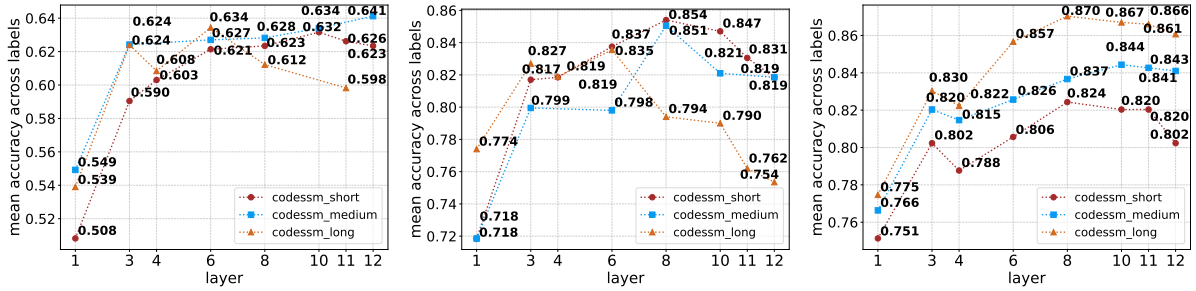
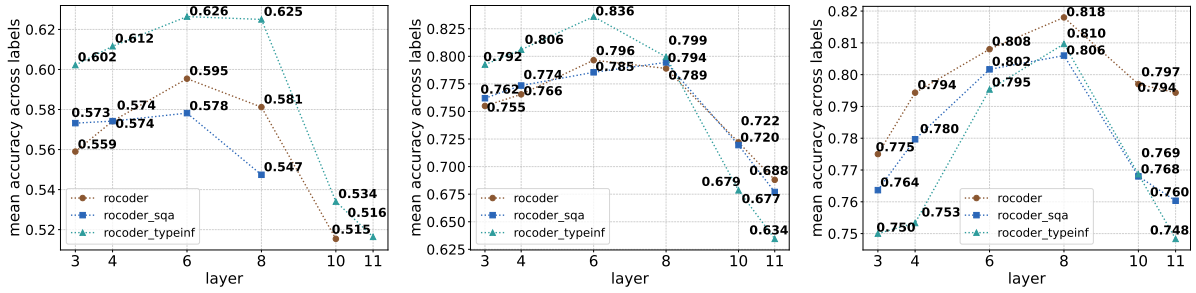
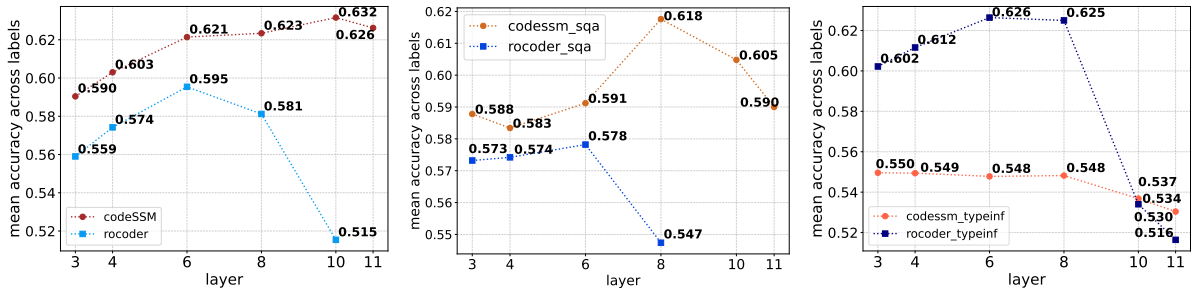


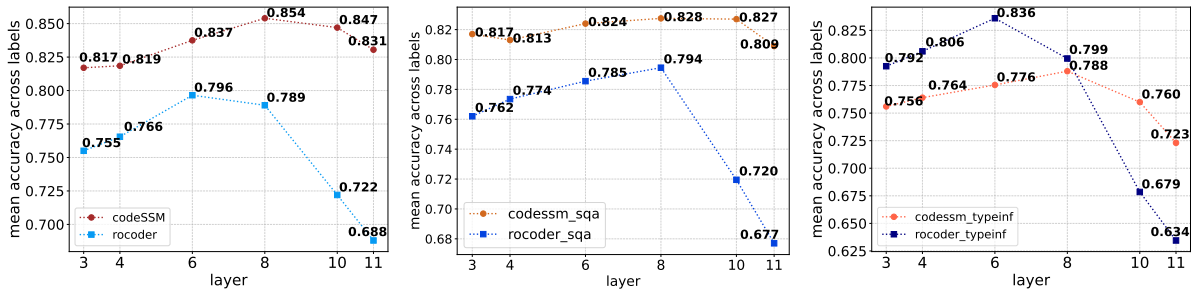
Figure 10: Comparison of hidden representation of CodeSSM on distance (left), siblings (center) and edge (right) prediction tasks with varying context lengths.



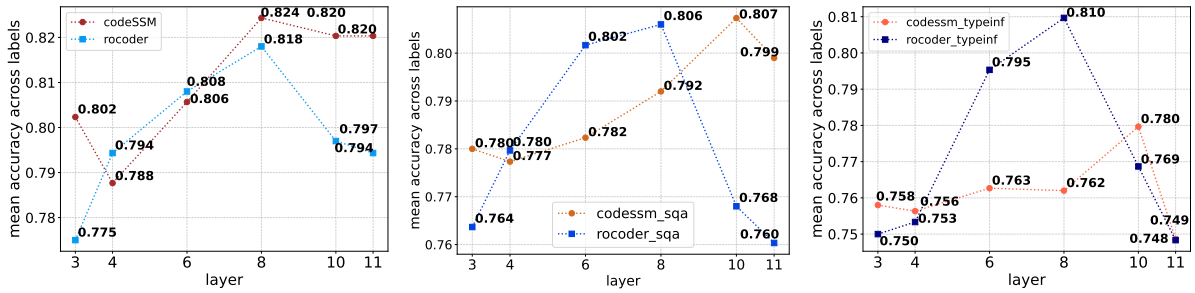
(a) Analysis results of RoCoder in comparsion with its finetuned variants.



(a) DirectProbe distance prediction task

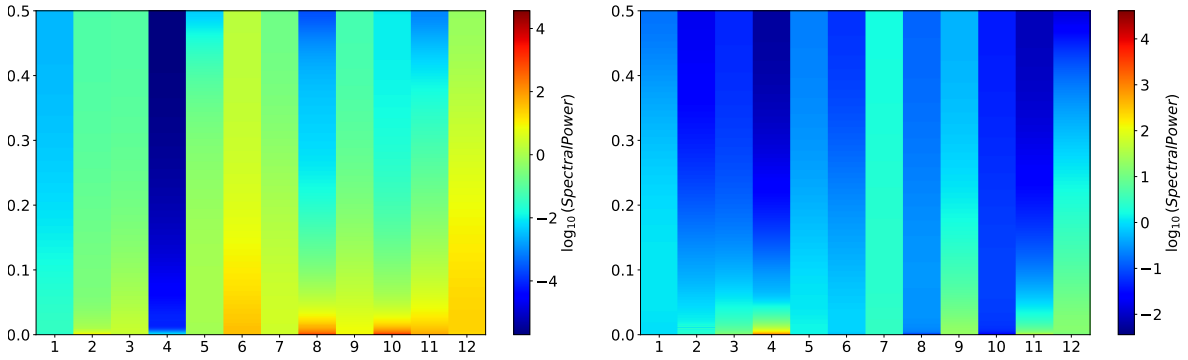


(b) DirectProbe siblings prediction task

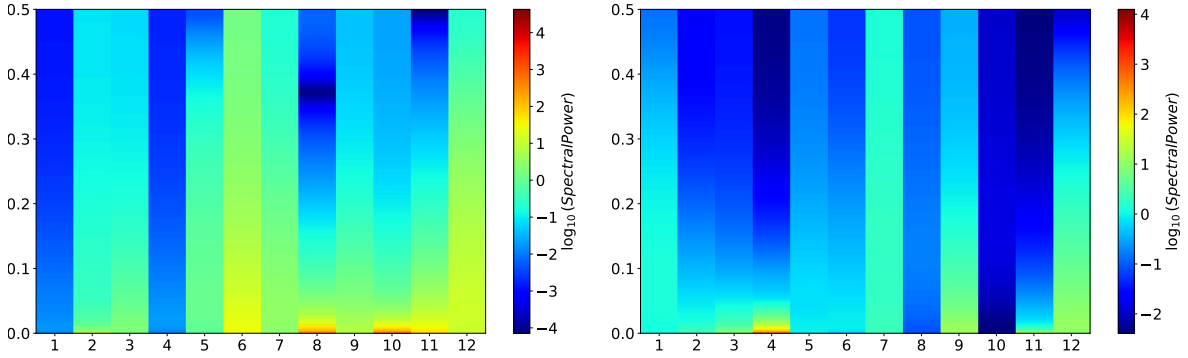


(c) DirectProbe DFG edge prediction task

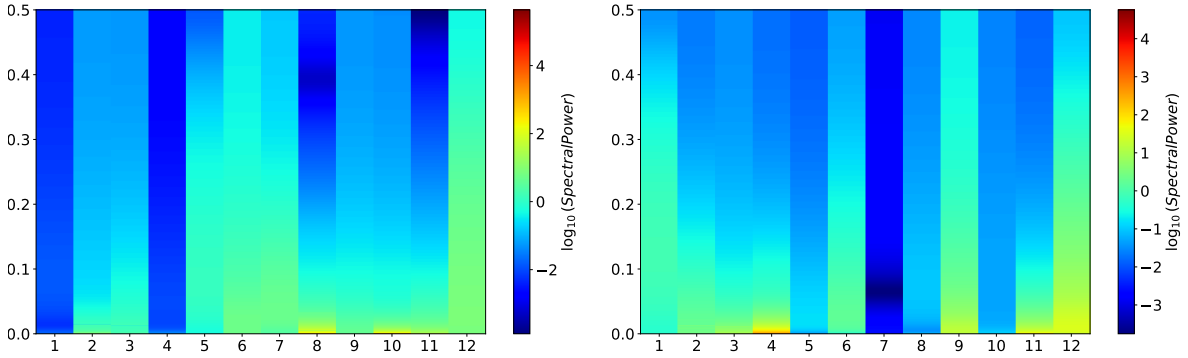
Figure 12: Comparative analysis of hidden representation of CodeSSM and RoCoder (left), after finetuning on SQA (center) and after finetuning on type inference (right) for some layers.



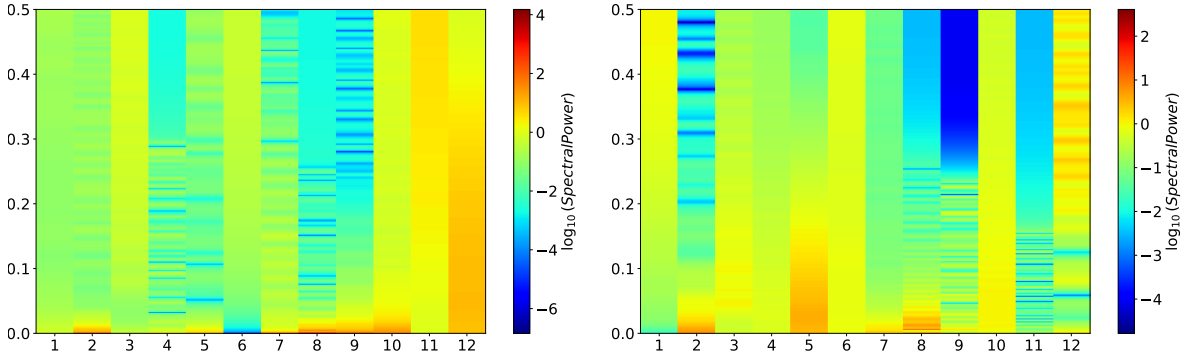
(a) CodeSSM



(b) CodeSSM_sqa

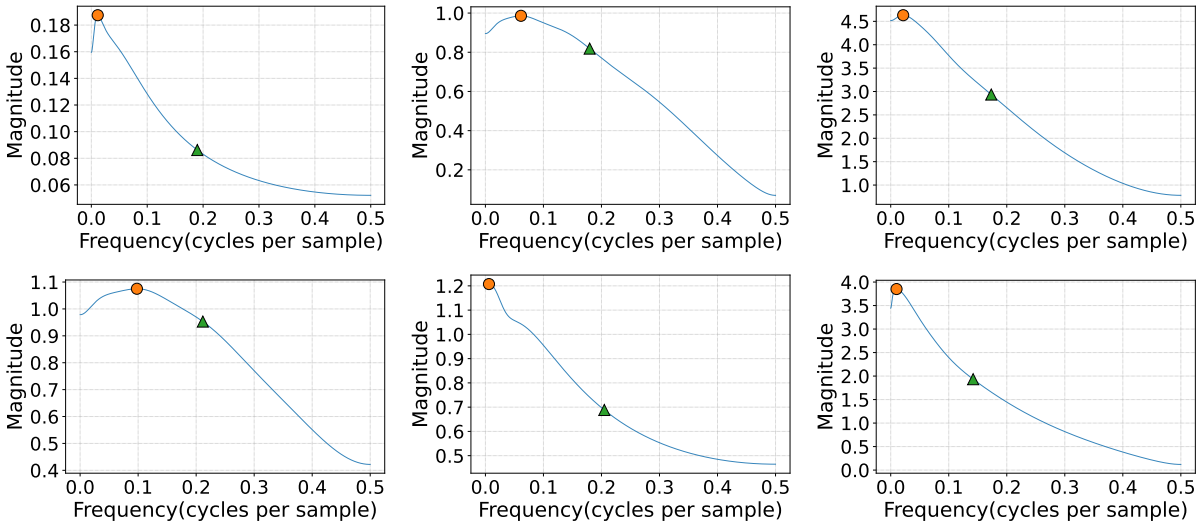


(c) CodeSSM_typeinf

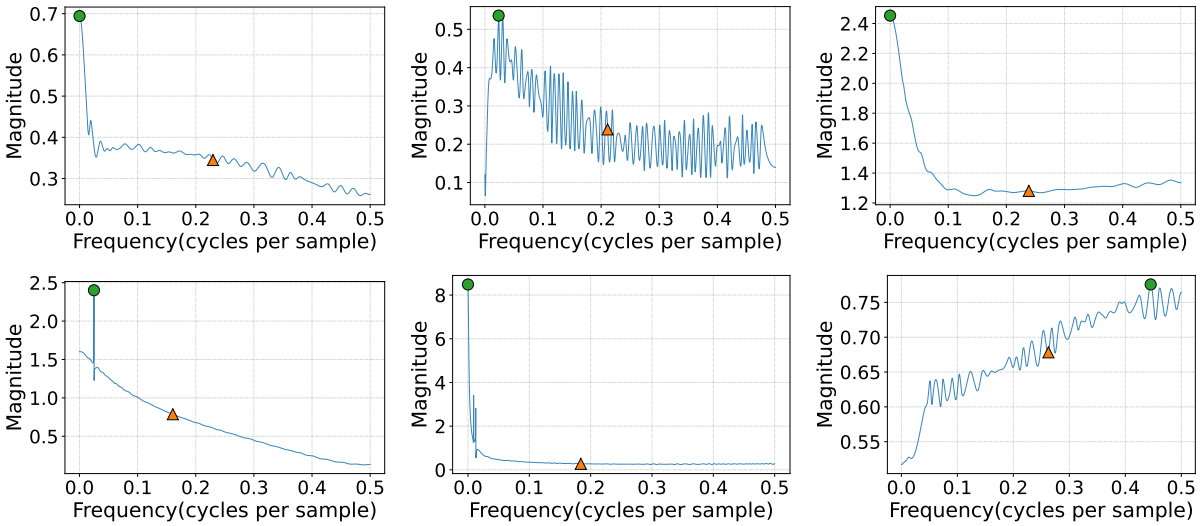


(d) CodeSSM-8k

Figure 13: Heatmap visualization of spectral powers of CodeSSM, CodeSSM_sqa, CodeSSM_typeinf and CodeSSM-8k. For the CodeSS-8k model, we take the mean over all kernels. The visualization shows the richer token relations learnt in the case of CodeSSM-8k.

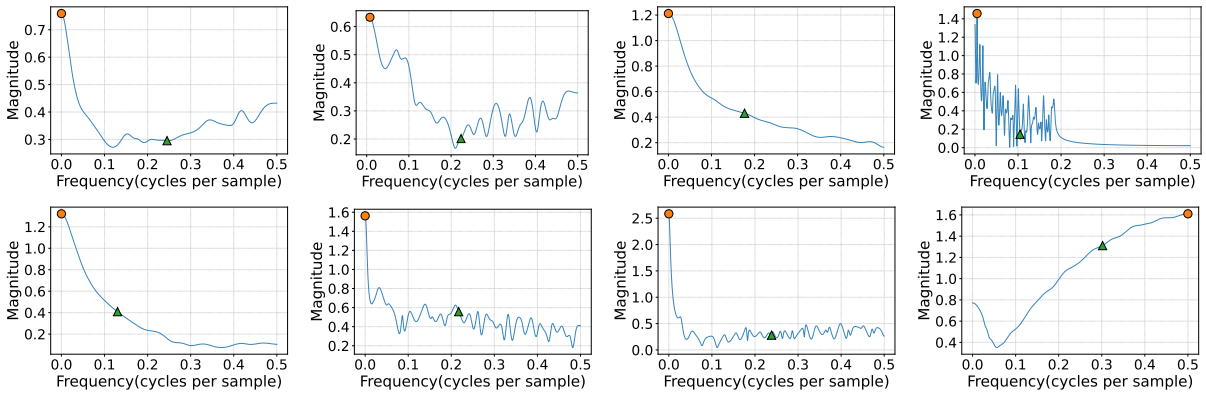


(a) CodeSSM Kernels

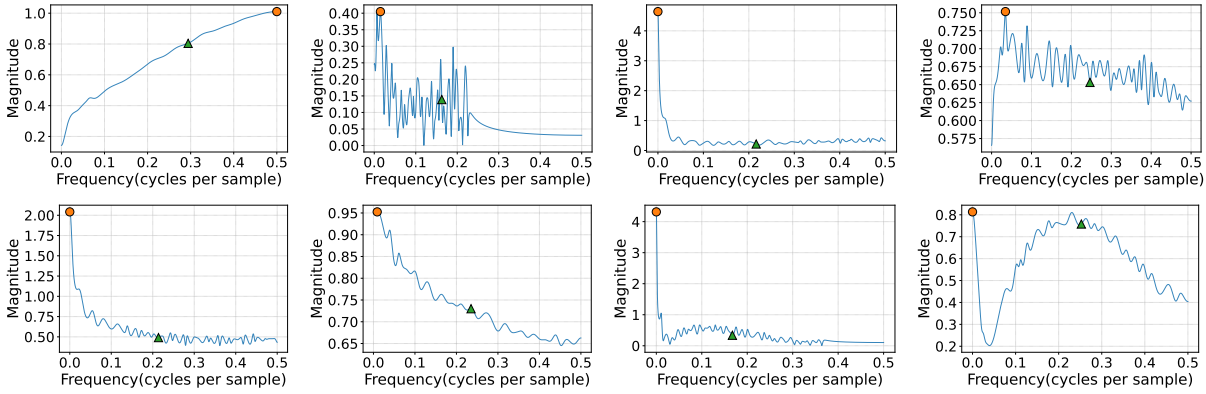


(b) CodeSSM-HF kernels

Figure 14: Additional kernel visualization showing kernels of layer 1 (left), layer 5 (center) and the last layer (right). In each sub-figure top row shows forward kernels and bottom row shows backward kernels. The visualization shows the richer kernels learned by CodeSSM-HF.



(a) Forward kernels of layer 1 of CodeSSM-8k.



(b) Backward kernels of layer 1 of CodeSSM-8k.

Figure 15: The forward and backward kernels of layer 1 of CodeSSM-8k model.

```
def Vgg19(rgb):
    start_time = time.time()
    print(\"build model started\")

    rgb_scaled = rgb * 255.0
    # Convert RGB to BGR
    red, green, blue = tf.split(rgb_scaled, 3, 3)

    if red.get_shape().as_list()[1:] != [224, 224, 1]:
        raise Exception(\"image size unmatch\")

    if green.get_shape().as_list()[1:] != [224, 224, 1]:
        raise Exception(\"image size unmatch\")
    if blue.get_shape().as_list()[1:] != [224, 224, 1]:
        raise Exception(\"image size unmatch\")

    bgr = tf.concat([
                blue - VGG_MEAN[0],
                green - VGG_MEAN[1],
                red - VGG_MEAN[2],
            ], axis=3)
    if bgr.get_shape().as_list()[1:] != [224, 224, 3]:
        raise Exception(\"image size unmatch\")

    # input layer
    net_in = InputLayer(bgr, name='input')
    # conv1
    net = Conv2dLayer(net_in, act=tf.nn.relu, shape=[3, 3, 3, 64], strides=[1, 1, 1, 1], padding='SAME', name='conv1_1')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 64, 64], strides=[1, 1, 1, 1], padding='SAME', name='conv1_2')
    net = PoolLayer(net, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', pool=tf.nn.max_pool, name='pool1')
    # conv2
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 64, 128], strides=[1, 1, 1, 1], padding='SAME', name='conv2_1')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 128, 128], strides=[1, 1, 1, 1], padding='SAME', name='conv2_2')
    net = PoolLayer(net, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', pool=tf.nn.max_pool, name='pool2')
    # conv3
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 128, 256], strides=[1, 1, 1, 1], padding='SAME', name='conv3_1')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 256, 256], strides=[1, 1, 1, 1], padding='SAME', name='conv3_2')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 256, 256], strides=[1, 1, 1, 1], padding='SAME', name='conv3_3')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 256, 256], strides=[1, 1, 1, 1], padding='SAME', name='conv3_4')
    net = PoolLayer(net, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', pool=tf.nn.max_pool, name='pool3')
    # conv4
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 256, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv4_1')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv4_2')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv4_3')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv4_4')
    net = PoolLayer(net, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', pool=tf.nn.max_pool, name='pool4')
    # conv5
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv5_1')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv5_2')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv5_3')
    net = Conv2dLayer(net, act=tf.nn.relu, shape=[3, 3, 512, 512], strides=[1, 1, 1, 1], padding='SAME', name='conv5_4')
    net = PoolLayer(net, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', pool=tf.nn.max_pool, name='pool5')
    # fc 6~8
    net = FlattenLayer(net, name='flatten')
    net = DenseLayer(net, n_units=4096, act=tf.nn.relu, name='fc6')
    net = DenseLayer(net, n_units=4096, act=tf.nn.relu, name='fc7')
    net = DenseLayer(net, n_units=1000, act=None, name='fc8')
    print(\"build model finished: %fs\" % (time.time() - start_time))
    return net
```

Figure 16: Example of code where DFG represents long range dependencies. The highlighted variables have a DFG edge and have 1099 tokens between them.

```
def create_risk_tear_sheet(positions, style_factor_panel=None, sectors=None, caps=None, shares_held=None, volumes=None,
                           percentile=None, returns=None, transactions=None, estimate_intraday='infer', return_fig=False):
    positions = utils.check_intraday(estimate_intraday, returns, positions, transactions)
    idx = positions.index & style_factor_panel.iloc[0].index & sectors.index & caps.index & shares_held.index & volumes.index
    positions = positions.loc[idx]
    vertical_sections = 0
    if style_factor_panel is not None:
        vertical_sections += len(style_factor_panel.items)
        new_style_dict = {}
        for item in style_factor_panel.items:
            new_style_dict.update({item: style_factor_panel.loc[item].loc[idx]})
            style_factor_panel = pd.Panel()
            style_factor_panel = style_factor_panel.from_dict(new_style_dict)
    if sectors is not None:
        vertical_sections += 4
        sectors = sectors.loc[idx]
    if caps is not None:
        vertical_sections += 4
        caps = caps.loc[idx]
    if (shares_held is not None) & (volumes is not None) & (percentile is not None):
        vertical_sections += 3
        shares_held = shares_held.loc[idx]
        volumes = volumes.loc[idx]
    if percentile is None:
        percentile = 0.1
    fig = plt.figure(figsize=[14, vertical_sections * 6])
    gs = gridspec.GridSpec(vertical_sections, 3, wspace=0.5, hspace=0.5)
    if style_factor_panel is not None:
        style_axes = []
        style_axes.append(plt.subplot(gs[0, :]))
        for i in range(1, len(style_factor_panel.items)):
            style_axes.append(plt.subplot(gs[i, :], sharex=style_axes[0]))
            j = 0
            for name, df in style_factor_panel.iteritems():
                sfe = risk.compute_style_factor_exposures(positions, df)
                risk.plot_style_factor_exposures(sfe, name, style_axes[j])
                j += 1
                if sectors is not None:
                    i += 1
                    ax_sector_longshort = plt.subplot(gs[i:i+2, :], sharex=style_axes[0])
                    i += 2
                    ax_sector_gross = plt.subplot(gs[i, :], sharex=style_axes[0])
                    i += 1
                    ax_sector_net = plt.subplot(gs[i, :], sharex=style_axes[0])
                    long_exposures, short_exposures, gross_exposures, net_exposures = risk.compute_sector_exposures(positions, sectors)
                    risk.plot_sector_exposures_longshort(long_exposures, short_exposures, ax=ax_sector_longshort)
                    risk.plot_sector_exposures_gross(gross_exposures, ax=ax_sector_gross)
                    risk.plot_sector_exposures_net(net_exposures, ax=ax_sector_net)
                if caps is not None:
                    i += 1
                    ax_cap_longshort = plt.subplot(gs[i:i+2, :], sharex=style_axes[0])
                    i += 2
                    ax_cap_gross = plt.subplot(gs[i, :], sharex=style_axes[0])
                    i += 1
                    ax_cap_net = plt.subplot(gs[i, :], sharex=style_axes[0])
                    long_exposures, short_exposures, gross_exposures, net_exposures = risk.compute_cap_exposures(positions, caps)
                    risk.plot_cap_exposures_longshort(long_exposures, short_exposures, ax_cap_longshort)
                    risk.plot_cap_exposures_gross(gross_exposures, ax_cap_gross)
                    risk.plot_cap_exposures_net(net_exposures, ax_cap_net)
                if volumes is not None:
                    i += 1
                    ax_vol_longshort = plt.subplot(gs[i:i+2, :], sharex=style_axes[0])
                    i += 2
                    ax_vol_gross = plt.subplot(gs[i, :], sharex=style_axes[0])
                    longed_threshold, shorted_threshold, grossed_threshold = risk.compute_volume_exposures(positions, volumes,
                                                                                                           percentile)
                    risk.plot_volume_exposures_longshort(longed_threshold, shorted_threshold, percentile, ax_vol_longshort)
                    risk.plot_volume_exposures_gross(grossed_threshold, percentile, ax_vol_gross)
    for ax in fig.axes:
        plt.setp(ax.get_xticklabels(), visible=True)
    if return_fig:
        return fig
```

Figure 17: Example of code where DFG represents long range dependencies. The highlighted variables have a DFG edge and have 1486 tokens between them.