Model Mapping: Decompiling Transformers into Python Programs

Dhruv Pai [1] **Benjamin Keigwin** [2] **Mason Wang** [1] **Shriyash Upadhyay** [2]

Abstract

Mechanistic interpretability aims to understand the internal mechanisms of transformers, ranging from circuit and single-neuron level detection methods to fully characterizing behavior on downstream data distributions. However, current methods fall short of providing a concise, macroscopic description of inductive biases and facts stored in transformer weights, even in small models on simple tasks. We train an end-to-end model to decompile transformers trained on outputs of RASP Seq2Seq programs into the abstract syntax tree (AST) for their training program. We extend our method to procedurally generated, complex Python Seq2Seq functions and demonstrate our model mapping method can generalize to programs/transformers not observed during train time. We demonstrate the feasibility of model mapping for transformer decomposition into human-interpretable, concise, and comprehensive programs on simple tasks.

1. Introduction

Transformer models have become the cornerstone of many NLP tasks, achieving state-of-the-art results across various benchmarks (Vaswani et al., 2017; Devlin et al., 2019; Radford et al., 2019). Despite their success, the internal operations of these models remain largely opaque, raising questions about how they accomplish these tasks.

With significant effort and insight, researchers have begun to understand how these models work by reverse-engineering their internal operations, a field known as mechanistic interpretability (Olah et al., 2020; Voss et al., 2021; Olah, 2022). This has led to discoveries of parts inside large models which are responsible for specific behaviors (Meng et al., 2022; Geva et al., 2023; Heimersheim & Janiak, 2023; Tigges et al., 2023; Hanna et al., 2023), as well as many interesting results on small models that help us better understand how these models operate (Olsson et al., 2022; Chughtai et al., 2023; Li et al., 2022; Gromov, 2023; Nanda et al., 2023).

However, the significant manual effort and insight required for progress in this field is a bug, not a feature. One of the major lessons in over the last decade of research has been the bitter lesson (Sutton, 2019) – the idea that techniques which can scale with compute (as opposed to with manual effort or insight) have been those which are most successful. Much of the success of transformers in fields like NLP come from the ability to scalably train such models (Vaswani et al., 2017; Radford et al., 2019; Brown et al., 2020). Although it might be possible to create successful language models using more traditional, manual methods from NLP, the approaches which got there most quickly were those which scaled with data and compute. If we aim to create techniques for understanding models which can advance as fast as models do, pursuing more scalable approaches appears necessary.

In this paper, we propose *model mapping*, a family of techniques for understanding models which are designed to scale with data and compute instead of human intervention.

Our contributions are as follows:

- We establish the use of model mapping for model interpretability, motivated by insights from category theory.
- We propose a family of models which decompile transformers into human-interpretable programs.
- We demonstrate the efficacy of model mapping for interpretability on simple sequence-to-sequence tasks.
- We provide preliminary insights on the scaling behavior of model mappings and the promises they may hold for model oversight at-scale.

2. Background

2.1. Circuit Analysis

Mechanistic interpretability attempts to explain the behavior of neural networks by understanding the algorithms they

[1]Department of Computer Science, Stanford University [2]Martian. Correspondence to: Dhruv Pai <dhruvpai@stanford.edu>, Benjamin Keigwin <ben@withmartian.com>, Shriyash Upadhyay <yash@withmartian.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

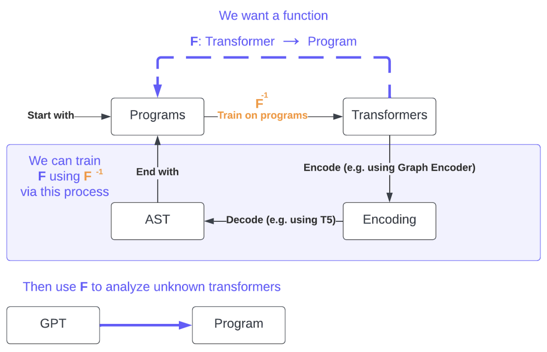

Figure 1. Model mapping is a method which can be used to analyze transformers without human-manual dissection of the weights. As a result, these methods can scale with data and compute instead of human effort. Above is an example of a model mapping technique which can turn transformers into programs.

learn. Such an understanding is typically achieved by studying "circuits" inside of these models: configurations of neurons and their connections that collectively perform a particular function (Olah et al., 2020).

Circuit studies are conducted by analyzing the activations within a network (Olah, 2022). Activations serve as memory states for the internal computation to be extracted from the models (Voss et al., 2021). By determining the information stored in specific locations, it becomes possible to identify the variables needed to construct a computation (program). Consequently, these methods prioritize localization (Hase et al., 2023), which involves pinpointing the specific memory locations where concepts are stored and understanding their interactions to accomplish tasks. This process may involve probing the model to detect activations that could be responsible for certain behaviors (Belinkov, 2021). Subsequently, counterfactual analyses, such as causal mediation analysis (Pearl, 2001; 2009), are employed to verify whether the identified activations genuinely explain the behavior. This type of counterfactual analysis is often referred to as activation patching (Zhang & Nanda, 2023), and it is also known as causal tracing, interchange intervention, or representation denoising.

Such techniques have been used to identify model weights that store and process factual information (Meng et al., 2022; Geva et al., 2023) or identify sub-networks responsible for specific behaviors (Wang et al., 2022; Geva et al., 2023; Lieberum et al., 2023). Deriving such information from mechanistic interpretability techniques, prior works have rectified model errors (Vig et al., 2020; Hernandez et al., 2022; Hase et al., 2023), steered model outputs (Li et al., 2023), and explained behaviors in training (Barak et al., 2022; Gromov, 2023; Nanda et al., 2023).

Because such techniques require expert human effort and intuition, they are difficult to scale to large models. As a result, most works tend to provide strong results on toy models (Olsson et al., 2022; Chughtai et al., 2023), such as finding a linear representation of board state in a model trained on sequences of moves in Othello (Li et al., 2022) and explaining grokking in a model trained to do modular arithmetic (Gromov, 2023; Nanda et al., 2023); or mixed results on strong models (Heimersheim & Janiak, 2023; Tigges et al., 2023), such as identifying simple circuits in GPT-2 for the greater than operation (Hanna et al., 2023) and indirect object identification (Wang et al., 2022).

In order to scale such methods, attempts have been made to automate the process of circuit analysis (Chan et al., 2022; Conmy et al., 2023; Bills et al., 2023; Syed et al., 2023). These methods have produced exciting results, such as identifying circuits that have been previous identified by manual effort. However, there is a lack of mechanistic interpretability methods that cheaply and correctly identify many of the circuits responsible for model behavior. If more scalable approaches to holistic circuit identification, e.g. a circuit assay, were developed, they would enable stronger interpretability and model oversight.

2.2. Category Theoretic Motivation

Our motivation for constructing transformer-to-program mappings is inspired by category theory, a branch of mathematics with two guiding principles:

1. It is more effective to study an object through its relationships with other objects.
2. If a question is difficult to solve in one domain, try translating it (faithfully) to another domain where it might be easier to address.

The second principle is particularly powerful in fields like algebraic geometry and algebraic topology, where challenging geometric or topological problems are often translated into simpler linear algebra problems. In our work, we apply a similar strategy: interpreting neural networks and transformers is complex, while interpreting programs is comparatively easier.

Though the translation process may lose some information, it can still be highly beneficial. For instance, in algebraic topology, singular (co)homology functors are extremely valuable for studying topological spaces, even though one cannot generally reconstruct a space from its (co)homology groups. Similarly, even if the resulting program does not exactly mirror the transformer's mechanisms, as long as there is sufficient similarity in key metrics (such as I/O behavior or quality preservation) the translation is beneficial.

Furthermore, there are additional theoretical reasons to choose a collection of programs as the codomain for this translation mapping. Specifically, one can prove statements about programs using the programs themselves. This behavior can be abstractly characterized by stating that types in a programming language correspond to propositions in an analogous logical system, and programs correspond to proofs in that logical system. Categories with this property, known as Cartesian closed categories, are ideal candidates for mapping neural networks into.

2.3. Model Mapping

Model maps can be constructed in various ways. One approach involves learning a mapping from a parallel corpus. Other methods, familiar to machine learning practitioners, also exist. For example, knowledge distillation (Hinton et al., 2015), a process mapping a larger model (the teacher) to a smaller model (the student), i.e., a function $F : \mathcal{T} \to \mathcal{T}$ is another such example. Model mapping can also be entirely procedural, as seen in (Friedman et al., 2023), which describes a constrained transformer architecture compilable into programs.

3. Methodology

3.1. RASP Data

To train a model which converts transformers into programs, we first construct a parallel corpus of programs and transformers. We do this by procedurally generating programs in the RASP language (Weiss et al., 2021), a human understandable programming language designed such that programs in this language can be compiled into sparse transformers. The RASP language defines a set of operations which emulate multi-head attention and the MLP operations found in transformers, and can be directly compiled into sparse transformer weights as a result. We compile the programs into transformers using a fork of the tracr compiler (Lindner et al., 2023). We describe the RASP language and the procedure we use to generate such programs in more detail in appendix A.

Generated programs combined sequence-to-sequence operations in a procedurally generated manner. We generated our dataset of $(\mathcal{T}, \mathcal{P})$ transformer-program pairs by training a bidirectional GPT model on each RASP program until convergence, whereby the evaluation accuracy exceeded 0.95. The architectural choices for the trained model are described in section 3.3. Models that satisfied the evaluation performance threshold at each epoch were added as data points.

To improve dataset quality, models which converged to the optimal loss basin for a program were sampled repeatedly during training to provide different transformer ”views” of the same programs, as a form of data augmentation. Programs were also screened for quality as valid list-to-list operators with bounded output behavior.

The advantage of using transformers trained on RASP I/O pairs is that RASP provides a useful model for sequence operations learnable by a transformer, including inductive biases and synthetics. However, the class of RASP programs is a limited subset of the hypothesis class for transformers. Therefore, we sought to test our approach on increasingly complex programs, which motivates the subsequent section.

3.2. Python Data

We follow a similar process to the above RASP program generation process to generate the Python programs we use. Each program has between 1 and 3 outer for loops, and within each for loop, there is optionally an additional nested for loop, conditional operation, and/or an assignment. We give an example of a typical program generated by this process in appendix B. The goal in using Python in this decompilation process is two-fold:

1. It is easier to write human-interpretable programs in Python than RASP, so we can better gauge what types of programs the model mapping is learning (or potentially failing to learn).

2. To demonstrate that the model mapping approach is not specific to RASP, and also applies to programs constructed in other languages.

3.3. Model Architecture & Learning

We discuss our model architecture for learning a transformer-to-program mapping. The same model architecture is used for mapping to both Python programs and RASP programs. Our approach utilizes an encoder-decoder architecture, detailed as follows:

1. **Input Formatting**: We begin by representing the input transformer T as a graph G_T. This is achieved by taking the n-layer transformer T and formatting it as a graph with n vertices. An edge is drawn from the i-th vertex to the $(i + 1)$-st vertex for $1 \leq i \leq n - 1$, representing the flow of information in transformer T. The parameter vector for the i-th vertex of G_T comprises the concatenated parameters of the i-th layer of transformer T.

2. **Graph Encoding**: The graph G_T from step 1 is fed into our graph encoder. The graph encoder consists of four layers: a fully-connected layer, a graph convolutional layer, a second fully-connected layer, and another graph convolutional layer. The output of this encoder is n vectors, each with a dimension equal to the embedding dimension of our decoder, where n is the number of vertices in the graph.

3. **Decoder Implementation**: The decoder employs a custom T5 architecture with three layers. Each layer features an intermediate feed-forward dimension of 32 units and utilizes 8 attention heads.

4. **Decoding Process**: The encoder state is provided to the decoder model to sample an abstract syntax tree (AST) autoregressively. The entire setup is trained end-to-end using a causal language modeling loss against the ground-truth AST for RASP, or against the ground-truth program for Python.

To test how well our encoder-decoder scales with larger inputs, we use an encoder-decoder of consistent size regardless of the size of input graphs and output programs/ASTs.

4. Experiments and Results

4.1. Reconstructed RASP programs

We assess the accuracy of the recovered RASP program by measuring the percentage of matching tokens between the predicted and target ASTs. This evaluation is performed on held-out model weights and program pairs using an 80/20 split. Tokens are derived from a custom dictionary representing the operations within the RASP program's abstract syntax tree. For a token to be considered correct, it must occupy the identical position in both the predicted and target linearized ASTs. Our findings show an average accuracy of 82.25%, calculated across all trained transformer sizes. Notably, we achieve over 80% accuracy for each model size sampled, ranging from 10K to 10M parameters, indicating effective scalability of the model mapping framework despite increasing over-parameterization.

4.2. Reconstructed Python Programs

We observe significantly better recovery for Python programs compared to RASP programs. This discrepancy can be attributed to two primary factors:

1. Tokenization Scheme: The custom tokenization for RASP is more penalizing. Although RASP and Python programs have a similar average number of lines, the linearized RASP ASTs are approximately four times shorter than the tokenized Python programs, which were tokenized using the GPT-2 tokenizer.

2. Order Independence of RASP Programs: RASP programs are likely more order-independent than Python programs, meaning that different sequences of the same lines of code can produce identical input-output behavior. Since the model is trained only on I/O pairs, it may struggle to predict the exact sequence of lines, resulting in decreased accuracy.

We employed the same 80/20 split for Python programs as for RASP, training models with varying hyperparameter configurations, resulting in models ranging from 10K to 10M parameters. The average accuracy of successfully recovered tokens for Python programs is 95.76%, calculated across all trained transformer sizes. An example comparison of a predicted Python program versus the target program is provided in Appendix. B.

5. Conclusion

In this paper, we introduced model mapping, a framework for mechanistic interpretability that converts neural networks into programs. This method enables the study of models via automatic conversion to programs, rather than by manual inspection of their internals, making it scalable for understanding neural networks.

We demonstrated the effectiveness of model mapping by converting thousands of transformers, ranging from 10K to 10M parameters, into programs. To apply these methods to state of the art models, they must generalize along two axes: size and distribution. First, models must scale to handle sizes up to trillions of parameters. While this work provides preliminary results on scaling, there is likely to be significant engineering work in this direction. Secondly, transformers analyzed in this work were generated procedurally, and are thus likely to have distributional differences from models trained on tasks like language modeling. However, if models can be made to generalize along these two axes, it seems plausible that model mapping methods could be used to convert very large models into code.

Furthermore, interpreting models as code requires better tools for understanding and refactoring extensive programs, which remains a challenging yet more tractable problem than direct model interpretation.

We hope this work facilitates exploration of the algorithms within models and that model mapping methods can be scaled more effectively than traditional approaches, leading to deeper understanding and practical applications of mechanistic interpretability.

Impact Statement

Mechanistic interpretability provides a means of mitigating the harms of AI and building safer AI systems. Systems that we do not understand may have implicit biases or other unfair or undesirable properties. These system may also be less robust and fail in unexpected cases. By understanding these models, we can then identify such issues and create tools to rectify them.

References

Ba, J., Kiros, J. R., and Hinton, G. E. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

Barak, B., Edelman, B. L., Goel, S., Kakade, S. M., Malach, E., and Zhang, C. Hidden progress in deep learning: Sgd learns parities near the computational limit. *ArXiv*, abs/2207.08799, 2022. URL https://api.semanticscholar.org/CorpusID:250627142.

Belinkov, Y. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48:207–219, 2021. URL https://api.semanticscholar.org/CorpusID:236924832.

Bills, S., Cammarata, N., Mossing, D., Tillman, H., Gao, L., Goh, G., Sutskever, I., Leike, J., Wu, J., and Saunders, W. Language models can explain neurons in language models, May 2023. URL https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T. J., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020. URL https://api.semanticscholar.org/CorpusID:218971783.

Chan, L., Garriga-Alonso, A., Goldowsky-Dill, N., Greenblatt, R., Nitishinskaya, J., Radhakrishnan, A., Shlegeris, B., and Thomas, N. Causal scrubbing: A method for rigorously testing interpretability hypotheses, 2022. URL https://www.alignmentforum.org/posts/JvZhhzycHu2Yd57RN/causal-scrubbing-a-method-for-rigorously-testing.

Chughtai, B., Chan, L., and Nanda, N. A toy model of universality: Reverse engineering how networks learn group operations. *ArXiv*, abs/2302.03025, 2023. URL https://api.semanticscholar.org/CorpusID:256615287.

Conmy, A., Mavor-Parker, A. N., Lynch, A., Heimersheim, S., and Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *ArXiv*, abs/2304.14997, 2023. URL https://api.semanticscholar.org/CorpusID:258418244.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *ArXiv*, abs/1810.04805, 2019.

Dong, Y., Cordonnier, J.-B., and Loukas, A. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. *ArXiv*, abs/2103.03404, 2021.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., A. Chen, T. Conerly, N. D. D. D. D. G. Z. H.-D. D. H. A. J. J. K. L. L. K. N. D. A. T. B. J. C. J. K. S. M., and Olah., C. A mathematical framework for transformer circuits, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Elhage, N., Hume, T., Olsson, C., Schiefer, N., Henigan, T., Kravec, S., Hatfield-Dodds, Z., Lasenby, R., Drain, D., Chen, C., and et al., Sep 2022. URL https://transformer-circuits.pub/2022/toy_model/index.html.

Friedman, D., Wettig, A., and Chen, D. Learning transformer programs. *ArXiv*, abs/2306.01128, 2023. URL https://api.semanticscholar.org/CorpusID:259064324.

Geva, M., Bastings, J., Filippova, K., and Globerson, A. Dissecting recall of factual associations in auto-regressive language models. *ArXiv*, abs/2304.14767, 2023. URL https://api.semanticscholar.org/CorpusID:258417932.

Gromov, A. Grokking modular arithmetic. *ArXiv*, abs/2301.02679, 2023. URL https://api.semanticscholar.org/CorpusID:255546130.

Hanna, M., Liu, O., and Variengien, A. How does gpt-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model. *ArXiv*, abs/2305.00586, 2023. URL https://api.semanticscholar.org/CorpusID:258426987.

Hao, Y., Angluin, D., and Frank, R. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810, 2022.

Hase, P., Bansal, M., Kim, B., and Ghandeharioun, A. Does localization inform editing? surprising differences in causality-based localization vs. knowledge editing in language models. *ArXiv*, abs/2301.04213, 2023. URL https://api.semanticscholar.org/CorpusID:255595518.

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

Hernandez, E., Schwettmann, S., Bau, D., Bagashvili, T., Torralba, A., and Andreas, J. Natural language descriptions of deep visual features. *ArXiv*, abs/2201.11114, 2022. URL https://api.semanticscholar.org/CorpusID:246285344.

Hinton, G. E., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015. URL https://api.semanticscholar.org/CorpusID:7200347.

Hornik, K., Stinchcombe, M. B., and White, H. L. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, K., Hopkins, A. K., Bau, D., Vi'egas, F., Pfister, H., and Wattenberg, M. Emergent world representations: Exploring a sequence model trained on a synthetic task. *ArXiv*, abs/2210.13382, 2022. URL https://api.semanticscholar.org/CorpusID:253098566.

Li, K., Patel, O., Vi'egas, F., Pfister, H.-R., and Wattenberg, M. Inference-time intervention: Eliciting truthful answers from a language model. *ArXiv*, abs/2306.03341, 2023. URL https://api.semanticscholar.org/CorpusID:259088877.

Lieberum, T., Rahtz, M., Kram'ar, J., Irving, G., Shah, R., and Mikulik, V. Does circuit analysis interpretability scale? evidence from multiple choice capabilities in chinchilla. *ArXiv*, abs/2307.09458, 2023. URL https://api.semanticscholar.org/CorpusID:259950939.

Lindner, D., Kram'ar, J., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *ArXiv*, abs/2301.05062, 2023.

Loshchilov, I. and Hutter, F. Fixing weight decay regularization in adam. *ArXiv*, abs/1711.05101, 2017.

Lu, Y., Li, Z., He, D., Sun, Z., Dong, B., Qin, T., Wang, L., and Liu, T.-Y. Understanding and improving transformer from a multi-particle dynamic system point of view. *ArXiv*, abs/1906.02762, 2019.

Meng, K., Bau, D., Andonian, A., and Belinkov, Y. Locating and editing factual associations in gpt. In *Neural Information Processing Systems*, 2022. URL https://api.semanticscholar.org/CorpusID:255825985.

Merrill, W. C. and Sabharwal, A. Log-precision transformers are constant-depth uniform threshold circuits. *ArXiv*, abs/2207.00729, 2022.

Mnih, V., Heess, N. M. O., Graves, A., and Kavukcuoglu, K. Recurrent models of visual attention. In *NIPS*, 2014.

Nanda, N., Chan, L., Lieberum, T., Smith, J., and Steinhardt, J. Progress measures for grokking via mechanistic interpretability. *ArXiv*, abs/2301.05217, 2023. URL https://api.semanticscholar.org/CorpusID:255749430.

Olah, C., 2022. URL https://transformer-circuits.pub/2022/mech-interp-essay/index.html.

Olah, C., Cammarata, N., Schubert, L., Goh, G., Petrov, M., and Carter, S. Zoom in: An introduction to circuits. 2020. URL https://api.semanticscholar.org/CorpusID:215930358.

Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T. J., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Johnston, S., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T. B., Clark, J., Kaplan, J., McCandlish, S., and Olah, C. In-context learning and induction heads. *ArXiv*, abs/2209.11895, 2022. URL https://api.semanticscholar.org/CorpusID:252532078.

Pearl, J. Direct and indirect effects. *Probabilistic and Causal Inference*, 2001. URL https://api.semanticscholar.org/CorpusID:5947965.

Pearl, J. Causality : Models , reasoning , and inference. 2009. URL https://api.semanticscholar.org/CorpusID:12575481.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Sutton, R. S., Mar 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Syed, A., Rager, C., and Conmy, A. Attribution patching outperforms automated circuit discovery. *ArXiv*, abs/2310.10348, 2023. URL https://api.semanticscholar.org/CorpusID:264147090.

Tigges, C., Hollinsworth, O. J., Geiger, A., and Nanda, N. Linear representations of sentiment in large language models. *ArXiv*, abs/2310.15154, 2023. URL https://api.semanticscholar.org/CorpusID:264591569.

Vaswani, A., Shazeer, N. M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NIPS*, 2017.

Vig, J., Gehrmann, S., Belinkov, Y., Qian, S., Nevo, D., Singer, Y., and Shieber, S. M. Investigating gender bias in language models using causal mediation analysis. In *Neural Information Processing Systems*, 2020. URL https://api.semanticscholar.org/CorpusID:227275068.

Voss, C., Cammarata, N., Goh, G., Petrov, M., Schubert, L., Egan, B., Lim, S., and Olah, C. Visualizing weights. *Distill*, 2021. URL https://api.semanticscholar.org/CorpusID:240920102.

Wang, K., Variengien, A., Conmy, A., Shlegeris, B., and Steinhardt, J. Interpretability in the wild: a circuit for indirect object identification in gpt-2 small. *ArXiv*, abs/2211.00593, 2022. URL https://api.semanticscholar.org/CorpusID:253244237.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. *ArXiv*, abs/2106.06981, 2021.

Xu, J., Sun, X., Zhang, Z., Zhao, G., and Lin, J. Understanding and improving layer normalization. *ArXiv*, abs/1911.07013, 2019.

Zhang, F. and Nanda, N. Towards best practices of activation patching in language models: Metrics and methods. *ArXiv*, abs/2309.16042, 2023. URL https://api.semanticscholar.org/CorpusID:263131114.

A. RASP appendix

A.1. The RASP Language

Recall that a transformer consists of two primary sub-components: multi-headed attention (MHA) and multi-layer perceptrons (MLP), each with residual connections. We can define multi-headed attention (Vaswani et al., 2017) in the form presented by (Elhage et al., 2021):

$$A^i = \text{softmax}(xW^i_{QK}x^T) \qquad \text{MHA}(x) = \sum_{i=1}^{H} A^i x W^i_{OV} \tag{1}$$

where x is the input sequence, W^i_{QK} and W^i_{OV} are the learnable parameters of the ith attention head. Note here that MHA does its computation in two steps: first, when computing attention, it compares each value in x against each other value in x and produces a probability distribution for how much each token in the sequence should be attended to, then it takes values from the W_{OV} matrices in proportion to the degree that they are attended to and sums them.

RASP (Weiss et al., 2021) is a human-readable programming language which aims to approximate the computations done by a transformer. It does so through *select-aggregate* operations. `select` takes in two sequences (key, query) and a comparison operator ($==, <, >, \leq, \geq, \neq$), and returns a list of positions that each token should attend to. This is analogous to the first part of the attention operation. `aggregate` takes in a `select` and a sequence of tokens, then for each token in the query of the `select`, averages the values of the tokens which were attended, producing a new sequence. This is analogous to the second part of the attention operation. The sequences which are used in the `select` operation are called sequence operations (s-ops). They can be constructed either from the tokens passed into a model (using the keyword `tokens`) or the indices of the token, representing the positional encodings usually used in transformers (using the keyword `indices`).

RASP also has a few other features to handle the kinds of computations found in a transformer. First, it allows simple elementwise operations, such as multiplying all the elements in a sequence by 3; these provide an analog to the MLP in a transformer, which can compute almost any such function accurately (Hornik et al., 1989). Second, it allows sequencing of operations – any sequence, s-op, or selector can be reused in a subsequent operation, mimicking the ability of residual connections to pass information from previous layers forward to new layers. Finally, for the convenience of the human programmer, functions can be defined which take in s-ops, sequences, selectors, or elements and define an operation. (For examples and descriptions of RASP programs, see (Weiss et al., 2021), (Lindner et al., 2023), or section 3.2 of this paper).

These "analogies" provided by RASP are not merely analogies. A compiler which turns RASP programs into transformers is a constructive proof that any RASP program can be losslessly represented by a transformer, and such a compiler was created in (Lindner et al., 2023).

A.2. Limitations of the RASP Language

This is not to say that RASP provides a perfect representation of all transformers. RASP expresses only a limited subset of transformers. Here, we enumerate the limitations of the RASP language, both to provide intuition about where the mapping trained from this corpus might fail and as a guide to future work. (The limitations are enumerated in decreasing order of perceived difficulty, according to the best knowledge of the authors):

- In practice, transformers tend to be trained with layer normalization (Ba et al., 2016) after the MHA and MLP layers. RASP does not include this, as the mechanisms underlying layernorm are not yet well-understood and a computational model for layernorm has not yet been developed (Xu et al., 2019; Lu et al., 2019; Loshchilov & Hutter, 2017). As the depth of a model increases, this may cause the behavior of RASP to differ significantly from normal transformers because layernorm may remove information relating to the absolute scale of model outputs. (Dong et al., 2021) provides preliminary evidence for this hypothesis.

- The `select` operation used in RASP is weaker than that used in most transformers. When a RASP program attends to a token using `select`, it either attends fully (1) or not at all (0), whereas traditional attention computations in transformers result in probability distributions ([0-1]) as a result of the softmax operation. It's worth noting that the `select` operation implemented by RASP is not hard-attention (Mnih et al., 2014), which has been shown to be much less expressive than traditional transformers, because RASP can recognize the majority language while hard-attention transformers cannot (Hao et al., 2022). Indeed, under certain conditions, the majority operation is sufficient to do most of the computations attention can do (Merrill & Sabharwal, 2022).

- The operations implemented by the MLPs inside of transformers are probably more complex than those permitted by the RASP language. In the original RASP paper (Weiss et al., 2021), the language permitted the use of arbitrary python functions for elementwise operations. However, we do not currently have an effective implementation of a mapping from arbitrary python code to MLPs. Therefore, we only allow basic elementwise operations (e.g. multiplication by a scalar) which can be easily converted into MLPs during compilation.
- As a result of being created from programs, transformers compiled from RASP programs tend to have different statistical properties than transformers trained on real data. For example, they lack superposition (Elhage et al., 2022).

For the purposes of this work, these limitations do not significantly impede our investigation of model mapping methods. Although these places limits on what can be expressed in the RASP language, they could be alleviated by other methods.

B. Description of Procedural Generations

B.1. Generating RASP programs

We randomly generate the RASP programs according to the following procedure:

1. We randomly select an integer n that indicates how many initial s-op variables should be created
2. The i-th initial s-op variable is created by randomly choosing one of the function primitives in the RASP language, and building that primitive out of variables previously defined which match the type required by the operation.

   For example, if at the 5th step, we randomly chose Select, the 5th variable might look like

   $$\text{var5 = Select(var3, tokens, } <)$$

   so long as var3 is syntactically possible to be an input to a select operation. Note the comparison (in this case $<$) is also randomly generated.
3. After the initial s-ops are created, we then ensure that each s-op created is actually used in the program, creating additional s-ops until this is satisfied
4. Finally, we return the most recently created s-op

B.2. Generating Python programs

We randomly generate Python programs according to the following procedure:

1. We begin by initializing the output out as a list of zeros
2. We then randomly choose between 1 and 3 outer for loops, and within each for loop, there is optionally an additional nested for loop
3. Inside each for loop, there is either a conditional operation or assignment. A typical conditional would look like:

   ```
   if arr[j] % 2 == 0:
   ```

   and a typical assignment would be of the form:

   ```
   out[j-1 % len(out)] = -arr[(j-1) % len(out)]
   ```
4. The output out is then returned

A typical predicted and target program:

```
def random_function(arr):
  n = len(arr)
  out = [0] * n
  for i in range(0, n):
    if arr[(i)] % 2 == 0:
      out[(i) % len(out)] = arr[(i) % len(arr)]
  for i in range(1, n):
    out[(i) % len(out)] = arr[(i) 1) % len(arr)]
    out[(i + 1) % len(out)] = arr[(i) % len(arr)]
  for i in range(1, n):
    out[(i) % len(out)] = arrarr[(i) % len(arr)]
    out[(i + 1) % len(out)] = arr[(i) % len(arr)]
    for j in range(1, n):
      out[(j) % len(out)] = arr[(j) % len(arr)]
      out[(j + 1) % len(out)] = arr[(j) % len(arr)]
      out[(j - 1) % len(out)] = arrarr[(j) 1) % len(arr)]
  return out
```

Figure 2. An example of a generated Python program.

```
def random_function(arr):
  n = len(arr)
  out = [0] * n
  for i in range(0, n):
    if arr[(i)] % 2!= 0:
      out[(i) % len(out)] = arr[(i) % len(arr)]
  for i in range(0, n):
    out[(i) % len(out)] = arr[(i - 1) % len(arr)]
    out[(i + 1) % len(out)] = arr[(i) % len(arr)]
  for i in range(1, 5):
    out[(i) % len(out)] = -arr[(i) % len(arr)]
    out[(i + 1) % len(out)] = arr[(i) % len(arr)]
    for j in range(1, n):
      out[(j) % len(out)] = arr[(j) % len(arr)]
      out[(j + 1) % len(out)] = arr[(j) % len(arr)]
      out[(j - 1) % len(out)] = -arr[(j + 1) % len(arr)]
  return out
```

Figure 3. An example of a target Python program.