SCALING SPARSE AUTOENCODER CIRCUITS FOR IN-CONTEXT LEARNING

Anonymous authors
Paper under double-blind review

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 encode the model's knowledge of which task to execute and 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-executing features through attention layer and MLP.

1 INTRODUCTION

Sparse autoencoders (SAEs; Ng (2011); Bricken et al. (2023); Cunningham et al. (2023)) are a promising method for interpreting large language model (LLM) activations. However, the full potential of SAEs in interpretability research remains to be explored, since most recent SAE research either i) interprets a single SAE's features rather than the model's computation as a whole (Bricken et al., 2023), or ii) does high-level interventions in the model, but does not interpret the downstream computation impacted by the interventions Templeton et al. (2024b). In this work, we address these limitations by interpreting in-context learning (ICL), a widely-studied phenomenon in LLMs. In brief, we show that SAEs enable a) the discovery of novel circuit components (**task-detection features**; Section 4.2) and b) making existing interpretations of ICL more precise, by e.g. decomposing task vectors (Todd et al., 2024; Hendel et al., 2023) into **task-execution features** (Section 3).

In-context learning (ICL; Brown et al. (2020)) is a fundamental capability of large language models that allows them to adapt to new tasks without fine-tuning. ICL is a significantly more complex and important task than other behaviors commonly studied in circuit analysis (such as IOI in Wang et al. (2022) and Kissane et al. (2024), or subject-verb agreement and Bias-in-Bios in Marks et al. (2024)). Recent work by Todd et al. (2024) and Hendel et al. (2023) has introduced the concept of task vectors, to study ICL in a simple setting, which we follow throughout this paper.[1] In short, task vectors are internal representations of tasks formed by language models during the processing of few-shot prompts, such as "hot $\rightarrow$ cold, big $\rightarrow$ small, fast $\rightarrow$ slow". These vectors can be extracted and added into different LLM forward passes to induce task performance 0-shot, i.e. make LLMs predict that "slow" follows "fast $\rightarrow$" without explicit context. Section 2.3 provides a full introduction.

To identify **task-execution features**, we decomposed task vectors using SAEs. To achieve this, we needed to go beyond existing methods for solving the classical dictionary problem of decomposing a vector into a sparse sum of dictionary vectors (Elad, 2010). To do this, we developed a bespoke method for LLMs we call the TASK VECTOR CLEANING **(TVC)** algorithm. By running the TVC

[1]Task vectors (Hendel et al., 2023) are also called "function vectors" (Todd et al., 2024), but we use "task vectors" throughout this paper for consistency.

algorithm, we found **task-execution features**: features that can partially replace task vectors by themselves alone and have highly interpretable max activating token patterns. We validate the causal relevance of these task features through a series of steering experiments on tasks, spanning several categories like translation or factual recall. The experiments demonstrate that identified task features encode crucial information about task execution, are causally implicated in the model's ICL capabilities and can play the same role as task vectors.

To find **task-detection features**, we adapted the Sparse Feature Circuits (SFC) methodology of Marks et al. (2024) to work on the more complex ICL task and the larger Gemma-1 2B model (Gemma Team, 2024). This adaptation allowed us to discover and analyze the subgraph of key SAE latents involved in ICL, providing a more comprehensive view of the ICL circuit. We found **task-detection features** with SFC: features that play a crucial role in identifying the specific task being performed earlier in the prompt. Task-detection features are tightly connected with task-execution features through attention, as part of the whole ICL circuit.

Our findings not only advance our understanding of ICL mechanisms but also demonstrate the potential of SAEs as a powerful tool for interpretability research on larger language models. By unifying the task vectors view with SAEs and uncovering two of the most important causally implicated feature families behind ICL, we pave the way for future work to control and monitor ICL further, to improve either safety or capabilities of models.

Our main contributions are as follows:

1. We demonstrate that SAEs can be effectively used to explain the mechanisms behind the complex ICL task (ICL) in a larger model (Gemma-1 2B), which has 20x more parameters than prior models typically studied at this depth in mechanistic interpretability research (Wang et al., 2022; Marks et al., 2024).
2. We identify two core bottlenecks in the ICL circuit – **task-detection features** and **task-execution features** – and study their interactions. This provides new insights into how LLMs process and execute ICL tasks. Specifically, we discover task-detection features that identify the task being performed earlier in the prompt, which are then moved by attention heads to trigger task-specific features.
3. We present a novel task vector cleaning method that decomposes task vectors into a small set of mostly task-relevant features, enabling more precise analysis of ICL mechanisms, and important linear directions in LLMs in general.

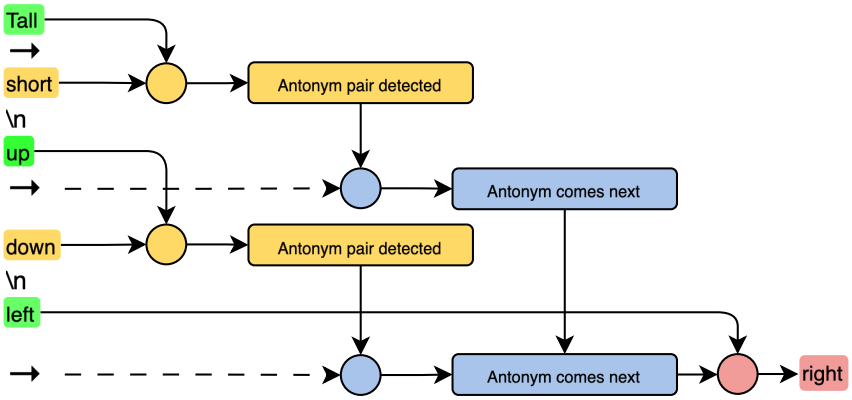


Figure 1: A diagram of the in-context learning circuit, showing task detection features (yellow) causing task execution features (blue) which cause the model to output the antonym (left → right).

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 are used to decompose the high-dimensional activations of language models into more interpretable features (Cunningham et al., 2023; Bricken et al., 2023). The basic idea behind SAEs is to train a neural network to reconstruct its input while constraining the hidden layer to have sparse activations. This process typically involves:

1. An encoder that maps the input to a sparse hidden representation.
   The SAE encoder typically has a linear encoder, a pre-activation bias and a post-encoder nonlinearity:

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

   In the JumpReLU SAE formulation (Rajamanoharan et al., 2024b), which we use[2], the activation function is JumpReLU:

$$\text{JumpReLU}_\theta(z) := zH(z - \theta) \tag{2}$$

   where θ is a learnable parameter for each SAE latent and H is the Heaviside step function.
2. A decoder that reconstructs the input from this sparse representation.
   The decoder is a simple affine projection in most formulations. The rows of the decoder are typically constrained to have unit norm with constrained optimization (Marks et al., 2024) or a custom loss penalty (Conerly et al., 2024).

$$\hat{\mathbf{x}}(\mathbf{f}) = \mathbf{W}_{\text{dec}}\mathbf{f} + \mathbf{b}_{\text{dec}} \tag{3}$$

3. A loss task that balances reconstruction accuracy with sparsity.
   Typically, the L_1 penalty on activations is used (Bricken et al., 2023) with some modifications (Rajamanoharan et al., 2024a; Conerly et al., 2024), although there are alternatives: Rajamanoharan et al., 2024b; Farrell, 2024; Riggs & Brinkman, 2024.

In our work, we train SAEs on residual stream activations and attention outputs, and also train transcoders[3] on MLP layers, all of which use the improved Gated SAE architecture (Rajamanoharan et al., 2024a).

2.2 Sparse Feature Circuits

Sparse Feature Circuits (SFCs) are a methodology introduced by Marks et al. (2024) to identify and analyze causal subgraphs of sparse autoencoder features that explain specific model behaviors. This approach combines the interpretability benefits of SAEs with causal analysis to uncover the mechanisms underlying language model behavior. The SFC methodology involves several key steps:

1. Decomposing model activations into sparse features using SAEs
2. Calculating the Indirect Effect (IE) of each feature on the target behavior
3. Identifying a set of causally relevant features based on IE thresholds
4. Constructing a circuit by analyzing the connections between these features

The IE of a model component is measured by intervening on that component and observing the change in the model's output. For a component a and a metric m, the IE is calculated as:

$$\text{IE}(m; a) = m(x|\text{do}(a = a')) - m(x) \tag{4}$$

Where $m(x|\text{do}(a = a'))$ represents the value of the metric when we intervene to set the value of component a to a', and $m(x)$ is the original value of the metric. In practice, attribution patching is used to approximate IE, allowing for efficient computation across many components simultaneously (Marks et al., 2024). We describe our modifications in Appendix D.

[2]We technically use Gated SAEs (Rajamanoharan et al., 2024a) and convert them to JumpReLU SAEs (Rajamanoharan et al., 2024b) using the procedure outline in the Gated SAEs paper (see Appendix B).

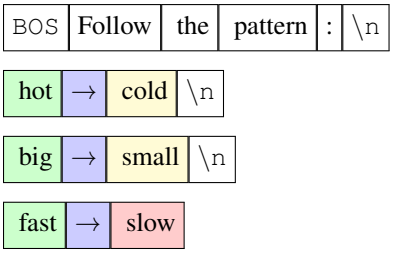
[3]Transcoders are a modification of SAEs that take MLP input and convert it into MLP output instead of trying to reconstruct the residual stream.

2.3 TASK VECTORS

Continuing from the high-level description in Section 1, task vectors were independently discovered by Hendel et al. (2023) and Todd et al. (2024). The key idea behind task vectors is that they 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. Task vectors have several important properties:

1. They can be extracted from the model's hidden states after processing ICL prompts.
2. When added to the model's activations in a zero-shot setting, they can induce task performance without explicit context.
3. They appear to encode abstract task information, independent of specific input-output examples.

To illustrate the concept, consider the following simple prompt for an antonym task, where boxes represent distinct tokens:



In this case, the task vector would encode the abstract notion of "find the antonym" rather than specific word pairs. Task vectors are typically collected by averaging the residual stream of "→" tokens at a specific layer across multiple ICL prompts for a given task. This averaged representation can then be used to study the model's internal task representations and to manipulate its behavior in zero-shot settings. We perform our analysis on the datasets derived from the Todd et al. (2024) paper. Details could be found in Appendix A.

3 DISCOVERING TASK-EXECUTION FEATURES

3.1 DECOMPOSING TASK VECTORS

To gain a deeper understanding of task vectors, we attempted to decompose them using sparse autoencoders (SAEs). However, several of our initial naive approaches faced significant challenges. Firstly, direct SAE reconstruction, i.e. passing the task vector as input to the SAE, produced noisy results with approximately 10-20 non-zero SAE features on layers of interest[4], most of which were irrelevant to the task. Moreover, this reconstruction noticeably reduced the vector's performance. These issues partly arose because task vectors are out-of-distribution inputs for SAEs, as they aggregate information from different residual streams rather than representing a single one.

We then explored inference-time optimization (ITO) (Smith, 2024) as an alternative. However, this method also failed to reconstruct task vectors using a low number of SAE features while maintaining high performance.

Given these observations, we developed a novel method called task vector cleaning. Our approach involves extracting task vectors from few-shot prompts for various tasks, reconstructing these vectors using trained SAEs, fine-tuning the SAE reconstruction weights to minimize negative log-likelihood loss on zero-shot prompts for the same task, and applying L1 regularization during fine-tuning to promote sparsity. This approach allows us to maintain the task vector performance while reducing the amoung of active SAE features to 2-4. The algorithm details can be found in Appendix C.

We compare it with the four baselines: original task vectors, naive SAE reconstruction, ITO with target L0 norm set to 5 and ITO with target L0 set to 20. To compare them, we steer the zero-shot prompt using the reconstructed task vector and calculate relative log-likelihood loss change. We then average it across all tasks. Layer-wise comparison results can be found on Figure 2.

[4]We found 3-5 interpretable features. Our cleaning algorithm can usually trim down the number to 2-4. The usual residual SAE L0 is around 44

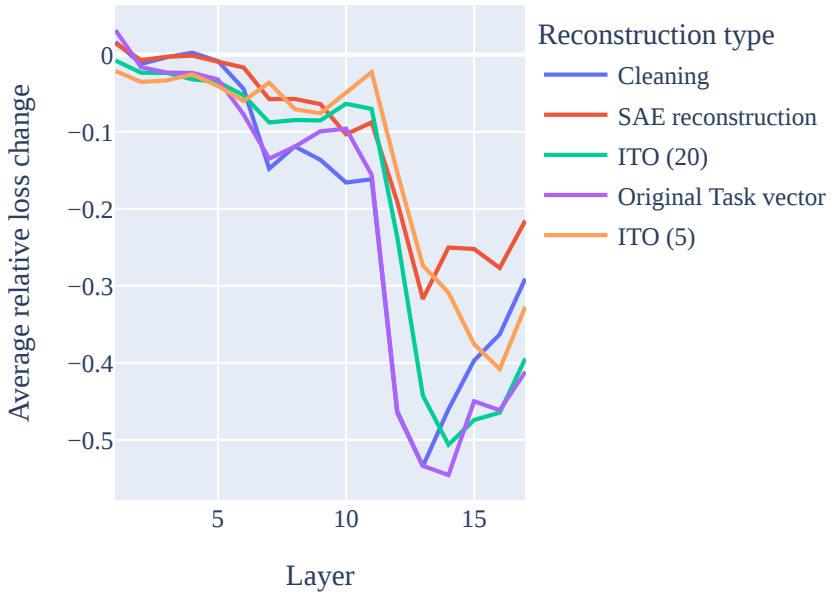


Figure 2: The effect on the model's loss by steering with different kinds of reconstructed task vectors, at each layer. We see that cleaning performs similarly to the original task vector until layer 14.

Using this method, we broke down task vectors into a small set of features. Many of these features were easy to interpret and clearly related to the task at hand. We found a particularly interesting group of features, which we called "task features." These task features have two key characteristics:

1. They activate when the model encounters examples of the relevant task in normal text.
2. In these encounters they activate on the token just before the task is completed.

For instance, imagine an antonym task feature processing the phrase "hot and cold." It would activate on the token "and," suggesting that the model expects an antonym to follow. This tells us that the model recognizes it's dealing with an antonym pair before seeing the complete pair.

3.2 Steering Experiments

To validate the causal relevance of our decomposed task features, we conducted a series of steering experiments. These experiments involved both positive and negative steering, allowing us to observe the features' impact on task performance across different contexts and model layers.

The experiments were performed on the dataset of diverse tasks taken from Todd et al. (2024). We first extracted relevant task features using our cleaning algorithm. Then steered the zero-shot prompt using them and calculated relative loss improvement, normalizing and clipping it after that. Further details can be found in Appendix F.

Figure 3 shows a heatmap of steering results for each pair of task and task-relevant feature. Higher values indicate greater improvement in the loss after steering. It can be seen that most tasks have a single feature with a high effect on them, and this feature generally does not significantly affect unrelated tasks. Another notable detail is that features from related tasks (like the translation group) at least partially affect all tasks within the group.

We have manually examined the features with the highest effect and found that their maximum activating dataset examples align with their hypothesized role in the ICL circuit. Interestingly, we observed that translation-to-English tasks all share a generic English-to-foreign task execution feature, thus requiring an additional language encoding feature for complete task encoding. This shared feature suggests a common mechanism for translation tasks, with language-specific information encoded separately.

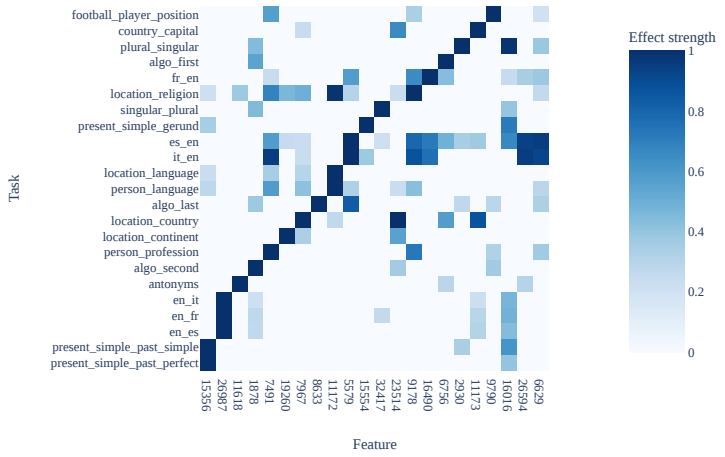


Figure 3: Heatmap showing the effect of steering with individual task-execution features for each task. We see that most features boost exactly one task, with a few exceptions for similar tasks like translating to English

4 Applying SFC to ICL

After identifying task-executing features through our task vector analysis, we sought to further expand our understanding of the in-context learning (ICL) circuit. To this end, we decided to apply the Sparse Feature Circuits (SFC) methodology (Marks et al., 2024) to the Gemma-1 2B model. However, due to the increased complexity of ICL tasks and the larger model size, the original SFC approach did not work out of the box. We had to implement several key modifications to address the challenges we encountered.

4.1 Our Modifications

4.1.1 Token Position Categorization and Feature Aggregation

We modified the SFC approach to better handle the structured nature of ICL prompts. Instead of treating each SAE feature as a separate node, we categorized token positions into the following groups:

- Prompt: The initial instruction tokens (e.g., "Follow the pattern:")
- Input: The last token before each arrow in an example pair
- Arrow: The arrow token itself ("$\rightarrow$")
- Output: The last token before each newline in an example pair
- Newline: The newline token
- Extra: Any tokens not covered by the above categories (e.g., in multi-token inputs or outputs)

This categorization allowed us to evaluate how features affect all tokens within the same category, separating features based on their role in the ICL circuit. It also enabled us to selectively disable parts of the circuit for one task while testing another, verifying the task-specificity of the found circuits.

4.1.2 Loss Function Modification

We modified the loss function to sum the loss across all pairs in the prompt, rather than calculating it only for the final pair. The original SFC paper suggested using log probabilities on a dataset of (x, y) pairs, where in the case of ICL, x would be the whole prompt, and y would be the output in the last pair. However, this approach often resulted in task-relevant features having high negative IEs on other example pairs in the prompt. This was likely due to the effect of the circuit on those pairs being lost to either diminishing gradients in backpropagation or copying circuits being much

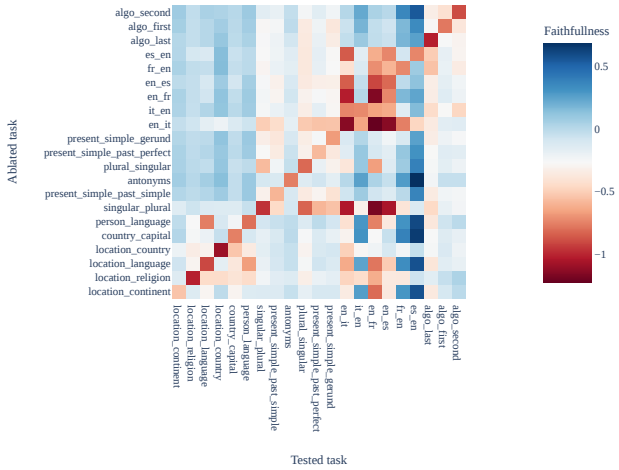


Figure 4: Heatmap showing, for each task, the change in faithfulness for every task when ablating the nodes with the highest IE for the original task.

more relevant to the prediction of the last pair. By considering all pairs, we amplified the effect of the task-solving circuit relative to the numerous cloning circuits that activate due to the repetitive nature of ICL prompts.

We also had to concentrate our analysis on two windows of layers: 11-17 and 9-11. Any extra layers in these windows resulted in important features often having highly negative IEs.

4.1.3 SFC Evaluation

To evaluate the quality of our SFC modification, we conducted a series of ablation experiments across the same dataset of ICL tasks. Our primary metric for evaluation was faithfulness, which measures how much of the original task performance is maintained after ablating specific features. We calculated faithfulness using the following formula:

$$F(M) = \frac{M - M_a}{M_n - M_a} \tag{5}$$

Where M is the current metric (loss), M_a is the fully ablated model metric, and M_n is the non-ablated model metric.

We evaluated the impact of ablating features for one task on the performance of all other tasks. Specifically, we ablated the highest Indirect Effect (IE) nodes first, continuing until we reached a faithfulness of 0.5 for the target task. This approach allowed us to assess both the specificity of the discovered circuits and their impact on related tasks. Our analysis revealed that it is possible to significantly reduce faithfulness by disabling just several hundred nodes. Furthermore, we found that we could reduce the number of active nodes to less than a thousand while keeping the performance almost intact. Extra details and faithfulness/completeness charts can be found in Appendix D.

Figure 4 presents a heatmap showing the change in faithfulness for various tasks when ablating the highest IE nodes for a single task. Several key observations can be made from this visualization:

- **Task Specificity**: Ablating most tasks does not significantly impact the performance of others, indicating that the discovered circuits are largely task-specific. This suggests that there are no common high-IE ICL-specific nodes across tasks.
- **Related Task Effects**: Tasks are grouped into categories, and we observe that ablation of related tasks has a higher effect on all tasks within the same group. This is visible as squares along the diagonal, particularly noticeable in the translation group.
- **Performance Improvement**: For some tasks, we observe that faithfulness rises well above 1.0 after ablation of other tasks. We hypothesize that this occurs because we reduce the confusion of the model by removing irrelevant execution paths.

It's worth noting that we excluded the **person_profession** and **football_player_position** tasks from Figure 4 due to the very small difference between their fully-ablated and non-ablated losses. This resulted in highly unstable faithfulness calculations for these tasks. We attribute this small difference partially to our modified loss function, as we found that calculating the loss only from the last pair results in a higher loss difference.

4.2 TASK-DETECTION FEATURES

Our modified SFC analysis revealed a second crucial component of the ICL mechanism: task-detection features. These features activate on instances of a complete task in the training data, specifically on the token that completes the task. Both task-detection and task-executing features showed high Indirect Effects (IEs) in the extracted sparse feature circuits, with task detection features connected to task execution features through attention output and transcoder nodes. We applied our task vector cleaning algorithm to extract task-detection features, identifying layer 11 as optimal for steering, preceding the layer 12 task-executing features. Details can be found in Appendix G. We

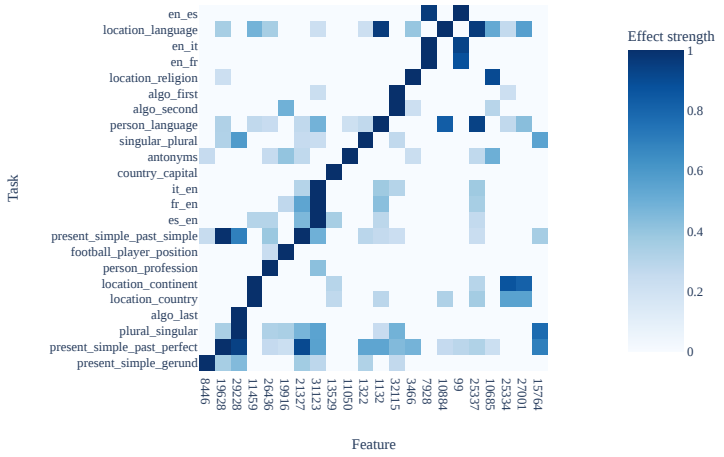


Figure 5: Heatmap showing the effect of steering with the task-detection feature most relevant to each task, on every task. We see that task detection features are typically specific to the task, with exceptions for similar tasks.

observed similar patterns as with task-executing features:

- **Task specificity**: Features strongly affect their corresponding tasks with minimal impact on unrelated tasks.
- **Related task effects**: Features from related tasks show some cross-task influence.

To evaluate the causal connection between task-detection features and task-execution features, we selected the most relevant detection and execution pairs based on steering effects and confirmed that their max activating patterns aligned with their hypothesized circuit roles. We then ablated detection directions while fixing attention patterns and measured the decrease in execution activations. Figure 6 presents the results.

The results of our causal connection analysis reveal several key insights. First, we observe strong causal connections between most task-detection and their corresponding task-executing features, supporting our hypothesis about their roles in the ICL circuit. Second, we note significant interconnectivity among translation tasks, suggesting shared circuitry for this group of related tasks. Interestingly, two tasks (**person_profession** and **present_simple_gerund**) showed unexpectedly weak connections between their detection and execution features, warranting further investigation.

These findings provide compelling evidence for the causal relationship between task-detection and task-executing features in the ICL circuit. They also highlight the interconnected nature of related tasks, particularly within the translation group.

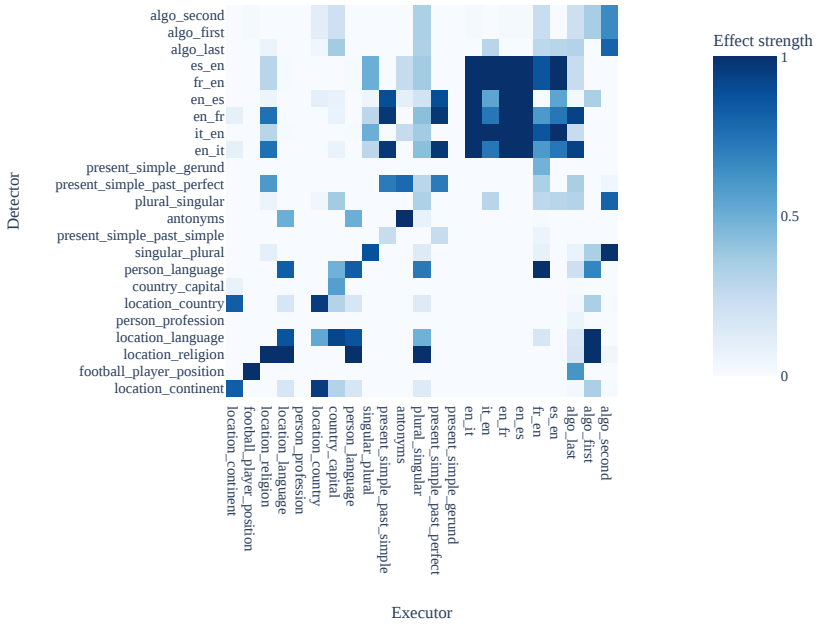


Figure 6: Heatmap showing the causal effect of the top task-detection features of each task, on the activation of the top task-executing features for every task. Averaged across all initial non-zero activations in all tasks.

5 Related work

Mechanistic Interpretability Olah et al. (2020) defines a framing for mechanistic interpretability in terms of *features* and *circuits*. It claims that neural network latent spaces have directions in them called features that correspond to meaningful variables. These features interact through model components sparsely to form circuits: interpretable computation subgraphs relevant to particular tasks. These circuits can be found through manual inspection in vision models (Cammarata et al., 2020). In language models, they can be found through manual patching (Wang et al., 2022; Hanna et al., 2023; Lieberum et al., 2023; Chan et al., 2022) or automated circuit discovery (Conmy et al. (2023); Syed et al. (2023); Bhaskar et al. (2024), though see Miller et al. (2024)). Marks et al. (2024) extends this research area to use **Sparse Autoencoders**, as discussed below.

In-Context Learning (ICL) ICL was first introduced in Brown et al. (2020) and refers to models learning to perform tasks from prompt information at test-time. There is a large area of research studying its applications (Dong et al., 2024), high-level mechanisms (Min et al., 2022) and limitations (Peng et al., 2023). Elhage et al. (2021) and Olsson et al. (2022) find *induction heads* partly responsible for in-context learning. However, since these attention heads do more than just induction (Goldowsky-Dill et al., 2023), and are not sufficient for complex task-following, induction heads alone cannot explain ICL. Anil et al. (2024, Appendix G) proposes a mechanistic hypothesis for an aspect of simple in-context task behavior. Hendel et al. (2023) and Todd et al. (2024) find that simple in-context learning tasks create strong directions in the residual stream adding which makes it possible for a network to perform tasks zero-shot, but does not explanation how task vectors form nor what interpretable components the task vectors are composed of.

Sparse Autoencoders A major roadblock to mechanistic interpretability research is superposition (Elhage et al., 2022b), where the interpretable units of neural network do not tend to align with the basis directions (e.g. neurons). Sparse autoencoders (Ng, 2011; Bricken et al., 2023) are one method of addressing this roadblock, and multiple works since proposed improvements to SAE training (Rajamanoharan et al., 2024b; Bussmann et al., 2024; Braun et al., 2024; Gao et al., 2024; Templeton et al., 2024b), and we use several more in our work (Rajamanoharan et al., 2024a; Adam Jermyn, 2024; Conerly et al., 2024). Cunningham et al. (2023), building on Bills et al. (2023), apply Conmy

et al. (2023) to find circuits in small language models. Marks et al. (2024) adapt Syed et al. (2023) in the SAE basis to find circuits and address a practical bias reduction problem. Kissane et al. (2024) apply a slightly different automated SAE algorithm (similar to ours in that it operates on single prompts) to IOI (Wang et al., 2022), using SAEs on the attention layer outputs and residual stream. Dunefsky et al. (2024) introduce *transcoders* (which are also briefly discussed in Templeton et al. (2024a) and Li et al. (2023)) to simplify analysis of circuits involving MLPs. We build on their work and train transcoders as part of our suite of Gemma-1 SAEs.

6 Conclusion

Limitations Our work focused on the simple task vector setting to study ICL (Section 2.3), which does not capture all ways that ICL is used in practice (generally involving far more tokens and open-ended tasks). We also only interpreted Gemma-1 2B, and therefore other LLM architectures or model sizes could lead to different results (though this is unlikely, since task vectors exist across models (Todd et al., 2024)). Finally, the complexity of the task studied meant our interpretations have some approximation error: attention heads matter for the detection-execution connection, but so does the succeeding MLP (Section 4.2).

Future Work Future work could extend SFC methods to work on more than a band of layers in the middle of the model (Section 2.2). Since there are many features corresponding to individual input tokens and output predictions (due to the three stages of inference in LLMs; Elhage et al. (2022a); Lad et al. (2024)), this will require further adaptation of the SFC methodology. Moreover, our multiple contributions will hopefully spur lots of further work that finds new tasks to interpret or explanations in greater depth than prior work, as discussed in our concluding paragraph below.

To summarise our work: we use SAEs to explain in context learning in greater detail than any prior mechanistic interpretability work. This provides strong evidence that Sparse Autoencoders are valuable circuit analysis tools, and the innovations developed: TVC (Section 3.1), SFC improvements (Section 2.2) and an SAE training codebase in JAX with open SAE weights (Section 7) are likely to help enable lots of other SAE research to tackle more ambitious tasks and larger models.

7 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 the penzai library 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 deeper exploration of our results. Due to the complexity of our infrastructure, we are only sharing 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, to be sufficient evidence of reproducibility.

REFERENCES

Adly Templeton Adam Jermyn. Ghost grads: An improvement on resampling, 2024. URL https://transformer-circuits.pub/2024/jan-update/index.html#dict-learning-resampling.

Cem Anil, Esin Durmus, Mrinank Sharma, Joe Benton, Sandipan Kundu, Joshua Batson, Nina Rimsky, Meg Tong, Jesse Mu, Daniel Ford, et al. Many-shot jailbreaking. 2024.

Adithya Bhaskar, Alexander Wettig, Dan Friedman, and Danqi Chen. Finding transformer circuits with edge pruning, 2024. URL https://arxiv.org/abs/2406.16778.

Steven Bills, Nick Cammarata, Dan Mossing, et al. Language models can explain neurons in language models. https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html, 2023.

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

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.

Dan Braun, Jordan Taylor, Nicholas Goldowsky-Dill, and Lee Sharkey. Identifying functionally important features with end-to-end sparse dictionary learning, 2024. URL https://arxiv.org/abs/2405.12241.

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.

Bart Bussmann, Patrick Leask, and Neel Nanda. Batchtopk: A simple improvement for topk-saes, 2024. URL https://www.alignmentforum.org/posts/Nkx6yWZNbAsfvic98/batchtopk-a-simple-improvement-for-topk-saes.

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

Lawrence Chan, Adria Garriga-Alonso, Nix Goldowsky-Dill, Ryan Greenblatt, Jenny Nitishinskaya, Ansh Radhakrishnan, Buck Shlegeris, and Nate Thomas. Causal scrubbing: A method for rigorously testing interpretability hypotheses, 2022. URL https://www.alignmentforum.org/posts/JvZhhzycHu2Yd57RN/causal-scrubbing-a-method-for-rigorously-testing.

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.

Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, et al. Towards automated circuit discovery for mechanistic interpretability. In *Proceedings of NeurIPS*, 2023.

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.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. A survey on in-context learning, 2024. URL https://arxiv.org/abs/2301.00234.

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, 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. URL https://transformer-circuits.pub/2021/framework/index.html.

Nelson Elhage, Tristan Hume, Catherine Olsson, Neel Nanda, Tom Henighan, Scott Johnston, Sheer ElShowk, Nicholas Joseph, Nova DasSarma, Ben Mann, Danny Hernandez, Amanda Askell, Kamal Ndousse, Jones, , Dawn Drain, Anna Chen, Yuntao Bai, Deep Ganguli, Liane Lovitt, Zac Hatfield-Dodds, Jackson Kernion, Tom Conerly, Shauna Kravec, Stanislav Fort, Saurav Kadavath, Josh Jacobson, Eli Tran-Johnson, Jared Kaplan, Jack Clark, Tom Brown, Sam McCandlish, Dario Amodei, and Christopher Olah. Softmax linear units. *Transformer Circuits Thread*, 2022a. https://transformer-circuits.pub/2022/solu/index.html.

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*, 2022b.

Eoin Farrell. Experiments with an alternative method to promote sparsity in sparse autoencoders, 2024. URL https://www.lesswrong.com/posts/cYA3ePxy8JQ8ajo8B/experiments-with-an-alternative-method-to-promote-sparsity.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. Scaling and evaluating sparse autoencoders, 2024. URL https://arxiv.org/abs/2406.04093.

Gemma Team. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

Nicholas Goldowsky-Dill, Chris MacLeod, Lucas Sato, and Aryaman Arora. Localizing model behavior with path patching, 2023. URL https://arxiv.org/abs/2304.05969.

Michael Hanna, Ollie Liu, and Alexandre Variengien. How does gpt-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model, 2023. URL https://arxiv.org/abs/2305.00586.

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.

Connor Kissane, Robert Krzyzanowsky, Arthur Conmy, and Neel Nanda. Attention output saes improve circuit analysis, 2024. URL https://www.alignmentforum.org/posts/EGvtgB7ctifzxZg6v/attention-output-saes-improve-circuit-analysis.

Vedang Lad, Wes Gurnee, and Max Tegmark. The remarkable robustness of llms: Stages of inference?, 2024. URL https://arxiv.org/abs/2406.19384.

Max Li, Sam Marks, and Aaron Mueller. dictionary_learning repository, 2023. URL https://github.com/saprmarks/dictionary_learning?tab=readme-ov-file#extra-functionalitysupported-by-this-repo. Accessed on September 30, 2024.

Tom Lieberum, Matthew Rahtz, János Kramár, et al. Does circuit analysis interpretability scale? evidence from multiple choice capabilities in chinchilla, 2023. URL https://arxiv.org/abs/2307.09458.

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.

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.

Joseph Miller, Bilal Chughtai, and William Saunders. Transformer circuit faithfulness metrics are not robust, 2024. URL https://arxiv.org/abs/2407.08734.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022. URL https://arxiv.org/abs/2202.12837.

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.

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.

Hao Peng, Xiaozhi Wang, Jianhui Chen, Weikai Li, Yunjia Qi, Zimu Wang, Zhili Wu, Kaisheng Zeng, Bin Xu, Lei Hou, and Juanzi Li. When does in-context learning fall short and why? a study on specification-heavy tasks, 2023. URL https://arxiv.org/abs/2311.08993.

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.

Logan Riggs and Jannik Brinkman. Improving sae's by sqrt()-ing l1 removing lowest activating features, 2024. URL https://www.lesswrong.com/posts/YiGs8qJ8aNBgwt2YN/improving-sae-s-by-sqrt-ing-l1-and-removing-lowest.

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.

Adly Templeton, Joshua Batson, Adam Jermyn, and Chris Olah. Predicting future activations, January 2024a. URL https://transformer-circuits.pub/2024/jan-update/index.html#predict-future. Accessed on September 30, 2024.

Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024b. URL https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html.

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.

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 (Gemma Team, 2024). The model's architecture is largely the same as that of Llama (Dubey et al., 2024) with the exception of 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 is as follows: Here's the list with task descriptions:

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 SAEs are trained with a learning rate of 1e-3 and Adam betas of 0.0 and 0.99 for 150M ($\pm$100) tokens. 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 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 hasn't fired in 2000 steps. We use Anthropic's input normalization and sparsity loss for Gemma 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 use 8 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 (Gemma Team, 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, and caching LM activations is not the main bottleneck for us. For this and other reasons, we don't do SAE sparsity coefficient sweeps so as 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 amount of dead features and led to a Pareto improvement over float32 SAEs.

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

C 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.

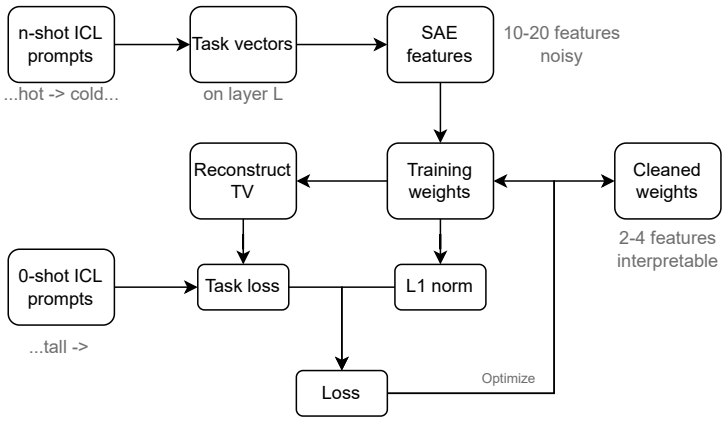


Figure 7: Overview of the task vector cleaning algorithm.

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 performance.

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:

BOS	Follow	the	pattern	:	\n

tall	→	short	\n

...

old	→	young	\n

hot	→	cold

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

The algorithm initializes 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 L0 norm of weights with coefficient l to the loss function. The algorithm implements early stopping when the L0 norm remains unchanged for n iterations.

The hyperparameters l, 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.

It's worth noting that we successfully applied this method to the recently released Gemma 2 2B model using Gemma Scope SAE suite (Lieberum et al., 2024).

D DETAILS OF OUR SFC IMPLEMENTATION

D.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 residuals to each layer. Note that we do not use SAEs during the stage.

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. Like Marks et al. (2024), we do not perform full ablation of circuit edges.

D.2 Faithfulness charts

Figure 8 shows averaged node trimming effect on faithfulness across all tasks. We follow methodology of Marks et al. (2024) thresholding removing nodes with low IE first. We can see that the circuits keep at least 0.8 faithfulness on average just with 1000 nodes.

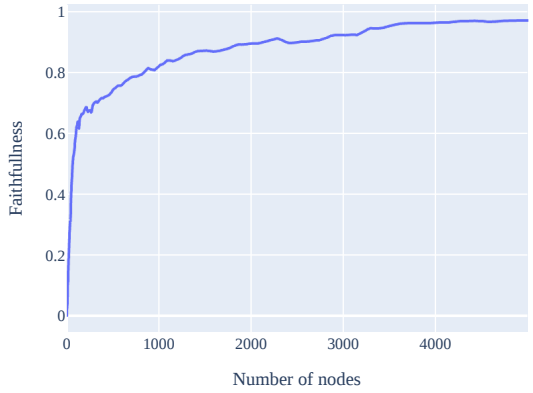


Figure 8: Average faithfulness across tasks depending on the amount of nodes left in the circuit.

Figure 9 shows averaged inverse node trimming effect on faithfulness across all tasks. Marks et al. (2024) calls this metric completeness and calculates it as faithfulness of the model just with the circuit ablated. We calculate it by thresholding nodes starting with those that have the highest IE. We can see that ablation of just even several hundred nodes have drastic impact on faithfulness.

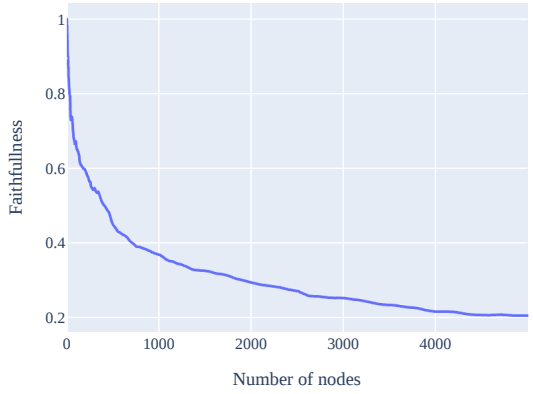


Figure 9: Average faithfulness across tasks depending on the amount of important nodes ablated from the circuit .

E EXAMPLE CIRCUITS

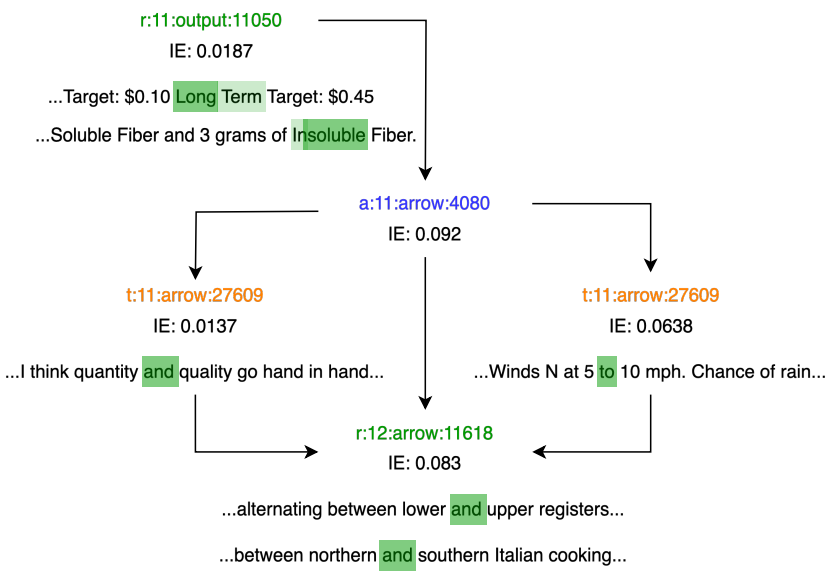


Figure 10: An example of a circuit found using our SFC variant. We focused on a subcircuit with high indirect effects. Maximum activating examples from the SAE training distribution are included.

An example output of our circuit cleaning algorithm can be found in Figure 10. 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.

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-executing features across all tasks. Through this process, we established an optimal steering scale of 25, which we then applied consistently across all subsequent experiments.

For each pair of task and feature, we performed steering 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.

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.

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

BOS	Follow	the	pattern	:	\n

hot	→	cold

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 as it happens on real prompts.

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 lay 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 5.

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

BOS	Follow	the	pattern	:	\n

X	→	Y	\n

hot	→	cold