SCALING SPARSE AUTOENCODER CIRCUITS FOR IN-CONTEXT LEARNING

Dmitrii Kharlapenko*
ETHZ
dkharlapenko@ethz.ch

Stepan Shabalin*
Georgia Institute of Technology
sshabalin3@gatech.edu

Neel Nanda, Arthur Conmy
Joint senior authors

ABSTRACT

Sparse autoencoders (SAEs) are a popular tool for interpreting large language model activations, but their utility in addressing open questions in interpretability remains unclear. In this work, we demonstrate their effectiveness by using SAEs to deepen our understanding of the mechanism behind in-context learning (ICL). We identify abstract SAE features that (i) encode the model's knowledge of which task to execute and (ii) whose latent vectors causally induce the task zero-shot. This aligns with prior work showing that ICL is mediated by task vectors. We further demonstrate that these task vectors are well approximated by a sparse sum of SAE latents, including these task-execution features. To explore the ICL mechanism, we adapt the sparse feature circuits methodology of Marks et al. (2024) to work for the much larger Gemma-1 2B model, with 30 times as many parameters, and to the more complex task of ICL. Through circuit finding, we discover task-detecting features with corresponding SAE latents that activate earlier in the prompt, that detect when tasks have been performed. They are causally linked with task-execution features through the attention and MLP sublayers.

1 INTRODUCTION

Sparse autoencoders (SAEs) have emerged as a promising method for interpreting large language model (LLM) activations (Ng, 2011; Bricken et al., 2023; Cunningham et al., 2023). However, current SAE research typically focuses on either analyzing individual features or performing high-level interventions without examining downstream effects. In this work, we demonstrate SAEs' broader potential by using them to interpret in-context learning (ICL), a fundamental LLM capability that enables models to adapt to new tasks from examples alone (Brown et al., 2020).

Recent work has shown that ICL behaviors can be captured by task vectors - internal representations that can be extracted and used to induce zero-shot task performance (Todd et al., 2024; Hendel et al., 2023). However, these vectors were difficult to interpret naively using SAEs. We address this challenge by developing the Task Vector Cleaning (TVC) algorithm, which decomposes task vectors into interpretable sparse features while preserving their functional properties.

Applying TVC to the Gemma-1 2B model (Team et al., 2024), we identify two key components of the ICL mechanism: task-execution features that implement specific operations, and task-detection features that identify which operation to perform. By extending the Sparse Feature Circuits methodology (Marks et al., 2024), we demonstrate how these components interact through attention and MLP layers to enable ICL behavior.

Our main contributions are:

1. We scale sparse feature circuit finding to Gemma-1 2B, a model **10 – 35× larger** than those previously studied with comparable depth in mechanistic interpretability (Wang et al., 2022; Marks et al., 2024).

*Equal contribution.

2. We identify and characterize two core ICL circuit components: task-detection features and task-execution features, revealing how they interact to process information across the prompt.
3. We develop TVC, a novel sparse decomposition method that enables precise analysis of task vectors and their constituent features.

2 BACKGROUND

2.1 SPARSE AUTOENCODERS (SAES)

Sparse autoencoders (SAEs) are neural networks designed to learn efficient representations of data by enforcing sparsity in the hidden layer activations (Elad, 2010). In the context of language model interpretability, SAEs decompose high-dimensional activations into interpretable features (Cunningham et al., 2023; Bricken et al., 2023). The encoding step is as follows, with $\mathbf{f}$ denoting the pre-activation features and $\mathbf{W}_{\text{enc}}$ and $\mathbf{b}_{\text{enc}}$ the encoder weights and biases respectively:

$$\mathbf{f}(\mathbf{x}) = \sigma(\mathbf{W}_{\text{enc}}\mathbf{x} + \mathbf{b}_{\text{enc}}) \tag{1}$$

For JumpReLU SAEs (Rajamanoharan et al., 2024b), the activation function and decoder are (with H being the Heaviside step function, θ the threshold parameter and $\mathbf{W}_{\text{dec}}$/$\mathbf{b}_{\text{dec}}$ the decoder affine parameters):

$$\hat{\mathbf{x}}(\mathbf{f}) = \mathbf{W}_{\text{dec}}(\mathbf{f} \odot H(\mathbf{f} - \theta)) + \mathbf{b}_{\text{dec}} \tag{2}$$

In our work, we train SAEs on residual stream activations and attention outputs, and also train transcoders on MLP layers using the improved Gated SAE architecture (Rajamanoharan et al., 2024a).

2.2 SPARSE FEATURE CIRCUITS

Sparse Feature Circuits (SFCs) (Marks et al., 2024) identify causal subgraphs of SAE features that explain specific model behaviors. The method involves decomposing model activations using SAEs, calculating their Indirect Effect (IE) on target behaviors, and analyzing connections between causally relevant features. In practice, attribution patching (Syed et al., 2023) approximates IEs for efficient computation across many components.

2.3 TASK VECTORS

Task vectors capture the essence of a task demonstrated in a few-shot prompt, allowing the model to apply this learned task to new inputs without explicit fine-tuning (Hendel et al., 2023; Todd et al., 2024). They can be extracted from the model's hidden states and, when added to the model's activations in a zero-shot setting, can induce task performance without explicit context. Consider the following prompt (Example 1) for an antonym task, where boxes represent distinct tokens:

BOS	Follow	the	pattern	:	\n
hot	→	cold	\n		
big	→	small	\n		
fast	→	slow			

Example 1: All token types in an example input: prompt, input, arrow, output, newline (target tokens for calculating the loss on included).

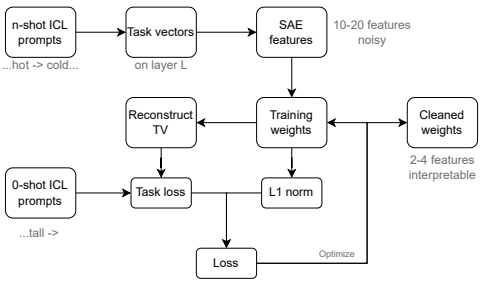


Figure 1: Overview of the task vector cleaning algorithm (see Figure 7; TV stands for task vector).

Task vectors are collected by averaging the residual stream of → tokens at a specific layer across multiple ICL prompts for a given task. For our experiments, we selected layer 12 as the target layer in the Gemma 1 2B model, where task vectors showed the strongest effects.

3 Discovering Task-Execution Features

3.1 Decomposing task vectors

Initial attempts to decompose task vectors using direct SAE reconstruction or inference-time optimization (Smith, 2024) encountered significant limitations. Direct reconstruction produced noisy results with excessive non-zero features that reduced task performance, while ITO failed to maintain effect on loss with sparse feature sets. To address these challenges, we developed the **Task Vector Cleaning (TVC)** algorithm. This novel method optimizes SAE decomposition weights $\theta \in \mathbb{R}^{d_{SAE}}$ through a three-step process:

1. Initialize weights from standard SAE decomposition
2. Reconstruct the task vector and measure performance on zero-shot prompts
3. Optimize weights using loss function $\mathcal{L} = \mathcal{L}_{NLL}(\theta) + \lambda\|\theta\|_1$

Figure 2: Performance comparison of reconstruction methods across layers, showing TVC maintains effectiveness through layer 14.

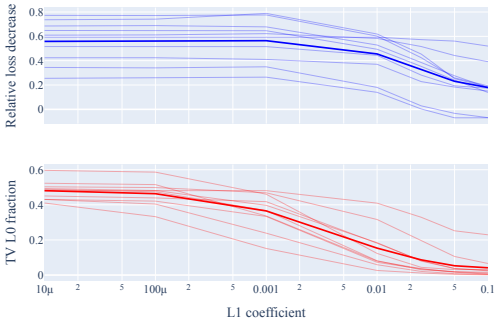


Figure 3: TVC evaluation across L_1 coefficients, showing consistent feature reduction while maintaining performance.

Extensive evaluation across multiple model scales and architectures demonstrated that TVC consistently reduces active features by 50-80% while preserving or improving task performance (Figure 3). The algorithm revealed interpretable "task-execution features" characterized by anticipatory activation patterns - they activate on arrow tokens right before the task completion (89.8% of activation mass, details in Appendix F). They can also partially replace task vectors themselves.

3.2 Steering Experiments

To validate our executing features' causal relevance, we conducted steering experiments on zero-shot prompts using features extracted by our cleaning algorithm. The results (Figure 4) show strong task specificity - most tasks have a single highly effective feature that minimally affects unrelated tasks. Features from related tasks, such as translations, show partial cross-task effects, suggesting shared mechanisms. Detailed results across multiple models are available in Appendix F.

4 Applying SFC to ICL

When applying SFC to analyze the ICL circuit, we aggregated Indirect Effects (IEs) over meaningful token categories (prompt , input , arrow , output , newline) from Example 1. This aggregation enabled us to better understand how features influenced different components of the ICL prompt structure.

Our evaluations demonstrated that the modified approach successfully scales to larger models while maintaining the ability to identify task-specific circuits. Using circuits of 500 nodes, we achieved an

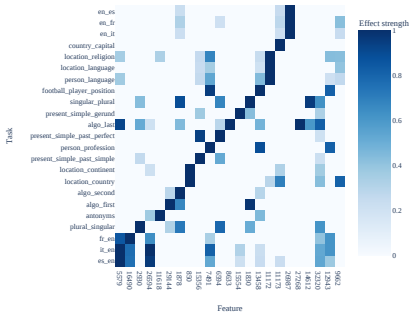


Figure 4: Heatmap showing the effect of steering with individual task-execution features for each task. Most features boost exactly one task, with a few exceptions for similar tasks like translating to English. Full and unfiltered versions of the heatmap are available in Appendix F.

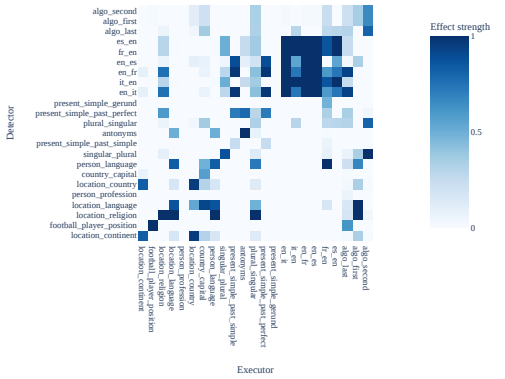


Figure 5: Heatmap showing the causal effect of the top task-detection features of each task, on the activation of the top task-execution features for every task. Averaged across all initial non-zero activations in all tasks.

average faithfulness of 0.6 across tasks, with strong task specificity evident in cross-task ablation studies (see Appendix E.2 for detailed results).

4.1 Task-Detection Features

Our SFC analysis revealed a second crucial component of the ICL mechanism: task-detection features. Unlike executor features that activate before task completion, these features activate specifically on output tokens where tasks are completed in the training data. We identified layer 11 as optimal for these features, preceding the layer 12 task-execution features (Figure 2).

To validate the causal relationship between detection and execution features, we conducted ablation experiments, matching the strongest features of each type based on their steering effects. Our analysis (Figure 5) showed that disabling task-detection features significantly reduced the activation of corresponding task-execution features, confirming their interdependence. Detailed results and token type activation patterns (Table 2) are available in Appendix G.

5 Related work

Mechanistic Interpretability Mechanistic interpretability studies how neural networks process information through identifiable features and circuits (Olah et al., 2020). Features represent meaningful directions in the network's latent space, while circuits are interpretable computation subgraphs formed by feature interactions. While early work focused on manual circuit discovery in vision models (Cammarata et al., 2020), recent advances have enabled automated discovery in language models through patching techniques (Wang et al., 2022) and sparse autoencoders (Marks et al., 2024).

In-Context Learning (ICL) In-context learning enables models to adapt to new tasks using only prompt examples (Brown et al., 2020). While early work identified induction heads as a key mechanism (Olsson et al., 2022), recent research shows they are insufficient to explain complex task behaviors. Particularly relevant to our work, Hendel et al. (2023) and Todd et al. (2024) discovered task vectors - strong directional signals that enable zero-shot task performance, though their internal composition remained unexplained. Recent findings by Park et al. (2024) demonstrate that language models adapt to new tasks by reorganizing existing object representations, suggesting our task execution features may manifest differently across various task types.

Sparse Autoencoders Sparse autoencoders (SAEs) address the challenge of superposition in neural networks, where interpretable features are misaligned with network directions (Elhage et al., 2022). Recent advances have improved SAE training (Rajamanoharan et al., 2024b) and enabled their

application to circuit discovery (Cunningham et al., 2023). Our work builds on these foundations, particularly incorporating transcoders (Dunefsky et al., 2024) for analyzing MLP circuits in the Gemma-1 model.

6 Reproducibility Statement

We are committed to fostering reproducibility and advancing research in the field of mechanistic interpretability. To support this goal, we plan to release the following resources upon successful acceptance of this paper:

1. Two JAX libraries optimized for TPU:
   - A library for Sparse Autoencoder (SAE) training
   - A library for SAE inference and model analysis, built upon Penzai with our custom Llama and Gemma ports
2. A full suite of SAEs for Gemma 2B, along with a dataset of their max activating examples
3. Two custom dashboards used in our analysis:
   - A dashboard for browsing max activating examples
   - An interactive dashboard for exploring extracted Sparse Feature Circuits (SFC)

These resources will enable researchers to replicate our experiments, extend our work, and conduct their own investigations using our tools and methodologies. The release of our custom dashboards will provide additional transparency and facilitate a deeper exploration of our results. Due to the complexity of our infrastructure, we only share anonymized versions of our analysis, cleaning, and SFC scripts, which still require our JAX libraries to run. We hope that reviewers will find this, along with the detailed methodologies described in the paper, sufficient evidence of reproducibility.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. Transformers as statisticians: Provable in-context learning with in-context algorithm selection.

Hritik Bansal, Karthik Gopalakrishnan, Saket Dingliwal, Sravan Bodapati, Katrin Kirchhoff, and Dan Roth. Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale. URL http://arxiv.org/abs/2212.09095.

Joseph Bloom. Open source sparse autoencoders for all residual stream layers of gpt2-small, 2024. URL https://www.alignmentforum.org/posts/f9EgfLSurAiqRJySD.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, 2023. https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Nick Cammarata, Shan Carter, Gabriel Goh, Chris Olah, et al. Thread: Circuits. *Distill*, 2020. doi: 10.23915/distill.00024. https://distill.pub/2020/circuits.

Siyu Chen, Heejune Sheen, Tianhao Wang, and Zhuoran Yang. Unveiling induction heads: Provable training dynamics and feature learning in transformers. URL http://arxiv.org/abs/2409.10559.

Tom Conerly, Adly Templeton, Trenton Bricken, Jonathan Marcus, and Tom Henighan. Update on how we train saes, 2024. URL https://transformer-circuits.pub/2024/april-update/index.html#training-saes.

Hoagy Cunningham, Aidan Ewart, Logan Riggs, et al. Sparse autoencoders find highly interpretable features in language models, 2023. URL https://arxiv.org/abs/2309.08600.

Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can GPT learn in-context? language models implicitly perform gradient descent as meta-optimizers. URL http://arxiv.org/abs/2212.10559.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable llm feature circuits, 2024. URL https://arxiv.org/abs/2406.11944.

Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, New York, 2010. ISBN 978-1-4419-7010-7. doi: 10.1007/978-1-4419-7011-4.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. Toy Models of Superposition. *arXiv preprint arXiv:2209.10652*, 2022.

Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes.

Roger Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, Evan Hubinger, Kamilė Lukošiūtė, Karina Nguyen, Nicholas Joseph, Sam McCandlish, Jared Kaplan, and Samuel R. Bowman. Studying large language model generalization with influence functions, 2023. URL https://arxiv.org/abs/2308.03296.

Xiaochuang Han, Daniel Simig, Todor Mihaylov, Yulia Tsvetkov, Asli Celikyilmaz, and Tianlu Wang. Understanding in-context learning via supportive pretraining data. URL http://arxiv.org/abs/2306.15091.

Roee Hendel, Mor Geva, and Amir Globerson. In-context learning creates task vectors, 2023. URL https://arxiv.org/abs/2310.15916.

Daniel D. Johnson. Penzai + treescope: A toolkit for interpreting, visualizing, and editing models as data, 2024. URL https://arxiv.org/abs/2408.00211.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in jax via callable pytrees and filtered transformations, 2021. URL https://arxiv.org/abs/2111.00254.

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2, 2024. URL https://arxiv.org/abs/2408.05147.

Johnny Lin. Neuronpedia: Interactive reference and tooling for analyzing neural networks, 2023. URL https://www.neuronpedia.org. Software available from neuronpedia.org.

Arvind Mahankali, Tatsunori B. Hashimoto, and Tengyu Ma. One step of gradient descent is provably the optimal in-context learner with one layer of linear self-attention. URL http://arxiv.org/abs/2307.03576.

Samuel Marks, Can Rager, Eric J. Michaud, et al. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. *Computing Research Repository*, arXiv:2403.19647, 2024. URL https://arxiv.org/abs/2403.19647.

Andrew Ng. Sparse autoencoder. *CS294A Lecture Notes*, 2011. Unpublished lecture notes.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 2020. doi: 10.23915/distill.00024.001. https://distill.pub/2020/circuits/zoom-in.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads, 2022. URL https://arxiv.org/abs/2209.11895.

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. URL http://arxiv.org/abs/2212.07677.

Jane Pan, Tianyu Gao, Howard Chen, and Danqi Chen. What in-context learning "learns" in-context: Disentangling task recognition and task learning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8298–8319. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.527. URL https://aclanthology.org/2023.findings-acl.527.

Core Francisco Park, Andrew Lee, Ekdeep Singh Lubana, Yongyi Yang, Maya Okawa, Kento Nishi, Martin Wattenberg, and Hidenori Tanaka. Iclr: In-context learning of representations, 2024. URL https://arxiv.org/abs/2501.00070.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Senthooran Rajamanoharan. Improving ghost grads, 2024. URL https://www.alignmentforum.org/posts/C5KAZQib3bzzpeyrg/progress-update-1-from-the-gdm-mech-interp-team-full-update#Improving_ghost_grads.

Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Tom Lieberum, Vikrant Varma, János Kramár, Rohin Shah, and Neel Nanda. Improving dictionary learning with gated sparse autoencoders, 2024a. URL https://arxiv.org/abs/2404.16014.

Senthooran Rajamanoharan, Tom Lieberum, Nicolas Sonnerat, Arthur Conmy, Vikrant Varma, János Kramár, and Neel Nanda. Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders, 2024b. URL https://arxiv.org/abs/2407.14435.

Allan Raventós, Mansheej Paul, Feng Chen, and Surya Ganguli. Pretraining task diversity and the emergence of non-bayesian in-context learning for regression. URL http://arxiv.org/abs/2306.15063.

Lingfeng Shen, Aayush Mishra, and Daniel Khashabi. Do pretrained transformers learn in-context by gradient descent? URL http://arxiv.org/abs/2310.08540.

Chenglei Si, Dan Friedman, Nitish Joshi, Shi Feng, Danqi Chen, and He He. Measuring inductive biases of in-context learning with underspecified demonstrations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11289–11310. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.632. URL https://aclanthology.org/2023.acl-long.632.

Lewis Smith. Replacing sae encoders with inference-time optimisation, 2024. URL https://www.alignmentforum.org/posts/C5KAZQib3bzzpeyrg/full-post-progress-update-1-from-the-gdm-mech-interp-team#Replacing_SAE_Encoders_with_Inference_Time_Optimisation.

Aaquib Syed, Can Rager, and Arthur Conmy. Attribution patching outperforms automated circuit discovery, 2023. URL https://arxiv.org/abs/2310.10348.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, et al. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

Eric Todd, Millicent L. Li, Arnab Sen Sharma, et al. Function vectors in large language models. In *Proceedings of the 2024 International Conference on Learning Representations*, 2024.

Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: A circuit for indirect object identification in GPT-2 small, 2022. URL https://arxiv.org/abs/2211.00593.

Lean Wang, Lei Li, Damai Dai, Deli Chen, Hao Zhou, Fandong Meng, Jie Zhou, and Xu Sun. Label words are anchors: An information flow perspective for understanding in-context learning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9840–9855. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.609. URL https://aclanthology.org/2023.emnlp-main.609.

Xinyi Wang, Wanrong Zhu, Michael Saxon, Mark Steyvers, and William Yang Wang. Large language models are latent variable models: Explaining and finding good demonstrations for in-context learning, 2024. URL https://arxiv.org/abs/2301.11916.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. URL http://arxiv.org/abs/2111.02080.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing, 2024. URL https://arxiv.org/abs/2406.08464.

Steve Yadlowsky, Lyric Doshi, and Nilesh Tripuraneni. Pretraining data mixtures enable narrow model selection capabilities in transformer models. URL http://arxiv.org/abs/2311.00871.

A MODEL AND DATASET DETAILS

For our experiments, we utilized the Gemma 1 2B model, a member of the Gemma family of open models based on Google's Gemini models (Team et al., 2024). The model's architecture is largely the same as that of Llama (Dubey et al., 2024) except for tied input and output embeddings and a different activation function for MLP layers, so we could reuse our infrastructure for loading Llama models. We train residual and attention output SAEs as well as transcoders for layers 1-18 of the model on FineWeb (Penedo et al., 2024).

Our dataset for circuit finding is primarily derived from the function vectors paper (Todd et al., 2024), which provides a diverse set of tasks for evaluating the existence and properties of function vectors in language models. We supplemented this dataset with three additional algorithmic tasks to broaden the scope of our analysis:

- Extract the first element from an array of length 4
- Extract the second element from an array of length 4
- Extract the last element from an array of length 4

The complete list of tasks used in our experiments with task descriptions is as follows:

Task ID	**Description**
location_continent	Name the continent where the given landmark is located.
football_player_position	Identify the position of a given football player.
location_religion	Name the predominant religion in a given location.
location_language	State the primary language spoken in a given location.
person_profession	Identify the profession of a given person.
location_country	Name the country where a given location is situated.
country_capital	Provide the capital city of a given country.
person_language	Identify the primary language spoken by a given person.
singular_plural	Convert a singular noun to its plural form.
present_simple_past_simple	Change a verb from present simple to past simple tense.
antonyms	Provide the antonym of a given word.
plural_singular	Convert a plural noun to its singular form.
present_simple_past_perfect	Change a verb from present simple to past perfect tense.
present_simple_gerund	Convert a verb from present simple to gerund form.
en_it	Translate a word from English to Italian.
it_en	Translate a word from Italian to English.
en_fr	Translate a word from English to French.
en_es	Translate a word from English to Spanish.
fr_en	Translate a word from French to English.
es_en	Translate a word from Spanish to English.
algo_last	Extract the last element from an array of length 4.
algo_first	Extract the first element from an array of length 4.
algo_second	Extract the second element from an array of length 4.

This diverse set of tasks covers a wide range of linguistic and cognitive abilities, including geographic knowledge, language translation, grammatical transformations, and simple algorithmic operations. By using this comprehensive task set, we aimed to thoroughly investigate the in-context learning capabilities of the Gemma 1 2B model across various domains.

B SAE TRAINING

Our Gemma 1 2B SAEs are trained with a learning rate of 1e-3 and Adam betas of 0.0 and 0.99 for 150M ($\pm$100) tokens of FineWeb (Penedo et al., 2024). The methodology is overall similar to (Bloom, 2024). We initialize encoder weights orthogonally and set decoder weights to their transpose. We initialize decoder biases to 0. We use Rajamanoharan (2024)'s ghost gradients variant (ghost gradients applied to dead features only, loss multiplied by the proportion of death features) with the additional modification of using softplus instead of exp for numerical stability. A feature is considered

dead when its density (according to a 1000-batch buffer) is below 5e-6 or when it has not fired in 2000 steps. We use Anthropic's input normalization and sparsity loss for Gemma 1 2B (Conerly et al., 2024). We found it to improve Gated SAE training stability. We modified it to work with transcoders by keeping track of input and output norms separately and predicting normed outputs.

We convert our Gated SAEs into JumpReLU SAEs after training, implementing algorithms like TVC and SFC in a unified manner for all SAEs in this format (including simple SAEs). The conversion procedure involves setting thresholds to replicate the effect of the gating branch. For further details, see Rajamanoharan et al. (2024b).

We use 4 v4 TPU chips running Jax (Bradbury et al., 2018) (Equinox (Kidger & Garcia, 2021)) to train our SAEs. We found that training with Huggingface's Flax LM implementations was very slow. We reimplemented LLaMA (Dubey et al., 2024) and Gemma (Team et al., 2024) in Penzai (Johnson, 2024) with a custom layer-scan transformation and quantized inference kernels as well as support for loading from GGUF compressed model files. We process an average of around 4400 tokens per second, which makes training SAEs and not caching LM activations the main bottleneck. For this and other reasons, we don't do SAE sparsity coefficient sweeps to increase TPU utilization.

For caching, we use a distributed ring buffer which contains separate pointers on each device to allow for processing masked data. The (in-place) buffer update is in a separate JIT context. Batches are sampled randomly from the buffer for each training step.

We train our SAEs in bfloat16 precision. We found that keeping weights and scales in bfloat16 and biases in float32 performed best in terms of the number of dead features and led to a Pareto improvement over float32 SAEs.

For training Phi 3 (Abdin et al., 2024) SAEs, we use data generated by the model unconditionally, similarly to (Xu et al., 2024)[1]. The resulting dataset we train the model on contains many math problems and is formatted as a natural-seeming interaction between the user and the model.

Each SAE training run takes us about 3 hours. We trained 3 models (a residual SAE, an attention output SAE, and a transcoder) for each of the 18 layers of the model. This is about 1 week of v4-8 TPU time.

Our SAEs and training code will be made public after paper acceptance.

C EXAMPLE CIRCUITS

An example output of our circuit cleaning algorithm can be found in Figure 6. We can see the flow of information through a single high-IE attention feature from a task-detection feature (activating on output tokens) to transcoder and residual execution features (activating on arrow tokens). The feature activates on antonyms on the detection feature #11050: one can assume the first sequence began as "Short Term Target", making the second half an antonym.

We will release a web interface for viewing maximum activating examples and task feature circuits.

D TASK VECTOR CLEANING ALGORITHM

The task vector cleaning algorithm is a novel approach we developed to isolate task-relevant features from task vectors. Figure 7 provides an overview of this algorithm.

Our process begins with collecting residuals for task vectors using a batch of 16 and 16-shot prompts. We then calculate the SAE features for these task vectors. We explored two methods: (1) calculating feature activation and then averaging across tokens, and (2) averaging across tokens first and then calculating the task vector. They had similar performances.

The cleaning process is performed on a training batch of 24 pairs, with evaluation conducted on an additional 24 pairs. All prompts are zero-shot. An example prompt is as follows:

[1]Phi-3 is trained primarily with instruction following data, making it an aligned chat model.

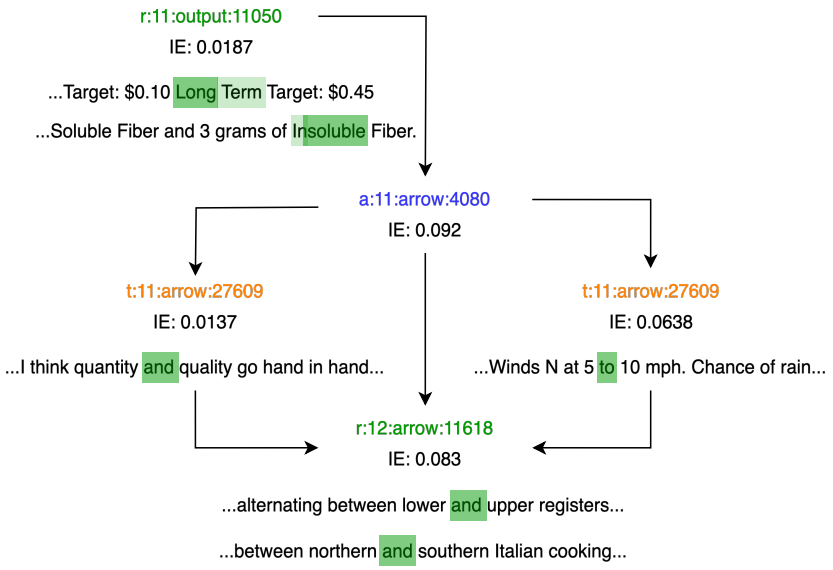


Figure 6: An example of a circuit found using our SFC variant. We focused on a subcircuit with high indirect effects. Maximum activating examples from the SAE training distribution are included.

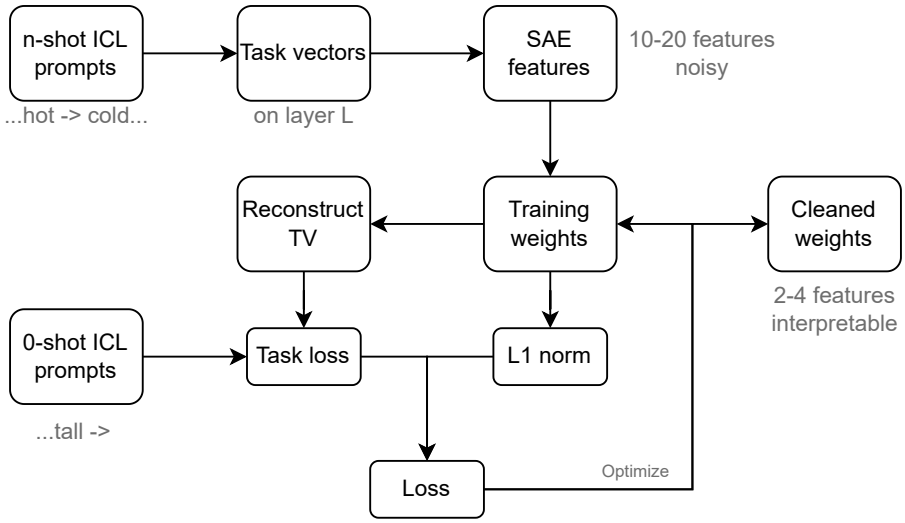


Figure 7: An overview of our Task Vector Cleaning algorithm. TV stands for Task Vector.

BOS	Follow	the	pattern	:	\n
tall	→	short	\n		
...					
old	→	young	\n		
hot	→	cold			

Example 2: The steered token is highlighted in red. Loss is calculated on the yellow token.

The algorithm is initialized with the SAE reconstruction as a starting point. It then iteratively steers the model on the reconstruction layer and calculates the loss on the training pairs. To promote sparsity, we add the L_1 norm of weights with coefficient λ to the loss function. The algorithm implements early stopping when the L_0 norm remains unchanged for n iterations.

```
def tvc_algorithm(task_vector, model, sae):
  initial_weights = sae.encode(task_vector)
  def tvc_loss(weights, tokens):
    task_vector = sae.decode(weights)
    mask = tokens == self.separator
    model.residual_stream[layer, mask] += task_vector
    # loss only on the ``output'' tokens,
    # ignoring input and prompt tokens
    loss = logprobs(model.logits, tokens, ...)
    return loss + l1_coeff * l1_norm(weights)
  weights = initial_weights.copy()
  optimizer = adam(weights, lr=0.15)
  last_l0, without_change = 0, 0  # early stopping
  for _ in range(1000):
    grad = jax.grad(tvc_loss)(weights, tokens)
    weights = optimizer.step(grad)
    if l0_norm(weights) != last_l0:
      last_l0, without_change = l0_norm(weights), 0
    elif without_change >= 50:
      break
  return weights
```

Algorithm 1: Pseudocode for Task Vector Cleaning.

The hyperparameters λ, n, and learning rate α can be fixed for a single model. We experimented with larger batch sizes but found that they did not significantly improve the quality of extracted features while substantially slowing down the algorithm due to gradient accumulation.

The algorithm takes varying amounts of time to complete for different tasks and models. For Gemma 1, it stops at 100-200 iterations, which is close to 40 seconds at 5 iterations per second.

It's worth noting that we successfully applied this method to the recently released Gemma 2 2B and 9B models using the Gemma Scope SAE suite (Lieberum et al., 2024). It was also successful with the Phi-3 3B model (Abdin et al., 2024) and with our SAEs, which were trained similarly to the Gemma 1 2B SAEs.

D.1 L_1 SWEEPS

To provide more details about the method's effectiveness across various models and SAE widths, we conducted L_1 coefficient sweeps with our Phi-3 and Gemma 1 2B SAEs, as well as Gemma Scope Gemma 2 SAEs. We chose two SAE widths for Gemma 2 2B and 9B: 16k and 65k. For Gemma 2 2B we also sweeped across several different target SAE L_0 norms. We studied only the optimal task vector layer for each model: 12 for Gemma 1, 16 for Gemma 2, 18 for Phi-3, and 20 for Gemma 2 9B. We used a learning rate of 0.15 with the Gemma 1 2B, Phi-3, and Gemma 2 2B 65k models, 0.3 with Gemma 2 2B 16k, and 0.05 with 200 early stopping steps for Gemma 2 9B.

Figures 8, 9, 10 compare TVC and ITO against original task vectors. The X-axis displays the fraction of active task vector SAE features used. The Y-axis displays the TV loss delta, calculated as $(L_{TV} - L_{Method})/L_{Zero}$, where L_{TV} is the loss from steering with the task vector, L_{Method} is the loss after it has been cleaned using the corresponding method, and L_{Zero} is the uninformed (no-steering) model loss. This metric shows improvement over the task vector relative to the loss of the uninformed model. Points were collected from all tasks using 5 different L_1 coefficient values.

We observe that our method often improves task vector loss and can reduce the number of active features to one-third of those in the original task vector while maintaining relatively intact performance. In contrast, ITO rarely improves the task vector loss and is almost always outperformed by TVC.

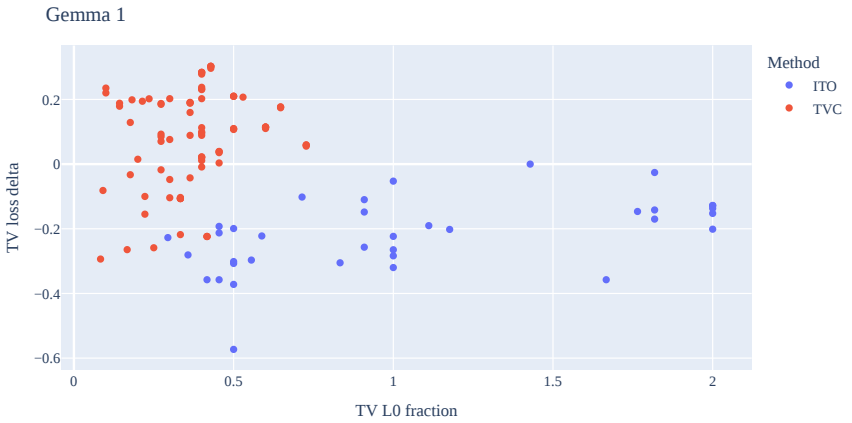


Figure 8: Performance of ITO and TVC across different tasks and optimization parameters compared to task vectors for Gemma 1 2B. The Y-axis shows relative improvement over task vector loss, while the X-axis shows the fraction of active TV features used. Metric calculation details are available in D.1

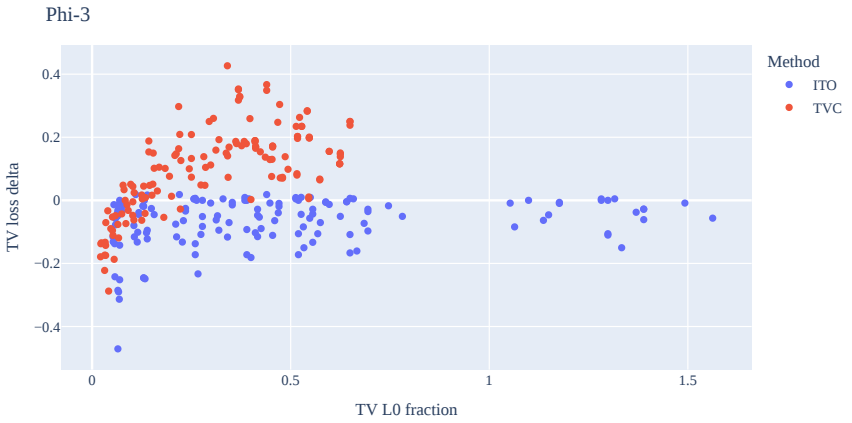


Figure 9: Performance of ITO and TVC across different tasks and optimization parameters compared to task vectors for Phi-3. The Y-axis shows relative improvement over task vector loss, while the X-axis shows the fraction of active TV features used. Metric calculation details are available in D.1

Figures 11, 12 and 13 show task-mean loss decrease (relative to no steering loss) and remaining TV features fraction plotted against L_1 sweep coefficients. We see that L_1 coefficients between 0.001 and 0.025 result in relatively intact performance, while significantly reducing the amount of active SAE features. From Figure 12 we can notice that the method performs better with higher target l0 SAEs, being able to affect the loss with just a fraction of active SAE features.

E Details of our SFC implementation

E.1 Implementation details

Our implementation of circuit finding attribution patching is specialized for Jax and Penzai.

We first perform a forward-backward pass on the set of prompts, collecting residuals and gradients from the metric to residuals. We collect gradients with `jax.grad` by introducing "dummy" zero-valued inputs to the metric computation function that are added to the residuals of each layer. Note that we do not use SAEs during this stage.

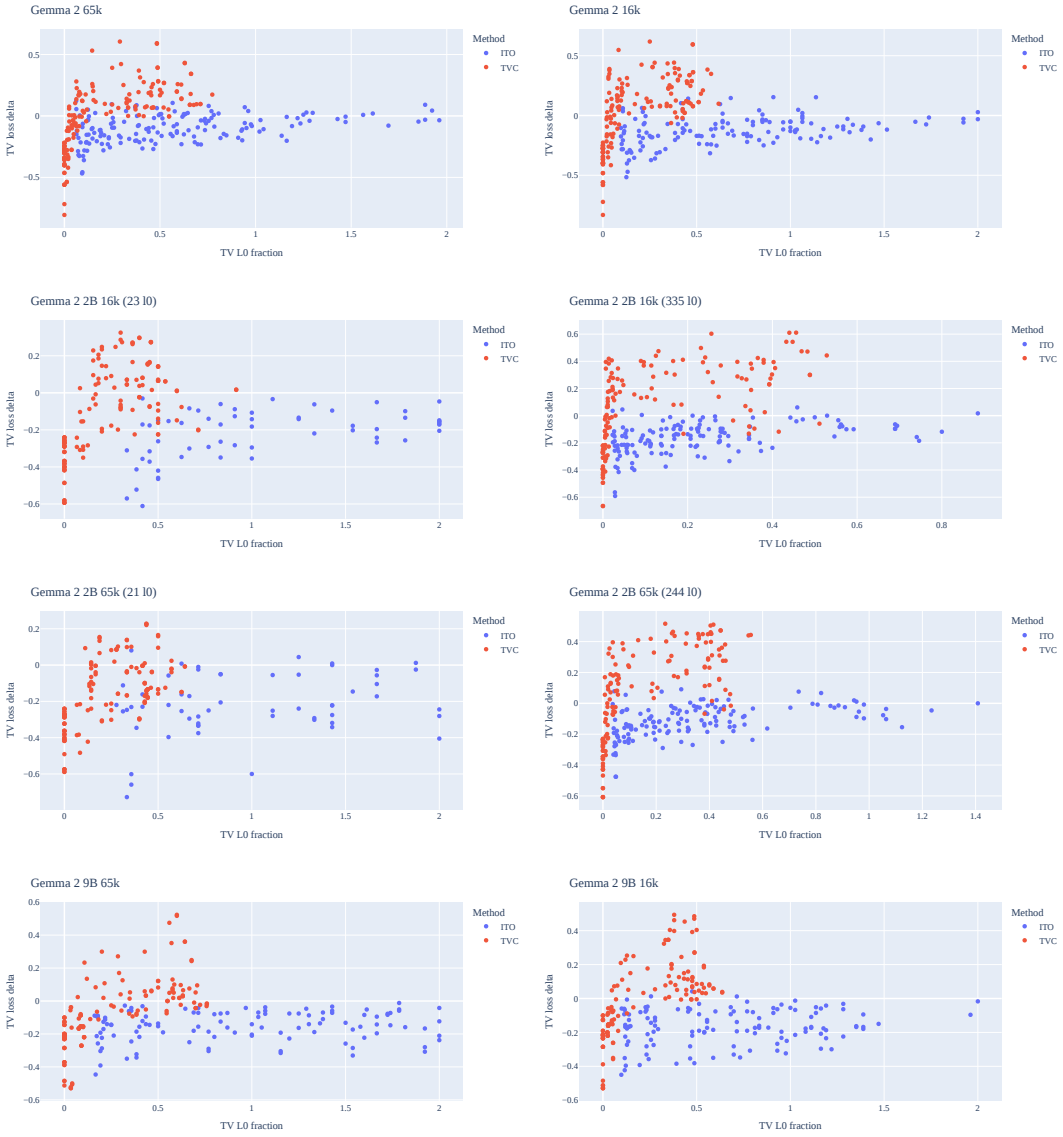


Figure 10: Performance of ITO and TVC across different tasks and optimization parameters compared to task vectors for Gemma 2 Gemma Scope SAEs. The Y-axis shows the relative improvement over the loss from steering with a task vector, while the X-axis shows the fraction of active TV features used. Metric calculation details are available in Appendix D.1.

We then perform an SAE encoding step and find the nodes (residual, attention output, and transcoder SAE features and error nodes) with the highest indirect effects using manually computed gradients from the metric. After that, we find the features with the top K indirect effects for each layer and position mask and treat them as candidates for circuit edge targets. We compute gradients with respect to the metric to the values of those nodes, propagate them to "source features" up to one layer above, and multiply by the values of the source features. This way, we can compute indirect effects for circuit edges and prune the initially fully connected circuit. However, like Marks et al. (2024), we do not perform full ablation of circuit edges.

We include a simplified implementation of node-only SFC in Algorithm 2.

```
# resids_pre: L x N x D - the pre-residual stream at layer L
# resids_mid: L x N x D - the middle of the residual stream
```

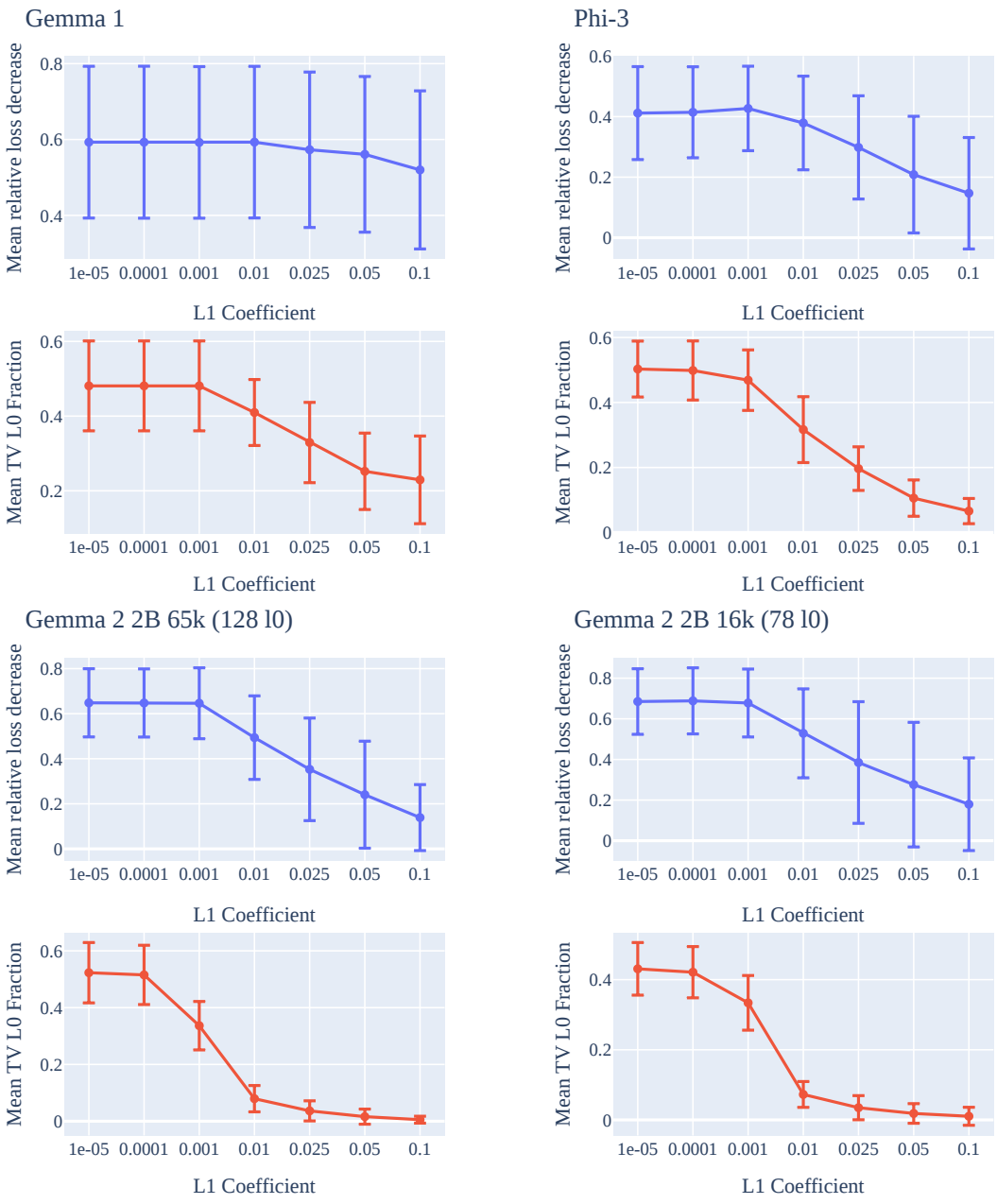


Figure 11: L_1 coefficient sweeps across different models and SAEs. All metrics are averaged across all tasks. Error bars show the standard deviation of the average for each case. Metric calculation details are available in D.1.

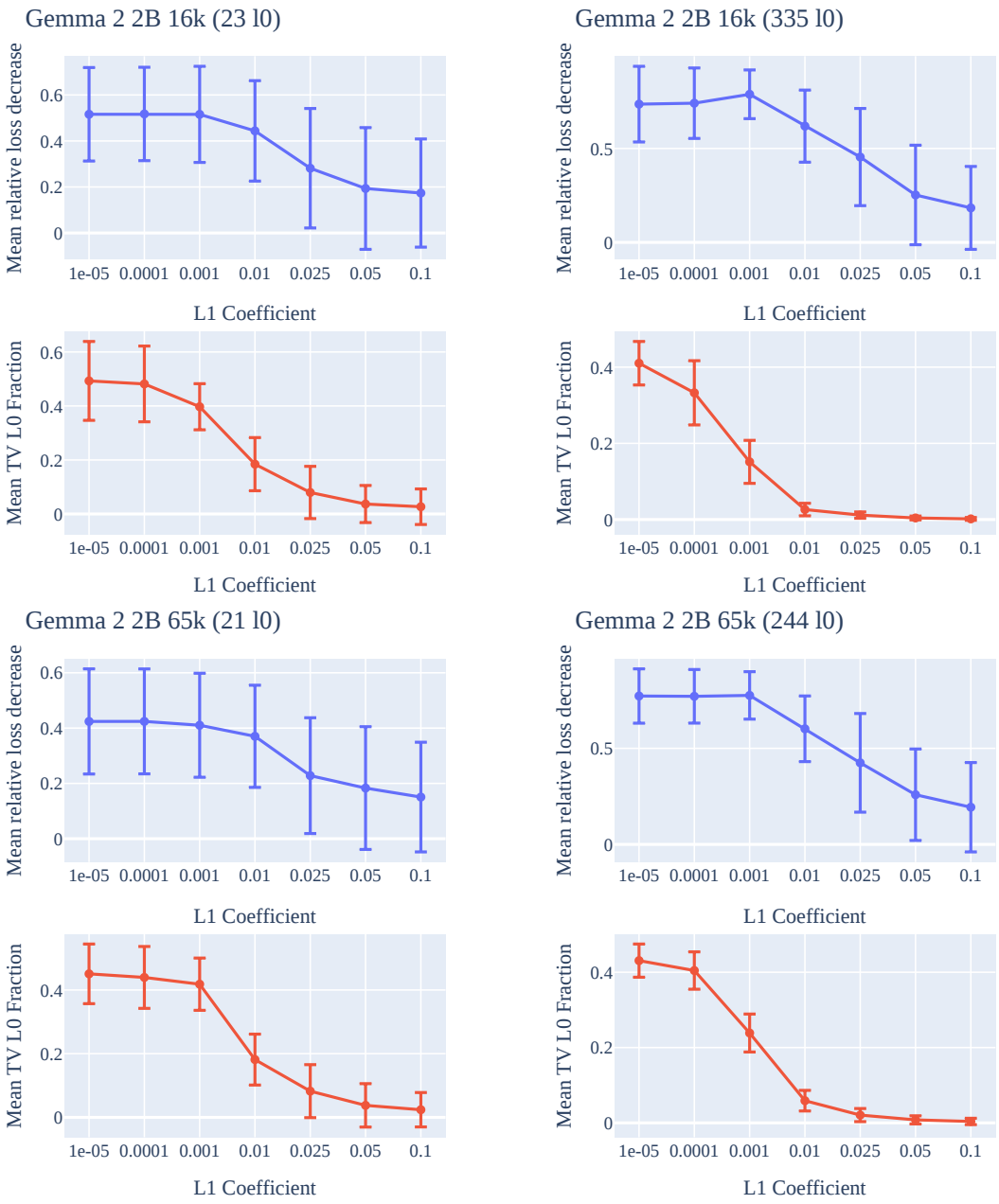


Figure 12: L_1 coefficient sweeps across different target SAE sparsities and widths for Gemma 2 2B. All metrics are averaged across all tasks. Error bars show the standard deviation of the average for each case. Metric calculation details are available in Appendix D.1.

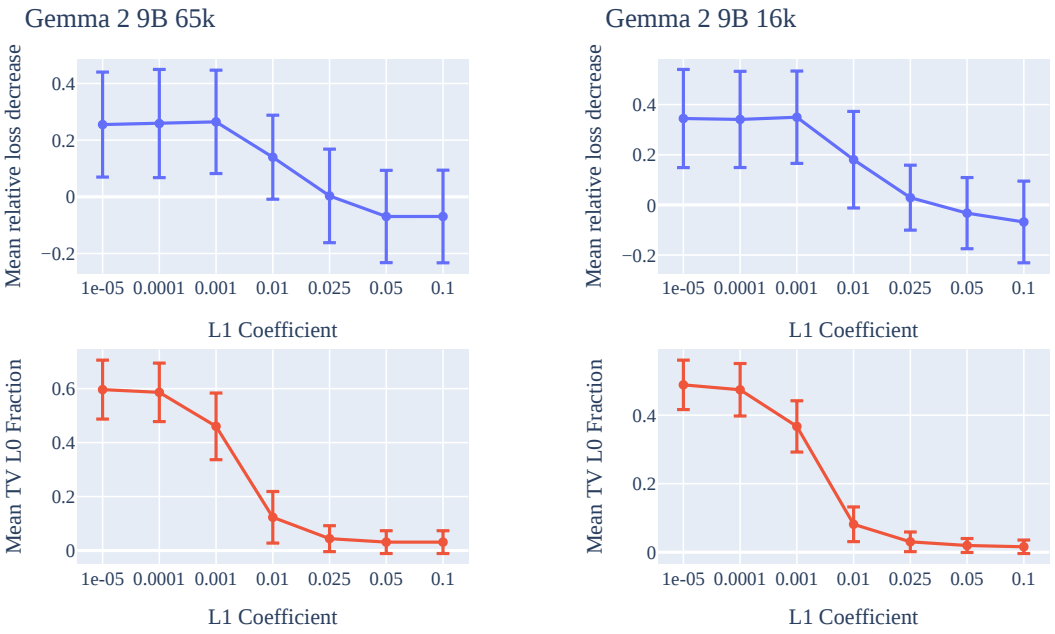


Figure 13: L_1 coefficient sweeps across two SAE widths for Gemma 2 9B. All metrics are averaged across all tasks. Error bars show the standard deviation of the average for each case. Metric calculation details are available in D.1.

```
# (between attention and MLP) at layer L
# grads_pre: L x N x D - gradients from the metric to resids_pre
# grads_mid: L x N x D - gradients from the metric to resids_mid
# all of the above are computed with a forward and backward
# pass without SAEs

# saes_resid: L - residual stream SAEs
# saes_attn: L - attention output SAEs
# transcoders_attn: L - transcoders predicting resids_pre[l+1]
# from resids_mid[l]

def indirect_effect_for_residual_node(layer):
    sae_encoding = saes_resid[layer].encode(
        resids_pre[layer])
    grad_to_sae_latents = jax.vjp(
        saes_resid[layer].decode,
        sae_encoding
    )(grads_pre[l])
    return (grad_to_sae_latents * sae_encoding).sum(-1)

def indirect_effect_for_attention_node(layer):
    sae_encoding = saes_attn[layer].encode(
        resids_mid[layer] - resids_pre[layer])
    grad_to_sae_latents = jax.vjp(
        saes_attn[layer].decode,
        sae_encoding
    )(grads_mid[l])
    return (grad_to_sae_latents * sae_encoding).sum(-1)

def indirect_effect_for_transcoder_node(layer):
    sae_encoding = transcoders[layer].encode(
```

```
        resids_mid[layer])
    grad_to_sae_latents = jax.vjp(
        transcoders[layer].decode,
        sae_encoding
    )(grads_pre[l+1])
    return (grad_to_sae_latents * sae_encoding).sum(-1)
```

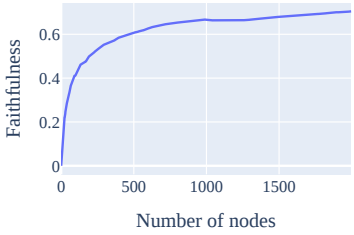
Algorithm 2: Pseudocode for Sparse Feature Circuits indirect effect calculation.

E.2 SFC Evaluation Details

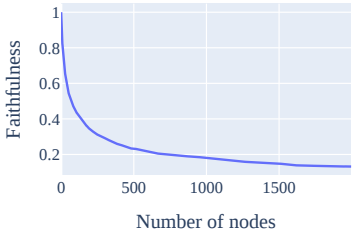
We evaluated our SFC modifications through comprehensive ablation studies, measuring faithfulness using the metric:

$$\text{Faithfulness}(C) = \frac{m(C) - m(\varnothing)}{m(M) - m(\varnothing)} \tag{3}$$

where $m(C)$ represents performance with circuit C, $m(\varnothing)$ the baseline, and $m(M)$ the full model's performance. Our ablation studies targeted both the discovered circuit C and its complement



(a) Faithfulness of C



(b) Faithfulness of $M \setminus C$

Figure 14: Faithfulness measurements for circuits and their complements.

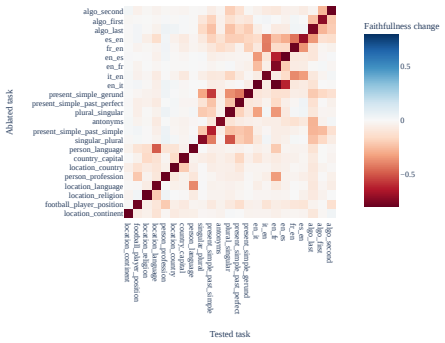


Figure 15: Cross-task impact of circuit ablation, showing strong task specificity.

$M \setminus C$, removing nodes according to their IE thresholds. As shown in Figure 14, circuits of 500 nodes achieved an average faithfulness of 0.6 across tasks. Cross-task ablation studies (Figure 15) demonstrated strong task specificity, with performance impacts largely confined to target tasks and closely related tasks like translation pairs. We focused our analysis on intermediate layers 10-17 of the model's 18 total layers. This choice was motivated by two factors: earlier layers primarily process token-level information rather than task-specific features, and our analysis showed more reliable IE approximations in these intermediate layers. This differs from previous work (Marks et al., 2024) which excluded fewer early layers (2 out of 6 versus our 10 out of 18), reflecting the increased complexity of analyzing larger models and ICL tasks. Appendix E.3 contains more details on faithfulness approximation quality. The results demonstrate that our modified SFC approach successfully scales to analyze complex ICL mechanisms in larger language models while maintaining the ability to identify task-specific circuits and their interactions.

E.3 IE APPROXIMATION QUALITY

Our IE calculation approach, which aggregates effects across all tokens of the same type, resulted in each layer having a limited number of non-zero nodes. This allowed us to directly examine the impact of disabling each of these nodes. We assessed the quality of the IE approximation by calculating correlation coefficients between the actual effects and their approximations. To further reduce computation time, we focused exclusively on nodes from the "input," "output," and "arrow" groups. Figure 16 displays the correlations averaged across all tasks for all SAE types combined, while Figure 17 presents the metric for each SAE type separately.

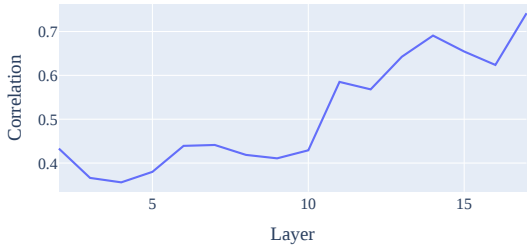


Figure 16: Average correlation of predicted and actual IEs across tasks for "input", "output" and "arrow" non-zero nodes.

Overall, we observe that the approximation quality remains relatively low before layer 6, which is much deeper in the model than layer 2, as reported by the original SFC paper. Non-residual stream SAEs begin to show adequate performance only in the last third of the model. This may be due to the quality of our trained SAEs, the increased task complexity, or token type-wise aggregation, and warrants further investigation. This is the primary reason our analysis focuses mainly on layers 10-15.

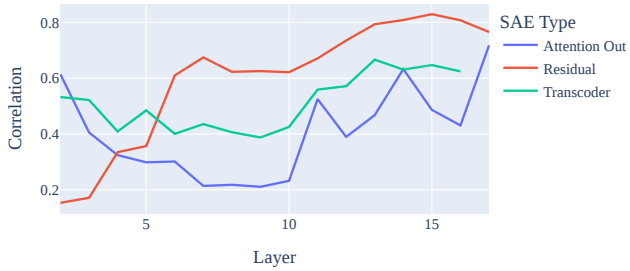


Figure 17: Average correlation of predicted and actual IEs across tasks for "input", "output" and "arrow" non-zero nodes for different SAE types.

F STEERING WITH TASK-EXECUTION FEATURES

To evaluate the causal relevance of our identified ICL features, we conducted a series of steering experiments. Our methodology employed zero-shot prompts for task-execution features, measuring effects across a batch of 32 random pairs.

We set the target layer as 12 using Figure 2 and extracted all task-relevant features on it using our cleaning algorithm. To determine the optimal steering scale, we conducted preliminary experiments using manually identified task-execution features across all tasks. Through this process, we established an optimal steering scale of 15, which we then applied consistently across all subsequent experiments.

For each pair of tasks and features, we steered with the feature and measured the relative loss improvement compared to the model's task performance on a prompt without steering. This relative improvement metric allowed us to quantify the impact of each feature on task performance.

Token Type	Mass (%)
arrow	89.80
output	6.46
input	3.2
newline	0.54
prompt	0.00

Table 1: Activation masses for **executor** features across different token types, averaged across all tasks. We can notice they activate largely on arrow tokens.

To normalize our results and highlight the most significant effects, we applied several post-processing steps:

- We clipped the effect to be no more than 1, thus ignoring any instances of loss increase.
- We then normalized the effects for all features within the same task to be in the 0 to 1 range.
- To remove clutter and highlight important features, we set effects lower than 0.2 to 0.
- Finally, we removed features with low maximum effect across all tasks to reduce the size of the resulting diagram. The full version of this diagram is present in Figure 18.

Prompt example with the steered token highlighted in red. Loss is calculated on the yellow token:

BOS Follow the pattern : \n
hot → cold

Example 3: Task-execution steering setup. The steered token is highlighted in red and the loss is calculated on the yellow token.

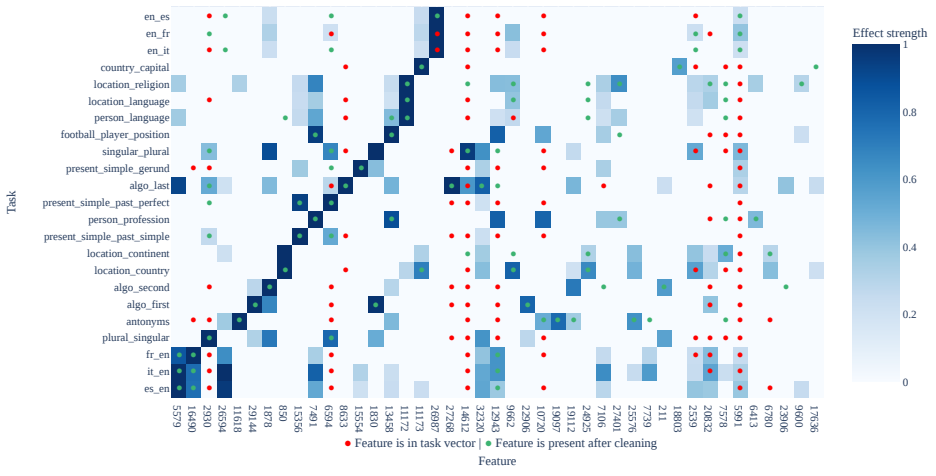


Figure 18: Full version of the heatmap in Figure 4 showing the effect of steering with individual task-execution features for each task. The features present in the task vector of the corresponding task are marked with dots (i.e. from the naive SAE reconstruction baseline in Section 3.1). Green dots show the features that were extracted by cleaning. Red dots are features present in the original task vector. Not all original features from the task vectors are present.

We also share the version of Figure 18 without normalization and value clipping. It is present in Figure 20. We see that task vectors generally contain just a few task-execution features that can boost the task themselves. The remaining features have much weaker and less specific effects.

F.1 NEGATIVE STEERING

To further explore the effects of the executor feature, we also conducted negative steering experiments. The setup involved a batch of 16 ICL prompts, each containing 32 examples for each task. We collected all features from the cleaned task vectors for every task. Similar to positive steering, we steered with features on arrow tokens, but this time multiplying the direction by -1. Prompts this time contained several arrow tokens, and we steered on all of them simultaneously.

An important distinction from positive steering is that performance degradation in negative steering may occur due to two factors: (1) our causal intervention on the ICL circuit and (2) the steering scale being too high. To address this, we measured accuracy across all pairs in the batch instead of loss, as accuracy does not decrease indefinitely. We also observed that features no longer share a common optimal scale. Consequently, for each task pair, we iterated over several scales between 1 and 30. For each feature, we then selected a scale that reduced accuracy by at least 0.1 for at least one task. Steering results at this scale were used for this feature across all tasks.

Figure 19 displays the resulting heatmap. While we observe some degree of task specificity — and even note that some executing features from Figure 18 have their expected effects — we also find that negative steering exhibits significantly lower task specificity. Additionally, we observe that non-task-specific features have a substantial impact in this experiment. This suggests that steering experiments alone may not suffice for a comprehensive analysis of the ICL mechanism, thus reinforcing the importance of methods such as our modification of SFC.

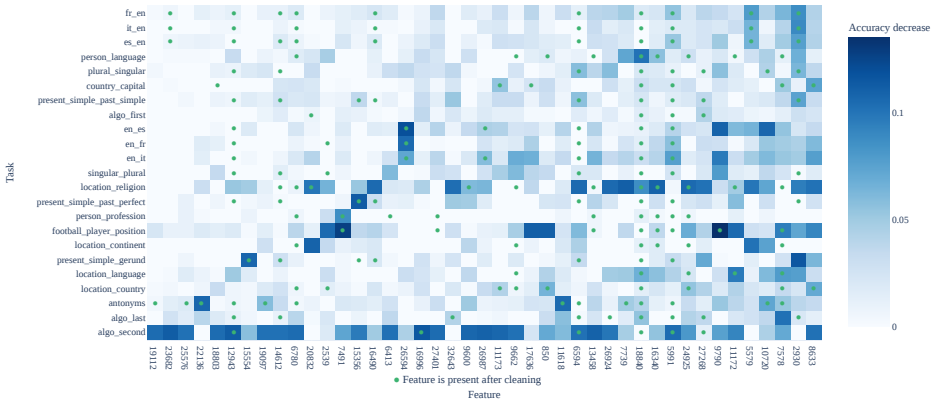


Figure 19: Negative steering heatmap. Displays accuracy decrease after optimal scale negative steering on full ICL prompts. Green circles show which features were present in the cleaned task vector of the corresponding task. More details in Appendix F.1.

F.2 GEMMA 2 2B POSITIVE STEERING

Additionally, we conducted zero-shot steering experiments with Gemma 2 2B 16k and 65k SAEs. Contrary to Gemma 1 2B, task executors from Gemma 2 2B did not have a single common optimal steering scale. Thus, we added an extra step to the experiment: for each feature and task pair, we performed steering with several scales from 30 to 300, and then selected the scale that had maximal loss decrease on any of the tasks. We then used this scale for this feature in application to all other tasks. Figure 21a and Figure 21b contain steering heatmaps for Gemma 2 2B 16k SAEs and Gemma 2 2B 65k SAEs respectively.

We observe a relatively similar level of executor task-specificity compared to Gemma 1. One notable difference between 16k and 65k SAEs is that 65k cleaned task vectors appear to contain more features with a strong effect on the task. However, this may be due to the l_1 regularization coefficient being too low.

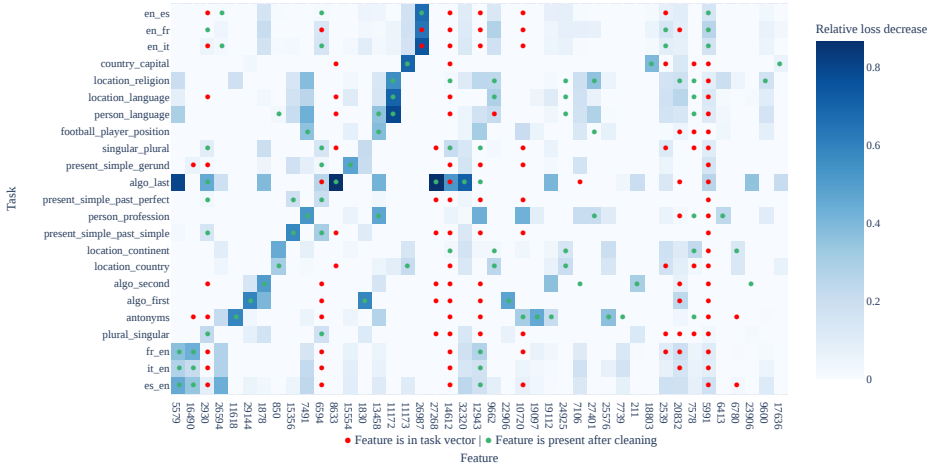


Figure 20: Unfiltered version of the heatmap in Figure 22 showing the effect of steering with individual task-execution features for each task. The features present in the task vector of the corresponding task are marked with dots. Green dots show the features that were extracted by cleaning. Red dots are the features present in the original task vector. Since the chart only contains features from cleaned task vectors, not all features from the original task vectors are present.

G TASK-DETECTION FEATURES

For our investigation of task-detection features, we employed a methodology similar to that used for task execution features, with a key modification. We introduced a fake pair to the prompt and focused our steering on its output. This approach allowed us to simulate the effect of the detection features the way it happens on real prompts. Table 2 and Figure 22 again show task and token specificity.

Our analysis revealed that layers 10 and 11 were optimal for task detection, with performance notably declining in subsequent layers. We selected layer 11 for our primary analysis due to its proximity to layer 12, where we had previously identified the task execution features. This choice potentially facilitates a more direct examination of the interaction between detection and execution mechanisms.

The steering process for detection features followed the general methodology outlined in Appendix F, including the use of a batch of 32 random pairs, extraction of task-relevant features, and application of post-processing steps to normalize and highlight significant effects. The primary distinction lies in the application of the steering to the prompt.

This approach allowed us to create a comprehensive representation of the causal relationships between task-detection features and the model's ability to recognize specific tasks, as visualized in Figure 22.

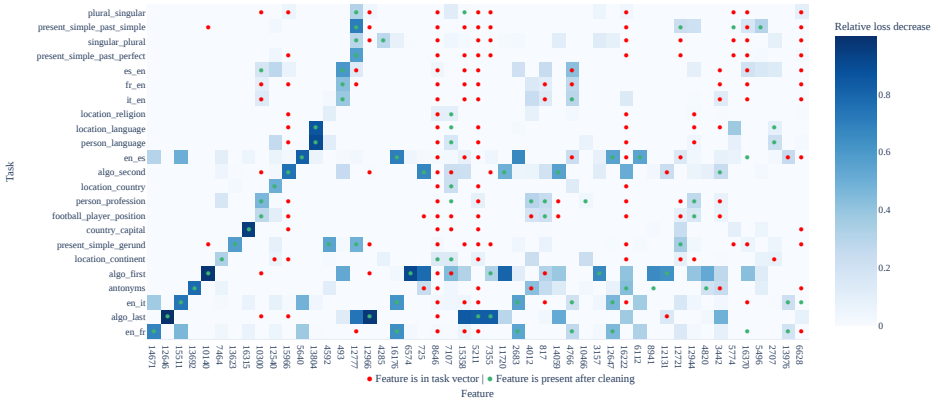
BOS	Follow	the	pattern	: \n
X	→	Y	\n	
hot	→	cold		

Example 4: Task-detection steering setup. The steered token is highlighted in red and the loss is calculated on the yellow token.

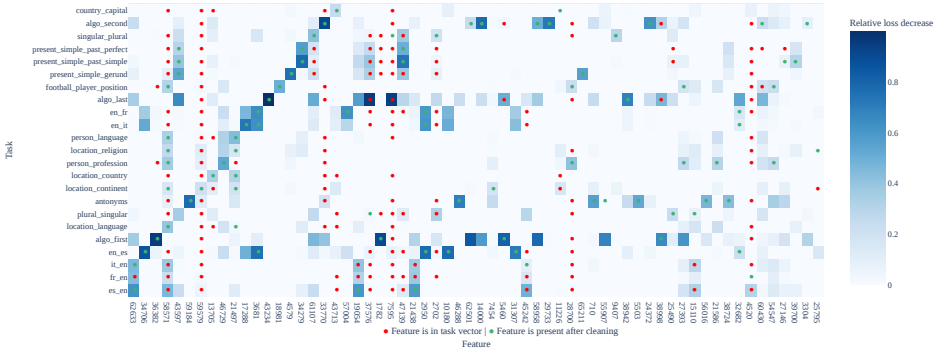
H ICL INTERPRETABILITY LITERATURE REVIEW

This section will cover work on understanding ICL not mentioned in Section 5.

Raventós et al. provides evidence for two different Bayesian algorithms being learned for linear regression ICL: one for limited task distributions and one that is similar to ridge regression. It



(a) Gemma 2 2B 16k



(b) Gemma 2 2B 65k

Figure 21: Unfiltered positive steering heatmap for Gemma 2 2B SAEs showing the effect of steering with individual task-execution features for each task. Steering scales were optimized for each feature. The features present in the task vector of the corresponding task are marked with dots. Green dots show the features that were extracted by cleaning. Red dots are the features present in the original task vector. Since the chart only contains features from cleaned task vectors, not all features from the original task vectors are present.

Token Type	Mass (%)
output	96.76
input	3.22
newline	0.01
arrow	0.0
prompt	0.0

Table 2: Activation masses for **task-detection** features across different token types, averaged across all tasks. We can notice that they activate almost exclusively on output tokens.

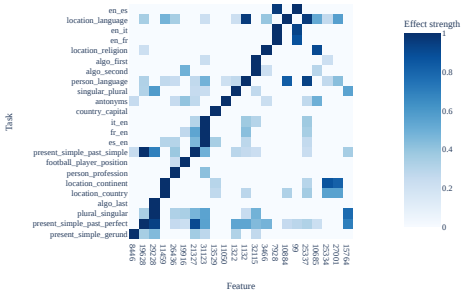


Figure 22: Heatmap showing the effect of steering with the task-detection feature most relevant to each task, on every task. We see that task detection features are typically specific to the task, with exceptions for similar tasks.

also intriguingly shows that the two solutions lie in different basins of the loss landscape, a phase transition necessary to go from one to the other. While interesting, it is not clear if the results apply to real-world tasks.

The existence of discrete task detection and execution features hinges on the assumption that in-context learning works by classifying the task to perform and not by learning a task. Pan et al. aims to disentangle the two with a black-box approach that mixes up outputs to force the model to learn the task from scratch. Si et al. look at biases in task recognition in ambiguous examples through a black-box lens. We find more clear task features for some tasks than others but do not consider whether this is linked to how common a task is in pretraining data.

Xie et al. proposes that in-context learning happens because language models aim to model a latent topic variable to predict text with long-range coherence. Wang et al. (2024) show following the two proposed steps rigorously improves results in real-world models. However, they do not endeavor to explain the behavior of non-finetuned models by looking at internal representations; instead, they aim to improve ICL performance.

Han et al. use a weight-space method to find examples in training data that promote in-context learning using a method akin to Grosse et al. (2023), producing results similar to per-token loss analyses in Olsson et al. (2022), and, similarly to the studies mentioned above, finds that those examples involve long-range coherence. Our method is also capable of finding examples in data that are similar to ICL, and we find crisp examples for many tasks being performed Appendix I.

Bansal et al. offers a deeper look into induction heads, scaling up Olsson et al. (2022) the way we scale up Marks et al. (2024). Crucially, it finds that MLPs in later layers cannot be removed while preserving ICL performance, indirectly corroborating our findings from Section 4.1. Chen et al. come up with a proof that states that gradient flow converges to a generalized version of the algorithm suggested by Olsson et al. (2022) when trained on n-gram Markov chain data.

Garg et al. studies the performance of toy models trained on in-context regression various *function classes*. Yadlowsky et al. find that Transformers trained on regression with multiple function classes have trouble combining solutions for learning those functions. Oswald et al. construct a set of weights for linear attention Transformers that reproduce updates from gradient descent and find evidence for the algorithm being represented on real models trained on toy tasks. Mahankali et al. proves that this algorithm is optimal for single-layer transformers on noisy linear regression data. Shen et al. questions the applicability of this model to real-world transformers. Bai et al. finds that transformers can switch between multiple different learning algorithms for ICL. Dai et al. find multiple similarities between changes made to model predictions from in-context learning and weight finetuning.

While important, we do not consider this direction of interpreting transformers trained on regression for concrete function classes through primarily white-box techniques. Instead, we aim to focus on clear discrete tasks which are likely to have individual features.

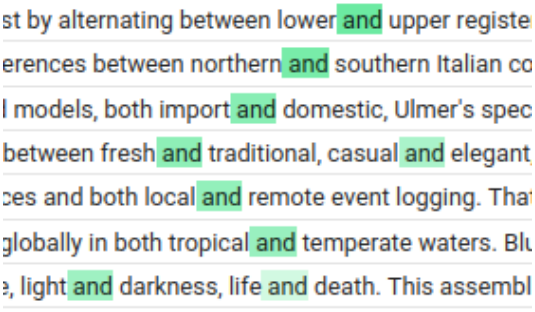
The results of Wang et al. are perhaps the most similar to our findings. The study finds "anchor tokens" responsible for aggregating semantic information, analogous to our "output tokens" (Section 2.3) and task-detection features. They tackle the full circuit responsible for ICL bottom-up and intervene on models using their understanding, improving accuracy. Like this paper, they do not deeply investigate later attention and MLP layers. Our study uses SAE features to find strong linear directions on output and arrow tokens corresponding to task detection and execution respectively, offering a different perspective. Additionally, we consider over 20 diverse token-to-token tasks, as opposed to the 4 text classification datasets considered in Wang et al..

I MAX ACTIVATING EXAMPLES

This section contains max activating examples for some executor and detector features for Gemma 1 2B, as described in (Bricken et al., 2023). They are computed by iterating over the training data distribution (FineWeb) and sampling activations of SAE features that fall within disjoint buckets for the activation value of span 0.5. We can observe that the degree of intuitive interpretability depends on the amount of task-similar contexts in the training data and SAE width.

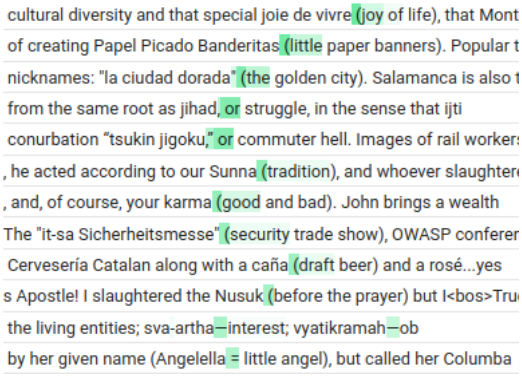
We also provide max activating examples for Gemma 2 2B executor features from Figures 21b and 21a. These max activating examples are taken from the Neuronpedia (Lin, 2023) and are available in Figures 26 and 25.

Here we can notice the main difference between executors and detectors: executors mainly activate **before the task completion**, while detectors activate on the **token that completes the task**. We also

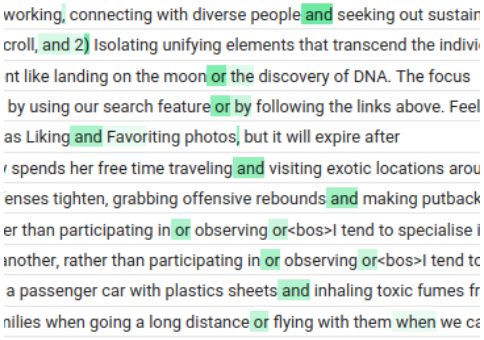


(a) Max activating examples for the antonyms executor feature 11618.

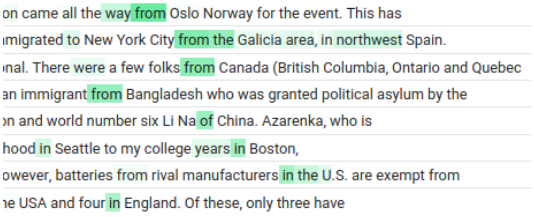
(b) Max activating examples for the English to foreign language translation executor feature 26987.



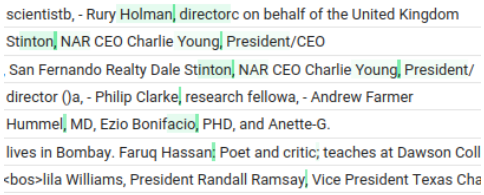
(c) Max activating examples for the translation to English executor feature 5579.



(d) Max activating examples for the "next comes gerund form" executor feature 15554.

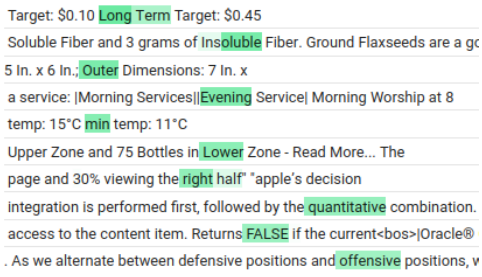


(e) Max activating examples for the prediction of city/-country feature 850.

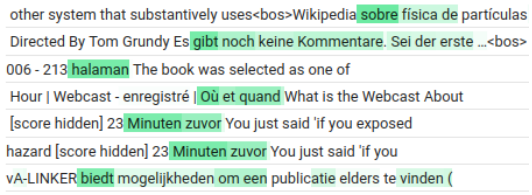


(f) Max activating examples for the person's occupation executor feature 13458.

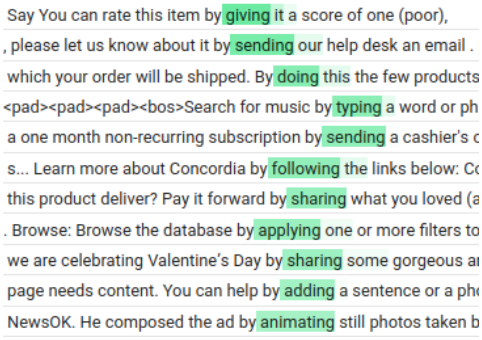
Figure 23: Max activating examples for executor features from Figure 4.



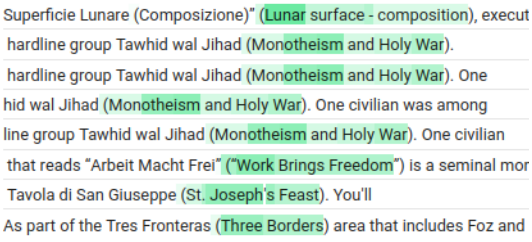
(a) Max activating examples for the antonyms detector feature 11050.



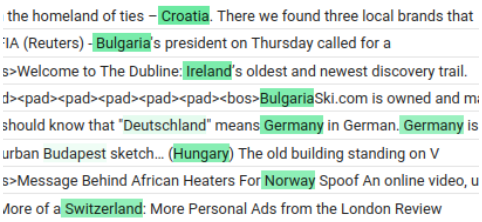
(b) Max activating examples for the English to foreign language switch detector feature 7928.



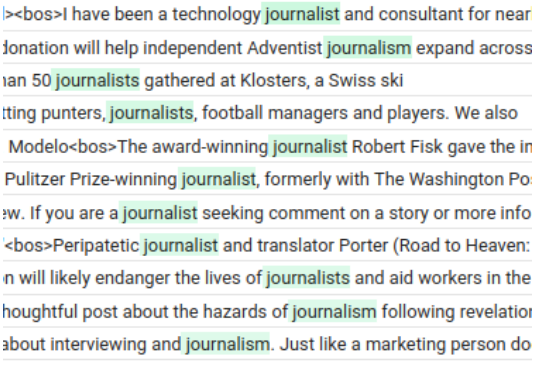
(c) Max activating examples for the gerund form detector feature 8446.



(d) Max activating examples for the translation to English detector feature 31123.

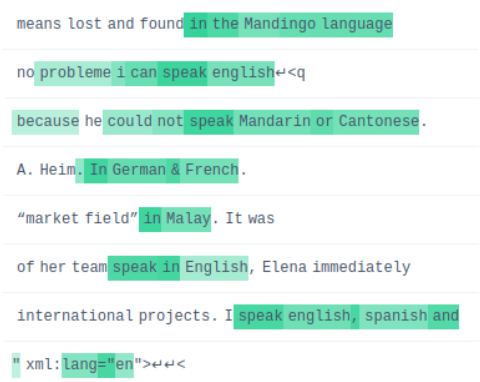


(e) Max activating examples for the country detector feature 11459.



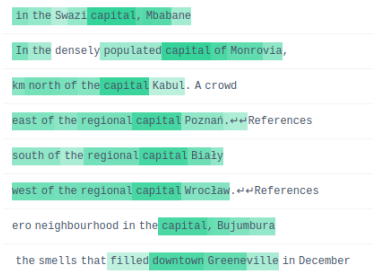
(f) Max activating examples for the journalist feature 26436. (The strongest detector for the person_profession task).

Figure 24: Max activating examples for detector features from Figure 22.



(a) Max activating examples for the language prediction executor feature 13804.

(b) Max activating examples for the repetition executor feature 12646. Extracted from the algo_last TV.

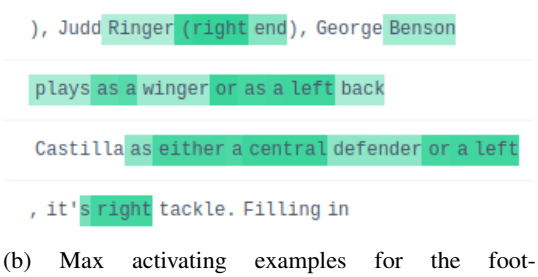


(c) Max activating examples for the capital prediction executor feature 16315.

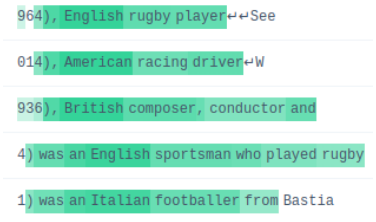
(d) Max activating examples for the translation feature 493.

Figure 25: Max activating examples for Gemma 2 2B 16k executor features from Figure 21a.

(a) Max activating examples for the antonyms executor feature 45288.



(b) Max activating examples for the football_player_position executor feature 18981.



(c) Max activating examples for the person_profession executor feature 46729.

(d) Max activating examples for translation to English executor feature 62633.

Figure 26: Max activating examples for Gemma 2 2B 65k executor features from Figure Figure 21b.

found that in Gemma 1 2B detector features for some tasks were split between several token-level features (like the journalism feature in Figure 24f), and they did not create a single feature before the task executing features activated. We attribute this to the limited expressivity of the SAEs that we used.