Can Graph Learning Learn Circuits?

Chester Tan[*]
Chair of Machine Learning for Complex Networks
Center for AI and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
chester.tan@uni-wuerzburg.de

Moritz Lampert[*]
Chair of Machine Learning for Complex Networks
Center for AI and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
moritz.lampert@uni-wuerzburg.de

Courtney Maynard[*]
Khoury College of Computer Sciences
Northeastern University
maynard.co@northeastern.edu

Ankit Ramakrishnan
Network Science Institute
Northeastern University
ramakrishnan.ank@northeastern.edu

Tina Eliassi-Rad[†] [‡]
Khoury College of Computer Sciences
Northeastern University
tina@eliassi.org

Ingo Scholtes[†]
Chair of Machine Learning for Complex Networks
Center for AI and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
ingo.scholtes@uni-wuerzburg.de

Abstract

Circuit localization is a mechanistic interpretability task whose goal is to identify a sparse subgraph of a transformer's computation graph sufficient to reproduce a particular behavior. Most established methods localize circuits independently for each model–task pair. We instead frame circuit localization as a graph machine learning problem in which the edges of a computation graph represent computational pathways, and graph neural networks (GNNs) model interactions among these pathways. We introduce *Graph Circuit Learning* (GCL), a supervised, amortized framework that trains a GNN across multiple model–task pairs and applies it to unseen cases. To provide sufficient data, we augment the INTERPBENCH benchmark with additional cases derived from the TRACRBENCH programs. Of the 14 evaluated GCL configurations, the highest scored a median edge AUROC of 0.902 (interquartile interval $[0.861, 0.942]$) on the 16 original held-out INTERPBENCH cases. This is close to the published INTERPBENCH median of 0.910 for EAP-IG while remaining below ACDC's 0.959. Removing all message-passing edges reduces the median to 0.825. We also adapt PGExplainer, a GNN explainability method, to circuit localization, obtaining a median edge AUROC of 0.858 on the same cases. These preliminary results suggest that graph machine learning offers a natural and potentially powerful perspective on circuit localization, and we hope this perspective encourages closer exchange between the two communities.

1 Introduction

Circuit localization is a task in mechanistic interpretability that seeks to identify a sparse subgraph of a neural network's computation graph sufficient to reproduce a specified behavior. Most existing methods infer a new circuit for each model–task pair, repeating the localization process and limiting the reuse of regularities learned from circuits in other models or tasks [1–7].

Circuits themselves are graphs. They admit many representations and granularity levels, exposing computational relationships and symmetries that graph learning methods are designed to exploit [8–

[*]Equal contribution.
[†]Equal contribution.
[‡]Also with the Network Science Institute at Northeastern University and the Santa Fe Institute.

Preprint. Preliminary work.

10]. In particular, graph structure identifies where pathways meet, which provides a natural inductive bias for modeling dependencies between computationally related pathways. As a motivation for why graph learning is a natural fit for circuit localization, we show that the effect of intervening on one pathway can depend on another pathway entering the same component (Section 2).

We investigate two complementary approaches to applying graph machine learning to circuit localization (Section 3). **(1)** We introduce *Graph Circuit Learning (GCL)*, a supervised, amortized framework that trains a graph neural network (GNN) across labeled model–task cases and applies it to unseen ones. **(2)** We adapt a popular GNN explainer, PGExplainer [11], to the canonical per–model–task setting. To support cross-case training and evaluation, we construct a benchmark that extends INTERPBENCH [12] with 30 TRACRBENCH-derived model–task pairs and a split designed to evaluate transfer to unseen pairs (Section 4). Our preliminary results show that the best GCL configuration is competitive with published per-case baselines on held-out cases.

2 Circuit Localization as a Graph Machine Learning Problem

For a trained transformer $\mathcal{M}$, we represent its computation as a component-level directed acyclic graph $G_{\mathcal{M}} = (V_{\mathcal{M}}, E_{\mathcal{M}})$, whose nodes are components such as attention heads or MLP blocks and whose edges are possible computational pathways through the residual stream [7, 20] (Figure 1a and Section A). A circuit is a sparse subgraph of $G_{\mathcal{M}}$ sufficient to reproduce a specified behavior. For a model–task case $(\mathcal{M}, \mathcal{T})$, the task $\mathcal{T}$ provides clean–corrupted prompt pairs (x, x') and a scalar metric $L_{\mathcal{T}}$. Circuit localization infers a binary mask $\mathbf{y} \in \{0,1\}^{|E_{\mathcal{M}}|}$ over the candidate pathways: $y_e = 1$ retains the clean message along edge e, whereas $y_e = 0$ replaces it with the corresponding corrupted message. We denote the metric under this masked computation by $L_{\mathcal{T}}(\mathcal{M}, x, x'; \mathbf{y})$.

At the component level, $G_{\mathcal{M}}$ is heterogeneous. Its nodes have different computational roles, and their features vary in both dimension and semantics within and across models. Therefore, learning across model–task cases is more challenging than learning on a fixed graph with a shared feature space. Similar challenges arise in heterogeneous graph foundation models [21]. Section A discusses this issue and alternative graph representations.

The motivation for modeling relationships between computational pathways comes from their interactions. Consider two edges e_1 and e_2, whose interventions induce fixed additive message changes $\Delta m_{e_1,p}$ and $\Delta m_{e_2,p}$ at the same read input $u_{t,p}$ of component t at token position p. Let $L_{t,p}(u)$ denote the scalar task metric obtained by replacing this read input with u and continuing the downstream computation. Writing $u^{\mathrm{cl}}_{t,p}$ for the clean read input, define $I_p(e_1, e_2)$ as the interaction contrast between applying both message changes jointly and applying each separately, as formalized in Equation (34) of Section E. Assume $L_{t,p}$ is twice continuously differentiable on a neighborhood containing the intervention surface and its Hessian is locally Lipschitz there. Let $Q_{t,p} = \nabla^2_u L_{t,p}(u^{\mathrm{cl}}_{t,p})$ denote the Hessian at the clean read input. Then, for sufficiently small message changes, we have:

$$I_p(e_1, e_2) = \Delta m^{\top}_{e_1,p} Q_{t,p} \Delta m_{e_2,p} + O\Big(\big(\|\Delta m_{e_1,p}\| + \|\Delta m_{e_2,p}\| \big)^3 \Big). \tag{1}$$

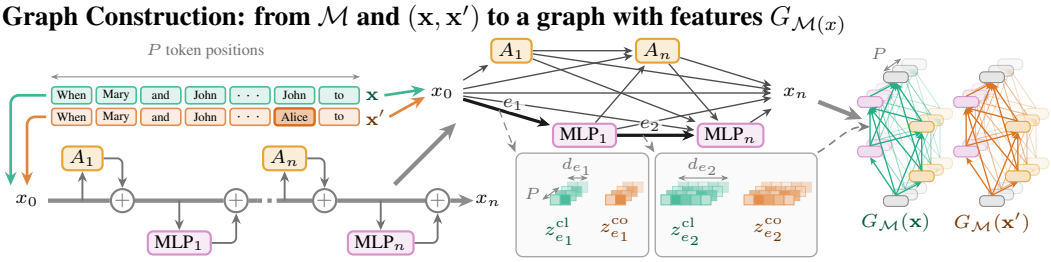
Section E contains the full derivation and proof. A nonzero mixed term indicates that the leading local interaction depends jointly on both message changes and the downstream computation. Thus, pathway relevance cannot always be decomposed into independent edge scores. Graph representations that expose shared read inputs and other relationships among pathways provide graph learners with useful structure for modeling such interactions.

3 Graph Machine Learning for Circuit Localization

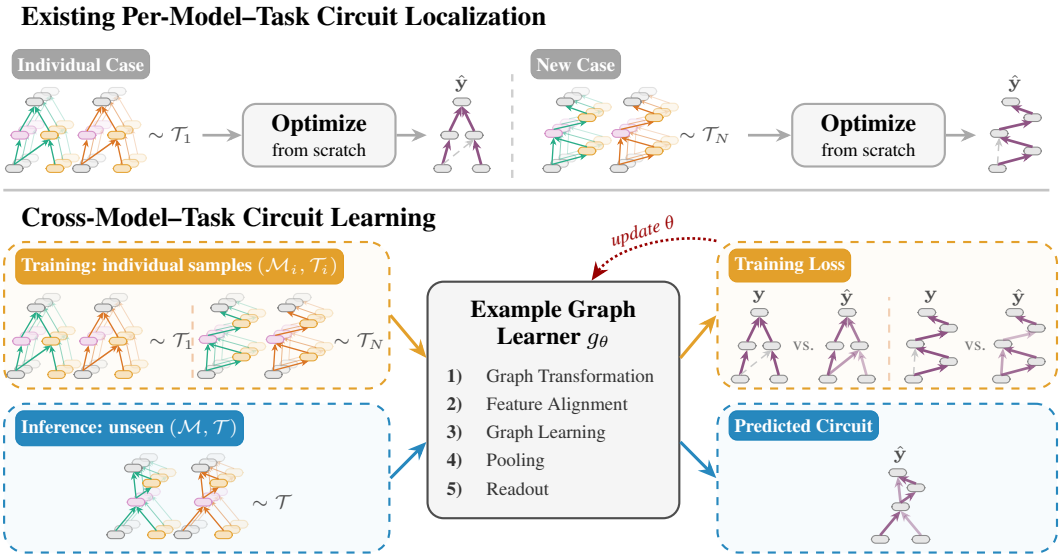
We investigate two complementary graph machine learning approaches for learning circuit masks.

Graph Circuit Learning (GCL). Rather than inferring a circuit mask independently for every model–task case, GCL trains a GNN across multiple model–task cases using their ground-truth masks, then applies the learned predictor to unseen model–task cases (Figure 1b).

Each training case $(\mathcal{M}_i, \mathcal{T}_i)$ consists of a transformer with component-level computation graph $G_{\mathcal{M}_i}$, a task $\mathcal{T}_i$, and a ground-truth circuit mask $\mathbf{y}_i$. We train one predictor across cases: $g_\theta\left(G_{\mathcal{M}}, \mathcal{T}\right) = \hat{\mathbf{y}} \in [0,1]^{|E_{\mathcal{M}}|}$ where $\hat{y}_e$ estimates whether edge e is in the ground-truth circuit, and training is



(a) Constructing a component-level computation graph with edge features from a transformer and a prompt pair. Each component reads from the residual stream and writes an additive update back to it. Unrolling these interactions yields the graph $G_{\mathcal{M}}$, whose nodes are components and whose edges are computational pathways. Running the model on a clean prompt $\mathbf{x}$ and a corrupted prompt $\mathbf{x}'$, each containing P token positions, produces the edge activations z_e. Together, the graph and activations form the clean (green) and corrupted (red) featured graphs passed to the learner in Figure 1b.



(b) Existing circuit-localization methods optimize a separate mask for each model–task case. The illustrated approach instead trains a shared graph learner g_θ across cases and applies it to unseen ones. Our example graph learner transforms the component-level computation graph into a directed line graph [13] (or an incidence graph [14, 15]). It then aligns heterogeneous features using TabPFN-style 1D feature attention [16], processes the transformed graph with a directed graph learner such as DirGNN [17] (or DAGformer [18]), and pools across token positions and prompt pairs using a permutation-invariant method [19]. Finally, it reads out one circuit-membership score for each edge in the original component-level computation graph. This architecture is one possible design. Section B details each stage.

Figure 1: An overview of (a) the component-level computation graph construction and (b) the per-case circuit localization vs. cross-case circuit learning.

end-to-end against the ground-truth masks with binary cross-entropy. Further implementation details are provided in Section B.

Figure 1b illustrates our primary graph-learning architecture, which comprises five stages: graph transformation, feature alignment, graph learning, pooling, and readout. This architecture represents only one way to learn from the component-level computation graph. Future work can explore alternative graph representations and learning architectures for circuit localization.

GNN Explainers for Circuit Localization. PGExplainer [11] is a GNN explainability approach that trains a multilayer perceptron (MLP) to generate edge masks explaining the predictions of a fixed, trained GNN. In our adaptation of PGExplainer, the MLP uses fixed features derived from transformer parameters to score each edge in the component-level computation graph. These scores

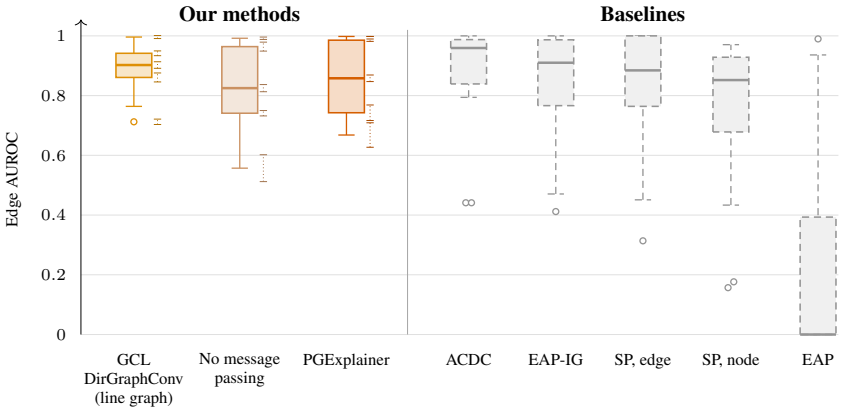


Figure 2: Edge AUROC across the 16 held-out INTERPBENCH cases. For our methods (solid box outlines), scores are averaged over 5 random seeds within each case; capped dotted intervals beside each box show ± 1 seed-level standard deviation for each summary statistic. Corresponding seed-level variability is unavailable for the published baselines (dashed box outlines). The highlighted GCL configuration uses DirGNN with a GraphConv aggregator (DirGraphConv) on the directed line graph and is the strongest of the 14 configurations evaluated on these cases (Section F). Baseline values are taken from Figure 9 of INTERPBENCH [12].

define a sampled mask that is applied during a transformer forward pass. The MLP is trained to preserve the clean model output. Unlike GCL, a separate explainer is fitted for each model–task case. Figure 5 illustrates how we adapt the original method from GNN explanation to circuit localization.

4 A Benchmark for Circuit Localization Across Model–Task Pairs

Evaluating circuit-localization methods requires knowing which circuit is correct. However, to our knowledge, there is no established way to obtain unique or provably correct ground-truth circuits for real-world transformers (Section H). INTERPBENCH [12] addresses this challenge using realistic semi-synthetic transformers with circuits known by construction.

Restricted Access Sequence Processing (RASP) is a domain-specific language for expressing sequence-processing algorithms in transformer-like operations [22]. Each RASP-derived case begins with a program whose primitives correspond to transformer operations. TRACR compiles the program into transformer weights [23], after which Strict Interchange Intervention Training (SIIT) trains a new transformer to preserve the same computation and ground-truth circuit while producing weight distributions closer to those of conventionally trained models [12].

However, INTERPBENCH was designed for the per–model–task setting and contains too few pairs to train a cross-case localizer or evaluate generalization to unseen pairs. To support cross-case learning, we apply the same SIIT training procedure to TRACRBENCH model–task pairs [24], adding 30 pairs to INTERPBENCH's 84 RASP-derived pairs. We **(1)** retain INTERPBENCH's 16 original evaluation pairs as the held-out test set, **(2)** exclude every related pair in any group containing a test case, and **(3)** use the remaining 50 pairs for grouped 5-fold cross-validation. Section D describes the added pairs and split design in detail.

5 Preliminary Evidence and Discussion

We evaluate both methods by edge AUROC on each held-out pair in our benchmark (Section F.1). We select from up to 18 hyperparameter settings using grouped 5-fold cross-validation on 50 cases and average results over 5 random seeds per held-out case.

Of the 14 configurations evaluated, Figure 2 compares the best-performing setup with **(1)** a control with all message-passing edges removed, **(2)** the adapted PGExplainer, and **(3)** the published INTERPBENCH baselines. The selected GCL configuration achieves a higher median edge AUROC (0.902) than both the control (0.825) and PGExplainer (0.858). GCL and our adapted PGExplainer also outperform every published baseline in minimum AUROC, and GCL does so at the first quartile. Because

the control retains every node and all node features, the comparison provides evidence that message passing over the observed computation graph contributes to performance. More broadly, these results suggest that amortized graph learning is a promising approach to circuit localization, while the adapted PGExplainer's performance demonstrates the feasibility of adapting graph explainability methods to this setting. Section F provides additional experimental results and ablations.

Discussion. This preliminary study focuses on synthetic model–task pairs with known ground-truth circuits. Its aim is not to identify the optimal graph-learning architecture, but to show that circuit localization admits a natural graph-learning formulation and that this formulation *can* improve performance in certain settings. Future work should evaluate this approach on larger, real-world models, using metrics such as faithfulness when ground-truth circuits are unavailable [3], and explore the broader design space of graph representations and graph-learning methods for circuit localization.

References

[1] Aaquib Syed, Can Rager, and Arthur Conmy. Attribution Patching Outperforms Automated Circuit Discovery. In Yonatan Belinkov, Najoung Kim, Jaap Jumelet, Hosein Mohebbi, Aaron Mueller, and Hanjie Chen, editors, *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 407–416, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.blackboxnlp-1.25. URL https://aclanthology.org/2024.blackboxnlp-1.25/. 1, 22, 23, 25

[2] Michael Hanna, Sandro Pezzelle, and Yonatan Belinkov. Have Faith in Faithfulness: Going Beyond Circuit Overlap When Finding Model Mechanisms. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=TZ0CCGDcuT. 22, 23, 25

[3] Aaron Mueller, Atticus Geiger, Sarah Wiegreffe, Dana Arad, Iván Arcuschin, Adam Belfki, Yik Siu Chan, Jaden Fried Fiotto-Kaufman, Tal Haklay, Michael Hanna, Jing Huang, Rohan Gupta, Yaniv Nikankin, Hadas Orgad, Nikhil Prakash, Anja Reusch, Aruna Sankaranarayanan, Shun Shao, Alessandro Stolfo, Martin Tutek, Amir Zur, David Bau, and Yonatan Belinkov. MIB: A Mechanistic Interpretability Benchmark. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 45069–45108. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/mueller25a.html. 5, 12

[4] Adithya Bhaskar, Alexander Wettig, Dan Friedman, and Danqi Chen. Finding Transformer Circuits With Edge Pruning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 18506–18534. Curran Associates, Inc., 2024. doi: 10.52202/079017-0587. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/20fdaf67581e6d7157376d1ed584040a-Paper-Conference.pdf. 10, 25

[5] Steven Cao, Victor Sanh, and Alexander Rush. Low-Complexity Probing via Finding Subnetworks. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 960–966, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.74. URL https://aclanthology.org/2021.naacl-main.74/. 24, 25

[6] Philipp Mondorf, Mingyang Wang, Sebastian Gerstner, Ahmad Dawar Hakimi, Yihong Liu, Leonor Veloso, Shijia Zhou, Hinrich Schuetze, and Barbara Plank. BlackboxNLP-2025 MIB Shared Task: Exploring Ensemble Strategies for Circuit Localization Methods. In Yonatan Belinkov, Aaron Mueller, Najoung Kim, Hosein Mohebbi, Hanjie Chen, Dana Arad, and Gabriele Sarti, editors, *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 537–542, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-346-3. doi: 10.18653/v1/2025.blackboxnlp-1.31. URL https://aclanthology.org/2025.blackboxnlp-1.31/.

[7] Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards Automated Circuit Discovery for Mechanistic Interpretability. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in*

Neural Information Processing Systems, volume 36, pages 16318–16352. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/34e1dbe95d34d7ebaf99b9bcaeb5b2be-Paper-Conference.pdf. 1, 2, 10, 24, 25

[8] Derek Lim, Haggai Maron, Marc T Law, Jonathan Lorraine, and James Lucas. Graph Metanetworks for Processing Diverse Neural Architectures. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 11430–11458, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/2feafd84b23a52b3fa434bc6d7682256-Paper-Conference.pdf. 1, 25

[9] Xiaolong Han, Zehong Wang, Bo Zhao, Binchi Zhang, Jundong Li, Damian Borth, Rose Yu, Haggai Maron, Yanfang Ye, Lu Yin, and Ferrante Neri. A Survey of Weight Space Learning: Understanding, Representation, and Generation, 2026. URL https://arxiv.org/abs/2603.10090. 13, 25

[10] Allan Zhou, Kaien Yang, Kaylee Burns, Adriano Cardace, Yiding Jiang, Samuel Sokota, J. Zico Kolter, and Chelsea Finn. Permutation Equivariant Neural Functionals. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 24966–24992. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4e9d8aeeab6120c3c83ccf95d4c211d3-Paper-Conference.pdf. 2, 25

[11] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized Explainer for Graph Neural Network. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 19620–19631. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/e37b08dd3015330dcbb5d6663667b8b8-Paper.pdf. 2, 3, 16, 17, 24, 25

[12] Rohan Gupta, Iván Arcuschin, Thomas Kwa, and Adrià Garriga-Alonso. InterpBench: Semi-Synthetic Transformers for Evaluating Mechanistic Interpretability Techniques. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 92922–92951. Curran Associates, Inc., 2024. doi: 10.52202/079017-2950. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/a8f7d43ae092d9a5295775eb17f3f4f7-Paper-Datasets_and_Benchmarks_Track.pdf. 2, 4, 11, 12, 24, 26

[13] Frank Harary and Robert Z. Norman. Some properties of line digraphs. *Rendiconti del Circolo Matematico di Palermo*, 9(2):161–168, May 1960. ISSN 1973-4409. doi: 10.1007/bf02854581. URL http://dx.doi.org/10.1007/BF02854581. 3, 14

[14] Alain Bretto. *Hypergraph Theory: An Introduction*. Springer International Publishing, 2013. ISBN 9783319000800. doi: 10.1007/978-3-319-00080-0. URL http://dx.doi.org/10.1007/978-3-319-00080-0. 3, 14

[15] Xiyuan Wang, Pan Li, and Muhan Zhang. Improving Graph Neural Networks on Multi-node Tasks with the Labeling Trick. *Journal of Machine Learning Research*, 26(23):1–44, 2025. URL http://jmlr.org/papers/v26/23-0560.html. 3, 14

[16] Noah Hollmann, Samuel Müller, Lennart Purucker, Arjun Krishnakumar, Max Körfer, Shi Bin Hoo, Robin Tibor Schirrmeister, and Frank Hutter. Accurate predictions on small data with a tabular foundation model. *Nature*, 637(8045):319–326, January 2025. ISSN 1476-4687. doi: 10.1038/s41586-024-08328-6. URL http://dx.doi.org/10.1038/s41586-024-08328-6. 3, 14, 25

[17] Emanuele Rossi, Bertrand Charpentier, Francesco Di Giovanni, Fabrizio Frasca, Stephan Günnemann, and Michael M. Bronstein. Edge Directionality Improves Learning on Heterophilic Graphs. In Soledad Villar and Benjamin Chamberlain, editors, *Proceedings of the Second Learning on Graphs Conference*, volume 231 of *Proceedings of Machine Learning Research*, pages 25:1–25:27. PMLR, 27–30 Nov 2023. URL https://proceedings.mlr.press/v231/rossi24a.html. 3, 15

[18] Yuankai Luo, Veronika Thost, and Lei Shi. Transformers over Directed Acyclic Graphs. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 47764–47782. Curran Associates,

Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/94e85561a342de88b559b72c9b29f638-Paper-Conference.pdf. 3, 15

[19] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander Smola. Deep Sets. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf. 3, 16

[20] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A Mathematical Framework for Transformer Circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html. 2, 10, 11, 12

[21] Zehong Wang, Zheyuan Liu, Tianyi Ma, Jiazheng Li, Zheyuan Zhang, Xingbo Fu, Yiyang Li, Zhengqing Yuan, Wei Song, Yijun Ma, Qingkai Zeng, Xiusi Chen, Jianan Zhao, Jundong Li, Meng Jiang, Pietro Lio, Nitesh Chawla, Chuxu Zhang, and Yanfang Ye. Graph Foundation Models: A Comprehensive Survey, 2025. URL https://arxiv.org/abs/2505.15116. 2

[22] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking Like Transformers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/weiss21a.html. 4, 26

[23] David Lindner, Janos Kramar, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. Tracr: Compiled Transformers as a Laboratory for Interpretability. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 37876–37899. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/771155abaae744e08576f1f3b4b7ac0d-Paper-Conference.pdf. 4, 11, 26

[24] Hannes Thurnherr and Jérémy Scheurer. TracrBench: Generating Interpretability Testbeds with Large Language Models. In *ICML 2024 Workshop on Mechanistic Interpretability*, 2024. URL https://openreview.net/forum?id=vNubZ5zK8h. 4, 16, 26

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 10

[26] Jack Merullo, Carsten Eickhoff, and Ellie Pavlick. Talking Heads: Understanding Inter-Layer Communication in Transformer Language Models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 61372–61418. Curran Associates, Inc., 2024. doi: 10.52202/079017-1962. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/70e5444e5f331f7f5431f302110b97af-Paper-Conference.pdf. 10, 12

[27] Aleksandar Makelov, Georg Lange, Atticus Geiger, and Neel Nanda. Is This the Subspace You Are Looking for? An Interpretability Illusion for Subspace Activation Patching. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 26486–26515, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/70b8505ac79e3e131756f793cd80eb8d-Paper-Conference.pdf. 10

[28] Areeb Ahmad, Abhinav Joshi, and Ashutosh Modi. Beyond Components: Singular Vector-Based Interpretability of Transformer Circuits. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 167772–167811. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f51afa708f3d4b3cae8dc9a8a9bbea87-Paper-Conference.pdf. 12

[29] Yixuan He, Xitong Zhang, Junjie Huang, Benedek Rozemberczki, Mihai Cucuringu, and Gesine Reinert. PyTorch Geometric Signed Directed: A Software Package on Graph Neural Networks for Signed and Directed Graphs. In Soledad Villar and Benjamin Chamberlain, editors, *Proceedings of the Second Learning on Graphs Conference*, volume 231 of *Proceedings of Machine Learning Research*, pages 12:1–12:27. PMLR, 27–30 Nov 2023. URL https://proceedings.mlr.press/v231/he24a.html. 12

[30] Yixuan He, Michael Perlmutter, Gesine Reinert, and Mihai Cucuringu. MSGNN: A Spectral Graph Neural Network Based on a Novel Magnetic Signed Laplacian. In Bastian Rieck and Razvan Pascanu, editors, *Proceedings of the First Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*, pages 40:1–40:39. PMLR, 09–12 Dec 2022. URL https://proceedings.mlr.press/v198/he22c.html.

[31] Ali Parviz and Yuichi Yoshida. Nonlinear Laplacians Improve Signed-Directed Graph Learning. In *New Perspectives in Graph Machine Learning*, 2025. URL https://openreview.net/forum?id=3CcP3VI6LW.

[32] Maya Bechler-Speicher, Yoel Gottlieb, Andrey Isakov, David Abensur, Ami Tavory, Daniel Haimovich, Ido Guy, and Udi Weinsberg. Billion-Scale Graph Foundation Models, 2026. URL https://arxiv.org/abs/2602.04768. 12

[33] Alexander Theus, Alessandro Cabodi, Sotiris Anagnostidis, Antonio Orvieto, Sidak Pal Singh, and Valentina Boeva. Generalized Linear Mode Connectivity for Transformers. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 136197–136225. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c6eadcc507edc04c1baf00f05cd18b1a-Paper-Conference.pdf. 13

[34] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric, 2019. URL https://arxiv.org/abs/1903.02428. 15

[35] Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaž Stojanovič, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan Eric Lenssen, and Jure Leskovec. PyG 2.0: Scalable Learning on Real World Graphs, 2025. URL https://arxiv.org/abs/2507.16991. 15

[36] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. ISBN 978-1-57735-809-1. doi: 10.1609/aaai.v33i01.33014602. URL https://doi.org/10.1609/aaai.v33i01.33014602. 15

[37] Haitz Sáez de Ocáriz Borde, Artem Lukoianov, Anastasis Kratsios, Michael Bronstein, and Xiaowen Dong. Scalable Message Passing Neural Networks: No Need for Attention in Large Graph Representation Learning, 2026. URL https://arxiv.org/abs/2411.00835. 15

[38] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking Attention with Performers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ua6zuk0WRH. 15

[39] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5156–5165. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/katharopoulos20a.html. 15

[40] Joseph D. Janizek, Pascal Sturmfels, and Su-In Lee. Explaining Explanations: Axiomatic Feature Interactions for Deep Networks. *Journal of Machine Learning Research*, 22(104):1–54, 2021. URL http://jmlr.org/papers/v22/20-1223.html. 22

[41] Hang Chen, Jiaying Zhu, Xinyu Yang, and Wenya Wang. Rethinking Circuit Completeness in Language Models: AND, OR, and ADDER Gates. In D. Belgrave, C. Zhang,

H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 150511–150540. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/dd37fdb24a4e1cfa3ed5c247217a7394-Paper-Conference.pdf. 22

[42] Kevin Ro Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 Small. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=NpsVSN6o4ul. 22

[43] Thomas McGrath, Matthew Rahtz, Janos Kramar, Vladimir Mikulik, and Shane Legg. The Hydra Effect: Emergent Self-repair in Language Model Computations, 2023. URL https://arxiv.org/abs/2307.15771.

[44] Cody Rushing and Neel Nanda. Explorations of Self-Repair in Language Models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 42836–42855. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/rushing24a.html. 22

[45] Callum McDougall, Arthur Conmy, Cody Rushing, Thomas McGrath, and Neel Nanda. Copy Suppression: Comprehensively Understanding an Attention Head, 2023. URL https://arxiv.org/abs/2310.04625. 22

[46] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996. ISSN 1557-7317. doi: 10.1145/227234.227246. URL http://dx.doi.org/10.1145/227234.227246. 23

[47] Barsat Khadka. MechRL: Reinforcement Learning Agents Perform Circuit Discovery for Mechanistic Interpretability, 2026. URL https://arxiv.org/abs/2605.26343. 25

[48] Shun Shao, Binxu Wang, Shay B. Cohen, Anna Korhonen, and Yonatan Belinkov. Differentiable Faithfulness Alignment for Cross-Model Circuit Transfer, 2026. URL https://arxiv.org/abs/2604.24302. 25

[49] Harrish Thasarathan, Matthew Kowal, Thomas Fel, and Konstantinos G. Derpanis. Cross-Model Circuit Discovery. In *Mechanistic Interpretability Workshop at ICML 2026*, 2026. URL https://openreview.net/forum?id=SliNacpKqc. 25

[50] Naiyu Yin, Dennis Wei, Tian Gao, Amit Dhurandhar, Karthikeyan Natesan Ramamurthy, and Yue Yu. Scalable Circuit Learning for Interpreting Large Language Models, 2026. URL https://arxiv.org/abs/2606.16939. 25

[51] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. GNNExplainer: Generating Explanations for Graph Neural Networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/d80b7040b773199015de6d3b4293c8ff-Paper.pdf. 25

[52] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. On Explainability of Graph Neural Networks via Subgraph Explorations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12241–12252. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/yuan21c.html. 25

[53] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=WznmQa42ZAx. 25

[54] Wanyu Lin, Hao Lan, and Baochun Li. Generative Causal Explanations for Graph Neural Networks. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 6666–6679. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/lin21d.html. 25

[55] Zehong Wang, Zheyuan Zhang, Nitesh V Chawla, Chuxu Zhang, and Yanfang Ye. GFT: Graph Foundation Model with Transferable Tree Vocabulary. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information*

Processing Systems, volume 37, pages 107403–107443. Curran Associates, Inc., 2024. doi: 10.52202/079017-3412. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/c23ccf9eedf87e4380e92b75b24955bb-Paper-Conference.pdf. 25

[56] Ben Finkelshtein, Ismail Ilkan Ceylan, Michael Bronstein, and Ron Levie. Equivariance Everywhere All At Once: A Recipe for Graph Foundation Models. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 30377–30434. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/2b89783b68cdcfa7c8d7b08c09f47b2a-Paper-Conference.pdf. 25

[57] Dan Friedman, Alexander Wettig, and Danqi Chen. Learning Transformer Programs. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 49044–49067. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/995f693b73050f90977ed2828202645c-Paper-Conference.pdf. 26

[58] Hannes Thurnherr and Kaspar Riesen. Neural Decompiling of Tracr Transformers. In Ching Yee Suen, Adam Krzyzak, Mirco Ravanelli, Edmondo Trentin, Cem Subakan, and Nicola Nobile, editors, *Artificial Neural Networks in Pattern Recognition*, pages 25–36, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-71602-7. 26

[59] Xinting Huang, Aleksandra Bakalova, Satwik Bhattamishra, William Merrill, and Michael Hahn. Discovering Interpretable Algorithms by Decompiling Transformers to RASP, 2026. URL https://arxiv.org/abs/2602.08857. 26

[60] Jason Gross, Rajashree Agrawal, Thomas Kwa, Euan Ong, Chun Hei Yip, Alex Gibson, Soufiane Noubir, and Lawrence Chan. Compact Proofs of Model Performance via Mechanistic Interpretability. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 79453–79515. Curran Associates, Inc., 2024. doi: 10.52202/079017-2523. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/90e73f3cf1a6c84c723a2e8b7fb2b2c1-Paper-Conference.pdf. 26

[61] Itamar Hadad, Guy Katz, and Shahaf Bassan. Formal Mechanistic Interpretability: Automated Circuit Discovery with Provable Guarantees. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Timsb74vIY. 26

A A Transformer as a Component-Level Directed Acyclic Graph (DAG)

In circuit localization, the computations inside a decoder-only transformer are often represented using a different, but mathematically equivalent, reformulation of the transformer definition given by Vaswani et al. [25]. While this formulation would be less computationally efficient if used as an implementation, it is easier to interpret because the residual stream, through which all components communicate, is linear [20]. The nonlinear components, such as attention heads and MLP blocks, communicate through this residual stream via two operations: (i) each component "reads" information from the residual stream through a projection W_I^*, and (ii) "writes" information back into the residual stream by adding its internal output, projected through W_O^* (see Figure 3). The read and write projections constrain which directions in the residual stream a component can access and modify, and thus provide a useful geometric view of how components interact through the shared residual stream [26, 27].

We can equivalently unroll the transformer's residual-stream representation into a component-level computation graph, a directed acyclic graph (DAG) in which each node is a model component and directed edges represent possible activation pathways from upstream writes to downstream reads [4, 7] (Figure 4). Because the residual stream is additive, every downstream component can in principle read information written by any earlier component. This induces edges not only between adjacent layers, but also between non-adjacent components connected through the residual stream. The resulting component-level DAG is not intended as an efficient implementation of the transformer, but as a faithful abstraction for reasoning about direct and mediated information flow between components.

Under this abstraction, an edge from an upstream component $*_i$ to a downstream component $*_j$ can be associated with an effective linear map, or "virtual weight" [20]. Using column-vector notation,

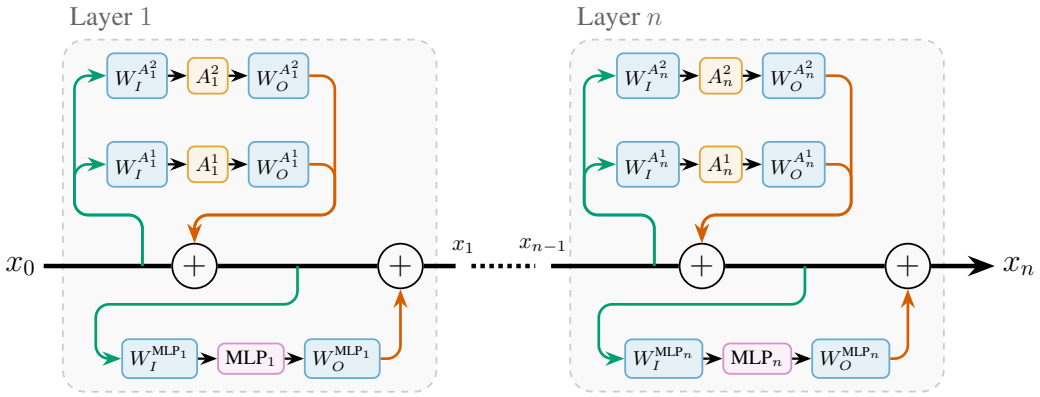


Figure 3: Residual-stream view of an n-layer decoder-only transformer. Attention heads (yellow) and MLP blocks (pink) read from the residual stream through input projections (blue) and write additive updates back through output projections (red).

this virtual weight is given by

$$W_I^{*_j} W_O^{*_i}, \tag{2}$$

where $W_O^{*_i}$ is the write projection of the upstream component and $W_I^{*_j}$ is the read projection of the downstream component. This virtual weight describes how directions written in the residual stream by component $*_i$ are visible to the input subspace read by component $*_j$. Thus, virtual weights provide a parameter-space explanation for why edges in the component-level computation graph are meaningful, while circuit-localization methods typically intervene on the corresponding activations along these edges.

The virtual weight covers only the residual-stream segment between an upstream write and a downstream read projection; component-internal nonlinearities, such as MLP activations and attention softmax, remain inside the nodes. Its linearity therefore requires that nothing nonlinear acts on the residual stream between writes and reads. This holds exactly for the models studied here: all benchmark transformers are Tracr-derived models trained with SIIT [12], and they inherit the Tracr-compiled architecture, which contains no normalization layers [23]. In typical transformers, every read is instead preceded by LayerNorm, whose scale and bias can be folded into the read projection but whose per-input rescaling remains nonlinear, so virtual weights there describe write–read interactions only up to an input-dependent rescaling [20]. Circuit-localization methods treat normalization as part of the downstream component, which keeps the component-level DAG exact; only the linear form of the edge map is specific to models without normalization layers.

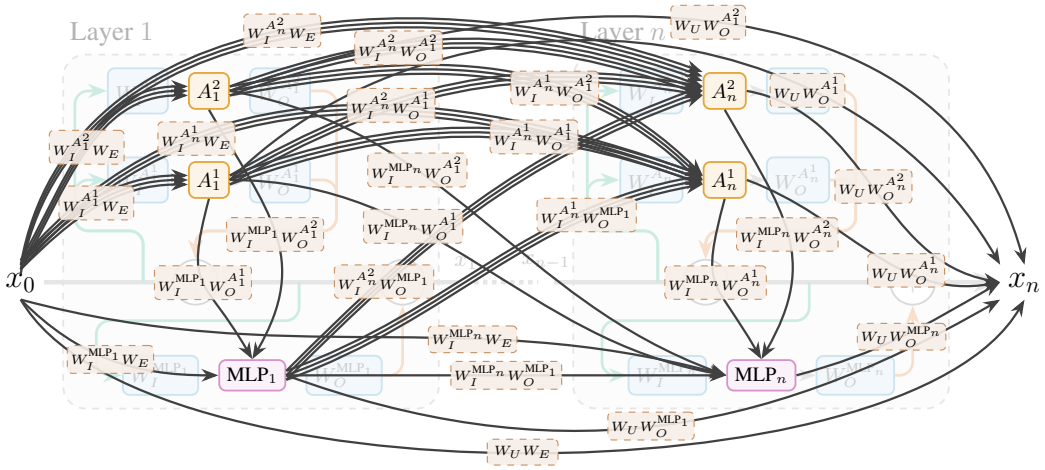


Figure 4: Component-level directed acyclic graph induced by the residual stream. Edges represent possible residual-stream-mediated activation pathways from upstream writes to downstream reads. Attention heads have separate query, key, and value read pathways.

The definition of the read projection W_I^* and write projection W_O^* depends on the component class of $*$. For MLPs of the general form

$$\mathrm{MLP}(\mathbf{x}) = W_{\mathrm{out}} h(W_{\mathrm{in}}\mathbf{x}), \tag{3}$$

where arbitrary hidden layers and nonlinear activations are abstracted into h, the input matrix $W_{\mathrm{in}} \in \mathbb{R}^{d_{\mathrm{mlp}} \times d_{\mathrm{model}}}$ and output matrix $W_{\mathrm{out}} \in \mathbb{R}^{d_{\mathrm{model}} \times d_{\mathrm{mlp}}}$ act as the read and write projections, respectively:

$$W_I^{\mathrm{MLP}} = W_{\mathrm{in}}, \qquad W_O^{\mathrm{MLP}} = W_{\mathrm{out}}. \tag{4}$$

Attention heads write to the residual stream through a single output projection W_O^A, which plays the same role for a head that W_{out} plays for an MLP, but read from the stream through three distinct projections that form their queries, keys, and values [20, 26]. Therefore, an upstream component $*_i$ can connect to a downstream attention head A_j through three distinct read pathways, governed by the virtual weights

$$\text{Value pathway:} \quad W_V^{A_j} W_O^{*_i}, \tag{5}$$

$$\text{Query pathway:} \quad W_Q^{A_j} W_O^{*_i}, \tag{6}$$

$$\text{Key pathway:} \quad W_K^{A_j} W_O^{*_i}. \tag{7}$$

These read-side virtual weights should be distinguished from the standard head-internal QK and OV circuits of the Transformer Circuits framework: under the column-vector convention used here, $W_Q^\top W_K$ determines how an attention head routes information across token positions, while $W_O W_V$ determines what information the head writes back into the residual stream [20].

Lastly, input tokens are projected into the residual stream via the embedding matrix $W_E \in \mathbb{R}^{d_{\mathrm{model}} \times d_{\mathrm{vocab}}}$, and the final residual stream state is transformed back to logits via the unembedding matrix $W_U \in \mathbb{R}^{d_{\mathrm{vocab}} \times d_{\mathrm{model}}}$. All bias terms are omitted from these formulations for simplicity. For details on folding biases into augmented weight matrices, see Ahmad et al. [28].

This component-level computation graph is the graph on which circuit-localization methods operate: nodes are transformer components, edges are possible activation pathways between components, and circuit localization aims to identify the sparse subset of edges that is relevant for a given model behavior.

Because INTERPBENCH and the Mechanistic Interpretability Benchmark (MIB) formulate circuit localization over model components and their connections, we use the corresponding component-level computation graph in this preliminary study [3, 12]. Other graph representations are possible, such as neuron-level graphs that treat individual neurons as nodes and their connections as edges. This representation avoids matrix edge features and heterogeneity in node features. It is, however, a much larger graph, which can exceed billions of edges even for a relatively small LLM. It is also directed and carries signed edge weights. Large size, edge direction, and signed weights together make the neuron-level graph a challenging and underexplored setting for graph learning [29–32]. We leave the study of graph representations beyond the component-level computation graph to future work, along with the challenges and opportunities they present for bringing graph learning to mechanistic interpretability.

B The Graph Circuit Learning Framework in Detail

This appendix specifies the Graph Circuit Learning architectures used in our experiments. It covers features, graph transformations, feature alignment, DirGNN and DAGformer graph learning backends, pooling, readout, and the supervised training procedure.

For a model–task case $(\mathcal{M}, \mathcal{T})$, let $G_{\mathcal{M}} = (V_{\mathcal{M}}, E_{\mathcal{M}})$ be the component-level computation graph defined in Section A, and let $\mathbf{y} \in \{0,1\}^{|E_{\mathcal{M}}|}$ be its ground-truth circuit mask. All architecture variants use the same raw feature construction and produce one logit for each edge in the original order of $E_{\mathcal{M}}$. They differ in their transformed graph, graph-learning architecture, and connectivity condition.

B.1 Features

A task $\mathcal{T}$ supplies a collection of clean–corrupted prompt pairs (x_b, x_b'). Let $\mathcal{B}_{\mathcal{T}}$ denote the clean–corrupted prompt pairs used to construct features for a model–task case. For each pair $b \in \mathcal{B}_{\mathcal{T}}$, let

$\mathcal{P}_b$ be the output positions where the benchmark compares the model's output with the task-specific target, and let $c = (b, p)$ denote the context for prompt pair b at position p. The component-level computation graph and ground-truth mask are fixed within a model–task case, whereas the activations and gradients vary across contexts. The implementation encodes each context (b, p) separately and pools the resulting edge representations only after graph learning.

Separate clean and corrupted forward passes provide component activations. A backward pass on the clean computation provides gradients of a task-specific scalar feature-extraction loss, denoted by $\ell_{\mathcal{T}}^{\text{feat}}$, specified by the benchmark for each model–task pair. The feature extractor uses KL divergence for tasks with discrete answer distributions, L_1 loss for floating-point targets, and, for tasks whose batches expose only class labels rather than full target distributions, the benchmark's logit-difference loss. For component v, let $a_{v,c}^{\text{cl}}$ and $a_{v,c}^{\text{co}}$ denote its clean and corrupted vectors at that component's input or output interface, and let

$$g_{v,c} = \nabla_{a_{v,c}^{\text{cl}}} \ell_{\mathcal{T}}^{\text{feat}} \tag{8}$$

denote the corresponding clean-computation gradient. For each component, we use the vector at the part of the computation represented by that component: attention writers use the per-head vector before the output projection, attention readers use the query, key, or value vector, MLP writers use the hidden vector after the nonlinearity, and MLP readers use the vector before the nonlinearity.

The forward pass records component states rather than a separate vector for every edge in the component-level computation graph. For an edge $e = (s, t)$, the map from the source component's vector to that edge's contribution to the target component's input is

$$W_e = W_I^t W_O^s, \qquad W_e : \mathbb{R}^{d_s} \to \mathbb{R}^{d_t}, \tag{9}$$

under the column-vector convention of Section A. The implementation retains the corresponding read and write factors and applies them successively, so it does not need to materialize every dense matrix W_e. The clean and corrupted pathway messages are

$$m_{e,c}^{\text{cl}} = W_e a_{s,c}^{\text{cl}}, \qquad m_{e,c}^{\text{co}} = W_e a_{s,c}^{\text{co}}. \tag{10}$$

These are the clean and corrupted edge messages denoted by z_e^{cl} and z_e^{co} in Figure 1a.

To calculate the clean gradient for each edge, we represent the target component's input by the sum of clean messages from its incoming edges, $u_{t,c}^{\text{lin}} = \sum_{e'=(s',t)} m_{e',c}^{\text{cl}}$. Since each clean message contributes additively, $\partial u_{t,c}^{\text{lin}} / \partial m_{e,c}^{\text{cl}} = I$ and

$$\nabla_{m_{e,c}^{\text{cl}}} \ell_{\mathcal{T}}^{\text{feat}} = g_{t,c}. \tag{11}$$

To express this gradient with respect to the source component's vector, we compute

$$\widetilde{g}_{e,c} = W_e^\top g_{t,c}. \tag{12}$$

The gradient $g_{t,c}$ is the same for all edges entering t, whereas $\widetilde{g}_{e,c}$ also depends on the map W_e for that edge.

For graph learning, we use the following six features for every edge $e = (s, t)$ and each context c:

$$a_{s,c}^{\text{cl}}, \quad a_{s,c}^{\text{co}}, \quad g_{t,c}, \quad m_{e,c}^{\text{cl}}, \quad m_{e,c}^{\text{co}}, \quad \widetilde{g}_{e,c}. \tag{13}$$

The first three vectors are the source component's clean state, corrupted state, and the gradient at the target component. The final three are the clean and corrupted contributions along the edge and the target gradient mapped back through that edge's linear map.

These features have a simple transformation law under a compatible reparameterization of the transformer by a change of basis at each component interface, following the general parameter-space symmetries studied for neural networks and transformers [9, 33]. Let A_v be the invertible change of coordinates for component v. If states transform as $a_{v,c} \mapsto A_v a_{v,c}$ and gradients transform as $g_{v,c} \mapsto A_v^{-\top} g_{v,c}$, then the edge operator transforms as $W_e \mapsto A_t W_e A_s^{-1}$ for $e = (s, t)$. The six features therefore transform as

$$\begin{aligned} a_{s,c}^{\text{cl}}, a_{s,c}^{\text{co}} &\mapsto A_s a_{s,c}^{\text{cl}}, A_s a_{s,c}^{\text{co}}, \qquad g_{t,c} \mapsto A_t^{-\top} g_{t,c}, \\ m_{e,c}^{\text{cl}}, m_{e,c}^{\text{co}} &\mapsto A_t m_{e,c}^{\text{cl}}, A_t m_{e,c}^{\text{co}}, \quad \widetilde{g}_{e,c} \mapsto A_s^{-\top} \widetilde{g}_{e,c}. \end{aligned} \tag{14}$$

Thus, the feature vectors are covariant representations of component coordinates. Their covariant roles make basis-invariant scalar combinations available:

$$g_{t,c}^{\top} m_{e,c}^{\mathrm{cl}} = \widetilde{g}_{e,c}^{\top} a_{s,c}^{\mathrm{cl}}, \quad g_{t,c}^{\top} m_{e,c}^{\mathrm{co}} = \widetilde{g}_{e,c}^{\top} a_{s,c}^{\mathrm{co}}. \tag{15}$$

The graph learner could learn to use such invariant combinations from the raw features.

The next subsection describes the two graph transformations that place these features on graph nodes.

B.2 Graph Transformations

Because the prediction target is an edge mask, both transformations produce one classifiable object for every original component-level edge. Let $\bar{e}$ denote the transformed node corresponding to $e \in E_{\mathcal{M}}$. The two transformations expose different relationships among these candidate pathways.

Both transformations are equivariant to a consistent reindexing of the component-level computation graph. For a component reindexing σ, the induced edge reindexing maps $e = (s, t)$ to $\sigma(e) = (\sigma(s), \sigma(t))$. The directed line graph then maps $\bar{e}$ to $\overline{\sigma(e)}$. The incidence graph maps component node v to $\sigma(v)$ and edge node $\bar{e}$ to $\overline{\sigma(e)}$. In each case, the transformed adjacency, relation types, attached features, and circuit labels are relabeled in the same way.

Directed line graph. The directed line graph [13] has node set

$$V_{\mathcal{L}} = \{\bar{e} : e \in E_{\mathcal{M}}\}. \tag{16}$$

For $e_1 = (s_1, t_1)$ and $e_2 = (s_2, t_2)$, the implementation treats e_1 and e_2 as composable if either $t_1 = s_2$ or t_1 is an internal read node and s_2 is the corresponding write node of the same attention head or MLP block. The second case is required because the component-level computation graphs in INTERPBENCH represent an attention head's query, key, or value read separately from its output write, and likewise represent an MLP's input read separately from its output write. Writing this implemented relation as $e_1 \prec e_2$, the directed line-graph adjacency is

$$E_{\mathcal{L}}^{\mathrm{obs}} = \{(\bar{e}_1, \bar{e}_2) : e_1 \prec e_2\}. \tag{17}$$

Thus, the directed line graph has $|E_{\mathcal{M}}|$ nodes, and circuit localization becomes binary node classification.

Incidence graph. The incidence graph [14, 15] retains the original component-level nodes as nodes while adding one node for each original component-level edge:

$$V_{\mathcal{I}} = V_{\mathcal{M}} \dot{\cup} \{\bar{e} : e \in E_{\mathcal{M}}\}. \tag{18}$$

The incidence graph has three families of directed edges that encode the source, target, and next relations:

$$s \xrightarrow{\text{source}} \bar{e}, \qquad \bar{e} \xrightarrow{\text{target}} t, \qquad \bar{e}_1 \xrightarrow{\text{next}} \bar{e}_2 \text{ when } e_1 \prec e_2, \tag{19}$$

for every $e = (s, t)$. Only the component-level edge nodes $\bar{e}$ receive circuit labels and enter the final readout. Component-level nodes carry component-level features and provide intermediate states through which incident edges can exchange information. Writing $n = |V_{\mathcal{M}}|$ and $m = |E_{\mathcal{M}}|$, the incidence graph representation has $n + m$ objects and $2m + |E_{\mathcal{L}}^{\mathrm{obs}}|$ directed edges.

Both transformations place the six features on graph nodes. In the directed line graph, each component-level edge becomes a graph node and receives all six vectors. In the incidence graph, each edge node receives the final three vectors, while each original component node v receives the component-indexed counterparts of the first three vectors evaluated at v itself, namely its clean state $a_{v,c}^{\mathrm{cl}}$, its corrupted state $a_{v,c}^{\mathrm{co}}$, and its own gradient $g_{v,c}$.

B.3 Feature Alignment

The six features described in the previous section naturally have variable widths and semantics as different transformers typically have different hidden dimensions and learn different hidden representations. We therefore use a feature alignment module, adapted from the official TabPFN implementation [16], to map every feature to a fixed-width representation of dimension d_{align}.

Let

$$q^{(\rho)} = \left(q_1^{(\rho)}, \ldots, q_{d_\rho}^{(\rho)}\right) \in \mathbb{R}^{d_\rho} \tag{20}$$

be a feature vector with role ρ and coordinate identifiers $\kappa_1, \ldots, \kappa_{d_\rho}$. Each scalar becomes one fixed-width feature token

$$z_j^{(\rho)} = A_{\text{val}} q_j^{(\rho)} + b_{\text{val}} + c_\psi(\kappa_j) + r(\rho). \tag{21}$$

Here, $A_{\text{val}} q_j^{(\rho)} + b_{\text{val}}$ is a learned scalar-value projection and $r(\rho)$ is a learned role embedding. The coordinate embedding $c_\psi(\kappa_j)$ assigns each graph-wide coordinate identifier a Gaussian random vector and maps that random vector into a fixed-dimensional space through a trainable projection. The six role embeddings $r(\rho)$, one for each of the six features, are learned from random initialization. A summary token is appended to the feature-token sequence, and its transformed representation is the fixed-width aligned representation for that feature:

$$F_\psi\left(q^{(\rho)}; \rho, \boldsymbol{\kappa}\right) \in \mathbb{R}^{d_{\text{align}}}. \tag{22}$$

B.4 Graph Learning Models

For each graph node v and context $c = (b, p)$, let $\phi_{v,c}$ denote the fixed-width representation formed by aligning and concatenating the features attached to that node, using zeros where a feature is absent. Let $\bar{\mathcal{G}}_{\mathcal{M}}$ denote the selected graph representation, either a directed line graph or an incidence graph. For each context $c = (b, p)$, the graph learner processes one copy of $\bar{\mathcal{G}}_{\mathcal{M}}$ with its context-specific features and returns one hidden state for every graph node:

$$\{h_{v,b,p}\}_{v \in V(\bar{\mathcal{G}}_{\mathcal{M}})} = \text{GraphLearner}_\theta\left(\bar{\mathcal{G}}_{\mathcal{M}}, \{\phi_{v,b,p}\}_{v \in V(\bar{\mathcal{G}}_{\mathcal{M}})}\right). \tag{23}$$

The parameters are shared across edges, clean–corrupted prompt pairs, positions, graph sizes, and training cases.

DirGNN. Our first graph learning backend uses PyTorch Geometric's [34, 35] DirGNN [17] construction with a GraphConv aggregator [36], which Figure 2 and Table 4 abbreviate as DirGraphConv. It additively aggregates incoming neighbors in the observed direction and in the reversed direction, and combines the two neighbor results with equal weight $\alpha = 0.5$. Following the block design of Scalable Message Passing Neural Networks [37], each layer applies the directed graph-convolution update and a per-node feed-forward transformation in separate pre-LayerNorm residual branches, with SiLU activations and learned residual scales.

DAGformer. Our second graph learning backend uses DAGformer, a transformer design for directed acyclic graphs [18]. For each graph node, the attention calculation considers only the nodes permitted by the selected graph's directed edges. It assigns weights to those connected nodes using dot products and a softmax, and does not use information from any other graph nodes.

We evaluate a factorized variant of DAGformer in three graph settings: the graph's original directed edges, no message-passing edges, and a complete directed acyclic graph in which every earlier node in a topological order is connected to every later node. We use this factorized variant rather than the quadratic variant for this complete graph because it makes the resulting all-to-all predecessor attention tractable. It approximates softmax attention with random features, following Performer [38] and related linear-attention methods [39]. Rather than calculate attention scores separately along the directed edges that enter each node, it maps queries and keys through a fixed random projection and combines the corresponding keys and values across those incoming edges. For each attention head and graph node i, let $P(i)$ denote the nodes permitted to send information to i. Written in the standard linear-attention convention, in which the output is a row vector rather than the column vector of Section A, the attention output is

$$o_i = \begin{cases} \dfrac{\varphi_q(q_i)^\top \left(\sum_{j \in P(i)} \varphi_{k,P(i)}(k_j) v_j^\top\right)}{\varphi_q(q_i)^\top \left(\sum_{j \in P(i)} \varphi_{k,P(i)}(k_j)\right)}, & P(i) \neq \emptyset, \\ 0, & P(i) = \emptyset. \end{cases} \tag{24}$$

Here, φ_q and $\varphi_{k,P(i)}$ are the query and key feature maps; the key map is normalized separately over $P(i)$ for numerical stability. Both feature maps add a fixed positive constant to every random feature, so the denominator is strictly positive whenever $P(i)$ is nonempty. The second case is stated separately because $P(i)$ can be empty under each of the three connectivity conditions, and the implementation returns exactly zero at such nodes without evaluating the ratio. When we keep the graph's original directed edges, $P(i)$ contains every node with an edge into i, and $P(i)$ is empty at every node with no incoming edge. In the complete directed acyclic graph, $P(i)$ contains every node earlier than i in a topological order, and the two sums are updated while traversing that order rather than materializing every edge from an earlier node to a later node; $P(i)$ is empty at the first node of that order. In the control with all message-passing edges removed, we keep the graph nodes and all of their features but remove every directed edge, so $P(i)$ is empty at every node. Each node's attention output is then zero, and the layer reduces to a transformation applied separately to each node. This control therefore measures the value of message passing on the observed graph.

B.5 Pooling, Readout, and Training

After graph learning, we obtain a representation $h_{e,b,p}$ for every edge $e \in E_{\mathcal{M}}$, retained clean–corrupted prompt pair b, and output position $p \in \mathcal{P}_b$. For either graph transformation, $h_{e,b,p}$ is the state of the graph node associated with e.

We use Deep Sets [19] to pool the context-specific representations for each edge. We first pool representations across the output token positions where the benchmark compares the model's output with the task-specific target, then pool the result across the clean–corrupted prompt pairs used to construct the features. We write $h_{i,e}$ for the representation of edge e in case i after both pooling stages.

A shared linear readout maps each pooled edge representation to one circuit-membership logit

$$\ell_{i,e} = w_{\text{out}}^{\top} h_{i,e} + b_{\text{out}}, \qquad \hat{y}_{i,e} = \sigma(\ell_{i,e}). \tag{25}$$

For training case i, let $C_i \subseteq E_{\mathcal{M}_i}$ be the set of edges in the ground-truth circuit. For each edge $e \in E_{\mathcal{M}_i}$, let $y_{i,e} = 1$ when $e \in C_i$ and $y_{i,e} = 0$ otherwise. Let $m_i = |E_{\mathcal{M}_i}|$, $n_i^{\text{circuit}} = |C_i|$, and $n_i^{\text{other}} = |E_{\mathcal{M}_i} \setminus C_i|$, and define

$$w_i^{\text{circuit}} = \max\left(1, \frac{n_i^{\text{other}}}{\max(n_i^{\text{circuit}}, 1)}\right). \tag{26}$$

The implemented binary cross-entropy-with-logits objective can equivalently be written as

$$\mathcal{L}_i = -\frac{1}{m_i} \sum_{e \in E_{\mathcal{M}_i}} \left[w_i^{\text{circuit}} y_{i,e} \log \sigma(\ell_{i,e}) + (1 - y_{i,e}) \log\left(1 - \sigma(\ell_{i,e})\right) \right]. \tag{27}$$

The mean is therefore taken over all computation-graph edges in the current case, and edges in the ground-truth circuit receive extra weight only when they are fewer than the other edges.

C Adapting PGExplainer to Circuit Localization

Figure 5 illustrates our adaptation of PGExplainer [11] from its original GNN explainability application to circuit localization.

D An Augmented INTERPBENCH Benchmark for Circuit Localization Across Model–Task Pairs

D.1 Creating Additional Model–Task Pairs

TRACRBENCH pairs RASP programs with input–output examples for sequence-processing tasks [24]. It does not provide the low-level transformers trained with Strict Interchange Intervention Training (SIIT) or the query/key/value-separated connection masks required by our INTERPBENCH-based circuit-localization setting. For each candidate task, we compile its RASP program with TRACR to obtain a high-level transformer and its ground-truth circuit. We then train a corresponding low-level

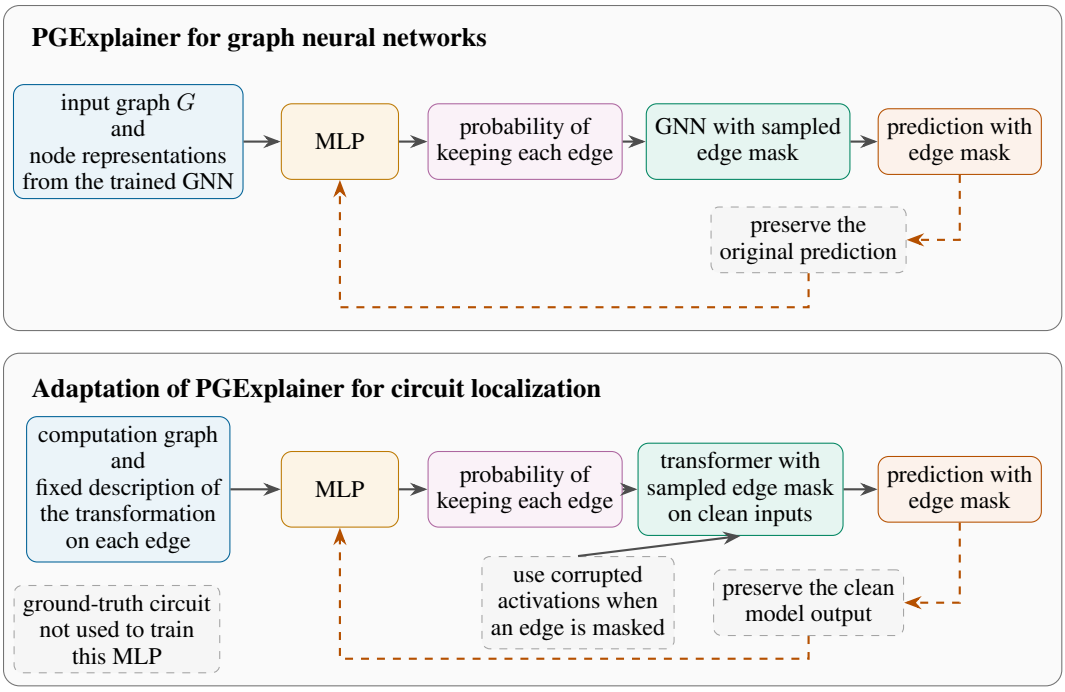


Figure 5: Comparison of the original PGExplainer procedure and our adaptation to circuit localization. In the upper panel, PGExplainer [11] scores each graph edge using representations produced by a trained GNN for the edge's two endpoint nodes. In the lower panel, our adaptation scores each computation-graph edge using a fixed representation of its associated linear transformation. In both settings, the scores define a sampled edge mask applied while the fixed predictor runs. In the transformer adaptation, masking an edge replaces its contribution with the corresponding corrupted activation. Ground-truth circuit annotations are not used to fit an individual explainer.

transformer through INTERPBENCH's SIIT procedure so that it implements the same computation while matching INTERPBENCH's architectural conventions and producing parameter distributions closer to those of conventionally trained models. We convert the resulting ground-truth circuit to the same query/key/value-separated connection-level definition used for INTERPBENCH cases.

We reload each trained transformer from its saved weights and check that it reproduces the source program's behavior on held-out inputs. We exclude candidates whose programs do not compile or whose trained transformers fail this reconstruction check. Of the 48 additional candidate pairs constructed in this way—the model–task pairs from TRACRBENCH that do not already appear in INTERPBENCH—30 pass both the behavior and the compilation checks and form the additional pool reported in Section 4.

D.2 Training and Evaluation Splits That Limit Leakage Across Model–Task Pairs

To test whether methods transfer to new model–task pairs, we keep related pairs on the same side of each training and evaluation split. We treat two model–task pairs as related when they match under at least one of the following checks:

1. **RASP program templates.** After standardizing function, helper, argument, and local names; removing docstrings, decorators, annotations, and type comments; and replacing string and numeric literal values with placeholders, two pairs match when their program templates are identical.
2. **Exact ground-truth circuits.** Two pairs match when their ground-truth circuits contain the same named model components, including components with no circuit connection, and the same directed component edges labeled as part of the circuit.

3. **Circuit structure.** When two circuits are not exact matches, we use directed graph isomorphism. Two pairs match when their circuit graphs can be put into one-to-one correspondence while preserving edge directions and broad component categories, but ignoring individual component names.

We treat these matches as transitive, so any two pairs joined by a sequence of matches belong to the same group.

We apply this split to 114 model–task pairs—84 INTERPBENCH pairs and the 30 additional pairs described in Section D.1. INTERPBENCH also contains two indirect-object-identification cases (IOI and IOI next-token), which are implemented directly as transformer models rather than compiled from RASP programs. We use neither, so 84 of its 86 model–task pairs enter the pool. We reserve the same 16 model–task pairs on which INTERPBENCH evaluated its circuit-localization baselines as our held-out test set. We then remove every group that contains a test pair, which discards 48 further pairs. These include one pair that uses the same program as a test pair and differs from it only in its vocabulary and maximum sequence length, so the first check places the two in the same group. The remaining 50 model–task pairs form 31 groups. We use them for 5-fold cross-validation, assigning each group as a whole to one validation fold and using the other four folds for training. Table 1 reports the size of each fold and of the held-out test set, how many of their pairs come from INTERPBENCH and from TRACRBENCH, and how many groups of related pairs each contains.

Table 1: Model–task pairs used for 5-fold cross-validation and the held-out test set. Each validation fold is used once for validation, while the other four folds are used for training.

Subset	Model–task pairs	Original INTERPBENCH pairs	TRACRBENCH-derived pairs	Groups of related pairs
Cross-validation pool	50	26	24	31
Validation fold 0	11	7	4	6
Validation fold 1	11	4	7	6
Validation fold 2	10	3	7	6
Validation fold 3	9	6	3	6
Validation fold 4	9	6	3	7
Held-out test set	16	16	0	10

Tables 2 and 3 describe the component-level computation graphs and ground-truth circuits represented in the cross-validation pool and held-out test set, in terms of their numbers of nodes and edges.

The three checks above keep the specific kinds of program and circuit similarity they identify from crossing between the held-out test set and the cross-validation pool. They do not give a complete theoretical account of when two model–task pairs should be treated as related; developing one is an important direction for future work.

Table 2: Sizes of the component-level computation graphs in each validation fold and in the held-out test set. Each "median $[Q_1$–$Q_3]$" entry gives the median and the interval from the first to the third quartile across the model–task pairs in that row. Each "min–max" entry gives the full range across the same pairs.

	Number of component nodes		Number of directed component edges	
Set	Median $[Q_1$–$Q_3]$	Min–max	Median $[Q_1$–$Q_3]$	Min–max
Fold 0 ($n = 11$)	38 [38–56]	38–74	110 [110–262]	110–479
Fold 1 ($n = 11$)	56 [56–56]	56–128	262 [262–262]	262–1520
Fold 2 ($n = 10$)	38 [38–38]	38–74	110 [110–110]	110–479
Fold 3 ($n = 9$)	38 [38–56]	38–56	110 [110–262]	110–262
Fold 4 ($n = 9$)	38 [38–56]	38–110	110 [110–262]	110–1108
Held-out test set ($n = 16$)	38 [38–38]	38–74	110 [110–110]	110–479

E Shared-Target Edge Interactions

Here, we motivate the graph-based design of GCL through an analysis of pairwise edge interactions. We first define interactions between arbitrary edge interventions. We then consider the specific case of two edges entering the same component input, derive an exact expression for their interaction, and obtain a local second-order approximation. Finally, we discuss the scope of the result and explain how the graph constructions expose the relevant edge relationships.

Table 3: Sizes of the ground-truth circuits in each validation fold and in the held-out test set. Each "median [Q_1–Q_3]" entry gives the median and the interval from the first to the third quartile across the model–task pairs in that row. Each "min–max" entry gives the full range across the same pairs.

Set	Number of circuit nodes		Number of circuit edges	
	Median [Q_1–Q_3]	Min–max	Median [Q_1–Q_3]	Min–max
Fold 0 ($n = 11$)	5 [5–13]	5–17	3 [3–9]	3–10
Fold 1 ($n = 11$)	13 [13–15]	13–28	9 [9–10]	9–22
Fold 2 ($n = 10$)	7 [2–11]	2–23	5 [1–7.25]	1–17
Fold 3 ($n = 9$)	8 [6–14]	4–18	4 [4–9]	3–12
Fold 4 ($n = 9$)	9 [6–11]	6–34	6 [4–6]	3–26
Held-out test set ($n = 16$)	2 [2–8]	2–14	1 [1–5.75]	1–16

E.1 Pairwise Edge Interactions

Circuit-localization methods commonly assign one scalar importance score to each edge. Such scores may contain contextual information, but they do not explicitly represent whether the effect of one edge intervention depends on another.

Fix a clean–corrupted prompt pair $(x, x') \sim \mathcal{T}$. For a set of jointly patched edges $S \subseteq E_{\mathcal{M}}$, let $\mathbf{y}^{(S)} \in \{0,1\}^{|E_{\mathcal{M}}|}$ denote the mask with $y_e^{(S)} = 0$ if $e \in S$ and $y_e^{(S)} = 1$ otherwise. Using the masked task metric defined in Section 2, the signed effect of jointly patching these edges is

$$\delta_S = L_{\mathcal{T}}\Big(\mathcal{M}, x, x'; \mathbf{y}^{(S)}\Big) - L_{\mathcal{T}}\Big(\mathcal{M}, x, x'; \mathbf{1}\Big). \tag{28}$$

For two edges, we define their interaction as the difference between their joint effect and the sum of their individual effects:

$$I(e_1, e_2) = \delta_{\{e_1,e_2\}} - \delta_{\{e_1\}} - \delta_{\{e_2\}}. \tag{29}$$

If $I(e_1, e_2) = 0$, the joint intervention effect equals the sum of the individual effects. A nonzero value indicates that the effect of patching one edge changes when the other edge is patched as well.

Such non-additivity can arise through different mechanisms. For composable edges, patching a downstream edge may overwrite or screen off an effect propagated from an upstream intervention. For causally parallel edges, neither intervention changes the message naturally produced by the other, but their effects may still interact through subsequent nonlinear computation. We analyze a particularly clear parallel setting in which two messages are added at the same component input.

E.2 Interactions at a Shared Read Input

Fix a token position p in the prompt pair (x, x'). Consider two edges

$$e_1 = (s_1, t), \qquad e_2 = (s_2, t) \tag{30}$$

whose messages enter the same read input of component t at this position. For each edge e_i, let

$$\Delta m_{e_i,p} = m^{\text{co}}_{e_i,p} - m^{\text{cl}}_{e_i,p} \tag{31}$$

denote the change in its transmitted message under patching. The construction of these messages is described in Section B.1.

Because both messages enter the same read input, the clean input to component t can be written as

$$u^{\text{cl}}_{t,p} = m^{\text{cl}}_{e_1,p} + m^{\text{cl}}_{e_2,p} + r^{\text{cl}}_{t,p}, \tag{32}$$

where $r^{\text{cl}}_{t,p}$ contains all remaining contributions to this input. This decomposition assumes that the read-side transformations represented by the component-level computation graph are included consistently in the edge messages. Any remaining read-side nonlinearity is treated as part of the downstream computation.

Patching e_1, e_2, or both therefore changes the shared input to

$$u^{\text{cl}}_{t,p} + \Delta m_{e_1,p}, \qquad u^{\text{cl}}_{t,p} + \Delta m_{e_2,p}, \qquad u^{\text{cl}}_{t,p} + \Delta m_{e_1,p} + \Delta m_{e_2,p}, \tag{33}$$

respectively. This setting is mathematically convenient because the two interventions are fixed additive changes to the same variable.

To isolate the consequences of changing this shared input, let $L_{t,p}(u)$ denote the task metric obtained by replacing the read input of component t at position p by u and continuing the downstream computation normally. Thus, $L_{t,p}$ is the original task metric viewed as a function of one internal model input.

Let $I_p(e_1, e_2)$ denote the interaction between the corresponding interventions localized to position p. I_p is the analogue of Equation (29) for interventions restricted to the position-p read input; patching an edge in Equation (29) changes read inputs at every position, so we study the position-localized quantity directly rather than deriving it from I. Using the three possible patched inputs above, this interaction is:

$$\begin{aligned} I_p(e_1, e_2) = L_{t,p}\left(u^{\mathrm{cl}}_{t,p} + \Delta m_{e_1,p} + \Delta m_{e_2,p}\right) \\ - L_{t,p}\left(u^{\mathrm{cl}}_{t,p} + \Delta m_{e_1,p}\right) - L_{t,p}\left(u^{\mathrm{cl}}_{t,p} + \Delta m_{e_2,p}\right) \\ + L_{t,p}\left(u^{\mathrm{cl}}_{t,p}\right). \end{aligned} \tag{34}$$

This expression measures how far the joint effect of the two message changes differs from the sum of their individual effects.

We assume that $L_{t,p}$ is twice continuously differentiable in a neighborhood containing the intervention surface defined in Equation (41) and that its Hessian is locally Lipschitz. Next, we derive an exact Hessian representation of Equation (34), from which the local approximation in Proposition E.1 follows.

E.3 Exact Interaction Identity

For compactness, let

$$d_1 = \Delta m_{e_1,p}, \qquad d_2 = \Delta m_{e_2,p}, \qquad u_0 = u^{\mathrm{cl}}_{t,p}. \tag{35}$$

We interpolate the strength of the two interventions independently using $s, r \in [0, 1]$ and define

$$\Psi(s, r) = L_{t,p}\left(u_0 + s d_1 + r d_2\right). \tag{36}$$

The four corners of this intervention surface represent the clean computation, the two individual interventions, and the joint intervention:

$$\Psi(0, 0) = L_{t,p}(u_0), \tag{37}$$
$$\Psi(1, 0) = L_{t,p}(u_0 + d_1), \tag{38}$$
$$\Psi(0, 1) = L_{t,p}(u_0 + d_2), \tag{39}$$
$$\Psi(1, 1) = L_{t,p}(u_0 + d_1 + d_2). \tag{40}$$

It follows from Equation (34) that

$$I_p(e_1, e_2) = \Psi(1, 1) - \Psi(1, 0) - \Psi(0, 1) + \Psi(0, 0). \tag{41}$$

Applying the fundamental theorem of calculus with respect to r gives

$$\Psi(1, 1) - \Psi(1, 0) = \int_0^1 \frac{\partial \Psi}{\partial r}(1, r)\, \mathrm{d}r, \tag{42}$$

$$\Psi(0, 1) - \Psi(0, 0) = \int_0^1 \frac{\partial \Psi}{\partial r}(0, r)\, \mathrm{d}r. \tag{43}$$

Subtracting these expressions and applying the fundamental theorem of calculus with respect to s yields

$$I_p(e_1, e_2) = \int_0^1 \int_0^1 \frac{\partial^2 \Psi}{\partial s\, \partial r}(s, r)\, \mathrm{d}s\, \mathrm{d}r. \tag{44}$$

By the chain rule,

$$\frac{\partial^2 \Psi}{\partial s\, \partial r}(s, r) = d_1^\top \nabla_u^2 L_{t,p}\left(u_0 + s d_1 + r d_2\right) d_2. \tag{45}$$

Consequently, the interaction has the exact representation

$$I_p(e_1, e_2) = \int_0^1 \int_0^1 d_1^\top \nabla_u^2 L_{t,p}\left(u_0 + s d_1 + r d_2\right) d_2\, \mathrm{d}s\, \mathrm{d}r. \tag{46}$$

The finite interaction is therefore determined by the mixed curvature of the task metric over the complete two-intervention surface. This exact identity also shows why the two edges must be considered jointly: the relevant term depends on both intervention directions d_1 and d_2.

E.4 Local Approximation

The exact identity depends on the Hessian throughout the intervention surface. For sufficiently small message changes, this Hessian can be approximated by its value at the clean read input. Define

$$Q_{t,p} = \nabla_u^2 L_{t,p}\left(u_{t,p}^{\mathrm{cl}}\right). \tag{47}$$

The matrix $Q_{t,p}$ describes how the sensitivity of the task metric changes locally when the shared read input of component t is varied.

The following proposition states that the leading local pairwise interaction is obtained by applying this curvature to the two message changes.

Proposition E.1 (Shared-target edge interactions)**.** *If the intervention surface remains within a neighborhood on which the Hessian of $L_{t,p}$ is Lipschitz, then*

$$I_p(e_1, e_2) = \Delta m_{e_1,p}^\top Q_{t,p} \Delta m_{e_2,p} + O\Big(\left(\|\Delta m_{e_1,p}\| + \|\Delta m_{e_2,p}\|\right)^3\Big). \tag{48}$$

The proposition has a direct interpretation. The first term measures whether the downstream computation responds differently when the two incoming messages are changed together rather than separately. If this term is nonzero, the leading local interaction is second-order and depends on the identities of both incoming edges.

Proof. From Equation (46),

$$I_p(e_1, e_2) = \int_0^1 \int_0^1 d_1^\top \nabla_u^2 L_{t,p}\left(u_0 + s d_1 + r d_2\right) d_2 \,\mathrm{d}s \,\mathrm{d}r. \tag{49}$$

Because the Hessian is Lipschitz on the intervention surface,

$$\nabla_u^2 L_{t,p}\left(u_0 + s d_1 + r d_2\right) = Q_{t,p} + O(s\|d_1\| + r\|d_2\|). \tag{50}$$

Substituting this expression into the exact identity gives

$$\begin{aligned} I_p(e_1, e_2) = &\int_0^1 \int_0^1 d_1^\top Q_{t,p} d_2 \,\mathrm{d}s \,\mathrm{d}r \\ &+ O(\|d_1\|\|d_2\|\left(\|d_1\| + \|d_2\|\right)). \end{aligned} \tag{51}$$

The leading integrand is constant in s and r, so

$$\int_0^1 \int_0^1 d_1^\top Q_{t,p} d_2 \,\mathrm{d}s \,\mathrm{d}r = d_1^\top Q_{t,p} d_2. \tag{52}$$

Furthermore,

$$\|d_1\|\|d_2\|\left(\|d_1\| + \|d_2\|\right) = O\Big(\left(\|d_1\| + \|d_2\|\right)^3\Big). \tag{53}$$

Restoring the original notation proves Equation (48). □

If the mixed second-order term in Equation (48) is nonzero, it is the leading local interaction. If it vanishes, the leading interaction may occur at third or higher order. For larger message changes, the exact identity in Equation (46) remains valid, but the clean-point Hessian may no longer provide an accurate approximation.

The smoothness assumption makes Proposition E.1 a conditional local illustration rather than a general statement. It does not apply directly at ReLU-style kinks, or to L_1, argmax, and other nonsmooth or discrete task metrics. The finite-difference definition in Equation (34) needs no such assumption and remains meaningful in those cases.

E.5 Implications for Circuit Localization and Graph Learning

Proposition E.1 shows that the first potentially nonzero local interaction between two edges entering the same read input is second-order. The matrix $Q_{t,p}$ describes how the downstream computation responds nonlinearly to changes in the shared read input. It depends on the shared read input, token position, and downstream computation, but not on the particular pair of incoming edges. The

pair-specific information is instead contained in the message changes $\Delta m_{e_1,p}$ and $\Delta m_{e_2,p}$. This structure is well suited to parameter sharing: a GNN can apply a common update rule across the graph while adapting its output to the features, shared components, and neighborhoods of individual edges. The learner can therefore reuse interaction patterns across edge pairs, computation graphs, and model–task cases.

Vanilla EAP estimates edge importance using a first-order Taylor approximation around the clean computation and therefore does not include this explicit mixed term [1]. EAP-IG can incorporate curvature and higher-order effects by evaluating gradients between corrupted and clean inputs, but allocates these effects among scalar edge scores rather than identifying interactions between particular pairs [2]. The Integrated Hessians method instead attributes pairwise feature interactions explicitly through mixed derivatives [40].

This distinction matters when an edge interacts with one pathway but not another, or when multiple pathways are redundant. Redundant OR-like mechanisms may be only partially recovered by standard circuit-discovery procedures [41]. Related dependencies appear in backup behavior and self-repair [42–44], as well as in copy suppression, whose effect depends on information written by earlier components [45]. These phenomena are not direct instances of Proposition E.1, but they demonstrate more generally that pathway relevance can depend on the surrounding computation.

A graph learner provides a direct mechanism for modeling such dependencies by exchanging representations between computationally related edges. The directed line graph and incidence graph expose these relationships differently. Composable edges are directly adjacent in the directed line graph. However, two edges entering the same target do not compose and are therefore not directly adjacent. If their shared target t has an outgoing edge $e_3 = (t, u)$, the directed line graph contains

$$\bar{e}_1 \to \bar{e}_3, \qquad \bar{e}_2 \to \bar{e}_3. \tag{54}$$

With bidirectional message passing, information can travel between the incoming edges through $\bar{e}_3$ in two message-passing steps.[1] In the incidence graph, both incoming edge nodes are incident to their shared component node t. With bidirectional propagation over the incidence relations, information can travel along

$$\bar{e}_1 \to t \to \bar{e}_2 \tag{55}$$

in two message-passing steps.

Once both edge representations lie within the same receptive field, the graph learner can condition the prediction for one edge directly on the observed features of the other. Thus, the graph structure provides an inductive bias by restricting this exchange to computationally related pathways rather than treating every possible pair of transformer edges as equally relevant.

F Additional Experimental Details, Results, and Ablations

Our implementation is openly available at https://github.com/graph-learning-circuits/graph-learning-circuits.

F.1 Additional Experimental Details

Here, we explain how we evaluate each method on the 16 held-out model–task pairs and how we choose hyperparameters and training duration using the cross-validation pool of 50 model–task pairs. See Section 4 for a description of the benchmark pairs

We evaluate circuit recovery separately on each held-out pair. Two score granularities are involved: GCL assigns separate scores to the query, key, and value input connections of each attention head because they represent distinct computational pathways, whereas INTERPBENCH evaluates these pathways at the level of the parent attention head. During training and validation, our binary cross-entropy (BCE) loss and AUROC operate at the query/key/value-connection level. For held-out evaluation, we apply the score-level equivalent of the official INTERPBENCH head-promotion rule. Specifically, each head-level connection receives the maximum of its corresponding query, key, and

[1] DirGNN aggregates along both directions and can realize this exchange. DAGformer variants attend only along the directed edges, so under the observed connectivity the two incoming edge representations do not reach one another.

value input-connection scores. We compute AUROC on these converted scores for comparability with the baselines reported by INTERPBENCH. Held-out results are averaged over 5 seeds within each model–task pair and then summarized across the 16 pairs.

For each GCL configuration, we compare a grid of 18 hyperparameter settings using grouped 5-fold cross-validation. For every setting and fold, we train on the other 4 folds and record validation AUROC on the remaining fold at regular intervals. Validation AUROC scores the query, key, and value input connections directly, without the head-level conversion used for held-out evaluation. Some settings did not complete all 5 folds because of out-of-memory errors, so the number of settings eligible for selection ranges from 9 to 18 across the 14 configurations. We score each completed setting at every training step where all 5 folds have a validation measurement, using a weighted average of the 5 per-fold scores. The weight of each fold is the number of groups of related cases it contains (Section D.2). We select the setting and training duration that jointly maximize this weighted average, breaking ties in favor of the earlier step. With the selected setting and duration, we reinitialize the predictor and train it on all 50 model–task pairs using 5 random seeds.

For every model–task pair and random seed, our PGExplainer adaptation fits a newly initialized MLP using only that pair's prompts and model outputs. To choose hyperparameters, we compare a grid of 18 settings using the same grouped 5-fold cross-validation procedure as for GCL. The ground-truth circuit masks of the 50 cross-validation pairs score the candidate settings but never enter an individual fit. With the selected setting, we fit the adaptation separately on each of the 16 held-out model–task pairs using 5 random seeds, and their ground-truth masks are used only for evaluation.

F.2 Additional Experimental Results and Ablations

Table 4 reports cross-validation selection scores and held-out edge-AUROC summaries for the 14 GCL configurations, our PGExplainer adaptation, and the published INTERPBENCH baselines. We chose the configuration highlighted in Figure 2 after evaluating all 14 on the held-out cases. Its median of 0.902 is the best result we observed, not one we could have predicted in advance, and should be read as a post-selection estimate. Cross-validation ranks this configuration second of the 14. Selecting on cross-validation scores alone would have chosen the DirGNN incidence-graph variant, whose held-out median is 0.876.

G Computational Costs

We use the standard work–span model of parallel complexity [46]. For a method X, W_X is its total work in terms of the total amount of computation required, and S_X is its span in terms of the number of sequential steps along the longest chain of dependencies.

We compare the work and span required to localize a circuit for one new model–task pair. For Graph Circuit Learning (GCL), this is its amortized inference cost after the shared predictor has been trained. We omit the cost of model selection or predictor training.

Let $(W_{\text{feat}}, S_{\text{feat}})$ be the fixed GCL feature construction for one pair, and let $(W_{\text{pred}}, S_{\text{pred}})$ be feature alignment, graph transformation, one trained-predictor evaluation, pooling, and readout. Let W_{EAP} be the work of one EAP calculation. W_{feat} and W_{EAP} have the same asymptotic order of target-model work for matched support sizes, because each runs a fixed number of forward and backward passes of the target model per prompt pair. Thus, GCL has per-pair costs:

$$(W_{\text{GCL}}, S_{\text{GCL}}) = (W_{\text{feat}} + W_{\text{pred}}, S_{\text{feat}} + S_{\text{pred}})\,. \tag{56}$$

EAP contracts its attribution quantities directly into edge scores [1], whereas GCL adds the predictor cost in Equation (56). Therefore, we do not claim a target-model-cost advantage over EAP.

EAP-IG averages EAP attribution across K evaluation points along the path from corrupted to clean activations, including the endpoints [2]. Let $(W_{\text{IG,pre}}, S_{\text{IG,pre}})$ and $(W_{\text{IG,reduce}}, S_{\text{IG,reduce}})$ be its shared preprocessing and final reduction costs, respectively. Writing (W_k, S_k) for the cost at integration point k, its work and ideal parallel span are

$$\begin{aligned} W_{\text{IG}} &= W_{\text{IG,pre}} + \sum_{k=1}^{K} W_k + W_{\text{IG,reduce}}, \\ S_{\text{IG}}^{\text{parallel}} &= S_{\text{IG,pre}} + \max_k S_k + S_{\text{IG,reduce}}, \quad S_{\text{IG,reduce}} = \mathcal{O}(\log K). \end{aligned} \tag{57}$$

Method	Graph transformation	Connectivity	Cross-validation Mean AUROC	Held-out test Min.	Q1	Median	Q3	Max.
Graph Circuit Learning								
Factorized DAGformer	Line	Observed	0.890	0.605 ± 0.033	0.771 ± 0.022	0.849 ± 0.014	0.962 ± 0.003	0.988 ± 0.006
Factorized DAGformer	Line	No message-passing edges	0.884	0.525 ± 0.062	0.710 ± 0.027	0.833 ± 0.011	0.924 ± 0.027	0.992 ± 0.007
Factorized DAGformer	Line	Complete DAG	0.912	0.606 ± 0.045	0.820 ± 0.031	0.871 ± 0.015	0.922 ± 0.012	0.963 ± 0.006
Factorized DAGformer	Incidence	Observed	0.910	0.699 ± 0.019	0.816 ± 0.017	0.887 ± 0.018	0.914 ± 0.023	0.965 ± 0.010
Factorized DAGformer	Incidence	No message-passing edges	0.819	0.380 ± 0.098	0.490 ± 0.132	0.506 ± 0.136	0.546 ± 0.135	0.638 ± 0.103
Factorized DAGformer	Incidence	Complete DAG	0.914	0.574 ± 0.091	0.806 ± 0.025	0.872 ± 0.019	0.915 ± 0.017	0.965 ± 0.017
DirGraphConv	Line	Observed	0.943	0.712 ± 0.009	0.861 ± 0.015	0.902 ± 0.011	0.942 ± 0.008	0.996 ± 0.005
DirGraphConv	Line	No message-passing edges	0.883	0.557 ± 0.045	0.741 ± 0.009	0.825 ± 0.012	0.964 ± 0.015	0.992 ± 0.004
DirGraphConv	Incidence	Observed	0.949	0.748 ± 0.012	0.824 ± 0.008	0.876 ± 0.018	0.936 ± 0.005	0.996 ± 0.005
DirGraphConv	Incidence	No message-passing edges	0.833	0.628 ± 0.020	0.760 ± 0.018	0.846 ± 0.007	0.961 ± 0.021	0.990 ± 0.006
Quadratic DAGformer	Line	Observed	0.899	0.675 ± 0.038	0.788 ± 0.033	0.831 ± 0.023	0.922 ± 0.027	0.976 ± 0.006
Quadratic DAGformer	Line	No message-passing edges	0.883	0.504 ± 0.033	0.718 ± 0.011	0.825 ± 0.026	0.919 ± 0.014	0.992 ± 0.004
Quadratic DAGformer	Incidence	Observed	0.908	0.623 ± 0.052	0.802 ± 0.033	0.879 ± 0.035	0.916 ± 0.034	0.976 ± 0.008
Quadratic DAGformer	Incidence	No message-passing edges	0.819	0.677 ± 0.201	0.716 ± 0.073	0.821 ± 0.060	0.884 ± 0.066	0.898 ± 0.056
GNN explainer								
PGExplainer			0.732	0.668 ± 0.041	0.743 ± 0.026	0.858 ± 0.011	0.986 ± 0.004	0.998 ± 0.000
Published INTERPBENCH *baselines*								
ACDC			—	0.441	0.839	0.959	0.987	1.000
EAP-IG			—	0.412	0.766	0.910	0.987	1.000
Edge-wise Subnetwork Probing			—	0.314	0.764	0.884	1.000	1.000
Node-wise Subnetwork Probing			—	0.157	0.678	0.852	0.928	0.971
EAP			—	0.000	0.000	0.000	0.393	0.990

Table 4: Cross-validation selection scores and edge-AUROC summaries across the 16 held-out test cases. The cross-validation column is the selection score defined in Section F.1, computed on the 50 pairs from a single seed, so it carries no interval. It scores at the connection level, where an attention head's query, key, and value inputs are separate edges, whereas the held-out columns use the official INTERPBENCH evaluation, which scores those inputs at the level of the parent attention head. The two groups of columns are therefore not comparable. Each GCL and our PGExplainer adaptation held-out entry summarizes 16 case scores after averaging 5 seeds within each case. The value after $\pm$ is the standard deviation of that statistic across seeds. The "No message-passing edges" rows remove every message-passing edge while retaining every node and all of its features. For the line-graph rows this isolates the value of message passing on the observed computation graph. For the incidence-graph rows, the comparison does not hold the usable input information fixed. In the incidence graph, three of the six feature roles are stored on component nodes, so removing the message-passing edges also removes the edge readout's access to them. Baseline values are taken from Figure 9 of the INTERPBENCH paper [12].

For $X \in \{\mathrm{PG}, \mathrm{SP}\}$, let U_X be the number of optimizer updates used to fit our PGExplainer adaptation or Subnetwork Probing instance for one pair. Its work and span are

$$W_X = W_{X,\mathrm{init}} + \sum_{u=1}^{U_X} W_{X,u} + W_{X,\mathrm{out}}, \qquad S_X = S_{X,\mathrm{init}} + \sum_{u=1}^{U_X} S_{X,u} + S_{X,\mathrm{out}}. \tag{58}$$

Our PGExplainer adaptation updates MLP parameters, whereas Subnetwork Probing updates a directly parameterized mask. Within each update, all mask variables are optimized jointly through a common objective. In both cases, update $u+1$ depends on the parameters produced by update u, so the updates are sequential [5, 11].

At a fixed ACDC threshold τ, let N_τ be the number of edges examined along its greedy trajectory, and let $(W_{\tau,j}, S_{\tau,j})$ be the cost of the intervention for the j-th examined edge. Its work and span are

$$W_{\mathrm{ACDC}}(\tau) = W_{\mathrm{pre}}(\tau) + \sum_{j=1}^{N_\tau} W_{\tau,j}, \qquad S_{\mathrm{ACDC}}(\tau) = S_{\mathrm{pre}}(\tau) + \sum_{j=1}^{N_\tau} S_{\tau,j}. \tag{59}$$

The intervention costs can differ by edge, and each intervention is evaluated against the circuit state produced by earlier accepted removals, so later decisions depend on earlier ones [7].

GCL performs target-model tracing with the same asymptotic order of target-model work as EAP. Unlike EAP-IG, it does not require repeated integration-point evaluations. Unlike our PGExplainer adaptation and Subnetwork Probing, it does not require a per-pair optimization loop. Unlike ACDC, it does not follow a sequential greedy intervention trajectory. This comparison characterizes marginal algorithmic work and dependency structure rather than establishing a universal wall-clock or total-cost advantage. Such an advantage also depends on predictor-training amortization, the scaling of (W_{pred}), hardware, batching, and model and graph sizes.

H Extended Related Work

Circuit localization methods. Circuit localization identifies a subgraph or *circuit* of a model's computation graph that is sufficient to reproduce a particular behavior. Automated methods for circuit localization reduce the manual effort required to select components or connections. ACDC tests removals of individual edges with causal interventions, whereas EAP and EAP-IG estimate edge effects from gradients [1, 2, 7]. Subnetwork Probing and Edge Pruning instead optimize directly parameterized circuit masks [4, 5]. These methods estimate or optimize a new circuit for each model–task pair and therefore provide the most direct baselines for Graph Circuit Learning.

Recent methods have also applied learning to circuit localization, although they differ in what is learned and whether it is reused. MechRL trains a policy with rewards from causal interventions [47]. Its learned policy can be applied to unseen behaviors without further training, whereas its warm-start variant adapts the policy to the new behavior. Differentiable Faithfulness Alignment learns to map node-importance scores from a source model to a target model, using a differentiable faithfulness objective on the target [48]. Universal Circuits jointly learn an aligned feature space across vision transformers and use it to identify class circuits that are shared across models or specific to individual models [49]. The alignment is learned jointly for the models under study, and the resulting circuits are defined over learned features rather than component connections. CircuitLasso does not reuse a localization model across cases. It fits a sparse regression model to activations from one model and task, producing circuits over neurons or sparse-autoencoder features [50].

By contrast, Graph Circuit Learning (GCL) is trained on synthetic model–task pairs with known circuits and then applied to unseen pairs. Beyond these differences in supervision and transfer, our study asks whether message passing over computation-graph structure improves circuit localization. This question motivates the connections to GNN explainers and weight-space learning.

GNN explainers. Post-hoc GNN explainers identify which part of an input graph supports a prediction made by a fixed graph neural network. GNNExplainer optimizes a subgraph for each queried prediction. SubgraphX searches candidate subgraphs for a given query [51, 52]. PGExplainer trains one edge-mask generator across inputs, allowing it to explain a new input without another round of mask fitting [11]. GraphMask likewise learns reusable gates that remove messages from a fixed GNN while preserving its predictions [53]. Gem derives training targets from the effects of edge removals and then trains a generator that explains new inputs [54].

Among these methods, PGExplainer provides a parameterized edge scorer and therefore offers a useful template for adapting graph explainability methods to circuit localization. Unlike GCL, however, our adaptation is fitted independently to each model–task pair.

Weight-space learning. Weight-space learning treats neural weights as a meaningful domain for analysis and modeling [9]. Permutation Equivariant Neural Functionals process model weights while respecting the fact that hidden units can be reordered without changing the represented function [10]. Graph Metanetworks encode a neural network as a graph, allowing one metanetwork to process different architectures while respecting parameter reorderings that leave the represented function unchanged [8]. Such methods describe or modify the network as a whole, for example by predicting a property or editing its parameters. GCL adopts the same broad view of a trained network as structured data, but adds activations and gradients from a specified task. Its output is one score for each candidate circuit connection rather than a prediction or transformation at the level of the whole network.

Foundation models and cross-domain transfer. Prior-data fitted networks provide a related example of training once across many tasks and applying the resulting predictor to unseen tasks. TabPFN is pretrained on millions of synthetic tabular datasets sampled from a prior, then uses labeled examples from an unseen dataset as context to predict its unlabeled examples without fitting a new model [16]. Graph foundation models pursue a related form of reuse across graph domains. GFT pretrains a transferable vocabulary of computation-tree patterns for downstream graph tasks, while Finkelshtein et al. construct graph foundation models with node- and label-permutation equivariance and feature-permutation invariance and evaluate them in zero-shot node classification across graphs with different features and labels [55, 56].

Program-based and formal mechanistic interpretability. Restricted Access Sequence Processing (RASP) expresses sequence algorithms through operations that transformers can implement, and TRACR compiles RASP programs into transformer weights with known internal structure [22, 23]. TRACRBENCH provides a large collection of validated RASP programs and compiled models. INTERPBENCH trains models to preserve program-derived circuits while giving them weights and activations closer to those of conventionally trained models [12, 24]. Together, these controlled models make it possible to study whether circuit labels can be learned across cases rather than inferred anew for every case.

Other work seeks to recover a program rather than a circuit. Transformer Programs restrict the model and its training procedure so that the trained network can be converted into a discrete program [57]. Neural Decompiling trains a model to recover complete RASP programs from TRACR weights [58]. Huang et al. [59] instead reparameterize a trained transformer as a RASP program and use causal interventions to isolate a small sufficient subprogram, recovering interpretable programs from small transformers trained on algorithmic and formal-language tasks. Formal approaches pursue stronger guarantees. Gross et al. [60] use a mechanistic interpretation of a model to prove lower bounds on its accuracy. Hadad et al. [61] use neural network verification to discover circuits with provable guarantees: agreement with the model over a continuous input region, robustness of that agreement under patching perturbations, and minimality. Both lines of work currently operate on small models, because verification and proof techniques are challenging to scale to modern transformers. These approaches certify behavior of either the whole model or a discovered circuit rather than supplying a unique ground-truth circuit. To the best of our knowledge, benchmarks whose circuits are known by construction provide the clearest established source of ground-truth supervision for circuit localization across model–task pairs.

I Generative AI Usage

Table 5 summarizes our use of generative AI assistants. We take full responsibility for all content in this work.

AI assistant from . . .	**Usage**
Anthropic	Theoretical analysis
OpenAI	Computational complexity analysis
Anthropic & OpenAI	Design dataset splits
Anthropic & OpenAI	Generate code for experiments
Anthropic & OpenAI	Generate figures and tables
Anthropic & OpenAI	Proofread the final manuscript

Table 5: Generative AI Usage. From Anthropic, we used Opus 4.8, Opus 5, and Fable 5. From OpenAI, we used GPT 5.5, GPT-5.6-Sol, and GPT-5.6-Terra. We independently verified all output from the AI assistants and made all final methodological and interpretive decisions.