Mechanistic Understanding of Language Models in Syntactic Code Completion

Anonymous submission

Abstract

Recently, language models (LMs) have shown impressive proficiency in code generation tasks, especially when fine-tuned on code-specific datasets, commonly known as Code LMs. However, our understanding of the internal decision-making processes of Code LMs, such as how they use their (syntactic or semantic) knowledge, remains limited, which could lead to unintended harm as they are increasingly used in real life. This motivates us to conduct one of the first Mechanistic Interpretability works to understand how Code LMs perform a syntactic completion task, specifically the closing parenthesis task, on the CodeLlama-7b model (Roziere et al. 2023). Our findings reveal that the model requires middle-later layers until it can confidently predict the correct label for the closing parenthesis task. Additionally, we identify that while both multi-head attention (MHA) and feed-forward (FF) sub-layers play essential roles, MHA is particularly crucial. Furthermore, we also discover attention heads that keep track of the number of already closed parentheses precisely but may or may not promote a correct number of closing parentheses that are still missing, leading to a positive or negative impact on the model's performance.

1 Introduction

Language models (LMs) have recently showcased impressive capabilities in code-related tasks, generating executable code from task specifications provided in natural language prompts (Jiang et al. 2024; Zan et al. 2023). To further enhance their capability, they are often instruction-tuned on code-specific datasets, resulting in specialized models known as Code LMs, such as CodeLlama (Roziere et al. 2023), StarCoder (Li et al. 2023), Code Gemma (Team et al. 2024), and DeepSeek-Coder (Guo et al. 2024). These advancements have led to widespread adoption by programmers and researchers, integrating LMs into their daily workflows to assist with coding tasks (Dakhel et al. 2023).

However, despite substantial progress in code generation, Code LMs can generate incorrect code, particularly when handling complex tasks (Dou et al. 2024; Tambon et al. 2024). These faulty codes can pose significant risks, especially when used by novice programmers working on critical applications (Dakhel et al. 2023). Furthermore, even when the generated code snippets are functionally correct, recent research found that these code snippets could contain security vulnerabilities (Siddiq and Santos 2022; Yang et al. 2024). Understanding the capabilities and limitations of Code LMs, such as where and how they invoke relevant knowledge internally, is therefore essential to mitigate potential harm in high-stakes scenarios. Yet, there is limited knowledge about the internals of these LMs while generating code for a given task. Developing deeper insights into their knowledge-relevant mechanisms is crucial for improving their reliability, performance, and safe deployment.

Mechanistic interpretability (MI) has recently emerged as a promising approach to understanding the internal mechanisms of LMs (Olah et al. 2020; Elhage et al. 2021; Rai and Yao 2024; Bereska and Gavves 2024). MI studies have investigated a range of LM behaviors, including in-context learning (Elhage et al. 2021; Bansal et al. 2022; Ren et al. 2024), reasoning (Stolfo, Belinkov, and Sachan 2023; Rai and Yao 2024; Dutta et al. 2024), and fact recall (Geva et al. 2023; Chughtai, Cooney, and Nanda 2024), providing valuable insights into how various LM components, such as multi-head attention (MHA) and feedforward (FF) sublayers, contribute to these capabilities. While substantial work has been done to investigate various behaviors of LMs, there has been limited focus on understanding how Code LMs internally use their knowledge in code generation tasks.

To address this gap, we present one of the first MI studies on Code LMs, where we investigate the internal workings of the CodeLlama-7b (Roziere et al. 2023) for the syntax completion task, the success of which requires an LM to not only locate its declarative knowledge of the programming language but also use the knowledge with its other capabilities (e.g., counting) smartly. Specifically, we study how CodeLlama-7b performs the closing parentheses task (e.g., `print(str(1 → ))`), where each opening parenthesis must be paired with a closing parenthesis. To this end, we first contribute a synthetic dataset for systematically studying a Code LM's syntax completion performance. Our dataset includes a total 168 prompts covering three sub-tasks with the number of closing parentheses in the target tokens being 2, 3, and 4, respectively. These prompts include recursive calls of class constructors including `str`, `list`, and `set`, with the number of open parentheses ranging from 2 to 12. With our dataset, we perform a series of analyses investigating the internal mechanisms of CodeLlama-7b, including projecting the intermediate (sub-)layers' activations via the logit lens (nostalgebraist 2020) to understand the typical timing

when the model realizes the correct token, measuring the logit difference between the correct and the counterfactual tokens to understand the effective contribution of each (sub-)layer to correct token predictions, and performing attention visualization analysis to discover the attention patterns inside the Code LM. Our experimental results reveal that:

- The Code LM can realize the correct target token only from the middle-to-late layers. For example, in case of needing two closing parentheses in the target token, the model on average only ranks the correct token within top 10 based on its projected logit value from layer 18, and being the top 1 from layer 25. We also discover that when the required closing parentheses increase, it becomes even more difficult for the model to identify the correct token, as illustrated by its even later layers for ranking the correct token within the top 10 or top 1.
- When looking into a comparative effect of predicting the correct token vs. predicting the counterfactual token, both MHA and FF sub-layers contribute to the task. However, MHA sub-layers make a more critical contribution to the prediction of the correct tokens. In addition, the contributions of the (sub-)layers generally follow a similar pattern when we vary the number of open parentheses and the class constructors in the input prompt.
- Finally, we identified and interpreted key attention heads responsible for performing the task. For instance, we discovered two attention heads, $L30H0$ and $L27H24$, both keeping track of the number of already closed parentheses precisely. However, while $L30H0$ consistently promotes the correct number of closing parentheses that are still missing across sub-tasks, $L27H24$ always promotes the token including exactly two closing parentheses, which we summarize as *incorrect knowledge association*. As a consequence, it was found to be crucial when the task requires two closing parentheses, but has a negative effect otherwise.

2 Methodology and Dataset

In this section, we introduce our methodology towards forming a mechanistic understanding of how a Code LM performs a syntax completion task that requires using its knowledge about a programming language in combination with others (e.g., counting). Our experiment will focus on Python code generation using CodeLlama-7b (Roziere et al. 2023), which is a state-of-the-art (SOTA) medium-size Code LM with a 32-layer decoder-only transformer architecture. The specific model checkpoint we use is "CodeLlama-7b-hf" (i.e., the base model). In what follows, we will first give an overview of our experiment design, then present our process of dataset generation to facilitate the experiment, and finally describe the methods we will use to analyze a Code LM.

2.1 Motivation and Overview

Syntax completion is a crucial and fundamental part of LM code generation, as syntactic correctness is essential for code to be executable. Dou et al. (2024) recently found that even SOTA Code LMs still suffer from syntactic issues to various extents in their generation. This has motivated us to carefully understand how a Code LM performs syntax completion. In our work, we select the closing parentheses task (e.g., `print(str(1 →`$\,$`))`) as our task, given it is one of the most common syntactical structures seen across programming languages; as a result, it is safe to assume that SOTA Code LMs have learned the necessary syntactic knowledge from their training. The input provided to a Code LM in this task is a partially complete line of code (e.g., `print(str(1`), which includes a varying number of function or class constructor calls but is missing some *final* number of closing parentheses that needs to be predicted as a whole in its next token.[1] The Code LM is then tasked with predicting the next token that consists of the necessary number of closing parentheses for the line of code to be syntactically correct (e.g., `))` in the running example).

In our experiments, we focus on analyzing the Code LM in Python programming language. This task was further broken down into three sub-tasks based on the number of closing parentheses required to correctly complete the line of code, which were two, three, and four closing parentheses. A partially completed line of code example for each of the sub-tasks can be seen in Table 1.

2.2 Data Generation

To evaluate the Code LM on the closing parentheses task, we created an initial synthetic dataset consisting of 168 input prompts, with each prompt consisting of a simple natural language instruction, in the form of a code comment that describes the desired semantic meaning of the following line of code, and a partially completed line of Python code. In our preliminary exploration, we found the code comment to be necessary to avoid semantic ambiguity, as otherwise there could be infinite plausible continuations of the same line of code (e.g., continuing "`print(str(1`" with more digits), which will make the analysis difficult. We began the dataset preparation process by searching for Python functions that were both commonly used in practice and could accept arguments of varying data types. To this end, we decided to initially focus on generating prompts that utilized the built-in `print` function while varying the argument supplied to the function. The argument was varied through the selection of a Python built-in class constructor, from a set containing `str`, `list`, and `set`, the integer value passed to the constructor, and the number of nested constructor calls (with the number of open parentheses ranging from 2 to 12 in our data synthesis), which was used to vary the number of required closing parentheses. Following, the various generated arguments were combined with a `print` function call to produce completed lines of code. The completed lines of

[1] Modern Code LMs may or may not predict parentheses one by one. For example, Codellama tokenizes `print(str(1))` into `print`, `(`, `str`, `(`, `1`, and `))`, a total of 5 tokens, with the two closing parentheses being predicted as one single token. In practice, we have observed that when Codellama completes the syntax correctly, it all follows this tokenization practice, namely, it directly generates "`))`" as a single token instead of generating "`)`" for two times. Therefore, in our task design, we consistently use the last token based on the Code LM's tokenization as the target token when analyzing the model's syntax completion performance.

Sub-Task	#of Examples	Prompt → Target Token	Counterfactual Token	Accuracy
Two Closing Paren	56	#print a list containing 2\n `print(list(list(tuple([2])) →))`	`)`	100.0%
Three Closing Paren	84	#print a string 12\n`print(str(str(12 →)))`	`))`	76.2%
Four Closing Paren	28	#print a set containing 123\n`print(set(set(set(set(tuple([123])) →))))`	`)))`	100.0%

Table 1: Examples of prompts provided to the Code LM for each sub-task. In our work, we synthesize a dataset of 168 prompts covering three sub-tasks with the number of closing parentheses in the target tokens being 2, 3, and 4, respectively. The sub-task design is based on the specific Code LM (Codellama)'s tokenization effect (see Section 2.2).

code were then combined with their respective natural language instructions and tokenized using the Code LM (i.e., Codellama)'s tokenizer to produce the final prompts along with their associated correct next (and final) token.

On this dataset, Codellama-7b was able to achieve an overall accuracy of 88%, with the breakdown for each sub-task shown in Table 1. Intriguingly, we observed that Codellama-7b achieves a lower accuracy on the Three Closing Parentheses sub-task than on the Four Closing Parentheses sub-task. Upon examination, we found that all failing cases in the Three Closing Parentheses sub-task occurred when the number of open parentheses in the prompt ranged from nine to eleven, whereas for the prompts in the Four Closing Parentheses sub-task, the largest number of open parentheses is only eight.[2] An intuitive conjecture is that, when the number of open parentheses gets larger, it becomes more challenging for the Code LM to correctly count both the number of open parentheses and the existing number of closing parentheses in the prompt, as well as calculate the difference as the required number of closing parentheses for next token prediction.

2.3 Methodology

We will understand how a Code LM completes a syntax completion task by mainly looking at the model's behaviors of predicting the correct next token, which requires proper use of its knowledge about the programming language. Specifically, we employ *logit lens* or *direct logit attribution* (nostalgebraist 2020) to analyze the contribution of each layer and its sublayers (MHA and FF) in predicting the correct next token. *Logit lens* allows us to view what the LM would have predicted in a given (sub-)layer by projecting the intermediate activations (denoted as $v \in R^d$, where d is the LM dimension) onto the logit distribution through multiplication with the unembedding parameter matrix (denoted as $W_U \in R^{d \times |\mathcal{V}|}$, where $\mathcal{V}$ is the vocabulary set and $|\mathcal{V}|$ denotes its size), i.e., $vW_U \in \mathcal{R}^{|\mathcal{V}|}$. As a result, we can examine the top-k candidate tokens for the next token prediction at each intermediate (sub-)layer by viewing the logit distribution. We refer readers to nostalgebraist (2020) or the recent survey paper of Rai et al. (2024) for a systematic and detailed explanation of the logit lens method.

In addition to analyzing the absolute logit value of the target token, our experiment also involves calculating the *logit difference* to evaluate the contribution of (sub-)layer for the correct token (e.g. "))) ") relative to a *counterfactual* token representing *incorrect knowledge* (e.g. ")) "), effectively filtering out (sub-)layers that indiscriminately increase the logit values of several tokens. We list the counterfactual tokens in Table 1. Such a comparative analysis has been widely adopted by prior work (Vig et al. 2020; Meng et al. 2022; Wang et al. 2022) for effectively discovering task-specific LM behaviors. When the logit difference becomes more positive (or negative, respectively), it implies that the LM is more (or less, respectively) capable of distinguishing between the correct and the misleading tokens.

Finally, when we are able to locate attention layers and heads making the most contribution to the prediction of the correct token, we will then apply *attention visualization* to scrutinize the model's attention pattern. In our experiment, we have found it a very helpful approach for interpreting the LM's behaviors in a human-understandable way.

Our experiments were carried out using the TransformerLens library (Nanda and Bloom 2022) to implement the logit lens and logit difference, and CircuitsVis (Cooney and Nanda 2023) for attention visualization.

3 Experimental Results

3.1 Overview of Experiments

Our experiments aim to answer three Research Questions (RQs). RQ1 and RQ2 examine the layer-wise phenomena in the process of the Code LLM generating the correct next token. Specifically, RQ1 leverages *logit lens* to understand what the model would have predicted in a given layer, from which we gauge from which layer the model typically can start picking the correct token. RQ2 then looks into how (e.g., effect of promotion (Geva et al. 2022)) each (sub-)layer contributes to the prediction of the correct token, particularly by contrasting the logit values between the correct token and a *counterfactual* token (Table 1). In this process, we also locate the critical attention layers that strongly promote the generation of the correct token. Following that, RQ3 then specifically investigates the patterns of these critical attention layers, such as how different attention heads in these layers play a role and how each attention head functions, aiming to form a clearer understanding of how the Code LM becomes aware of the required number of closing parentheses.

3.2 RQ1: At what layer does the Code LM start picking the correct token?

The overarching goal of this RQ is to better understand at what points during an inference phase the Code LM has

[2]We note that the range of the number of open parentheses in each sub-task is largely an effect of Codellama's tokenization and cannot be enforced during the dataset generation.

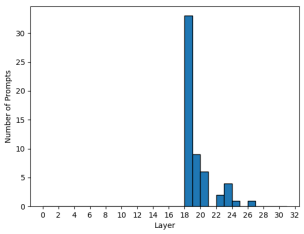
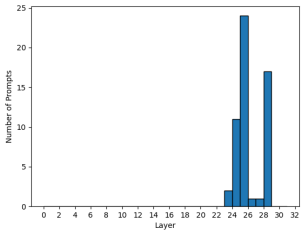
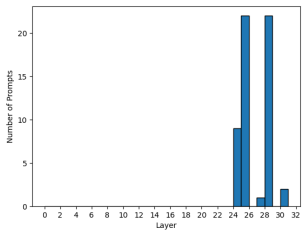
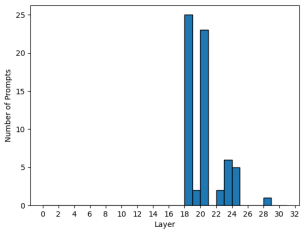
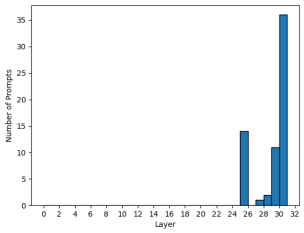
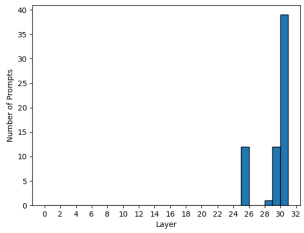
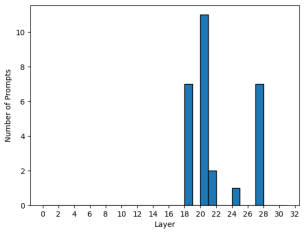
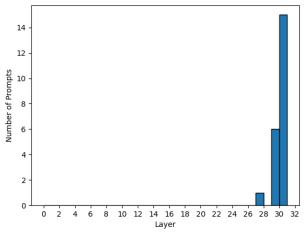
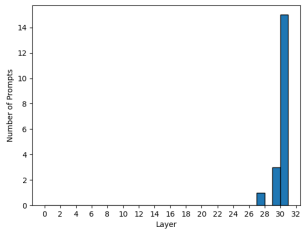
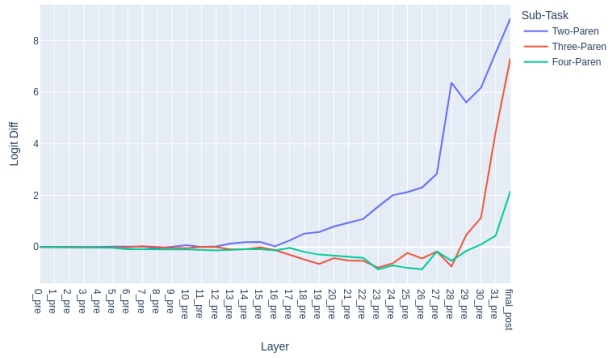
Sub-Task	L_{Top10}	L_{Top1}	$L_{\text{ConsistentTop1}}$
Two Closing Paren			
Three Closing Paren			
Four Closing Paren			

Table 2: Aggregated at the sub-task level, we report the layer distribution when the correct token's logit value is ranked within top 10 (L_{Top10}), top 1 (L_{Top1}), and consistently top 1 afterward ($L_{\text{ConsistentTop1}}$), respectively. We observed that, compared to the Three and the Four Closing Parenthesis sub-tasks, on the Two Closing Parenthesis sub-task the model can identify the correct token at an earlier layer, implying that the latter sub-task is considered easier than the former two.

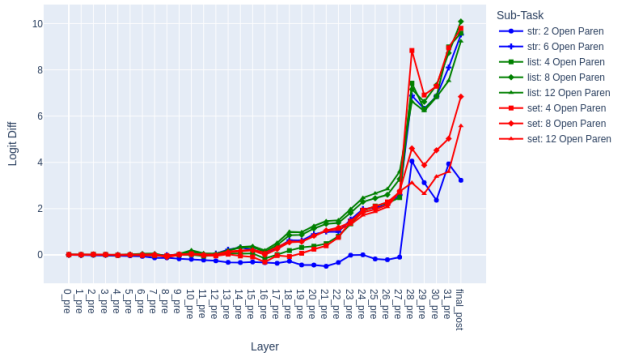
an understanding of what the next token prediction should be, which we define as the correct token's logit value being within the top ten logit values at a layer. This experiment aims to answer the overall question by obtaining answers to three related sub-questions: (1) *At what layer does the correct token's logit value first break into the top 10 logits?* (2) *When does the Code LM first consider that the correct token should be predicted as the next token (i.e., the correct token is associated with the highest logit value)?* and (3) *When does the correct token's logit value consistently rank as the highest logit value for all subsequent layers?* The respective answers to these questions for each of the sub-tasks can be found in Table 2, where for each sub-task we report the median layer (zero-indexed) across all prompts of that sub-task. The logit values in each layer are calculated by applying the logit lens to the residual-stream activation of that layer. We find that the Two Closing Parentheses sub-task has a lower median layer across the considered metrics for all sub-tasks. This was especially apparent for the median first layer where the correct token has the highest overall logit (L_{Top1}) and the median first layer where the correct token is consistently ranked as the top token for all subsequent layers ($L_{\text{ConsistentTop1}}$), where the Two Closing Parentheses sub-task reaches these milestones in layer 25 and the Three Closing Parentheses and Four Closing Parentheses sub-tasks reach these milestones in the final layers of the Code LM. We conjecture that the Code LM views the Two Closing Parentheses sub-task as being easier than the Three and Four Closing Parentheses sub-tasks, which the Code LM appears to view as having similar difficulty.

3.3 RQ2: How does each (sub-)layer contribute to the correct token prediction?

Layer-level Analysis To understand each layer's contribution to correct token prediction, we first measure the *logit difference* between the correct token and the counterfactual token on the residual stream across all layers. Specifically, Figure 1a showcases the logit difference across layers of the residual stream for each sub-task, averaged over prompts in the same sub-task. Figure 1b displays the same metric but focuses on the results for the Two Closing Parentheses sub-task aggregated at a *prompt type* level—here, we define each prompt type by the class constructor (i.e., `str`, `list`, or `set`) and the number of open parentheses. The results in Fig-



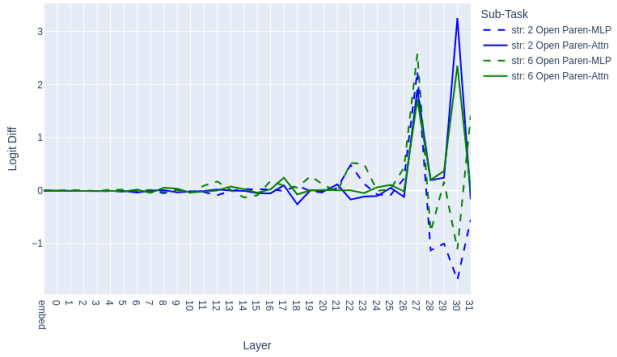
(a) Logit difference for each sub-task (averaged over prompts of the same sub-task).



(b) Logit difference for the Two Closing Paren sub-task, grouped by different prompt types (averaged over prompts of the same type).

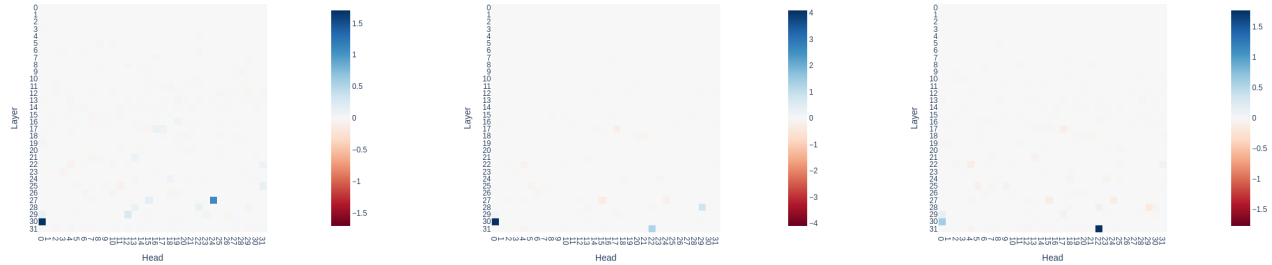
Figure 1: Logit difference of the Code LM between the correct and the counterfactual tokens across layers of the residual stream. "L_pre" and "L_post" indicate residual-stream activations before and after layer L, respectively.

(a) Sub-layer logit difference for each sub-task (averaged over prompts of the same sub-task).



(b) Sub-layer logit difference for the Two Closing Paren sub-task when the class constructor is `str` (averaged over prompts of the same type).

Figure 2: Sub-layer logit difference of the Code LM between the correct and counterfactual tokens contribution to the residual stream. Figures of sub-layer logit difference for other class constructors are shown in the Appendix A.



(a) Two Closing Paren (b) Three Closing Paren (c) Four Closing Paren

Figure 3: Logit differences between the correct and counterfactual tokens of various attention layers and heads for each sub-task. We observed that the contribution to the logit difference was dominantly made by a few heads (e.g., $L30H0$ and $L27H24$ for the Two Closing Parenthesis task).

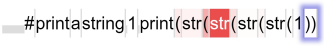
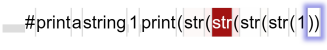
Sub-Task	$L30H0$	$L27H24$
Two Closing Paren		
Three Closing Paren		
Four Closing Paren		

Table 3: Attention patterns of the attention heads $L30H0$ and $L27H24$ for example prompts from each sub-task. When predicting the next token, both $L30H0$ and $L27H24$ predominantly attend to the innermost unclosed function call across all sub-tasks, suggesting that these attention heads are capable of tracking the number of unclosed parenthesis or function calls.

ure 1a show that the Code LM gains an understanding of the Two Closing Parentheses sub-task much earlier than the Three and Four Closing Parentheses sub-tasks, as evidenced by the residual stream having a positive logit difference in earlier layers. The result also corroborates our conjecture from Section 3.2 that the Code LM has more difficulty performing the Three and the Four Closing Parentheses sub-tasks. The layers with the largest contributions (defined by "peaks" of positive logit difference) to the residual stream for the Two, Three, and Four Closing Parentheses sub-tasks were layers 27 and 30, layers 30 and 31, and layers 31 and 26, respectively.

The prompt type level results for the Two Closing Parentheses sub-task found in Figure 1b confirm the positive logit difference contributions of layers 27 and 30 for all prompt types. For prompt types that utilize either the `list` or `set` class constructors, we find that the Code LM gains an understanding of the Two Closing Parentheses sub-task around layer 18. Similar behavior can be seen for the prompt types that utilize the `str` class constructor as the number of opening parentheses becomes large, in this case having 6 open parentheses. It appears that as the prompt type becomes sufficiently difficult, whether that be through the inclusion of more open parentheses or utilization of the `list` or `set` constructor calls, the mid-late layers play a larger role in the promotion of the correct token.

Sub-Layer Analysis The logit difference contribution to the residual stream at a sub-layer (i.e., FF or MHA) level can be seen in Figure 2a and Figure 2b. Similar to above, Figure 2a displays each of the sub-layer contributions to the residual stream aggregated at a sub-task level, while Figure 2b showcases the results for the Two Parentheses Subtask when the class constructor is `str`. At the sub-layer level, the logits of each sub-layer were calculated by projecting the sub-layer's immediate activation output to the vocabulary space via the logit lens. The results in Figure 2a reveal the importance of both FF and MHA sub-layers for the closing parentheses task, as all sub-tasks have both FF and MHA sub-layers in their top five positively contributing sub-layers. When comparing the overall contribution of the MHA sub-layers and the FF sub-layers for all sub-tasks, *the MHA sub-layers have a similar or larger positive contribution*—in fact, the most salient negative contributions to the logit difference for the Two and Three Closing Paren sub-tasks both come from MLP. Their positive contribution is especially apparent in the Three Closing Parentheses and Four Closing Parentheses sub-tasks. The MHA sub-layers that exhibit the strongest promotion of the correct tokens against the counterfactual ones for the Two Closing Parentheses, Three Closing Parentheses, and Four Closing Parentheses sub-tasks are the MHA sub-layers in layers 30 and 27, layers 30 and 31, and layers 31 and 30, respectively.

Figure 2b similarly illustrates the importance of MHA sub-layers to positive logit difference. Interestingly, the patterns when the number of open parentheses is 2 and 6 respectively are pretty similar, with MHA's positive contributions peaking at layers 27 and 30, MLP's positive contributions peaking at layers 27 and 22, and MLP's negative contributions dipping significantly at layers 28 and 30.

3.4 RQ 3: How do attention heads contribute to the promotion/suppression of correct tokens?

Given the insight from the experimental results of RQ2 that MHA sub-layers appear to contribute in a more significant fashion to the syntax completion task, we ran an additional experiment to see how individual attention heads contribute to the promotion or suppression of the logit difference between the correct token and the counterfactual token. We were especially interested in identifying the attention heads that had a large positive or negative contribution to the promotion of the logit difference in the previously identified MHA sub-layers. The heat map showing the logit difference projected from individual attention layers and heads for each sub-task can be seen in Figure 3. For all sub-tasks, we identify that most heads have a strong positive contribution to the logit difference (marked as deep blues), whereas only a few heads have a small negative contribution (marked as light reds). In particular, the positive contribution was dominantly made by only a few heads. For the Two Closing Parentheses sub-task, we find that the largest contributing heads are $L30H0$ (i.e., layer 30, head 0) and $L27H24$ (i.e., layer 27, head 24), which both have positive contributions. Interestingly, while the $L30H0$ attention head exhibits similar positive contribution behavior in the Three and Four Closing Parentheses sub-tasks as in the Two Closing Parentheses sub-task, we find that the $L27H24$ head negatively contributes the correct output for Three Closing Parentheses and Four Closing Parentheses sub-tasks.

We identified that, despite their functional differences, $L30H0$ and $L27H24$ have similar attention patterns for all sub-tasks. Specifically, they both were found to effectively track the number of already closed parentheses by attending to the function name up to the point where the parentheses are already closed, as shown in Table 3. However, while $L30H0$ dynamically promotes the correct number of closing parentheses based on the count of those already closed, $L27H24$ always promotes two closing parentheses regardless of the number of remaining open parentheses. In other words, this head is promoting incorrect knowledge despite being able to correctly understand the context. We summarize this the phenomenon as *"incorrect knowledge association"*. Consequently, this behavior of $L27H24$ results in a negative contribution for tasks requiring three or four closing parentheses.

4 Related Works

Analysis of Code LMs Code LMs (Abdin et al. 2024; Team et al. 2024; Guo et al. 2024; Li et al. 2023; Roziere et al. 2023; Chen et al. 2021), are a class of LMs specifically developed to enhance code generation capabilities of LMs through fine-tuning and additional training techniques (Chan et al. 2023). Although these models have demonstrated remarkable capabilities in code generation tasks (Yu et al. 2024; Zhuo et al. 2024; Lai et al. 2023; Cassano et al. 2023; Hao et al. 2022; Srivastava et al. 2022; Hendrycks et al. 2021), they remain susceptible to various syntactic and semantic (or logical) errors (Yu et al. 2024; Tambon et al. 2024; Dou et al. 2024). Prior studies have focused on empirically examining various types of bugs across a range of coding tasks and programming languages (Dou et al. 2024; Tambon et al. 2024; Dakhel et al. 2023) or proposing benchmark datasets to characterize these models' shortcomings (Wang et al. 2023; Siddiq and Santos 2022; Yang et al. 2024). For instance, Dou et al. (2024) observed that LMs are especially prone to syntactical errors (e.g., incomplete syntax structure, and indentation issues) when generating code for complex or lengthy problems. While these studies provide valuable insights into when Code LMs are likely to make mistakes, our understanding of the underlying internal mechanisms enabling code-generation capabilities remains limited. To bridge this gap, our study investigates how Code LMs perform syntax completion tasks.

Mechanistic Interpretability (MI) MI is a subfield of interpretability that aims to reverse-engineer LM by understanding their internal components and computational processes (Elhage et al. 2021; Olah et al. 2020; Rai et al. 2024; Bereska and Gavves 2024). Recent MI studies have investigated various LM behaviors, including sequence completion task (Elhage et al. 2021), Indirect Object Identification (Wang et al. 2022), Python docstring completion (Heimersheim and Janiak 2023; Conmy et al. 2023) and modular addition tasks (Nanda et al. 2023), by discovering circuits, a subset of LM components responsible for implementing these LM behaviors. These circuits can be explained in terms of human-understandable algorithms after interpreting the circuit components, which has led to the discovery of several interpretable attention heads such as induction heads (Elhage et al. 2021), suppression heads (McDougall et al. 2023), and previous token heads (Elhage et al. 2021). Building upon these advancements, we study the internal mechanisms of code LMs to understand the syntax completion capability of Code LMs. As far as we know, there has no been prior work carefully studying MI techniques in the application of LMs for code generation. As the first step, in this work, we have focused on discovering how the CodeLlama model identifies the correct next token and contrasts it against the counterfactual token in the syntax completion task. We include a discussion of our future extension along this line of research in Section 5.

5 Discussion and Future Works

In this work, we presented preliminary findings toward a mechanistic understanding of how Code LMs use their internal knowledge to complete syntactic completion tasks, identifying multiple attention heads that play a critical role in this task. Building on these results, we suggest the following directions for future research.

Circuit Discovery Seeing the relatively more important role the MHA sub-layers play in prioritizing the correct token over the counterfactual one, our work has focused on analyzing the MHA patterns in a Code LM. However, future work should extend the analysis to cover the MLP sublayers (which were found to implement knowledge look-up in transformers (Geva et al. 2021)), and eventually portray a complete *circuit* of how a Code LM associates various components in its transformer architecture towards successfully using its knowledge in syntax completion.

Universality of the Interpretation We hypothesize that there is significant overlap among similar sub-tasks involved in syntax completion tasks, both within and across programming languages. For example, we investigated CodeLlama's ability to perform the parenthesis completion task, which requires the model to track the number of open parentheses that have been closed. Similar counting mechanisms might also be needed for managing indentation in Python or closing curly braces in JavaScript. Our future work will look into whether a Code LM reuses the same components across tasks and languages for similar roles.

Improving the LM Performance Finally, we aim to utilize our interpretation result to enhance a Code LM's performance in real life. For example, in experiments we have identified an attention head, $L27H24$, that performs *incorrect knowledge association* and erroneously promotes two closing parentheses even when the model needs to generate three or four closing parentheses. Furthermore, recent work of Geva et al. (2022) and Rai and Yao (2024) have showcased the potential of directly controlling a model's generation or task performance via manipulating its neuron activation. In the future, we will similarly explore if suppressing such less precise attention heads can improve the accuracy of the Code LM in closing the parentheses and beyond.

References

Abdin, M.; Aneja, J.; Awadalla, H.; Awadallah, A.; Awan, A. A.; Bach, N.; Bahree, A.; Bakhtiari, A.; Bao, J.; Behl, H.; et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Bansal, H.; Gopalakrishnan, K.; Dingliwal, S.; Bodapati, S.; Kirchhoff, K.; and Roth, D. 2022. Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale. *arXiv preprint arXiv:2212.09095*.

Bereska, L.; and Gavves, E. 2024. Mechanistic Interpretability for AI Safety–A Review. *arXiv preprint arXiv:2404.14082*.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7): 3675–3691.

Chan, A.; Kharkar, A.; Moghaddam, R. Z.; Mohylevskyy, Y.; Helyar, A.; Kamal, E.; Elkamhawy, M.; and Sundaresan, N. 2023. Transformer-based vulnerability detection in code at EditTime: Zero-shot, few-shot, or fine-tuning? *arXiv preprint arXiv:2306.01754*.

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

Chughtai, B.; Cooney, A.; and Nanda, N. 2024. Summing Up the Facts: Additive Mechanisms Behind Factual Recall in LLMs. *arXiv preprint arXiv:2402.07321*.

Conmy, A.; Mavor-Parker, A.; Lynch, A.; Heimersheim, S.; and Garriga-Alonso, A. 2023. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36: 16318–16352.

Cooney, A.; and Nanda, N. 2023. CircuitsVis. https://github.com/TransformerLensOrg/CircuitsVis.

Dakhel, A. M.; Majdinasab, V.; Nikanjam, A.; Khomh, F.; Desmarais, M. C.; and Jiang, Z. M. J. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203: 111734.

Dou, S.; Jia, H.; Wu, S.; Zheng, H.; Zhou, W.; Wu, M.; Chai, M.; Fan, J.; Huang, C.; Tao, Y.; et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv preprint arXiv:2407.06153*.

Dutta, S.; Singh, J.; Chakrabarti, S.; and Chakraborty, T. 2024. How to think step-by-step: A mechanistic understanding of chain-of-thought reasoning. *arXiv preprint arXiv:2402.18312*.

Elhage, N.; Nanda, N.; Olsson, C.; Henighan, T.; Joseph, N.; Mann, B.; Askell, A.; Bai, Y.; Chen, A.; Conerly, T.; DasSarma, N.; Drain, D.; Ganguli, D.; Hatfield-Dodds, Z.; Hernandez, D.; Jones, A.; Kernion, J.; Lovitt, L.; Ndousse, K.; Amodei, D.; Brown, T.; Clark, J.; Kaplan, J.; McCandlish, S.; and Olah, C. 2021. A Mathematical Framework for Transformer Circuits. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2021/framework/index.html.

Geva, M.; Bastings, J.; Filippova, K.; and Globerson, A. 2023. Dissecting recall of factual associations in auto-regressive language models. *arXiv preprint arXiv:2304.14767*.

Geva, M.; Caciularu, A.; Wang, K.; and Goldberg, Y. 2022. Transformer Feed-Forward Layers Build Predictions by Promoting Concepts in the Vocabulary Space. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 30–45.

Geva, M.; Schuster, R.; Berant, J.; and Levy, O. 2021. Transformer Feed-Forward Layers Are Key-Value Memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 5484–5495.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hao, Y.; Li, G.; Liu, Y.; Miao, X.; Zong, H.; Jiang, S.; Liu, Y.; and Wei, H. 2022. Aixbench: A code generation benchmark dataset. *arXiv preprint arXiv:2206.13179*.

Heimersheim, S.; and Janiak, J. 2023. A circuit for Python docstrings in a 4-layer attention-only transformer. *URL: https://www. alignmentforum. org/posts/u6KXXmKFbXfWzoAXn/acircuit-for-python-docstrings-in-a-4-layer-attention-only*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515*.

Lai, Y.; Li, C.; Wang, Y.; Zhang, T.; Zhong, R.; Zettlemoyer, L.; Yih, W.-t.; Fried, D.; Wang, S.; and Yu, T. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, 18319–18345. PMLR.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

McDougall, C.; Conmy, A.; Rushing, C.; McGrath, T.; and Nanda, N. 2023. Copy suppression: Comprehensively understanding an attention head. *arXiv preprint arXiv:2310.04625*.

Meng, K.; Bau, D.; Andonian, A.; and Belinkov, Y. 2022. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems*, 35: 17359–17372.

Nanda, N.; and Bloom, J. 2022. TransformerLens. https://github.com/TransformerLensOrg/TransformerLens.

Nanda, N.; Chan, L.; Lieberum, T.; Smith, J.; and Steinhardt, J. 2023. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*.

nostalgebraist. 2020. Interpreting GPT: the logit lens. *AI Alignment Forum*. https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

Olah, C.; Cammarata, N.; Schubert, L.; Goh, G.; Petrov, M.; and Carter, S. 2020. Zoom In: An Introduction to Circuits. *Distill*. Https://distill.pub/2020/circuits/zoom-in.

Rai, D.; and Yao, Z. 2024. An Investigation of Neuron Activation as a Unified Lens to Explain Chain-of-Thought Eliciting Arithmetic Reasoning of LLMs. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7174–7193. Bangkok, Thailand: Association for Computational Linguistics.

Rai, D.; Zhou, Y.; Feng, S.; Saparov, A.; and Yao, Z. 2024. A practical review of mechanistic interpretability for transformer-based language models. *arXiv preprint arXiv:2407.02646*.

Ren, J.; Guo, Q.; Yan, H.; Liu, D.; Zhang, Q.; Qiu, X.; and Lin, D. 2024. Identifying semantic induction heads to understand in-context learning. *arXiv preprint arXiv:2402.13055*.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Siddiq, M. L.; and Santos, J. C. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 29–33.

Srivastava, A.; Rastogi, A.; Rao, A.; Shoeb, A. A. M.; Abid, A.; Fisch, A.; Brown, A. R.; Santoro, A.; Gupta, A.; Garriga-Alonso, A.; et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Stolfo, A.; Belinkov, Y.; and Sachan, M. 2023. A mechanistic interpretation of arithmetic reasoning in language models using causal mediation analysis. *arXiv preprint arXiv:2305.15054*.

Tambon, F.; Dakhel, A. M.; Nikanjam, A.; Khomh, F.; Desmarais, M. C.; and Antoniol, G. 2024. Bugs in large language models generated code. *arXiv preprint arXiv:2403.08937*.

Team, C.; Zhao, H.; Hui, J.; Howland, J.; Nguyen, N.; Zuo, S.; Hu, A.; Choquette-Choo, C. A.; Shen, J.; Kelley, J.; et al. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Vig, J.; Gehrmann, S.; Belinkov, Y.; Qian, S.; Nevo, D.; Singer, Y.; and Shieber, S. 2020. Investigating gender bias in language models using causal mediation analysis. *Advances in neural information processing systems*, 33: 12388–12401.

Wang, K.; Variengien, A.; Conmy, A.; Shlegeris, B.; and Steinhardt, J. 2022. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small. arXiv:2211.00593.

Wang, S.; Li, Z.; Qian, H.; Yang, C.; Wang, Z.; Shang, M.; Kumar, V.; Tan, S.; Ray, B.; Bhatia, P.; et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 13818–13843.

Yang, Y.; Nie, Y.; Wang, Z.; Tang, Y.; Guo, W.; Li, B.; and Song, D. 2024. SecCodePLT: A Unified Platform for Evaluating the Security of Code GenAI. *arXiv preprint arXiv:2410.11096*.

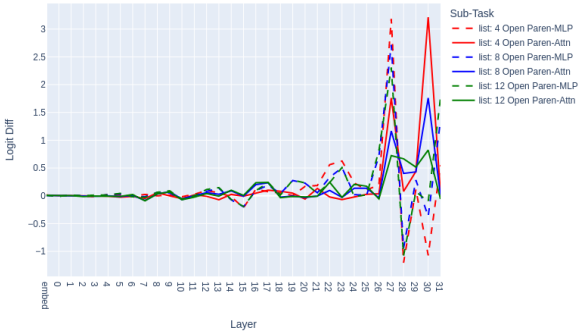
Yu, H.; Shen, B.; Ran, D.; Zhang, J.; Zhang, Q.; Ma, Y.; Liang, G.; Li, Y.; Wang, Q.; and Xie, T. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.

Zan, D.; Chen, B.; Zhang, F.; Lu, D.; Wu, B.; Guan, B.; Yongji, W.; and Lou, J.-G. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7443–7464.

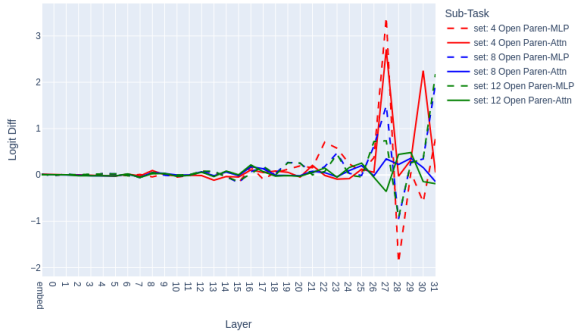
Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Additional Results of Sub-Layer Logit Differences

In Figure 4, we present the sub-layer logit difference curves for the other two class constructors, i.e., `list` and `set`, in the Two Closing Parenthesis sub-task.



(a) Sub-layer logit difference for the Two Closing Paren subtask when the class constructor is **`list`** (averaged over prompts of the same type).



(b) Sub-layer logit difference for the Two Closing Paren subtask when the class constructor is **`set`** (averaged over prompts of the same type).

Figure 4: Sub-layer logit difference of the Code LM between the correct and counterfactual tokens contribution to the residual stream. "embed" indicates the word embedding.