Are Code and Math Reasoning Distinct in Large Language Models? A Mechanistic Interpretability Study

Ishita Kakkar [*] **Prashant Gokhale** [*] **Rishabh Sharma** [*] **Sai Krishna Ponnam** [*]

Abstract

Large language models (LLMs) demonstrate strong performance on both mathematical reasoning and program synthesis tasks, yet it remains unclear whether these abilities arise from shared internal mechanisms or from distinct, specialized circuits. In this work, we present a mechanistic interpretability study comparing math and code reasoning in a modern transformer model. Using Activation-aware Singular Value Decomposition (ActSVD), we identify low-rank subspaces within model weights that are causally important for each domain and analyze their overlap through orthogonal disentanglement. Complementarily, we adapt causal tracing methods to controlled math and code factual prompts to localize when and where domain-specific information becomes causally relevant during inference. Our findings suggests that math reasoning operates as a foundational subspace upon which code reasoning partially builds, while code reasoning relies on additional domain-specific structures that do not contribute to mathematical performance.

1. Introduction

Large language models achieve strong performance on both mathematical reasoning tasks and programming tasks. They solve math word problems, generate proofs or calculations, and write executable code that performs multi step reasoning. Despite this surface level success, it is still unclear whether these capabilities arise from shared internal mechanisms or from distinct, specialized circuits inside the model.

Understanding this distinction matters for both scientific and practical reasons. From a scientific perspective, mechanistic interpretability aims to explain how learned representations and circuits inside neural networks give rise to specific behaviors. Prior work has shown that language models contain identifiable and sometimes highly specialized circuits for particular linguistic and reasoning functions, such as indirect object identification and in context learning induction patterns (Olsson et al., 2022), (Wang et al., 2023). Math reasoning and code reasoning are often grouped together as forms of logical or symbolic reasoning, but they differ in structure, supervision, and execution. Math problems are often solved through natural language chains of thought or symbolic manipulation, while code reasoning frequently relies on algorithmic structure, control flow, and sometimes execution feedback. Whether these forms of reasoning rely on overlapping or distinct internal components remains an open question.

From a practical perspective, this question is relevant for data selection and fine tuning. Modern models are frequently adapted to new domains using limited additional data. If math and code reasoning share internal circuits, then fine tuning on one domain may transfer effectively to the other. If they rely on distinct mechanisms, then mixing data across domains may lead to inefficient training or unintended interference. A clearer understanding of the correlation and overlap between math and code reasoning inside models can inform decisions about dataset composition, curriculum design, and targeted fine tuning for reasoning heavy applications.

Recent prompting methods further motivate this investigation. Program Aided Language Models and Program of Thought prompting show that models can improve math performance by generating and executing code to perform computations (Chen et al., 2022), (Gao et al., 2022). This suggests that models may switch between different internal solution modes when reasoning in math versus code, or even when solving the same math problem via text based chain of thought versus program execution. These observations raise the question of whether different internal pathways are being engaged, and whether those pathways are shared across domains.

In this paper, we study whether code reasoning and mathematical reasoning rely on distinct or shared internal mechanisms in large language models. We use tools from mechanistic interpretability, including activation patching and path patching, to localize causal components responsible for performance in each domain (Zhang & Nanda, 2024) (Goldowsky-Dill et al., 2023). We measure the degree of specialization and overlap of these components across tasks, models, and solution formats. We also perform cross domain patching experiments to test whether components identified for one domain causally support performance in the other. Our goal is to provide a quantitative and mechanistic characterization of the relationship between math and code reasoning inside language models.

2. Related Works

2.1. Mechanistic interpretability and causal analysis

Mechanistic interpretability seeks to explain model behavior by identifying circuits, composed of neurons, attention heads, and connections, that causally contribute to specific functions. Early work demonstrated that small transformer models contain interpretable attention heads and neurons that implement recognizable algorithms or features. More recent work has focused on causal interventions to move beyond correlation.

Activation patching is a core technique in this literature. It measures the causal importance of internal activations by replacing them with activations from a different input and observing the effect on model output (Zhang & Nanda, 2024). Path patching extends this idea by isolating specific computational paths through the network, allowing researchers to localize behavior to subgraphs rather than individual components (Goldowsky-Dill et al., 2023). These methods have been used to identify circuits for tasks such as indirect object identification in GPT 2 Small (Wang et al., 2023) and to study induction heads that support in context learning (Olsson et al., 2022).

Automated circuit discovery methods further scale these ideas by searching for minimal sets of components that explain a behavior (Conmy et al., 2023). Together, this line of work provides a toolkit for answering questions about whether different behaviors depend on shared or distinct internal mechanisms. Our work builds directly on these methods and applies them to the comparison between math and code reasoning.

2.2. Math reasoning in language models

Language models have been extensively evaluated on mathematical reasoning tasks, including grade school arithmetic, algebraic word problems, and competition style questions. (Cobbe et al., 2021b) introduced training verifiers to improve math problem solving, highlighting the importance of reasoning quality and solution verification rather than surface level accuracy. Subsequent work has shown that chain of thought prompting significantly improves math performance by encouraging models to produce intermediate reasoning steps.

These results demonstrate that models can represent and manipulate mathematical structure, but they do not explain how this reasoning is implemented internally. Mechanistic studies of math reasoning remain limited, especially compared to work on linguistic circuits. This gap motivates a deeper investigation into the internal components that support mathematical reasoning and how they relate to other forms of structured reasoning.

2.3. Code reasoning and program based prompting

Code generation and code reasoning have become central benchmarks for large language models. Models can write functions, reason about program behavior, and solve algorithmic problems. Importantly, several works have shown that generating and executing code can improve performance even on non programming tasks.

Program Aided Language Models delegate parts of the reasoning process to external code execution, improving numerical and symbolic reasoning accuracy (Gao et al., 2022). Program of Thought prompting further separates reasoning from computation by having models generate executable programs that carry out calculations (Chen et al., 2022). These approaches suggest that code like representations may play a special role in how models handle precise computation.

The success of these methods raises the possibility that models internally reuse code related circuits when solving math problems, especially when prompted to produce program like solutions. Alternatively, the improvement may arise from external execution rather than shared internal mechanisms. Our work aims to distinguish between these possibilities by directly intervening on internal activations.

2.4. Comparing reasoning domains

Several papers implicitly compare reasoning domains by evaluating transfer across tasks, but few study the internal mechanisms underlying such transfer. The interpretability literature has shown that some circuits are highly task specific, while others are reused across behaviors (Olsson et al., 2022), (Wang et al., 2023). However, most studies focus on linguistic or synthetic tasks rather than complex reasoning domains like math and code. Our work connects these lines of research by asking whether math reasoning and code reasoning share causal circuits inside large language models. By combining mechanistic interpretability methods with task comparisons and cross domain interventions, we aim to move beyond performance level comparisons and provide insight into how these reasoning abilities are implemented internally.

3. Methodology

3.1. ActSVD

To investigate whether mathematical and code reasoning rely on separable low-rank components within the model weights, we employ ActSVD (Activation-aware Singular Value Decomposition), a weight attribution method recently proposed by Wei et al. (2024). Unlike standard SVD, which decomposes weight matrices based solely on the magnitude of the weights, ActSVD identifies singular vectors that are functionally important for a specific distribution of data. This allows us to decompose a weight matrix into subspaces specialized for "Math" and "Code."

The ActSVD pipeline consists of three stages: activation collection, low-rank decomposition, and orthogonal disentanglement.

Activation Collection. Standard low-rank compression techniques often approximate a weight matrix W directly. However, a direction in W with large magnitude may not be relevant if the input activations along that direction are consistently zero during inference. ActSVD addresses this by considering the layer's output response to specific data.

We define two domain-specific calibration datasets: $\mathcal{D}_{math}$ (derived from GSM8K) to capture reasoning activation patterns, and $\mathcal{D}_{code}$ (derived from HumanEval) to capture syntactic and algorithmic activation patterns. For a given linear layer $W \in \mathbb{R}^{d_{out} \times d_{in}}$, we run the model on these datasets and collect the input activations $X_{in} \in \mathbb{R}^{d_{in} \times N}$, where N is the number of tokens. This yields two activation stacks: X_{math} and X_{code}.

ActSVD Decomposition. We seek a low-rank approximation $\hat{W}$ that minimizes the reconstruction error of the layer's *outputs* on the calibration data, rather than the weights themselves. The optimization objective is:

$$\hat{W} = \underset{\text{rank}(\hat{W}) \leq r}{\arg\min} \|WX_{in} - \hat{W}X_{in}\|_F^2 \tag{1}$$

Following Wei et al. (2024), the closed-form solution is obtained by performing SVD on the output activations $Z = WX_{in}$. Let $U\Sigma V^\top \approx Z$ be the rank-r SVD of the outputs. The optimal low-rank weight approximation is given by projecting the original weights onto the subspace spanned by the top left singular vectors U:

$$\hat{W} = \Pi W, \quad \text{where } \Pi = UU^\top \tag{2}$$

Here, $\Pi \in \mathbb{R}^{d_{out} \times d_{out}}$ is the orthogonal projection matrix onto the most significant output directions for the given task. By computing this separately for our datasets, we obtain Π_{math} (the subspace critical for math) and Π_{code} (the subspace critical for code).

Orthogonal Disentanglement. To determine if Math reasoning uses mechanisms distinct from Code reasoning, we cannot simply look at the intersection of Π_{math} and Π_{code}, as high-dimensional subspaces may overlap incidentally. Instead, we use orthogonal projection to isolate the "pure" Math subspace.

We construct a modification ΔW that retains the most critical Math ranks but strictly projects them onto the orthogonal complement of the Code subspace. The update is defined as:

$$\Delta W = (I - \Pi_{code})\Pi_{math}W \tag{3}$$

In this formulation, Π_{math} extracts the components necessary for Math, while $(I - \Pi_{code})$ removes any directions that are also significant for Code. If the model's Math performance persists when restricted to this orthogonal update (or conversely, if removing this update degrades Math but preserves Code), it provides evidence that the internal mechanisms for mathematical reasoning are distinct from those used for code generation.

3.2. Causal Tracing

We use causal tracing to localize where specific factual information is represented and causally used inside the model during inference. Our approach follows the intervention-based framework introduced by Meng et al. (2022), which measures how restoring internal activations from a clean forward pass can recover correct predictions in an otherwise corrupted run.

Clean and Corrupted Model States. Given an input prompt x, we construct two model executions. In the *clean run*, the original prompt is passed through the model, producing hidden activations $h_i^{(l)}$ at token position i and layer l. In the *corrupted run*, the same prompt is used, but the embeddings corresponding to the subject tokens are corrupted by injecting Gaussian noise. This corruption is designed to disrupt the model's ability to recall the correct answer while keeping the remainder of the prompt unchanged.

The effectiveness of the corruption is quantified by the *Total Effect (TE)*, defined as the drop in probability assigned to the correct answer token between the clean and corrupted runs.

Activation Patching and Indirect Effect. To localize where the relevant information is used, we perform targeted causal interventions. For a given layer l and token position i, we replace the hidden activation in the corrupted run with the corresponding activation from the clean run, while leaving all other activations unchanged. The model is then run forward with this patched activation.

We measure the *Indirect Effect (IE)* of patching at (l, i)

as the increase in probability assigned to the correct answer token relative to the corrupted baseline. Repeating this procedure across layers and token positions yields a two-dimensional map that identifies which internal states causally mediate recall of the target information.

Adaptation to Math and Code Facts. The original causal tracing framework is formulated for factual associations of the form (s, r, o), where the object o is a single next-token prediction. Directly applying this method to multi-step reasoning tasks (e.g., grade-school math problems or code generation benchmarks) is not appropriate, as such tasks do not admit a well-defined single-token target.

To adapt causal tracing to math and code domains while preserving the assumptions of the method, we construct math and code facts as structured (s, r, o) triplets:

- **Subject (s):** a short natural-language phrase identifying a mathematical or programming concept.
- **Relation (r):** a fixed relational phrase (e.g., “is” or “equals”).
- **Object (o):** a single-token answer that completes the factual association.

Each triplet is instantiated as a simple declarative prompt of the form *“The s r o.”*

This construction ensures that the correct answer is a single token, the subject tokens are causally necessary for predicting the object, and corruption of the subject reliably disrupts recall. We apply the same causal tracing procedure to both math facts and code facts, enabling a controlled comparison of how factual information from each domain is represented and retrieved within the model.

4. Experimental Setup

Models We conducted our experiments on the Qwen 2.5 1.5B open-weight transformer model. This model was chosen due to its strong performance on reasoning tasks, while remaining computationally feasible for the intensive demands of mechanistic interpretability techniques such as ActSVD and Causal Tracing.

Datasets To isolate reasoning behaviors, we use two domain-specific calibration datasets, as outlined in our methodology:

- **Math Dataset** ($\mathcal{D}_{math}$): We used a subset of GSM8K (Cobbe et al. (2021a)) problems requiring multi-step arithmetic and logical reasoning to capture activation patterns specific to mathematical reasoning.
- **Code Dataset** ($\mathcal{D}_{code}$): We used problems from HumanEval (Chen et al. (2021)) that emphasize program synthesis, control flow, and symbolic manipulation to capture syntactic and logical activation patterns specific to code generation.

For each dataset, we collect activations from intermediate layers by running the model on a subset of prompts. These activations are stacked across samples to form the input activation matrix X_{in} used for ActSVD decomposition.

Causal Tracing Dataset: We create two small, controlled datasets of math and code facts, tailored for causal inference. Each example is written as an (s, r, o) triplet, where the subject names a math or programming concept, the relation is a short fixed phrase, and the object is a single-token answer. Prompts are constructed to end immediately before the object token so that predicting the next token corresponds directly to recalling the fact. We choose facts that do not require multi-step reasoning or symbolic computation and remove examples where the model does not reliably predict the correct object or where subject corruption has little effect.

Evaluation We evaluate reasoning performance using accuracy on GSM8K for mathematical reasoning and Pass@1 on HumanEval for code reasoning.
To analyze mechanistic overlap, we additionally measure:

- Performance degradation as a function of rank removal.
- Cross-domain effects when ranks important to one domain are removed.

For causal tracing experiments, we quantify the causal contribution of specific layers and token positions using the Indirect Effect (IE) metric.

4.1. ActSVD Decomposition and Rank Attribution

We apply ActSVD to identify low-rank subspaces in each transformer layer that are most critical for preserving task-specific outputs. For each domain (math and code), we compute a task-specific projection matrix Π_{task} from the top-r singular vectors. Task-critical ranks are determined by iteratively removing individual ranks and evaluating the corresponding degradation in task performance. The resulting directions are then ranked according to their causal contribution to task accuracy. This procedure produces two ordered sets of directions: math-critical ranks and code-critical ranks.

To test whether math and code reasoning rely on shared or distinct mechanisms, we conduct orthogonal disentanglement by projecting the math-critical subspace onto the orthogonal complement of the code-critical subspace. This ensures that the resulting update isolates math-specific directions while explicitly removing any overlap with code reasoning directions. We then perform ablation studies by systematically removing ranks identified as critical for either math or code and re-evaluating model performance on GSM8K and HumanEval to measure cross-domain effects. Ablations are conducted over removal thresholds ranging from $r = 1$ to $r = 20$.

5. Results

5.1. Rank Removal and Performance Sensitivity

Dropping Code-Critical Ranks

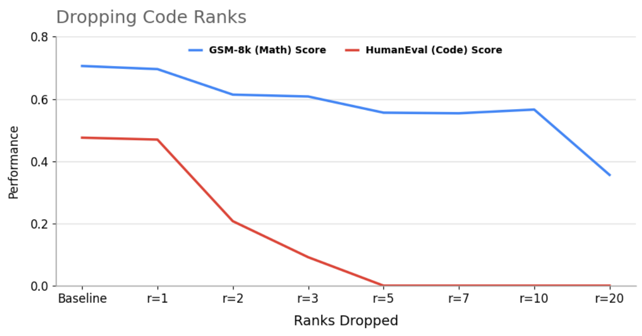

Figure 1. Performance on GSM8K and HumanEval as code-critical ranks are progressively removed. Code performance degrades rapidly, while math performance remains stable, indicating minimal reliance of math reasoning on code-specific subspaces.

When progressively removing ranks most important for code reasoning:

- **Impact on Code Performance:** As expected, removing code-critical ranks caused a degradation in performance on the HumanEval benchmark. The score dropped sharply from a baseline of ≈ 0.5 to near 0.0 when approximately 5 to 6 ranks were removed, confirming that these ranks are causally necessary for code generation.
- **Impact on Math Performance:** In contrast, the removal of these same code ranks had a negligible effect on mathematical reasoning. The GSM8K score remained stable at the baseline of ≈ 0.7, maintaining high performance even as Code performance collapsed. A decline was only observed at extreme ablation levels ($r \geq 10$).

This suggests that math reasoning does not strongly depend on code-specific subspaces.

Dropping Math-Critical Ranks

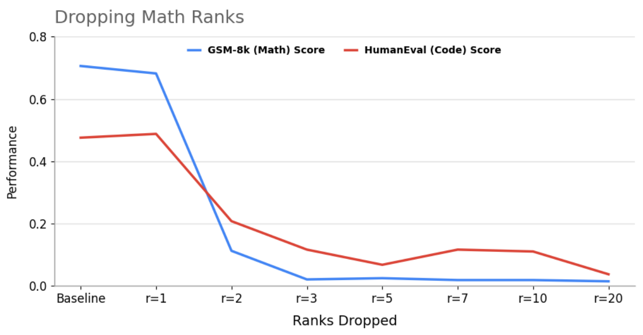

Figure 2. Performance on GSM8K and HumanEval as math-critical ranks are removed. Both math and code performance degrade significantly, demonstrating that code reasoning partially depends on math-critical directions.

In contrast, removing math-critical ranks leads to:

- **Impact on Math Performance:** Removing math-critical ranks resulted in a steady decline in GSM8K scores, dropping from the baseline to below 0.4 as more ranks were removed.
- **Impact on Code Performance:** Removing math ranks had a severe negative impact on Code performance, with HumanEval scores exhibiting a substantial decline despite no code-specific ranks are removed.

This indicates that code reasoning partially depends on math reasoning subspaces, but not vice versa.

5.2. Sparsity of Reasoning Directions

A key observation is that math reasoning depends on a small and sparse set of critical directions. Removing the top-ranked math directions leads to a sharp decrease in performance, whereas removing additional ranks beyond this set yields diminishing marginal effects. In contrast, code reasoning exhibits a more distributed reliance across ranks, consistent with the need to represent a wide range of syntactic and semantic structures.

5.3. Causal Tracing Results

For math facts, we observe that early-relation tokens begin contributing non-trivially in the middle layers of the network. The Indirect Effect for math rises steadily starting around the first third of layers, indicating that math relations are integrated into the model's reasoning pipeline relatively early. This suggests that numerical relationships and symbolic constraints are actively transformed and propagated through intermediate representations well before the final prediction layers.

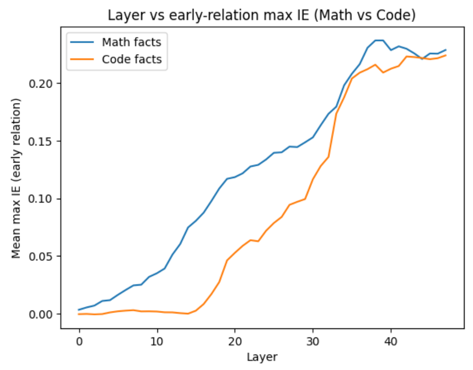

Figure 3. This figure compares where, across layers, information from the relation part of the prompt becomes causally important for predicting the correct answer. For math facts, early-relation tokens start contributing strongly in the middle layers, while for code facts this contribution is weaker at first and only rises substantially in later layers. By the final layers, both curves converge, indicating that math and code ultimately rely on similar high-level representations near the output. Overall, this suggests that math facts are integrated into the model's reasoning pipeline earlier, whereas code facts depend more on later-stage processing before influencing the final prediction.

In contrast, code-related facts exhibit a markedly different trajectory. The Indirect Effect for code remains near zero across early and mid layers, only increasing substantially in later layers. This implies that information about program structure, variable bindings, or control flow is not immediately causally relevant, but instead is deferred until higher-level representations closer to the output. Such behavior is consistent with code reasoning relying more heavily on late-stage synthesis and token-level decision-making rather than early relational abstraction.

In the final layers, the curves for math and code converge, with both domains exhibiting similarly high Indirect Effect values. This convergence suggests that although math and code facts are processed differently early in the network, they ultimately rely on shared high-level representations near the output layer to generate the final answer.

5.4. Processing Depth

The causal tracing results add a temporal dimension to our ActSVD findings and help explain the asymmetric dependency between math and code reasoning. While ActSVD revealed that code performance depends on math-critical ranks, causal tracing clarifies why this dependency arises by showing when each type of information becomes causally active during forward computation.

Specifically, math facts begin to exert a significant causal influence relatively early in the network, whereas code-related facts only become causally important much later. The the model first establishes a logical or math scaffold—encoding quantities, relations, and constraints—before applying domain-specific syntactic and structural transformations required for code generation. In this view, math reasoning serves as a foundational computation upon which more specialized reasoning is built.

This temporal ordering also explains the asymmetry observed in our ablation experiments. Because code reasoning is deferred to later layers, it likely operates on latent representations produced by earlier math- and logic-oriented layers. When math-critical ranks are removed, these early precursor representations are disrupted, preventing downstream code-specific circuits from functioning effectively and leading to a degradation in code performance.

6. Discussion and Future Work

The primary objective of this project was to determine whether LLMs use shared or distinct internal mechanisms for code versus math reasoning. Our findings provide strong evidence that math and code reasoning are neither fully symmetric nor independent in large language models. Although the two domains exhibit partial overlap, their dependency structure is asymmetric. Math reasoning constitutes a foundational subspace upon which code reasoning partially builds, while code reasoning introduces additional domain-specific structure that does not substantially contribute back to math reasoning. This asymmetry explains the empirical observation that models with poor math reasoning also tend to perform poorly on code tasks, whereas the converse does not hold.

The extreme sensitivity to a small number of math-critical ranks highlights a potential brittleness in math reasoning mechanisms. This aligns with prior work showing that reasoning behaviors can depend on narrow internal circuits. This has implications for robustness, as small perturbations or fine-tuning updates may significantly impair reasoning, while targeted interventions could efficiently enhance math performance.

6.1. Limitations

Our analysis is limited to a single model size and architecture, and the observed effects may not generalize to larger models. The degree of reasoning specialization and sparsity may vary with model scale, architectural choices, or training data composition. Additionally, our experiments rely on a single primary benchmark per domain—GSM8K for math reasoning and HumanEval for code reasoning. Although these benchmarks are widely adopted and broadly representative, they capture only a limited subset of each reasoning domain.

6.2. Future Work

A natural extension of this work is to examine how the identified math- and code-critical subspaces evolve across model scale, architecture, and training objective. Comparing base models to instruction-tuned or reinforcement learning-aligned variants could also reveal whether alignment procedures strengthen, redistribute, or entangle reasoning subspaces.

7. Conclusion

In this study, we utilized mechanistic interpretability techniques, specifically ActSVD and Causal Tracing, to investigate the internal relationship between mathematical and code reasoning within large language models. Our results demonstrate that these two domains are neither identical nor fully independent; rather, they exhibit a distinct asymmetric dependency.

Through experiments, we show that code reasoning degrades significantly when math-critical ranks are removed, whereas mathematical reasoning remains largely robust to the ablation of code-critical ranks. This asymmetry suggests that code generation depends on a subspace of mathematical reasoning, but not vice versa. Causal tracing further clarifies this relationship by revealing a temporal hierarchy: mathematical concepts are processed in earlier to middle layers, providing a logical scaffold upon which code reasoning is constructed in later layers.

Overall, these findings suggest that mathematical reasoning functions as a foundational capability for the more complex, structurally specific task of code generation.

8. Additional Info

Each member contributed roughly equally towards this project.

References

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

Chen, W. et al. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021a.

Cobbe, K. et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021b.

Conmy, A., Mavor-Parker, A. N., et al. Towards automated circuit discovery for mechanistic interpretability. In *NeurIPS*, 2023.

Gao, L. et al. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

Goldowsky-Dill, N., MacLeod, C., Sato, L., and Arora, A. Localizing model behavior with path patching. *arXiv preprint arXiv:2304.05969*, 2023.

Meng, K., Bau, D., Andonian, A., and Belinkov, Y. Locating and editing factual associations in gpt. *Advances in neural information processing systems*, 35:17359–17372, 2022.

Olsson, C. et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Wang, K. R. et al. Interpretability in the wild: A circuit for indirect object identification in gpt-2 small. In *ICLR*, 2023.

Wei, B., Huang, K., Huang, Y., Xie, T., Qi, X., Xia, M., Mittal, P., Wang, M., and Henderson, P. Assessing the brittleness of safety alignment via pruning and low-rank modifications. *arXiv preprint arXiv:2402.05162*, 2024.

Zhang, F. and Nanda, N. Towards best practices of activation patching in language models: Metrics and methods. In *International Conference on Learning Representations (ICLR)*, 2024. URL `https://openreview.net/forum?id=Hf17y6u9BC`.