Beyond Syntax: How Do LLMs Understand Code?

Marc North
CS, Durham University
Durham, UK
marc.north@durham.ac.uk

Amir Atapour-Abarghouei
CS, Durham University
Durham, UK
amir.atapour-abarghouei@durham.ac.uk

Nelly Bencomo
CS, Durham University
Durham, UK
nelly.bencomo@durham.ac.uk

***Abstract*—Within software engineering research, Large Language Models (LLMs) are often treated as 'black boxes', with only their inputs and outputs being considered. In this paper, we take a machine interpretability approach to examine how LLMs internally represent and process code.**

We focus on variable declaration and function scope, training classifier probes on the residual streams of LLMs as they process code written in different programming languages to explore how LLMs internally represent these concepts across different programming languages. We also look for specific attention heads that support these representations and examine how they behave for inputs of different languages.

Our results show that LLMs have an understanding — and internal representation — of *language-independent* coding semantics that goes beyond the syntax of any specific programming language, using the same internal components to process code, regardless of the programming language that the code is written in. Furthermore, we find evidence that these language-independent semantic components exist in the middle layers of LLMs and are supported by language-specific components in the earlier layers that parse the syntax of specific languages and feed into these later semantic components.

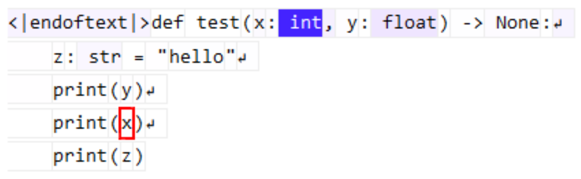
Finally, we discuss the broader implications of our work, particularly in relation to concerns that AI, with its reliance on large datasets to learn new programming languages, might limit innovation in programming language design. By demonstrating that LLMs have a language-independent representation of code, we argue that LLMs may be able to flexibly learn the syntax of new programming languages while retaining their semantic understanding of universal coding concepts. In doing so, LLMs could promote creativity in future programming language design, providing tools that augment rather than constrain the future of software engineering.

***Index Terms*—Mechanistic interpretability, Large Language Models (LLMs), Software engineering**

I. Introduction

Despite the recent progress of Large Language Models (LLMs) in coding tasks, the internal mechanisms that LLMs use to understand and process code are poorly understood. Specifically, how LLMs process code written in different programming languages — and whether LLMs have a semantic understanding of underlying coding concepts, rather than relying on a shallow processing of the specific syntax of each programming language — remains unclear.

This is the central research question that our work aims to answer: **Is LLMs' understanding of code language specific, or do LLMs have an internal representation of coding logic that is independent of any specific programming language syntax?**



```
<|endoftext|>def test(x: int, y: float) -> None:↵
    z: str = "hello"↵
    print(y)↵
    print(x)↵
    print(z)
```

Fig. 1: Attention pattern of head 12@21 in Pythia 2.8b for the token 'x' (highlighted in red). The attention head is moving information from the variable type token 'int' to the variable token 'x'.

A. Syntax vs Semantics

To briefly elucidate this central question, Figure 1 shows the attention pattern of an attention head in Pythia 2.8b while processing some Python code. As can be seen, the head strongly attends to the token corresponding to the type of the variable *x*; that is, this attention head is *moving information from the type token 'int' to the variable token 'x'*. It is unsurprising that such an attention head would exist in an LLM – after all, a variable's type seems like useful information to move to the variable's token position. However, it does raise the question of how an LLM might perform this task for inputs of different programming languages; there are two plausible methods one can imagine:

- An LLM could have separate circuits (sub-graphs of connections between internal components) for each programming language it has seen in its training data – e.g., a 'variable type' circuit for Python code and a completely separate 'variable type' circuit for Java code.
- An LLM could have one 'variable type' circuit that uses the same model components to understand variable types regardless of the language of the input.

In this paper, we show that is the second — in our view more interesting — method that LLMs use to understand code written in different programming languages. That is, **LLMs use the same attention head to move variable type information for inputs from different programming languages**. Furthermore, the variable type information is represented internally the same way regardless of programming language. This idea is illustrated in Figure 2.

We find that this separate syntax-dependent to common syntax-independent path is used by LLMs of different archi-

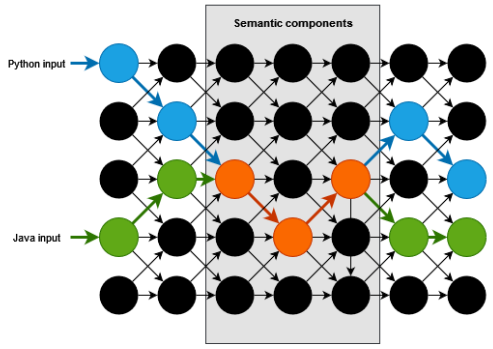


Fig. 2: Conceptual illustration of the path through a transformer of separate inputs written in Python and Java respectively.

tectures and sizes, suggesting that this behaviour is universal.

B. *Why Does This Matter?*

40% of code being committed to GitHub is written using AI, and developers who use AI have been found to be 55% more productive [1]. There are, however, concerns about the long-term effects that AI will have on software engineering.

In education, for instance, there is a growing worry that students may become overly reliant on AI tools to generate and debug code [2]. Moreover, that a reliance on AI will entrench old ideas and stifle new innovation.

When examining how the rise of AI-assisted programming has negatively affected knowledge sharing, Burtch et al. [3] found that AI has already had a noticeable impact, commenting that "reliance on AI-generated solutions might contribute to stagnation... impeding progress across various fields, including programming and other technical domains."

Of specific relevance to our research is the future impact of AI-assisted programming on the development of programming languages. While LLMs have made huge strides in programming ability, recently achieving 92% on the HumanEval benchmark [4], it was found that AI "typically performs better on languages for which bigger datasets are available" [5], with the authors further speculating that the varying abilities of LLMs "will likely determine which language will be used in the future". This is a cause for concern, as the development of new programming languages helps make software more "maintainable, robust, and performance-guaranteed" [6].

This is the context in which we explore how LLMs understand code. It is our hope that knowledge of LLMs' internal representation of code will help us avoid this prophesied stagnation of software engineering.

The rest of the paper sets out the background that this work builds on, details the methods via which we discover and quantify our findings, and discusses the practical implications and future plans of our work.

II. Background and Related Work

A. *Mechanistic Interpretability*

Since LLMs are trained with the objective of minimising loss on next-token prediction [7], they are often thought of as simply token predictors, or 'stochastic parrots' [8]. An alternative idea, with links to neuroscience research [9], is that this next-token prediction objective — given enough model capacity and training data — is sufficient for a model to develop internal, abstract 'world models' [10] that go beyond parroting language, enabling LLMs to develop structured reasoning abilities and internal representations of complex ideas such as — as we explore in our work — programming concepts and semantics.

Mechanistic interpretability (MI) is a burgeoning field of AI research that aims to reverse engineer machine learning models' internal mechanisms into human-understandable algorithms — going beyond previous black-box interpretability techniques [11] — that, as well as being of inherent academic merit, potentially has practical application for AI alignment [12], model steering [13], and AI safety [14] in many different domains [15] [16].

A central concept in mechanistic interpretability is the residual stream and its role as a channel through which model components communicate [17]. In MI, the residual stream is thought of as the cumulative sum of all model components, which can quantify each individual component's contribution to not only the model's final output, but also to the input of each downstream component within the model [18].

One MI technique that can be used to demystify the residual stream is *probing* [19]. A probe is a classifier model — separate from the LLM being interpreted — that is trained to classify a linguistic feature using an LLM's internal representation as input. For example, we might use the values of the residual stream at a certain layer to classify the part-of-speech at token positions. The classifier's performance is used to infer the extent to which the LLM's internal representation captures the feature being classified.

Another useful technique for understanding the behaviour of LLMs is ablation [20], where specific components of the model are disabled, typically either by setting the component's output to zero or to the component's mean output across a dataset, to observe their impact on performance. This allows us to quantify how much the model depends on that component, or set of components, for a given task.

Recent interpretability research has also developed techniques to examine the multilayer perceptron (MLP) layers of LLMs, such as sparse autoencoders [21] and transcoders [22]. However, in this paper we focus on the attention layers, which contain *attention heads* that move information between tokens [23], as we are exploring how LLMs understand and reason about code. LLMs have been found to do their reasoning in the attention layers [24], while the MLP layers act as the model's knowledge store [25]. As such, while MLP interpretability is an important area of study and may become relevant to our future work, it is not the focus of this paper.

In the field of natural language, Jawahar et al. [26] found that BERT captures linguistic information in a similar way to classical language tree structures, and that different layers capture different levels of language information. More recently, research has been carried out into the abilities of LLMs to understand multiple natural languages, with Zhao et al. [27] finding that LLMs process different natural languages in language-specific components in the early layers, before using the same model components across all languages to solve the task in the mid-layers, before splitting back up into language-specific components in the later layers.

While much mechanistic interpretability research has focused on finding specific circuits for modal algorithms, our focus in this work is not to exhaustively describe the circuits we are examining, but rather to demonstrate that components within these circuits are independent of the syntax of specific programming languages and instead process programming concepts at a semantic level.

III. Methods and Results

A. *Type-Mover Heads*

We first look for attention heads that use variable declaration statements to move information from tokens containing the variable type to the token containing the corresponding variable name. In order to explore LLMs generally, we examine three different models with diverse model architectures and different sizes: LlaMa 2 7B [28], Pythia 2.8B [29], and GPT-J 6B [30].

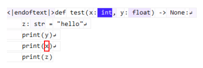
The attention pattern in an attention head for a destination token with sequence index i is given by:

$$\text{Attention}(i) = \text{softmax}\left(\frac{(X\,W_q)[i](X\,W_k)^T}{\sqrt{d_\text{head}}}\right), \qquad (1)$$

where W_q and W_k are the head query and key matrices respectively, X is the n x d_{model} input matrix to the attention head, where n is the sequence length and d_{model} is the dimensionality of the model residual stream, and d_{head} is the attention head dimensionality.

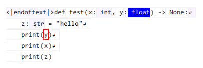
We look for attention heads that move information from *variable type* tokens to *variable name* tokens by finding heads for which $\arg\max(\text{Attention}(v)) = t$ (where v is the index in the input sequence of the *variable name* token and t is the index of the *variable type* token) for every example of generated datasets of Python, Java, and Go code. That is, heads for which the highest attention score of the *variable name* token is the *variable type* token. After finding a list of type-attending attention heads for each language, we filter to heads that are on all three lists, i.e., heads that attend to variable type for *all three languages*.

We found such attention heads present in each of the models we investigated; one such attention head can be seen in figure 3, which shows the attention pattern of the same attention head on inputs of Python, Java, and Go, where for the three different variables in each code sample, the head always attends to that variable's type token. This attention head's behaviour is *independent of the programming language of the input*, relying



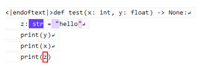
```
<|endoftext|>def test(x: int, y: float) -> None:↵
    z: str = "hello"↵
    print(y)↵
    print(x)↵
    print(z)
```

(a) Python: token 'x'



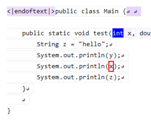
```
<|endoftext|>def test(x: int, y: float) -> None:↵
    z: str = "hello"↵
    print(y)↵
    print(x)↵
    print(z)
```

(b) Python: token 'y'



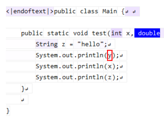
```
<|endoftext|>def test(x: int, y: float) -> None:↵
    z: str = "hello"↵
    print(y)↵
    print(x)↵
    print(z)
```

(c) Python: token 'z'



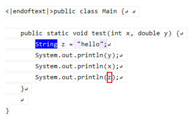
```
<|endoftext|>public class Main {↵ ↵

    public static void test(int x, double y) {↵
        String z = "hello";↵
        System.out.println(y);↵
        System.out.println(x);↵
        System.out.println(z);↵
    }↵
    ↵
}
```

(d) Java: token 'x'



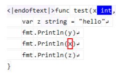
```
<|endoftext|>public class Main {↵ ↵

    public static void test(int x, double y) {↵
        String z = "hello";↵
        System.out.println(y);↵
        System.out.println(x);↵
        System.out.println(z);↵
    }↵
    ↵
}
```

(e) Java: token 'y'



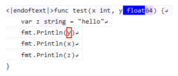
```
<|endoftext|>public class Main {↵ ↵

    public static void test(int x, double y) {↵
        String z = "hello";↵
        System.out.println(y);↵
        System.out.println(x);↵
        System.out.println(z);↵
    }↵
    ↵
}
```

(f) Java: token 'z'



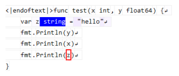
```
<|endoftext|>func test(x int, y float64) {↵
    var z string = "hello"↵
    fmt.Println(y)↵
    fmt.Println(x)↵
    fmt.Println(z)↵
}
```

(g) Go: token 'x'



```
<|endoftext|>func test(x int, y float64) {↵
    var z string = "hello"↵
    fmt.Println(y)↵
    fmt.Println(x)↵
    fmt.Println(z)↵
}
```

(h) Go: token 'y'



```
<|endoftext|>func test(x int, y float64) {↵
    var z string = "hello"↵
    fmt.Println(y)↵
    fmt.Println(x)↵
    fmt.Println(z)↵
}
```

(i) Go: token 'z'

Fig. 3: Attention patterns of Pythia 2.8b head 12@21 on three different inputs at three different token positions. This **single attention head** attends to the destination token's declaration type for input written in Python, Java, or Go, even though the declaration syntax is different in each language.

on the language-specific syntax processing of earlier model layers.

That there are individual heads in each model that perform this function for code written in *different programming languages* shows that **these heads are programming language independent, and that these attention heads are working based on *code semantics*, not just *code syntax*.**

B. *Type-Mover Attention Head Classification Probes*

Following on from our finding that attention heads exist that move information from *variable type* tokens to the corresponding *variable name* tokens, we next explore what information these heads are moving and — importantly — whether the information being moved is *language independent*.

We train probes to classify variable types at the *variable name* token position, using the activations of these attention heads across a dataset of code examples as inputs. These classifiers are simple dense networks with one hidden layer that is 1/4 the size of the input layer, using ReLU activation functions. We use Adam optimisation and cross entropy loss. We train one classifier per programming language, using generated datasets of Python, Java, and Go code.

These head-activation classifiers achieved perfect accuracy, confirming our intuition that the information being moved by these heads is indeed the variable token's corresponding variable type. That variable-type classifiers can be trained on the attention-head activations of *variable name* tokens shows that LLMs move variable type information from *variable type* tokens to the corresponding *variable name* tokens when processing code. This result is interesting, but not surprising. More significantly, **these trained classifiers could also accurately classify the types of variables in inputs written in programming languages *that they were not trained on***; e.g., a classifier trained on the residual streams of Python inputs could accurately classify variable types in Java code. This

	Training Dataset	Unablated Test Dataset			Ablated Test Dataset		
		Python	Java	Go	Python	Java	Go
(a) LLaMa 2 7B	Python	1.00	1.00	1.00	0.55	0.65	0.35
	Java	1.00	1.00	1.00	0.45	0.85	0.75
	Go	0.95	0.97	1.00	0.20	0.50	0.40
(b) GPT-J 6B	Python	1.00	1.00	1.00	0.30	0.45	0.45
	Java	1.00	1.00	1.00	0.25	0.95	0.55
	Go	1.00	0.95	1.00	0.40	0.60	0.85
(c) Pythia 2.8B	Python	1.00	0.97	0.91	0.44	0.50	0.56
	Java	0.85	1.00	0.85	0.28	0.98	0.42
	Go	0.99	0.72	1.00	0.29	0.49	0.91

TABLE I: Accuracy of the classifier probes trained on the residual stream values of each model with inputs of each language. Each probe is tested against the test dataset of each language, not just the language it was trained on, as well as the ablated versions of each model.

further shows that the internal representation of variable types are represented internally by LLMs in a *language independent* way.

C. Residual Stream Classification Probes

We similarly train classifiers on the *residual stream* of each model after the attention blocks identified in section III-B at the *variable name* token position, using the same classifier architecture and training as described above. The accuracy of these residual-stream classifiers is shown in Table I; we find that these classifiers can accurately classify the variable type of a token based on the residual stream of the *variable name* token.

As with the attention-head classifiers, **these residual-stream classifiers can also accurately classify the types of variables in inputs written in programming languages *that they were not trained on***, showing that the models' internal representation of variable type is language independent.

D. Semantic Component Ablation

In section III-B, we identified attention heads in each model that move variable type information to *variable name* tokens, and — importantly — do so for inputs written in different programming languages, i.e. they are semantic, rather than syntactic.

We create ablated versions of each of the LLMs by zero-ablating all of these semantic-level type-moving heads in order to to test how this will affect the model's ability to identify variable types. We quantify this by testing the same residual-stream trained classifiers from III-C on the residual stream of these ablated models. As table I shows, the classifiers' accuracy dropped significantly, i.e. **ablating these semantic-level attention-heads removed type information from the *variable name* tokens**.

Interestingly, the Java- and Go-trained classifiers were still able to detect variable types in the language they were trained on, while their accuracy on languages they weren't trained on dropped. This perhaps suggests that when we ablate the semantic-level model components, the classifiers still have syntax-level features in the residual stream to fall back on. The same doesn't seem to be true of the Python-trained classifier.

One interesting possibility is that the way that LLMs process code is by having an internal representation of one programming language – in this case Python — and then translating code from any other language into this one representation in its early layers, i.e. LLMs have language-specific early-layer components and then in the mid-layers process *all* code as if it were Python code. This would be similar to Zhao et al.'s argument that multilingual LLMs first translate other languages into English in their early layers before processing everything as English [27]. This interpretation would explain our finding that ablating semantic components has a much larger negative effect on a model's internal representation of Python than on that of Java or Go; in this interpretation, a model's code semantic components *are* its Python components. While an interesting thought, more investigation is required to explore this idea.

IV. Future Plans

Other programming features. We have looked at how LLMs process variable types; future work will look at other programming features, such as function scope and control structures, to explore if they are similarly processed in a language-independent way.

Full LLM circuits. This work has focused on identifying semantic-level components of LLMs. Future work will join this up fully with earlier model components, using mechanistic interpretability techniques such as direct logit attribution, to develop a full picture of code-related circuits in LLMs and how earlier language-specific syntax components connect to these later semantic components.

Syntax-level fine-tuning. A future research direction of our work is fine-tuning LLMs by freezing language-independent semantic components while allowing the language-specific layers to adapt. Future work will explore whether this approach allows an LLM to learn new programming languages, or enhance its understanding of a less common programming language — without altering its core understanding of coding semantics — more effectively and efficiently than existing fine-tuning techniques.

MLP language independence. Training SAEs and transcoders on MLP layers to identify the directions in the model's vector space of programming concepts across different programming languages in order to determine whether programming knowledge is stored in a language-agnostic way in the MLP layers of LLMs.

Python as LLMs' internal representation of code. Following on from our observation in Section III-D that ablating semantic-level LLM components has a greater effect on models' ability to detect variable types in the residual stream of Python code, future work will explore the idea that LLMs use Python as their internal representation of code, and whether there are specific features of Python or of widely used LLM training datasets that would lead to this.

REFERENCES

[1] Microsoft, "Morgan stanley tmt conference," 2023. [Online]. Available: https://www.microsoft.com/en-us/Investor/events/FY-2023/Morgan-Stanley-TMT-Conference

[2] M. Abbas, F. A. Jam, and T. I. Khan, "Is it harmful or helpful? examining the causes and consequences of generative ai usage among university students," *International Journal of Educational Technology in Higher Education*, vol. 21, no. 1, p. 10, Feb. 2024.

[3] G. Burtch, D. Lee, and Z. Chen, "The consequences of generative ai for ugc and online community engagement," *SSRN Electronic Journal*, Jan. 2023.

[4] Anthropic, "Introducing claude 3.5 sonnet." [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[5] A. Buscemi, "A comparative study of code generation using chatgpt 3.5 across 10 programming languages," 2023. [Online]. Available: https://arxiv.org/abs/2308.04477

[6] "How do developers discuss and support new programming languages in technical qa site? an empirical study of go, swift, and rust in stack overflow," vol. 137, p. 106603, Sep. 2021.

[7] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035

[8] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big? ," in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '21. New York, NY, USA: Association for Computing Machinery, Mar. 2021, p. 610–623. [Online]. Available: https://dl.acm.org/doi/10.1145/3442188.3445922

[9] T. Salvatori, A. Mali, C. L. Buckley, T. Lukasiewicz, R. P. N. Rao, K. Friston, and A. Ororbia, "Brain-inspired computational intelligence via predictive coding," no. arXiv:2308.07870, Aug. 2023, arXiv:2308.07870 [cs]. [Online]. Available: http://arxiv.org/abs/2308.07870

[10] J. Kulveit, C. von Stengel, and R. Leventov, "Predictive minds: Llms as atypical active inference agents," no. arXiv:2311.10215, Nov. 2023, arXiv:2311.10215 [cs]. [Online]. Available: http://arxiv.org/abs/2311.10215

[11] S. Casper, C. Ezell, C. Siegmann, N. Kolt, T. L. Curtis, B. Bucknall, A. Haupt, K. Wei, J. Scheurer, M. Hobbhahn, L. Sharkey, S. Krishna, M. Von Hagen, S. Alberti, A. Chan, Q. Sun, M. Gerovitch, D. Bau, M. Tegmark, D. Krueger, and D. Hadfield-Menell, "Black-box access is insufficient for rigorous ai audits," in *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, Jun. 2024, p. 2254–2272, arXiv:2401.14446 [cs]. [Online]. Available: http://arxiv.org/abs/2401.14446

[12] A. Arditi, O. Obeso, A. Syed, D. Paleka, N. Panickssery, W. Gurnee, and N. Nanda, "Refusal in language models is mediated by a single direction," no. arXiv:2406.11717, Jul. 2024, arXiv:2406.11717 [cs]. [Online]. Available: http://arxiv.org/abs/2406.11717

[13] A. M. Turner, L. Thiergart, G. Leech, D. Udell, J. J. Vazquez, U. Mini, and M. MacDiarmid, "Activation addition: Steering language models without optimization," no. arXiv:2308.10248, Jun. 2024, arXiv:2308.10248 [cs]. [Online]. Available: http://arxiv.org/abs/2308.10248

[14] L. Bereska and E. Gavves, "Mechanistic interpretability for ai safety – a review," no. arXiv:2404.14082, Aug. 2024, arXiv:2404.14082 [cs]. [Online]. Available: http://arxiv.org/abs/2404.14082

[15] M. North, A. Atapour-Abarghouei, and N. Bencomo, "Code gradients: Towards automated traceability of llm-generated code," Aug. 2024. [Online]. Available: https://durham-repository.worktribe.com/output/2433851

[16] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. v. d. Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," no. arXiv:2203.15556, Mar. 2022, arXiv:2203.15556 [cs]. [Online]. Available: http://arxiv.org/abs/2203.15556

[17] N. Elhage, N. Nanda, C. Olsson, T. Henighan, N. Joseph, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, N. DasSarma, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah, "A mathematical framework for transformer circuits." [Online]. Available: https://transformer-circuits.pub/2021/framework/index.html

[18] Z. Yu and S. Ananiadou, "Exploring the residual stream of transformers," no. arXiv:2312.12141, Dec. 2023, arXiv:2312.12141 [cs]. [Online]. Available: http://arxiv.org/abs/2312.12141

[19] C. Singh, J. P. Inala, M. Galley, R. Caruana, and J. Gao, "Rethinking interpretability in the era of large language models," no. arXiv:2402.01761, Jan. 2024, arXiv:2402.01761 [cs]. [Online]. Available: http://arxiv.org/abs/2402.01761

[20] Y. Yao, X. Xu, and Y. Liu, "Large language model unlearning," no. arXiv:2310.10683, Feb. 2024, arXiv:2310.10683 [cs]. [Online]. Available: http://arxiv.org/abs/2310.10683

[21] Anthropic, "Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet." [Online]. Available: https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html

[22] J. Dunefsky, P. Chlenski, and N. Nanda, "Transcoders find interpretable llm feature circuits," Jun. 2024. [Online]. Available: https://arxiv.org/abs/2406.11944v1

[23] M.-H. Guo, C.-Z. Lu, Z.-N. Liu, M.-M. Cheng, and S.-M. Hu, "Visual attention network," no. arXiv:2202.09741, Jul. 2022, arXiv:2202.09741 [cs]. [Online]. Available: http://arxiv.org/abs/2202.09741

[24] B. Liao and D. V. Vargas, "Attention-driven reasoning: Unlocking the potential of large language models," no. arXiv:2403.14932, Mar. 2024, arXiv:2403.14932 [cs]. [Online]. Available: http://arxiv.org/abs/2403.14932

[25] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," no. arXiv:2012.14913, Sep. 2021, arXiv:2012.14913 [cs]. [Online]. Available: http://arxiv.org/abs/2012.14913

[26] G. Jawahar, B. Sagot, and D. Seddah, "What does bert learn about the structure of language?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, A. Korhonen, D. Traum, and L. Màrquez, Eds. Florence, Italy: Association for Computational Linguistics, Jul. 2019, p. 3651–3657. [Online]. Available: https://aclanthology.org/P19-1356

[27] Y. Zhao, W. Zhang, G. Chen, K. Kawaguchi, and L. Bing, "How do large language models handle multilingualism?" 2024. [Online]. Available: https://arxiv.org/abs/2402.18815

[28] Meta, "Llama 2: Open foundation and fine-tuned chat models," no. arXiv:2307.09288, Jul. 2023, arXiv:2307.09288 [cs]. [Online]. Available: http://arxiv.org/abs/2307.09288

[29] S. Biderman, H. Schoelkopf, Q. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika, and O. van der Wal, "Pythia: A suite for analyzing large language models across training and scaling," no. arXiv:2304.01373, May 2023, arXiv:2304.01373 [cs]. [Online]. Available: http://arxiv.org/abs/2304.01373

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," Jul. 2021. [Online]. Available: https://arxiv.org/abs/2107.03374v2