Looking into Black Box Code Language Models

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Language Models (LMs) have shown their application for tasks pertinent to code and several code LMs have been proposed recently. The majority of the studies in this direction only focus on the improvements in performance of the LMs on different benchmarks, whereas LMs are considered black boxes. Besides this, a handful of works attempt to understand the role of attention layers in the code LMs. Nonetheless, feed-forward layers remain under-explored which consist of two-thirds of a typical transformer model's parameters. In this work, we attempt to gain insights into the inner workings of code language models by examining the feed-forward layers. To conduct our investigations, we use two state-of-the-art code LMs, Codegen-Mono and Ploycoder, and three widely used programming languages, Java, Go, and Python. We focus on examining the organization of stored concepts, the editability of these concepts, and the roles of different layers and input context size variations for output generation. Our empirical findings demonstrate that lower layers capture syntactic patterns while higher layers encode abstract concepts and semantics. We show concepts of interest can be edited within feed-forward layers without compromising code LM performance. We anticipate these findings will facilitate better understanding, debugging, and testing of code LMs.

1 Introduction

Code language models (code LMs), leveraging the transformers architecture Vaswani et al. [2017], have emerged as powerful productivity tools in software development. Inspired by the success of natural language processing (NLP) transformers (e.g., BERT Devlin et al. [2019], GPT Radford et al. [2018b]), these models have been trained on vast repositories of code from open-source projects. Through this training, code LMs have acquired the ability to capture complex patterns, syntax, and semantics of programming languages. Consequently, code LMs have demonstrated significant success across various coding tasks, including code generation, completion, editing, and documentation. Notably, many of them including GitHub Co-pilot GitHub [2021] and Amazon CodeWhisperer Amazon [2023] are getting incorporated into integrated development environments (IDEs) as assistants to improve developers' productivity.

Existing work on code LMs, such as CodeBERT Feng et al. [2020], GraphCodeBERT Guo et al. [2020], CodeGPT CodeGPT [2023], and CodeT5 Wang et al. [2021] primarily focus on the performance improvement of the code LMs on different benchmarks and treat code LMs as a *black box*. Specifically, 96% of studies focus on improving the predictive accuracy of code LMs Jiarpakdee et al. [2021]. These studies overlook a crucial aspect: understanding the underlying mechanisms by which these models make predictions or generate code. As a consequence, the inner workings of code LMs remain largely obscure, potentially resulting in the generation of vulnerable code Pearce et al. [2022], challenges in debugging Huang et al. [2023], Guo et al. [2024], and difficulties in updating the codebase Barke et al. [2023]. Moreover, the lack of interpretability undermines developers' confidence in these models and their ability to effectively leverage them in practical software development scenarios. Enhancing the interpretability of code LMs is critical for enhancing transparency, trust, compliance, and accountability in software development.

Recognizing the importance of interpretability in code LMs, Authors in Mohammadkhani et al. [2023] focused on understanding the role of attention layers in code LMs. Their study examined the distribution of attention weights across input sequences, shedding light on a crucial aspect of model behavior. Nonetheless, it is noteworthy that attention layers constitute only one-third of a typical code LM. The remaining two-thirds, primarily constituted by feed-forward (FF) layers, have largely remained unexplored in existing research. Moreover, in NLP literature, FF layers are considered the databases (i.e., memory) of the model, represented in the form of keys and values Geva et al. [2021]. In this work, we aim to bridge this gap by concentrating on the FF layers of code LMs, aiming to explain their role and impact in code LMs.

Specifically, for a given code prefix as input, we compute the activation coefficient for a selected key in a certain layer. Then, we obtain the top code prefixes whose representation produced the highest inner product with the given key. Upon analyzing these prefixes, we discover interesting syntactic and semantic patterns associated with each key. Likewise, when we mask keys related to a specific concept of interest (e.g., `numpy`), we observe a notable decrease in the performance of the code LMs concerning that particular concept. However, other programming constructs do not exhibit significant performance deterioration following the masking of the same keys. Additionally, we transform each value vector into a probability distribution by multiplying it with the output embedding matrix. Then, we assess how the predictions at each layer align with the final output of the model. Furthermore, we manipulate the context size to investigate the impact of varying context lengths on this alignment.

In our investigation, we employ two well-known autoregressive code LMs: Codegen-Mono-2.7B Nijkamp et al. [2022] and Polycoder-2.7B Xu et al. [2022]. Codegen specializes in the Python programming language, while Polycoder encompasses multiple languages, where our focus is on three diverse programming languages: Java, Go, and Python. To conduct our exploration, we collected 5,000 code files from active GitHub repositories with more than 50 stars for each programming language.

Specifically, our study focuses on the following research questions (RQs).

RQ1: *What information is stored in the feed-forward layers of code LMs?*

Considering the unexplored role of FF layers in code LMs, our inquiry aims to uncover what information is stored in FF layers. We examine the top 50 input sequences against each key in the FF layers, which exhibit the highest activation in that key relative to all other sequences in the dataset. We then qualitatively and quantitatively explored these keys to see how the model is storing information to uncover insights into the nature of information representation in the FF layers, particularly in relation to code generation tasks. Our investigation revealed that the FF layers of code LMs are responsible for capturing a wide range of information, spanning from fundamental syntactic patterns such as keywords and n-grams to more abstract concepts and semantics. Notably, the initial layers predominantly capture low-level syntactic elements (e.g., keywords, n-grams), while the higher layers capture more abstract and higher-level semantics, such as iterators and other complex programming constructs.

RQ2: *Can we precisely edit a concept of interest in code LMs, and how does such editing affect the general performance of code LMs?*

If we truly understand how information is stored in the FF layers, then we must be able to edit it. We aim to find out the feasibility of accurately editing the concept of interest in code LMs and to evaluate the subsequent impact on the model's overall performance. This inquiry is motivated by the need to quantify the adaptability of code LMs to new information, particularly concerning deprecated methods or application programming interfaces (APIs). To address this question, adopt a systematic approach. Initially, we identify and filter keys associated with APIs of interest, such as `numpy` in Python, across various programming languages using regular expressions, focusing on those keys where our concept of interest ranks among the top 50 triggers. Subsequently, we apply masking techniques to these keys and observe the effect on the model's performance concerning the concept of interest. Conversely, we evaluate the impact of masking the same keys on the model's performance on everything except the concept of interest, aiming to quantify any potential side effects on general performance. Our findings indicate a significant decrease in accuracy concerning the concept of interest, implying that the model's knowledge is highly localized. Additionally, we did not observe a noteworthy decline in the model's performance regarding all other aspects except for the concept of interest. This empirical evidence demonstrates the viability of editing operations without detrimentally affecting its general performance.

Summary of findings. We explore how FF layers encode syntactic and semantic information of programming languages and their role in generating output tokens in code LMs. Our empirical findings demonstrate that lower layers capture syntactic patterns, while higher layers encode abstract concepts and semantics. We also show that concepts of interest can be edited within FF layers without compromising the performance of code LMs. Additionally, we observe that initial layers serve as "thinking" layers, while later layers are crucial for predicting the next tokens of code.

Contributions. In summary, this work makes the following contributions:

- We explore and describe the role of feed-forward layers in code language models, which consist of two-thirds of a typical transformer model's parameters.
- We demonstrate the viability of editing a concept of interest in code language models and empirically show the impact of editing concepts on model performance.

We present our investigations and findings on information storage and editing in Sec. 2 We discuss related work in Sec. 3 and provide conclusions of this work in Sec. 4. Additionally, we explore two additional research questions along with the background in the Appendix.

2 Information storage and editing

The following section describes our methods and experiments to find out how information is stored in FF layers (RQ1), how can we edit stored concepts, and the impact of editing on code LMs (RQ2).

2.1 Information Storage

2.1.1 Capturing Top Trigger Examples.

Let us denote our dataset as D, which consists of n code prefixes represented as $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$. A code prefix $\mathbf{x}_i$ is passed through the model and an activation coefficient $a_i = \max(x_i^l \cdot k_i^l)$ is computed for every key $\mathbf{k}_i^l$ in layer l, where x_i^l denotes the representation of $\mathbf{x}_i$ at layer l, and k_i^l is the key vector corresponding to the i-th hidden dimension at layer l. This process is repeated for all the prefixes in D. Then a ranking of $\mathbf{x} \in D$ is established for each key $\mathbf{k}_i^l$ based on the activation coefficient a. For each key $\mathbf{k}_i^l$ in layer l, we then identify t trigger examples $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_t\} \subset D$, which produce activation coefficients that rank in the top 50 of a particular key $\mathbf{k}_i^l$.

Authors in Geva et al. [2021] suggest that these keys act as detectors for specific patterns from the input data. By examining top-t triggers for a key, we can deduce what patterns that key is responsive to. This method of probing allows us to uncover the encoded patterns in a given code LM's keys. That is, we can discover how the model encodes and interprets information and the model's operational logic.

2.1.2 Pattern Analysis using Regular Expression Filtering

Once we have successfully gathered top-k triggers for all the keys, we face the challenge of dealing with an extensive search space, which makes obtaining meaningful quantitative results a daunting task. For example, both the models under investigation in this work (i.e., Codegen-Mono, and Polycoder) are autoregressive models with 32 layers and 2560 hidden dimensions containing a total of $2,560 \times 4 \times 32 = 327,680$ keys. Navigating through this vast expanse is no small feat.

To tackle this issue, we employ a strategic approach. We implement regular expression (regex) filtering, targeting various application programming interfaces (APIs) such as `numpy` and `torch`, as well as fundamental programming concepts like loops and conditionals. This process helps us narrow down the search space, focusing on keys related to our areas of interest within the expansive search space.

For all of the explorations on keys, we extensively use regex filtering, along with other heuristics that are based on the frequency of occurrence of our concept of interest amongst the top triggers of each

Key	Pattern	Triggers

k_1^1	Keyword `frame`	`frame_ab.shape[2]`
		`start_frame_num = start_frame`
		`os.path.join(pred_frame_path,`

k_1^7	Keyword `assert`	`self.assertEqual(self.buffer.read(), original)`
		`self.assertIsNone(ret.exception)`
		`self.assertIsNotNone(getattr(ctx.obj, name))`

k_4^{11}	Slicing in python	`weights = weights[:, 1:1 + P - 2]`
		`lrs[:, -5:-4, :, :]`
		`priors[:, 2:])`

k_{3187}^{14}	math	`np.sin(pi * x / 2) + np.finfo(np.float32).eps)`
		#$\lvert B - A * W \rvert^2$
		`m = np.max(np.abs(covmean.imag))`

k_{18}^{26}	Image related concepts	`rgb = color.lab2rgb(lab.astype(np.float64))`
		`elif isinstance(pic, np.ndarray):`
		`low resolution photo of the {}.`

k_2^{32}	Loss concepts	`g_vggloss *= self.lambda_vgg`
		`= math.ceil(math.log(sr_factor,1 / self.scale_factor))`
		`(l_d_real + l_d_fake).backward()`

Table 1: Sample trigger examples for Python from Codegen model.

key (e.g., in key 5 out of all 50 triggers, 40 are related to `numpy`), accordingly, to handle the wast search space, and get meaningful insights.

2.1.3 Qualitative Analysis of keys

We conduct a qualitative analysis across all layers to examine the patterns of information triggered by the keys using a subset of chosen keys from each layer.

To get a better representation of chosen keys than random, we filter the keys for a particular API, for example `numpy`. We then divide the filtered keys into five different ranges based on the frequency of occurrences of the concept of interest and select five randomly selected keys from each range. In theory, it should give us 25 keys per layer for each concept of interest, which is not always the case because frequencies of occurrence of the concept of interest are not uniformly distributed across all ranges. Nonetheless, we manually go through approximately 15-25 keys per layer for each concept of interest. Doing this gives us a heuristic to get a better representation of the vast search space.

We present results from some of these selected keys in Tables 1, 2, 3, and 4. We showcase a few examples from each key, specifically, we highlight three instances that exemplify the main pattern observed among the 50 triggers for that key. We have also included the original text files associated with the keys' triggers to ensure completeness.

The quantitative analysis reveals that the initial layers, or lower layers, of the model, predominantly focus on identifying keywords and n-grams, such as the example frame in key 1 layer 1 (i.e., k_1^1) of Codegen-Mono on python in Table 1. Progressing deeper into the model, the layers exhibit an enhanced semantic understanding. A notable example is key 3187 in layer 14 (i.e., k_{3187}^{14}) of Codegen-Mono on python in Table 1, which demonstrates the model's capability not only to cluster similar mathematical functions like `np.math` and `np.sin` but also to recognize a comment that contains a mathematical equation, despite it not being code. This progression underscores a significant increase in the model's semantic understanding. As we move further into the higher layers, the model's ability to grasp and interpret complex semantic concepts, including but not only limited to loss, image, math, and slicing, becomes increasingly apparent. Through these select examples, it is clear that the model evolves to understand higher-level semantic concepts with greater depth as we ascend through its layers.

Analysis of Python in Codegen-Mono. Table 1 provides triggers for Python the Codegen-Mono model, in the first two examples it is shown that the keys are capturing keywords: frame in key 1 layer 1 (i.e., k_1^1) and keyword assert in key 1 layer 7 (i.e.;k_1^7), after these initial layers it is predominantly higher level semantics, key 4 layer 11 (i.e., k_4^{11}) is about slicing of arrays in python, though it very well might be just capturing the sign `:`, key 3187 layer 14 (i.e., k_{3187}^{14}) as mentioned above is an interesting one because the model seems to group concepts of maths in this key, even equation

Key	Pattern	Triggers

k_4^1	Keyword `runtime`	`runtime, _ := getTestModuleInstance(t)`
		`return r.runtime.regExpExec(execFn, r, s)`
		`if runtime.GOOS ==`

k_{24}^8	Keyword `func`	`func (*AnyValue_StringValue) isAnyValue_Value()`
		`func PrintToPDF() *PrintToPDFParams {}`
		`func newBaseGoCollector() baseGoCollector`

k_{7142}^{15}	Longer then token length string	`func (t *Transport) time.Duration (original)`
		`NativeHistogramMinResetDuration time.Duration`
		`Timeout time.Duration`

k_{24}^{22}	Setting Flag Values	`SYS_PPOLL = 336`
		`DLT_LINUX_LAPD = 0xb1`
		`TCP_BBR_PACE_PER_SEC = 0x43e`

k_2^{26}	Checks and errors on internet services	`conn, err := net.DialUDP("udp", nil, udpAddr)`
		`expectedFilenameURL: &url.URL{Scheme: "file", Path: ""},`
		`ConnectionTimeout: s.opts.connectionTimeout,`

k_{22}^{30}	Comments	`/* For block sizes below 64 kB, we never need`
		`// Invariant: we have a 4-byte match at s, and`
		`// before the CAS operation. So, we need to check`

Table 2: Sample trigger examples for Go from Polycoder model.

Key	Pattern	Triggers

k_3^1	Keyword `has`	`hasRouterField = true;`
		`hasLeadership = false;`
		`hasFields = writeIfNotEmpty(out,`

k_{35}^8	errors and exceptions	`log.error("Trying to reap: " + holder.path, e);`
		`System.err.println("syntax error "": " + command);`
		`Assert.fail("Expected auth exception was not thrown");`

k_{30}^{15}	Time	`long start = System.nanoTime();`
		`put("nano", System.nanoTime());`
		`Assert.assertTrue(listener.await(10, TimeUnit.SECONDS)`

k_3^{21}	Internet Protocols	`URL url = new URL(tcpUrl.replaceFirst("tcp", "http"));`
		`return localInetAddress.getHostAddress();`
		`ftpFtpConnection.ftp.changeWorkingDirectory(..);`

k_{4347}^{26}	Logs and errors	`LOG.info(String.format("-----", count));`
		`throw new IOException(String.format("Incorrect version`
		`log.error(String.format("Connection timed out,`

k_3^{31}	Loops	`for (Element mb : members) {`
		`i < constructorBean.parameterTypes.size(); i++) {`
		`while (mc.find()) {`

Table 3: Sample trigger examples for Java from Polycoder model.

comments are in that key. Key 18 in layer 26 (i.e., k_{18}^{26}) captures concepts of image, from RGB to resolution to checking if the instance is an image, the understanding of the model is so profound about concepts related to images that in a key, which we have not showcased here, it even captured an array initialization of [0,255], without any mention of the image at all. Key 2 layer 32 (i.e., k_2^{32}) captures concepts of loss in deep learning in Python. In the given triggers it captures from loss weightage to backpropagating loss to a manual equation of a loss function with no mention of loss keyword.

Analysis of Go in Polycoder. In Table 2, we present triggers for the Go on the Polycoder model. In the first two rows are examples of model capturing keywords: `runtime` in key 4 layer 1 (i.e., k_4^1) and func in key 24 in layer 8 (i.e., k_{24}^8). In the next row, we show a key 7142 layer 14 (i.e., k_{7142}^{14}) that is not necessarily capturing semantics but is capturing a longer string `time.Duration` which is not just a single keyword. In the next row, the key 24 layer 22 (i.e., k_{24}^{22}) captures the setting of different flags with hex values. The next key 2 layer 26 (i.e., k_2^{26}) is interesting as it captures checks and errors specifically on internet services, from exceptions on file name URL to connection error and timeout, it is interesting that this key not only knows about checks and errors but also checks and errors specifically on internet services. Lastly, the key 22 layer 30 (i.e., k_{22}^{30}) is capturing different comments, even with different styles of commenting too (i.e., `//` or `/*`), from this key, it is evident that the model knows the difference between comments and code.

Key	Pattern	Triggers

k_4^1	Keyword `add`	`add_tokens=True`
		`add(values)`
		`add_image_summaries=True`

k_{246}^8	Keyword `randn`	`np.random.randn(10,) * 0.1`
		`= self.rng.randn(`
		`rnn['Bin'] = rng.randn(N)/np.sqrt(1.0)`

k_{131}^{15}	Load and Save	`plt.savefig(f)`
		`test_labels = np.load(file_obj)`
		`pickle.dump(data, f)`

k_{17}^{22}	Datasets	`datasets.random_mlp(5, 1000), 100)`
		`dset.CIFAR10(args.data_path,transform=train_transform)`
		`dataset = datasets.EMPTY_DATASET`

k_{2788}^{26}	Labels	`labels = np.array([], dtype=bool)`
		`groundtruth = np.array([], dtype=bool)`
		`targets = np.zeros([batch_size, num_steps], np.int32)`

k_{5533}^{31}	Declarations with arrays	`expected_y_min = np.array([3.0, 14.0], dtype=float)`
		`Pixels = np.zeros((2 * d, 2 * d, 2), dtype=np.int32)`
		`labels = tf.constant([1, 2], dtype=tf.int32)`

Table 4: Sample trigger examples for Python from Polycoder model.

Analysis of Java in Polycoder. Table 3 presents triggers for Java on the Polycoder model. The first row is a key 2 layer 1 (i.e., k_3^1) that captures the keyword `has`, and is not different from the other experiment tables, but the next key 35 layer 8 (i.e., k_{35}^8) is different from the previously discussed tables as it seems to have a higher level of semantic understanding of errors and exceptions in Java, from logging the error to asserting and printing errors. Given this key is not in the first few layers, it is not unexpected to capture semantics, but considering other examples where keys in this range of layers were mostly capturing keywords, it is an interesting result, showcasing that the boundary of where semantic understanding of the model starts is not super clear. Next key 30 layer 15 (i.e., k_{30}^{15}) is capturing instances of time. Next key 3 layer 21 (i.e., k_3^{21}) captures concepts of network connections and network protocols specifically from FTP to TCP to the local host (i.e., connections), in contrast to key 2 layer 26 (i.e., k_2^{26}) in Table 2 which was also capturing network services, but it was specifically capturing errors and logs. This shows the understanding of the model in different semantics. Next is a key 4347 layer 26 (i.e., k_{4347}^{26}) with logs and errors from throw to logging of info and errors. Next key 3 layer 32 (i.e., k_3^{31}) is unique in the sense that it captures an actual programming concept of loops.

Analysis of Python in Polycoder. The triggers for Python on the Polycoder model are presented in Table 4. First, two rows are examples of the model capturing keywords: key 4 layer 1 (i.e., k_4^1) for `add` and key 246 layer 8 (i.e., k_{246}^8) for `randn` which is in line with the findings in other settings. Next key 131 layer 15 (i.e., k_{131}^{15}) captures codes for saving and loading different types of objects. Next key 17 layer 22 (i.e., k_{17}^{22}) captures codes for dataset initializations of different types. Next is a key 2788 layer 26 (i.e., k_{2788}^{26}) which captures labels for training. In the next row, we show a key 5533 layer 31 (i.e., k_{5533}^{31}) that captures different types of array declarations. In all of the higher-level semantic keys, keywords are rarely repeated among different triggers, which proves that these keys are actually capturing the said higher-level concepts and are not just capturing keywords.

Polysemantic Keys. In NLP interpretability literature, the concept of polysemous keys is recognized Fan et al. [2024]. Polysemantic keys are unique in their ability to engage in the representation of multiple, often unrelated, concepts or functions. Unlike their counterparts that encode singular, straightforward functions, these neurons showcase a multifaceted nature, showing a more complex and interconnected representation within the model.

Interpreting what individual neurons/keys in a neural network are doing is a daunting task, exacerbated by the complexity of polysemantic neurons. Interpretability methods aim to map these neurons' functions, striving to demystify the model's internal mechanisms. However, the polysemantic nature of some neurons adds a significant layer of complexity, as these neurons do not adhere to the simplicity of encoding a single function or concept.

Key	Triggers

k^{18}_{1187}	`labels = np.frombuffer(buf, dtype=np.uint8).astype(np.int32)`
	`euler_angles = np.asarray(euler_angles,dtype=np.float32)`
	`array_frombytes(buffer, data)`

k^{29}_{1265}	`data = np.frombuffer(buf, dtype=np.uint8)`
	`uint8image: a [height, width, depth]`
	`class EditProfileViewTest(TestCase):`

k^{30}_{770}	`+= 1 - np.array(self.env.dones)`
	`x = np.round(xyt[:,[0]]).astype(np.int32)`
	`logvar.set_shape(size__xz)`

Table 5: Polysemantic Keys with trigger examples in Codegen (Python).

Polysemantic keys in Codegen-Mono for Python. In our exploration, we also come across these polysemantic keys in coding models. In Table 5, we present examples of some polysemantic keys for Python on the Codegen-Mono model. The first row shows a key 1187 layer 18 (i.e., k^{18}_{1187}) which is capturing labels, array from bytes, and euler_angles, all of these do not belong to any one concept so it is evident that this key is not learning a singular function, instead it is a polysemantic key. The next row shows key 1265 layer 29 (i.e., k^{29}_{1265}) which contains examples of data from the buffer, a class declaration, and comments about an image, these triggers also do not have any common theme so this key is also polysemantic. Next key 770 layer 30 (i.e., k^{30}_{770}) also tells a similar story of being polysemantic.

Findings. This qualitative analysis aids in revealing the nature of the patterns and semantics captured by the code LMs. It enables us to observe the extent to which the model comprehends various high-level semantics, such as grouping a mathematical equation with math operations or capturing an array ranging from 0 to 255 within a key associated with image-related functions. This analysis answers our first research question of uncovering the underlying nature of the stored data in FF layers. We notice a consistent pattern in the information stored in FF layers. Specifically, the initial layers of the model tend to predominantly capture keywords, while higher layers tend to capture higher-level semantics.

2.2 Editing Concept of Interest

To answer our next research question about the possibility of editing a concept of interest from the model and how the editing will affect the performance of the model (RQ2), we perform the following experiments. In this work, we focus on a special case of editing: masking.

2.2.1 Masking

The first step to mask keys related to the concept of interest is to identify these keys across all layers. To do this, we filter through top-t triggers for each key $\mathbf{k}^l_i$ in layer l using regex, and identify the keys that are related to the concept of interest, among all 327,680 keys in the model. We mark a key as a key $\mathbf{k}^l_i$ as a key related to a concept of interest only if the concept of interest (e.g., `numpy`) is used amongst the top-t triggers of that key.

After identifying the keys that are related to the concept of interest, we can mask them by zeroing out the weights of the key. That is, we set $k^l_i = 0$ if the key has been identified as a key related to the concept of interest, in the previous step. Zeroing out weights is a known strategy to remove parts of the model, since zeroing out weights results in that key or part of the model not taking part in the model's output formation Haider and Taj [2021].

2.2.2 Performance on concepts of Interest

To gauge the performance of the models on concepts of interest, we use 10,000 lines of code from our curated dataset for each language and model setting, containing concepts of interest, to perform this experiment. We first select two highly used APIs or functions from each language, and then we filter the keys with top triggers for these functions or APIs using regex.

Model		Name of API of Interest	API of Interest		Concepts of non-Interest	
			Baseline	Masked	Baseline	Masked
CodeGen Mono-2B	Python	`np.`	61.06	41.07 ↓ **19.99**	61.53	58.10 ↓ **3.43**
		`torch.`	59.32	48.36 ↓ **10.96**	61.74	60.70 ↓ **1.04**
Polycoder 2.7B	Python	`np.`	55.19	41.26 ↓ **13.93**	80.18	76.18 ↓ **4.0**
		`torch.`	54.61	34.77 ↓ **19.84**	79.92	77.38 ↓ **4.0**
	Go	`log.`	69.23	62.13 ↓ **7.10**	71.52	70.60 ↓ **0.92**
		`time.`	67.23	35.42 ↓ **31.81**	71.52	64.56 ↓ **6.96**
	Java	`.equals(`	75.59	63.09 ↓ **12.5**	79.91	77.87 ↓ **2.04**
		`.get(`	47.67	23.52 ↓ **24.15**	79.77	68.87 ↓ **10.9**

Table 6: Making results indicate that masking keys associated with the API of interest notably degrades the performance of models specifically for that API. However, the overall performance of the models across all other constructs is not significantly affected.

In the case of Python and Go we see the model's performance on generating the next token right after the `API.` call. An example for `numpy` in Python would be the performance of the model to produce the right method after the `np.` (e.g., context is `val = np.`, ground truth is `array`). In the case of Java, there is no `API.` type of calls so we went with the prediction of actual method names. An example would be `System.out.println(mystr` as context and `.equals(` as the ground truth. Exact regexes used for all the APIs, and functions are shown in the Table 6 column "API of Interest". We make sure that the selected filtered examples remain consistent between both, masked and unmasked, experiments.

In Table 6, accuracies are reported for "API of Interest" in column Baseline, where we provide performance accuracies of the unmasked models (i.e., unchanged pre-trained model) on concepts of interest, while in column Masked we provide performance accuracies of the masked models (i.e., model keys related to the concept of interest are masked by the masking technique discussed above) on concepts of interest, along with the drop in accuracy from baseline unmasked experiment.

General Performance. To gauge the general performance of models, excluding selected concepts of interest, we check the model's performance on the next token prediction on 10,000 lines of code in each setting. 10,000 lines of code are filtered from the dataset through regex to not have the concepts of interest used in any of them.

Table 6 also reports results for "concepts of non-Interest", in column Baseline, where we provide general performance accuracies of the unmasked models(i.e., unchanged pre-trained model), and in column Masked we provide general performance accuracies of the masked models(i.e., model keys related to the concept of interest are masked by the masking technique discussed above), along with the drop in accuracy from baseline unmasked experiment.

Findings. The results in Table 6 help us answer RQ2, which is about the inquiry of the effects of precise editing in the network keys for a particular concept of interest. A notable drop in the model's performance can be seen for the concept of interest when the keys related to that concept are masked. Moreover, there was no significant decrease in the model's performance in areas other than the concept of interest. This provides empirical proof that it is possible to make editing changes without adversely impacting the overall performance of the model. This finding suggests that the model's knowledge is localized, and the keys we are identifying to be related to a concept of interest are indeed related to that concept. This also proves that precise editing of the model's knowledge is plausible. Nonetheless, one might wonder why the performance drops drastically but does not completely diminish. There are multiple factors contributing to this phenomenon. (*i*) We only select the top 50 triggers, which implies that we deactivate a small percentage of keys in total. Intuitively, the performance should not have dropped to zero for the API of interest. (*ii*) We did not mask polysemantic keys, which are capable of learning multiple functions. Masking these keys could potentially lead to unintended consequences on the model's overall performance. Further exploration in this direction is left to future research. Previous studies have also underscored that polysemous keys present a considerable challenge for model editing Fan et al. [2023].

3 Relater Work

Understanding the mechanisms behind the predictions of models is crucial for their deployment in real-world applications. Interpretability focuses on uncovering the rationale of model decisions, providing insights into model behavior, and enhancing the trustworthiness of models. We organize related work into two categories: interpretability in machine learning and interpretability in code LMs.

3.1 Interpretability in Machine Learning

The methods for achieving interpretability in machine learning models can be broadly categorized into three main types: (*i*) counterfactual interventions, (*ii*) hyper-network structures, and (*iii*) probing-based methods. The counterfactual intervention methods investigate how the changes in input features influence model outputs by modifying inputs and observing resultant output variations. These methods include techniques like removing or replacing input words to determine their effect on model decisions, with examples being the extraction of key sentences from labeled documents. The works Li et al. [2016] and Ribeiro et al. [2018] are examples of counterfactual interventions. The hyper-network structure approaches involve creating a learnable mask over the neurons of a frozen pre-trained model, where an L1-norm or L2-norm is applied to the masks Haider and Taj [2021]. These masks serve as indicators of neuron importance in the targeted area, examples of hyper-network structure approaches are Radford et al. [2019] and Lakretz et al. [2019]. Lastly, there are probing-based methods, which involve aligning model neurons or components with specific concepts by identifying patterns of co-occurrence between neuron activations and the target concept Geva et al. [2021], Durrani et al. [2020]. Our method of probing the model keys falls under this general category of interpretability.

3.2 Interpretability in Code LMs

Interpretability within code generation models remains a relatively under-explored area of research, and most of the research in the field focuses on the attention part of the model. Authors in Mohammadkhani et al. [2023] examine CodeBERT and GraphCodeBERT in the context of software engineering tasks. By analyzing attention scores across different token types, the study reveals patterns in how these models allocate attention to various parts of the code. Authors in Liu et al. [2024] examine the effectiveness of pre-trained language models like CodeT5 and CodeGPT in generating, translating, and repairing code, They use attention interpretability specifically focusing on how these models pay attention to different parts of the code during the generation process. Authors inPaltenghi and Pradel [2021] compare the attention mechanisms of neural models analyzing code to the attention of skilled human developers. It introduces a method for capturing human attention on code and compares it with the attention weights of neural models. To understand what coding models capture about the source code's structure and semantics, authors in Wan et al. [2022] use attention analysis along with probing on word embeddings, and syntax tree induction. All of these works focus on analyzing attention weights and activations to understand where the model directs its attention throughout the input sequence.

4 Conclusions

This work targets a key problem in code MLs – understanding the inner workings and interpretability of code language models. Our study focused on feed-forward layers of LMs, which consist of two-thirds of a typical transformer model's parameters. In our investigations, we employ two state-of-the-art code language models, Codegen-Mono and Polycoder, and leverage three widely-used programming languages, Java, Go, and Python, as the basis for our analyses. Our empirical findings show lower layers capture syntax while higher layers encode abstract concepts and semantics. We demonstrate concepts can be edited in feed-forward layers without compromising the code language model's performance. Initial layers serve as "thinking" layers, while later layers crucially predict subsequent tokens. Earlier layers can accurately predict smaller contexts, whereas the role of later layers becomes critical in facilitating better predictions. We anticipate that these findings will lay the groundwork for developing a more comprehensive understanding, enabling more effective debugging and testing methodologies for code language models

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Amazon. Amazon codewhisperer: Build applications faster and more securely with your ai coding companion. `https://aws.amazon.com/codewhisperer/`, 2023.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023. doi: 10.1145/3586030. URL `https://doi.org/10.1145/3586030`.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

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

CodeGPT. Codegpt: Jetbrains extension providing access to state-of-the-art llms, such as gpt-4, claude 3, code llama, and others, all for free. `https://github.com/carlrobertoh/CodeGPT`, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

Nadir Durrani, Hassan Sajjad, Fahim Dalvi, and Yonatan Belinkov. Analyzing individual neurons in pre-trained language models. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4865–4880, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.395. URL `https://aclanthology.org/2020.emnlp-main.395`.

Yimin Fan, Fahim Dalvi, Nadir Durrani, and Hassan Sajjad. Evaluating neuron interpretation methods of nlp models. In *Thirty-seventh Conference on Neural Information Processing Systems*, Dec 2023. URL `https://openreview.net/forum?id=YiwMpyMdPX`.

Yimin Fan, Fahim Dalvi, Nadir Durrani, and Hassan Sajjad. Evaluating neuron interpretation methods of nlp models. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. *arXiv preprint arXiv:2012.14913*, 2021.

GitHub. Github copilot: Your ai pair programmer. `https://copilot.github.com/`, 2021.

GitHub. The top programming languages. `https://octoverse.github.com/2022/top-programming-languages`, 2023.

GitHub. Github. https://www.github.com/, 2024.

Google. Bigquery public datasets. https://cloud.google.com/bigquery/public-data, 2023.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623306. URL https://doi.org/10.1145/3597503.3623306.

Muhammad Umair Haider and Murtaza Taj. Comprehensive online network pruning via learnable scaling factors. In *2021 IEEE International Conference on Image Processing (ICIP)*, pages 3557–3561. IEEE, 2021.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 432–443. IEEE, 2021.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.

Yair Lakretz, German Kruszewski, Theo Desbordes, Dieuwke Hupkes, Stanislas Dehaene, and Marco Baroni. The emergence of number and syntax units in lstm language models. *arXiv preprint arXiv:1903.07435*, 2019.

Jiwei Li, Will Monroe, and Dan Jurafsky. Understanding neural networks through representation erasure. *arXiv preprint arXiv:1612.08220*, 2016.

Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmatif. Explaining transformer-based code models: What do they learn? when they do not work? In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 96–106. IEEE, 2023.

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

Matteo Paltenghi and Michael Pradel. Thinking like a developer? comparing the attention of humans with neural models of code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 867–879. IEEE, 2021.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2022. doi: 10.1109/SP46214.2022.9833571.

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

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), jan 2020a. ISSN 1532-4435.

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

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Herve Jegou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 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*, pages 2377–2388, 2022.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

Encoder-Decoder (e.g., T5)	Encoder (e.g., BERT)	Decoder (e.g., GPT)

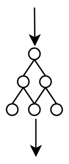
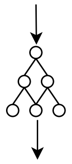
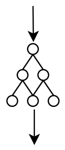
`# Function to MASK0`
`def factorial(n):`
`    if n == 0:`
`        return 1`
`    return n * MASK1(n-1)`

`MASK0 = find factorial of given number`
`MASK1 = factorial`
	`# Function to find factorial of given number`
	`def factorial(n):`
	`    if n == 0:`
	`        return 1`
	`    return n * MASK(n-1)`
	
	`MASK = factorial`
		`# Function to find factorial of given number`
		`def factorial(n):???`
		
		`    if n == 0:`
		`        return 1`
		`    return n *factorial(n-1)`

Table 7: Code LMs follow transformer architecture, which comes in three variations: encoder-decoder, encoder-only, and decoder-only.

A Background

In this section, we discuss the necessary background on transformer-based language models, code LMs, and neural memories.

A.1 Transformer-based Language Models

The Transformer architecture Vaswani et al. [2017] employs interconnected attention blocks and feed-forward layers. The attention block Bahdanau et al. [2014] facilitates the model's ability to weigh the significance of individual tokens in a sequence, thus capturing long-range dependencies across the input sequence. Concurrently, the feed-forward layers enable the model to retain crucial information derived from the training data Geva et al. [2021]. The transformer-based LMs are trained using extensive text data in a self-supervised manner. Their substantial parameter space, often reaching billions or even trillions, gives them an impressive ability to absorb broad semantic and syntactic knowledge and strong memorization skills. These models have achieved state-of-the-art performance for various NLP tasks, and the utilization of transformer-based LMs has emerged as a highly promising research direction in NLP Vaswani et al. [2017], Radford et al. [2018b], Devlin et al. [2019], Radford et al. [2019], Raffel et al. [2020b].

Transformer-based LMs have three variations in their architecture. Table 7 illustrates these architectures. Encoder-decoder models, such as T5 Raffel et al. [2020b], adhere to the original transformer architecture, with both encoder and decoder stacks. They formulate tasks by framing them as text-to-text problems, enabling unified training and inference. Encoder models, such as Bidirectional Encoder Representations from Transformers (BERT) Devlin et al. [2018], utilize the encoder stack and adopt a masked language modeling objective during training. They leverage bidirectional context understanding to comprehend text effectively. Decoder models, such as Generative Pre-trained Transformer (GPT) Radford et al. [2019], leverage the decoder stack. They are trained to predict the next tokens based on preceding ones, they excel in language generation tasks. Due to the simplicity of the decoder architecture and the prevalence of text generation tasks, decoder models have become a de facto standard for various language modeling tasks.

A.2 Code Language Models

Following the success of transformer architecture in NLP, code LMs have adopted this architecture. In code LMs, there are primarily three categories, mirroring the classifications of transformer models. These are Masked LMs, Encoder-Decoder, and Decoder-only autoregressive models.

Masked LMs in the context of coding generate code for masked tokens by classifying them based on the adjacent tokens on either side Devlin et al. [2019]. The advantage of Masked LMs over autoregressive models lies in their ability to consider the context from both sides of a masked token,

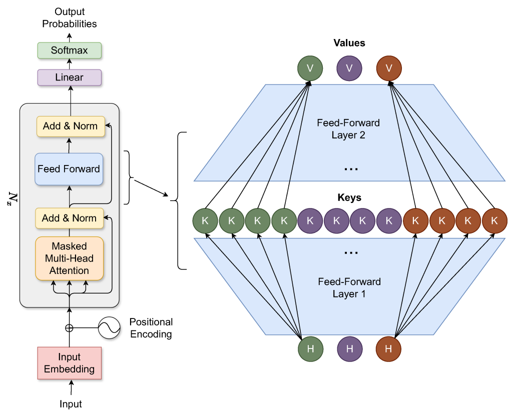


Figure 1: Feed Forward layers act as key-value memories of the model. Feed Forward layers constitute two-thirds of a typical code LM.

providing a richer base of information for predicting the masked token. Examples of Masked LMs tailored for coding include CodeBert Feng et al. [2020] and CuBERT Kanade et al. [2020].

The predominant category in code LMs is the auto-regressive models, which focus on predicting the subsequent token based on the preceding context. The GPT models Radford et al. [2018a] belong to of decoder-only category and the T5 models Raffel et al. [2020a] are encoder-decoder models. In Encoder-Decoder models an encoder encodes the input, which is then passed to a decoder akin to GPT for multiple mask prediction. The Code-specific Encoder-Decoder models include CodeT5 Wang et al. [2021] and PLBART Ahmad et al. [2021]. Lastly, Decoder-Only models (i.e., GPTs) estimate the likelihood of the next token based on previous ones. In the broader field of NLP, GPT-like models have achieved prominence, a trend that extends to code LMs as well. Decoder-Only models for code feature Lu et al. [2021], Xu et al. [2022], Nijkamp et al. [2022], Chen et al. [2021], Black et al. [2022], among others.

The widespread adoption of auto-regressive models, including GPT variants, is primarily due to their sequential left-to-right token prediction capability. This trait enables their application in a variety of contexts, such as code completion, generating comments for code, or converting plain text into code Feng et al. [2020], Guo et al. [2020].

A.3 Neural Memories

Authors in Sukhbaatar et al. [2019] has shown that feed-forward layers act as key-value memories, emulating memory networks Sukhbaatar et al. [2015]. For a given input context $\mathbf{x}$, we can compute the distribution over keys: $p(k_i \mid x) \propto \exp(\mathbf{x} \cdot \mathbf{k}_i)$ and memory of $\mathbf{x}$ can be expressed as $M(\mathbf{x}) = \sum_{i=1}^{d_m} p(k_i \mid x)\mathbf{v}_i$. That is, we can represent FF layers as $\text{FF}(\mathbf{x}) = f(\mathbf{x} \cdot K^\top) \cdot V$, where $\mathbf{x} \in R^d$ is the text input, $K, V \in R^{d_m \times d}$ represent parameter matrices and f denotes a non-linearity Geva et al. [2021].

Code LMs follow the transformer architecture Vaswani et al. [2017], which incorporates interconnected self-attention and FF layers. Each FF layer operates as a position-wise function, independently processing input vectors. The FF layers function using two matrices: one representing keys and the other values. The first matrix serves as a set of key vectors, while the second matrix serves as a set of corresponding values for these keys. Specifically, transformers employ ReLU non-linearity and the function of FF layers can be expressed as: $\text{FF}(\mathbf{x}) = \text{ReLU}(\mathbf{x} \cdot K^\top) \cdot V$, where $\mathbf{x}$ represents the input vector, K represents the output of the first matrix acting as keys, and V represents the output of the second matrix acting as values. Figure 1 illustrates the zoomed-in view of FF layers, emphasizing the keys and values.

Language	Number of Files	Number of Repositories	Minimum number of stars (GitHub)	Date Last Active (GitHub)
Python	5,000	50	50	01-01-2020
Go				
Java				

Table 8: Criteria for dataset collection.

B Approach

In this section, we discuss our approach to conducting our study, including selected code models, dataset, and research questions.

B.1 Selected Models

For our choice of models, we chose two state-of-the-art mid-sized models for our investigation. One is a mono-language model and the other is a multi-language model.

Codegen-Mono-2.7B. CodegenNijkamp et al. [2022] a 2.7 billion parameter GPT model, with 32 layers, it is trained sequentially on three datasets, called THEPILE Gao et al. [2020], BIGQUERY Google [2023], and BIGPYTHON. THEPILE is an 825.18 GB English text corpus for language modeling. The dataset is constructed from 22 diverse high-quality subsets, one of which is programming language data collected from GitHub repositories with more than 100 stars that constitute 7.6% of the dataset. The multi-lingual dataset BIGQUERY is a subset of Google's publicly available BigQuery dataset, which consists of code in multiple programming languages. For the multi-lingual training, the following 6 programming languages are chosen: C, C++, Go, Java, JavaScript, and Python. The monolingual dataset BIGPYTHON contains a large amount of data in the Python programming language.

Polycoder-2.7B. Polycoder Xu et al. [2022] is also a 2.7 billion parameter GPT model, with 32 layers, it was trained on cloned repositories for 12 popular programming languages with at least 50 stars (stopping at about 25K per language to avoid a too-heavy skew towards popular programming languages) from GitHub in October 2021. For each project, each file belonging to the majority language of that project was extracted, yielding the initial training set. This initial, unfiltered dataset spanned 631GB and 38.9M files.

B.2 Dataset

We leverage GitHub GitHub [2024] to access publicly available source code, which hosts a wide array of programming languages and diverse projects. To establish a comprehensive dataset, we systematically cloned the most prominent repositories associated with three popular programming languages; Java, Go, and Python GitHub [2023]. Our selected programming languages are representative of popular programming paradigms, imperative, dynamic, and object-oriented. Moreover, the open ecosystem in these programming languages allows us to be selective while collecting dataset to maintain high quality. To maintain the quality, we selected repositories with a minimum of 50 stars (similar to Polycoder Xu et al. [2022]). We curate 5,000 files for each of the selected programming languages, Table 8 presents the criteria we imposed while collecting the dataset from Github repositories.

We use the GitHub API using GraphQL to retrieve and list repositories, based on the following criteria: 'pushed:>2020-01-01 (i.e., active), fork:false(i.e., are repos and not forks), sort:stars'. For any specified programming language, we sort the resulting repositories by their star count and select the top 50 repositories, at the time of the search.

Table **??** provides a summary of the characteristics of our dataset. In terms of source lines of code, Python, Go, and Java each one has over 1.4M, 2.1M and 572K, respectively. Python files exhibit over 38K classes and 93K functions, while Go files has 23K `struct` counts (Go does not have class) and 101K functions defined, and in Java files, there are 15K classes containing 27K methods.

B.3 Research Questions

We consider the following research questions for our study on selected code LMs using our dataset described above.

RQ1: *What information is stored in the feed-forward layers of code LMs?* Given the unexplored nature of the role of FF layers in code LMs, Our investigation aims to clarify the precise information stored within these layers. Considering the nature of programming languages, we want to explore how syntactic information and semantics are stored in different code LMs.

RQ2: *Can we precisely edit a concept of interest from code LMs, and how does such editing affect the general performance of code LMs?* Often in programming languages and frameworks, certain methods or APIs are deprecated and code LMs would need to adapt to the changes. We explore the possibility of updating the learned concept by editing the concept of interest from code LMs. Along with the possibility of editing, we also want to measure the performance impact of the editing performed.

RQ3: *How does local information in each layer agree to the final output of code LMs?* The capability to generate output stands as a fundamental strength of code LMs. Our objective is to investigate how this output is formulated and delineate the distinct roles of various layers in this process.

RQ4: *How does the context size impact the agreement between layers in code LMs?* We want to understand the role of context size and its impact on producing output. This research question is driven by the desire to evaluate how the complexity of the model's task evolves with changes in context size.

The next two sections (Sec. 2 and C) explore FF layers in selected code LMs to answer these questions and discuss the findings of our study.

C Information Aggregation

This section elaborates on our approach and experiments to investigate the alignment between local information at different layer levels and the final output (RQ3) and study the effects of varying context sizes on these alignments (RQ4).

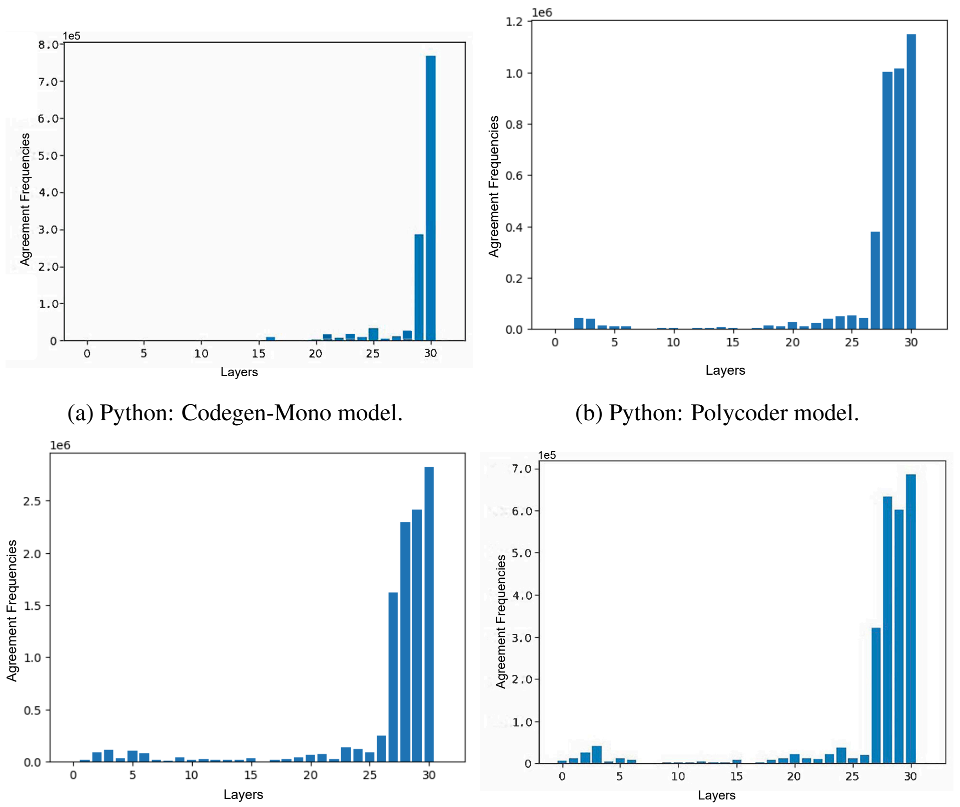
C.1 Layer Agreements to Final Output

To understand how different layers aggregate information to form the model's final output and whether different layers agree with the model's final output, we conduct the following experiment. We transform each value vector (i.e., hidden dimension output of the second feed-forward layer), denoted as v_i^l in layer l, into a probability distribution over the vocabulary and select the token with the highest probability. That is, we perform multiplication of v_i^l for each layer l with the output embedding matrix of the model E, and subsequently applying softmax function: $p_i^l = \text{softmax}(v_i^l \cdot E)$. We then apply argmax function $o_i^l = \text{argmax}(p_i^l)$ to get o_i^l which is the top predicted token by layer l, when $x_i \in D$ is passed as input to the model.

It is important to note that the resulting probability distribution p_i^l is not calibrated. However, it is worth mentioning that the ranking established by p_i^l remains unaffected, allowing for meaningful analysis. To compute agreement we compare the top token prediction o_i^l from each layer l with the final output of the model o_i^L, where L represents the last layer. If $o_i^l = o_i^L$, then layer l agrees with the model's final output when $x_i \in D$ is passed as input to the model.

To conduct the agreement experiment, we utilize our entire dataset. For each line of code, we generate multiple examples by considering all prefixes of the line, resulting in n examples, where n represents the number of tokens in the line of code. Figure 2 presents the results of this experiment, where it is evident that the agreement of initial layers in all the settings is quite low but as we move ahead into the model the agreement starts to increase, and in the last few layers it is exponentially high.

Python on Codegen-Mono. In Figure 2 (a), we present agreement results for the Codegen-Mono model on Python language. From the graph of frequencies, it can be seen that the initial half layers of



(a) Python: Codegen-Mono model.

(b) Python: Polycoder model.

(c) Go: Polycoder model.

(d) Java: Polycoder model.

Figure 2: Agreement between different layers and model's final output.

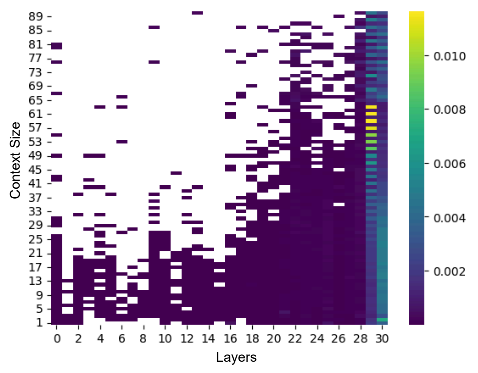
the model till layer 15 have no agreement with the final output of the model, at layer 16 there is some agreement, but it drops again till layer 20 after layer 20 it gradually increases till 28 to 29 layer, and then we see a sudden exponential increase in the agreement till the second last layer of the model.

Python on Polycoder. We present agreement results for the Polycoder model on Python language in Figure 2 (b). In these results, we see a little different agreement pattern where there is a little agreement in the initial layers till layer 5 then it goes down but does not become completely zero. After layer 20 it starts to gradually increase and after layer 25 it increases exponentially and is quite high in the last few layers of the model. This behavior is different from the one we noticed in the previous results in Figure 2 (a), but is consistent with all the other results on the Polycoder model. We believe that this behavior is dependent on the model and is influenced by the nature of their respective training processes, with one being monolingual and the other multilingual.

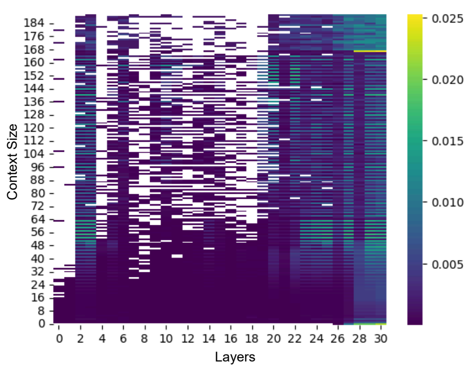
Go on Polycoder. Figure 2 (c) presents agreement results for the Polycoder model on Go Language. It shows a similar story to the agreement graph of the Polycoder model on Python language in Figure 2 (b), there is some agreement in the initial layers till layer 7, then it drops but never goes to zero, then after layer 20 it gradually increases and after layer 25 it exponentially increases, and the peak of last few layers is close to each other, unlike Codegen-Mono model on Python.

Java on Polycoder. In Figure 2 (d) agreement results for the Polycoder model on Java language are presented. These results are similar to the other results of the Polycoder model on other languages. There is some agreement in the initial layers, then it drops till layer 20 and after layer 20 it gradually increases and in the last 4 layers it is exponentially high, and the peaks for the last layers are closer to each other.

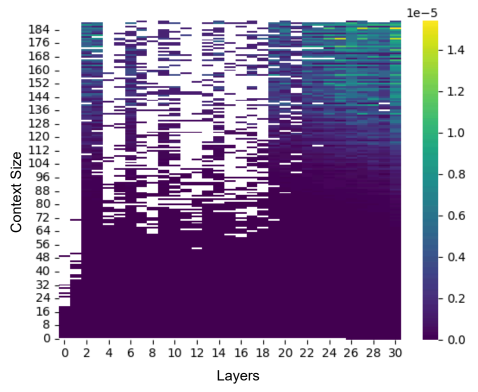
Findings. The results in Figure 2 help us to answer RQ3, how local information in each layer agrees with the final output of the code LMs. We observe that the early layers of the model show minimal



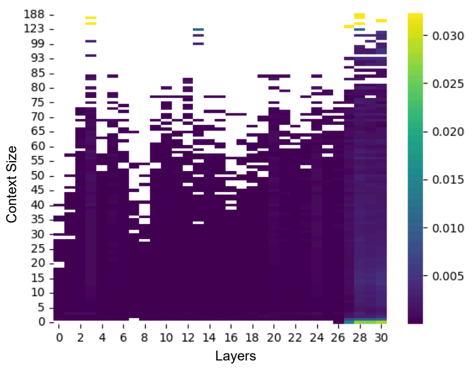
(a) Python: Codegen-Mono model.



(b) Python: Poly-Coder model.



(c) Go: Poly-Coder model.



(d) Java: Poly-Coder model.

Figure 3: Layer agreements with the model's final output, as we vary the length of the context.

agreement with the model's final output, implying that their primary role is akin to processing or "thinking" rather than having a direct impact on the output. Conversely, as we move deeper into the model, there is a noticeable rise in agreement, implying that the later layers, with more refined information, are more important in forming the final output.

We also observe a difference in behaviors between the two models where results for the Polycoder model have some agreement in the initial layers, which then drops and goes up again after layer 20, this is in contrast to the result for the Codegen-Mono model where initial half of the layers have no agreement with the final output of the model. We also observe that the peaks on high agreement in the last few layers in the Polycoder model are closer to each other, this is in contrast to the Codegen-Mono model where the peaks in the last layers are also exponential to each other. We posit that it is a model-dependent behavior and has to do with the nature of training of these two models, one being monolingual while the other being multilingual.

C.2 Impact of Variance in Context Size to Layer Agreements

To answer RQ4, how context size affects the output formulation and agreement of layers with the final output of the model, we repeat the same experiment as above with varying context sizes from 1 to 188. We analyze the agreement among layers and token counts and present our results using 2D heatmaps in Figure 3. We found that initial tokens are generally easier to predict, thus showing higher agreement between initial layers and the final output. This might be attributed to the model capturing more salient features in the early stages of processing. In contrast, later tokens, which are more challenging to predict, tend to have higher agreement with the upper layers and the final output. This suggests that the later stages of processing, possibly involving more abstract or contextual information, play a more significant role in predicting these complex tokens.

Python on Codegen-Mono. In Figure 3 (a), we provide agreement results for the model Codegen-Mono on Python language of different layers with the final output of the model along with varying context sizes from 1 to 89. From the heatmap, we see that when the context size is small we see agreement even in the initial layers but as the context size increases only the later layers after layer 18 have agreement with the final output of the model.

Python on Polycoder. Figure 3 (b) presents agreement results for the model Polycoder on Python language of different layers with the final output of the model along with varying context sizes from 1 to 184. We observe similar results about the agreement, wherein as the context size increases, only the later layers exhibit agreement with the model's final output. However, there is a notable difference in the behavior of the agreement from Figure 3 (a). Across all results for the Polycoder model, we observe a peak in agreement across all context sizes in the initial few layers. After these first few layers, this behavior aligns with our observation of the agreement results for the Polycoder model (in Figure 2), where the agreement increases for a few layers and then decreases after the first few layers. We posit that this behavior stems from the inherent differences in the nature of both models. Nonetheless, our core assertion remains valid, as even the first layer can generate accurate predictions when the context size is relatively small.

Go on Polycoder. In Figure 3 (c), we provide agreement results for the model Polycoder on the Go language of different layers with the final output of the model along with varying context sizes from 1 to 184. We observe a similar trend to another finding in the Polycoder model, where the agreement across all context sizes initially increases around layer 5 before declining. However, after that, as the context size increases, only the last layers exhibit agreement with the final output of the model.

Java on Polycoder. Figure 3 (d) presents the agreement results of different layers within the Polycoder model, trained on the Java programming language, with the final output of the model. These agreement measurements are provided across varying context sizes, ranging from 1 to 188 tokens. This result is also in line with the other results of the Polycoder model model where the agreement of all context sizes increases around layer 5 and then goes down, but then as the context size increases only the last layers agree to the final output of the model.

Findings. The results in Figure 3 help us answer RQ4: understand the behavior of the models with varying context sizes. From these results, it is evident that the complexity of the task for the model changes with varying context sizes. Our findings reveal that even the earlier layers, as early as the very first layer, across all four settings, can predict some tokens correctly in the smaller context size. However, as the context size increases only the later layers can make the correct prediction except for model-dependent behavior in the results on the Polycoder model, where there was some agreement around layer 4 to layer 6 in both agreement experiments, across all settings. This signifies that as the context size becomes larger, the task of accurate prediction becomes difficult for the model. This behavior may look counter-intuitive at first because a larger context size has more information for the model to make predictions. But a larger context also requires the model to have a higher semantic understanding of the input, which our findings from the exploration of keys suggest that only higher layers possess (refer to RQ1). With a smaller context size, there is a possibility that even completing n-grams and keywords could result in the correct prediction. Our investigation into the keys has revealed that initial layers indeed demonstrate an aptitude for understanding keywords and n-grams, thereby enabling them to occasionally predict the correct output when the context size is sufficiently small.