Interpretability in Parameter Space: Minimizing Mechanistic Description Length with Attribution-based Parameter Decomposition

Dan Braun* **Lucius Bushnaq*** **Stefan Heimersheim*** **Jake Mendel**

Lee Sharkey†

Apollo Research‡

Summary

Mechanistic interpretability aims to understand the internal mechanisms learned by neural networks. Despite recent progress toward this goal, it remains unclear how best to decompose neural network parameters into mechanistic components. We introduce *Attribution-based Parameter Decomposition* (APD), a method that directly decomposes a neural network's parameters into components that (i) are faithful to the parameters of the original network, (ii) require a minimal number of components to process any input, and (iii) are maximally simple. Our approach thus optimizes for a minimal length description of the network's mechanisms. We demonstrate APD's effectiveness by successfully identifying ground truth mechanisms in multiple toy experimental settings: Recovering features from superposition; separating compressed computations; and identifying cross-layer distributed representations. While challenges remain to scaling APD to non-toy models, our results suggest solutions to several open problems in mechanistic interpretability, including identifying minimal circuits in superposition, offering a conceptual foundation for 'features', and providing an architecture-agnostic framework for neural network decomposition.

*Core research contributor

†Correspondence to lee@apolloresearch.ai

‡Contributions statement below

Contents

1 Introduction

Mechanistic interpretability aims to improve the trustworthiness of increasingly capable AI systems by making it possible to understand their internals. The field's ultimate goal is to map the parameters of neural networks to human-understandable algorithms. A major barrier to achieving this goal is that it is unclear how best to decompose neural networks into the individual mechanisms that make up these algorithms, if such mechanisms exist [Bussman et al., 2024] [Sharkey et al 2025., (forthcoming)]. This is because the mechanistic components of neural networks do not in general map neatly onto individual architectural components, such as individual neurons [Hinton, 1981, Churchland and Shenoy, 2007, Nguyen et al., 2016], attention heads [Janiak et al., 2023, Jermyn et al., 2023], or layers [Yun et al., 2021, Lindsay et al., 2024, Meng et al., 2023b].

Sparse dictionary learning is currently the most popular approach to tackling this problem [Lee et al., 2007, Yun et al., 2021, Sharkey et al., 2022, Cunningham et al., 2023, Bricken et al., 2023]. This method decomposes the neural activations of the model at different hidden layers into sets of sparsely activating latent directions. Then, the goal is to understand how these latent directions interact with the network's parameters to form circuits (or 'mechanisms') that compute the activations at subsequent layers [Cammarata et al., 2020, Olah, 2023, Sharkey, 2024, Olah, 2024a]. However, sparse dictionary learning appears not to identify canonical units of analysis for interpretability [Bussman et al., 2024]; suffers from significant reconstruction errors [Makelov et al., 2024, Gao et al., 2024]; optimizes for sparsity, which may not be a sound proxy for interpretability in the limit [Chanin et al., 2024, Till, 2024, Ayonrinde et al., 2024]; and leaves feature geometry unexplained [Engels et al., 2024a, Mendel, 2024], among a range of other issues (see Sharkey et al. 2025 (forthcoming) for review). These issues make it unclear how to use sparsely activating directions in activation space to identify the network's underlying mechanisms. Here, we investigate an approach to more directly decompose neural networks parameters into individual mechanisms.

There are many potential ways to decompose neural network parameters, but not all of them are equally desirable for mechanistic interpretability. For example, a neuron-by-neuron description of how a neural network transforms inputs to outputs is a perfectly accurate account of the network's behavior. But this description would be unnecessarily long and would use polysemantic components. This decomposition fails to carve the network at its joints because it does not reflect the network's deeper underlying mechanistic structure.

We therefore ask what properties an ideal mechanistic decomposition of a neural network's parameters should have. Motivated by the minimum description length principle, which states that the shortest description of the data is the best one, we identify three desirable properties:

- **Faithfulness**: The decomposition should identify a set of components that sum to the parameters of the original network.[1]
- **Minimality**: The decomposition should use as few components as possible to replicate the network's behavior on its training distribution.
- **Simplicity**: Components should each involve as little computational machinery as possible.

Insofar as we can decompose a neural network's parameters into components that exhibit these properties, we think it would be justified to say that we have identified the network's underlying *mechanisms*: Faithfulness ensures that the decomposition reflects the parameters of and computations implemented by the network. Minimality ensures the decomposition comprises specialised components that that play distinct roles. And simplicity encourages the components to be individual, basic computational units, rather than compositions of them.

To this end, we introduce *Attribution-based Parameter Decomposition (APD)*, a method that decomposes neural network parameters into components that are optimized for these three properties. In brief, APD involves decomposing the parameter vector of any neural network into a sum of

[1] Faithfulness to the original network's parameters is subtly different from the 'behavioral faithfulness of a circuit', which has been studied in other literature [Wang et al., 2022]. Components that sum to the parameters of the original network will necessarily exhibit behavior that is faithful to the original network (assuming all components are included in the sum). But behavioral faithfulness does not imply faithfulness to a network's parameters, since different parameters may exhibit the same behavior. Our criterion is therefore stricter, and relates to a decomposition of parameters rather than Wang et al. [2022]'s definition of a circuit.

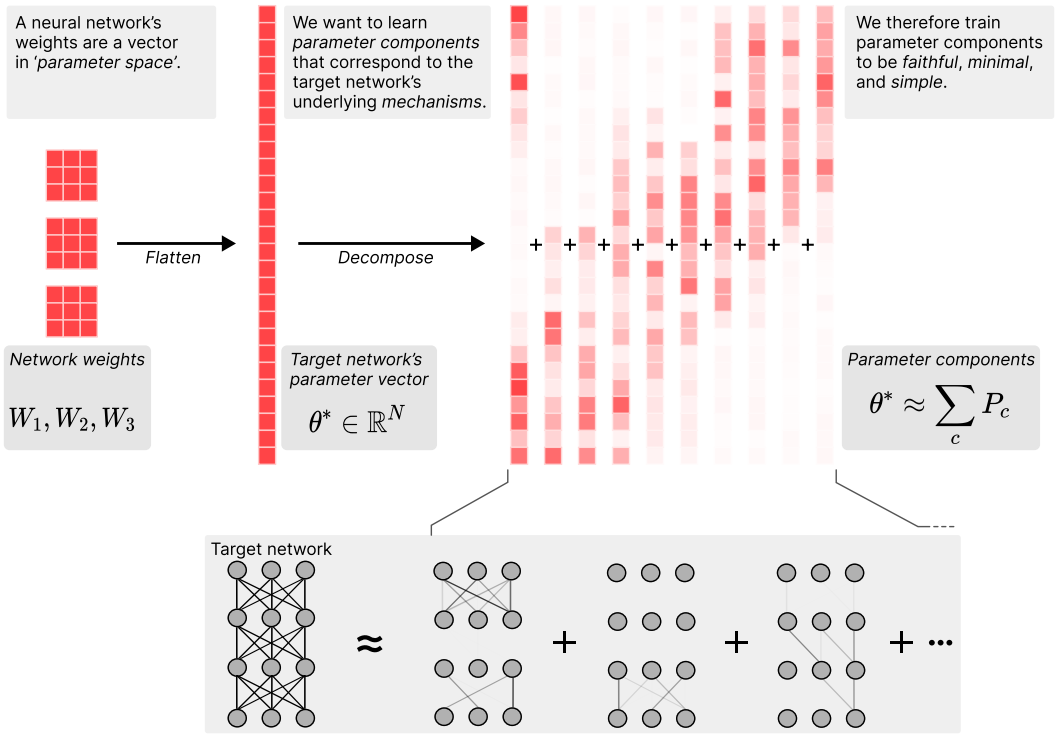

Figure 1: Decomposing a target network's parameters into parameter components that are faithful, minimal, and simple.

parameter components. They are optimized such that they sum to the target parameters while only a minimal number of them are necessary for the causal process that computes the network's output for any given input. They are also optimized to be less complex individually than the entire network, in that they span as few directions in activation space as possible across all layers.

Our approach leverages the idea that, for any given input, a neural network should not require all of its mechanisms simultaneously [Veit et al., 2016, Zhang et al., 2022, Dong et al., 2023]. On any given input, it should be possible to ablate unused mechanisms without influencing the network's computations. This would let us study the mechanisms in relative isolation, making them easier to understand. For example, suppose a neural network uses only one mechanism to store the knowledge that '*The sky is blue*' in its parameters. Being a 'mechanism', as defined above, it is maximally simple, but may nevertheless be implemented using multiple neurons scattered over multiple layers of the model. Despite being spread throughout the network, we contend that there is a single vector in parameter space that implements this knowledge. On inputs where the model uses this stored knowledge, the model's parameters along this direction cannot be varied without changing the model's output. But on inputs where the model does not use this fact, ablating the model parameters along this direction to zero should not change the output.

Our method has several connections to other contemporary approaches in mechanistic interpretability, such as sparse dictionary learning [Sharkey et al., 2022, Cunningham et al., 2023, Bricken et al., 2023, Braun et al., 2024, Dunefsky et al., 2024, Ayonrinde et al., 2024, Lindsay et al., 2024], causal mediation analysis [Vig et al., 2020, Wang et al., 2022, Conmy et al., 2024, Syed et al., 2023, Kramár et al., 2024, Geiger et al., 2024], weight masking [Mozer and Smolensky, 1988, Phillips et al., 2019, Csordás et al., 2021, Cao et al., 2021a], and others, while attempting to address many of their shortcomings. It also builds on work that explores the theory of computation in superposition [Vaintrob et al., 2024, Bushnaq and Mendel, 2024].

This paper is structured as follows: We first describe our method in Section 2. In Section 3, we provide empirical support for our theoretical work by applying APD to three toy models where we have access to ground-truth mechanisms. First, in a toy model of superposition, APD recovers mechanisms corresponding to individual input features represented in superposition (Section 3.1).

Second, in a model performing compressed computation – where a model is tasked with computing more nonlinear functions than it has neurons – APD finds parameter components that represent each individual function (Section 3.2). Third, when extending this model of compressed computation to multiple layers, APD is still able to learn components that represent the individual functions, even those that span multiple layers (Section 3.3). In Section 4, we discuss our results, the current state of APD, and possible next steps in its development, with conclusions in Section 5. We include a detailed discussion on related work in Section 6.

2 Method: Attribution-based Parameter Decomposition

In this section, we outline our method, Attribution-based Parameter Decomposition (APD). First, we outline why we define 'mechanisms' as vectors in parameter space (Section 2.1). Then, we discuss how our method optimizes parameter components to be faithful, minimal, and simple, thus identifying the network's mechanisms (Section 2.2). While a brief description of APD suffices to understand our experiments, a more detailed description and motivation can be found in Appendix A.

2.1 Defining 'mechanism space' as parameter space

To identify a neural network's mechanisms, we must first identify the space in which they live. The weights of neural networks can be flattened into one large parameter vector in *parameter space* (Figure 1). During learning, gradient descent iteratively etches a neural network's mechanisms into its parameter vector. This makes it natural to look for mechanisms in the same vector space as the whole network.

Vectors in parameter space also satisfy a broad range of criteria that we require individual mechanisms to have. *Mechanism space* should:

- **Span the same functional range as the target network**: We want mechanisms that perform a subcomponent of the algorithm implemented by the target neural network. We therefore expect mechanisms to lie somewhere in between "*Doing everything the target network does*" and "*Doing nothing*". Parameter space contains such mechanisms: The target network's parameter vector does everything that the target network does. And the zero parameter vector serves as a 'null mechanism'. Vectors that lie 'in between' serve as candidates for individual mechanisms[2].
- **Accommodate basis-unaligned mechanisms**: It has long been known that neural representations may span multiple neurons [Hinton, 1981, Churchland and Shenoy, 2007, Nguyen et al., 2016]. However, even more recent work suggests that representations may span other architectural components, such as separate attention heads [Janiak et al., 2023, Jermyn et al., 2023] or even layers [Yun et al., 2021, Lindsay et al., 2024, Meng et al., 2023b]. Vectors in parameter space span all of these components and can therefore implement computations that happen to be distributed across them.
- **Accommodate superposition**: Neural networks appear to be able to represent and perform computation on variables in superposition [Elhage et al., 2022, Vaintrob et al., 2024, Bushnaq and Mendel, 2024]. We would like a space that can compute more functions than they have neurons. Vectors in parameter space support this requirement, theoretically [Bushnaq and Mendel, 2024] and in practice, as we will demonstrate in our experiments.
- **Accommodate multidimensional mechanisms**: Some representations in neural networks appear to be multidimensional [Engels et al., 2024a]. We therefore want to be able to identify mechanisms that can do multidimensional computations on these representations. Vectors in parameter space satisfy this requirement.

[2]For this to be meaningful, we need a reasonable definition of what it means for vectors to lie 'in between' the target network's parameter vector and the zero vector. One reasonable definition is that vectors that are 'in between' should have a lower magnitude than the target network's parameter vector and have positive cosine similarity with the target network's parameters. This definition is implied by the method introduced in this work, although it does not optimize for these properties directly.

Having defined mechanism space as parameter space, we now want a method to identify a set of parameter components that correspond to the network's underlying mechanisms. In particular, we want to identify parameter components that satisfy the faithfulness, minimality, and simplicity criteria.

2.2 Identifying networks' mechanisms using Attribution-based Parameter Decomposition

APD aims to minimize the total description length of the mechanistic components used by the network *per data point* over the training dataset. It decomposes the network's parameters $\theta^* \in \mathbb{R}^N$ into a set of parameter components and directly optimizes them to be faithful, minimal, and simple. A more detailed discussion of how our method relates to the Minimum Description Length principle can be found in Appendix A.2.

Optimizing for faithfulness: We decompose a network's parameters $\theta^*_{l,i,j}$, where l indexes the network's weight matrices and i, j index rows and columns, by defining a set of C parameter components $P_{c,l,i,j}$. Their sum is trained to minimize the mean squared error (MSE) with respect to the target network's parameters, $\mathcal{L}_{\text{faithfulness}} = \text{MSE}(\theta^*, \sum_{c=1}^{C} P_c)$.

Optimizing for minimality: The parameter components are also trained such that, for a given input, a minimal number of them is used to explain the network's output (Figure 2). To achieve this, we use two steps:

1. **Attribution step:** We want estimate the causal importance of each parameter component P_c for the network's output on each datapoint $f_{\theta^*}(x)$. In this step, we therefore calculate the *attributions* of each parameter component with respect to the outputs, $A_c(x) \in \mathbb{R}$. It would be infeasibly expensive to compute this exactly, since it would involve a large number of causal interventions, requiring one forward pass for every possible combination of component ablations. We therefore use an approximation. In this work, we use gradient attributions [Mozer and Smolensky, 1988, Molchanov et al., 2017, Nanda, 2022a, Syed et al., 2023], but other attribution methods may work as well. This step therefore involves one forward pass with the target model to calculate the output and one backward pass to compute the attributions with respect to the parameters, which are used to calculate the attributions with respect to each parameter component (Equation 8).
2. **Minimality training step**: We sum only the top-k most attributed parameter components, yielding a new parameter vector $\kappa(x) \in \mathbb{R}^N$, and use it to perform a forward pass. We train the output of the top-k most attributed parameter components to match the target network's outputs by minimizing $\mathcal{L}_{\text{minimality}} = D(f_{\theta^*}(x), f_{\kappa(x)}(x))$, where D is some distance or divergence measure (Equation 17). This step trains the *active* parameter components to better reconstruct the target network's behavior on a given data point. This should increase the attribution of active components on that data. In some cases, we also train some of the hidden activations to be similar on both forward passes, since it may otherwise be possible for APD to learn solutions that produce the same outputs using different computations (See Appendix A.2.2 for details).

In our experiments, we use batch top-k [Bussmann et al., 2024] to select a fixed number of active parameter components for the minimality training step (a.k.a sparse forward pass) per batch. This sidesteps the issue of needing to select a specific number of active parameter components for each sample, although does present other issues (see Appendix C.2).

Optimizing for simplicity: The components are also trained to be 'simpler' than the parameters of the target network. We would like to penalize parameter components that span more ranks or more layers than necessary by minimizing the sum of the ranks of all the matrices in active components: $\sum_{c=1}^{C} s_c(x) \sum_l \text{rank}(P_{c,l})$, where $s_c(x) \in \{0, 1\}$ indicates active components. In practice, we minimize the L_p norm of the singular values of weight matrices in active components using a loss $\mathcal{L}_{\text{simplicity}}(x) = \sum_{c=1}^{C} s_c(x) \sum_{l,m} ||\lambda_{c,l}||_p^p$, where $\lambda_{c,l,m}$ are the singular values of parameter component c in layer l. This is also known as the Schatten-p norm[3]. For a discussion of how to calculate Schatten norms efficiently, see Appendix A.2.2.

[3]Since $p \in (0, 1)$, the Schatten-p norm and L_p norms here are technically quasi-norms. For brevity, we refer to them as norms throughout.

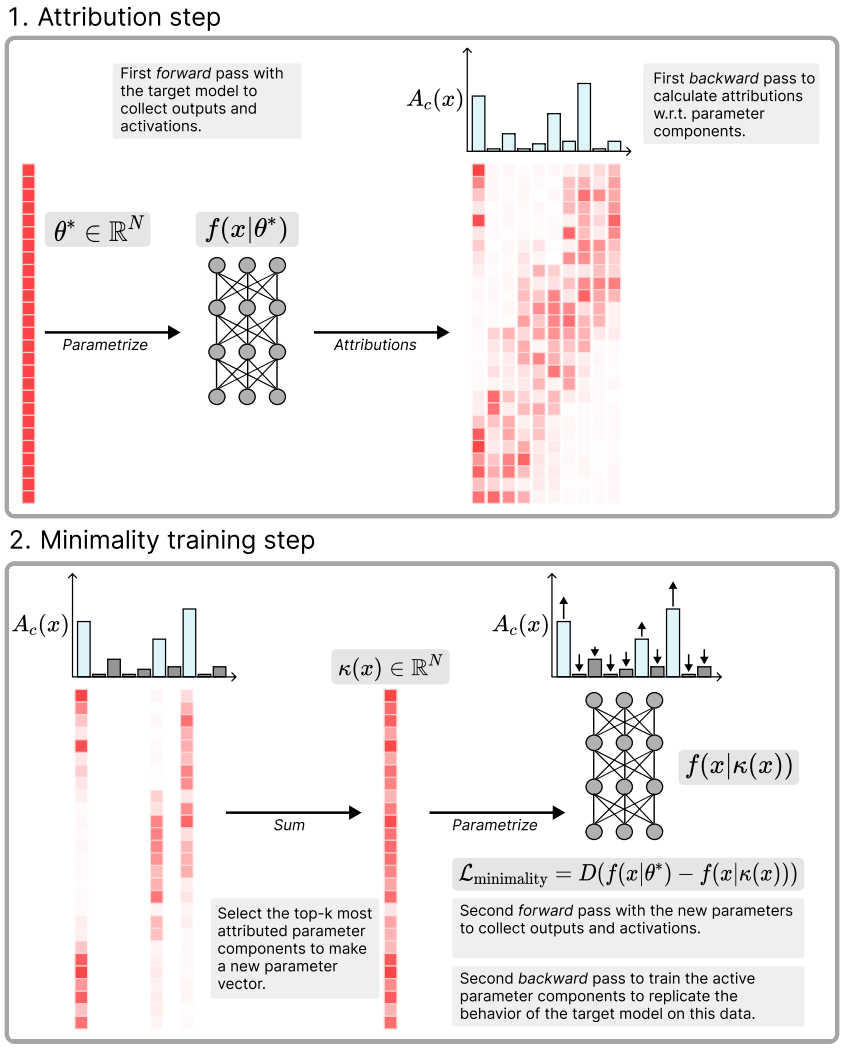

Figure 2: **Top:** Step 1: Calculating parameter component attributions $A_c(x)$. **Bottom:** Step 2: Optimizing minimality loss $\mathcal{L}_{\text{minimality}}$.

Biases Currently, we do not decompose the network's biases. Biases can be folded into the weights by treating them as an additional column in each weight matrix, meaning they can in theory be decomposed like any other type of parameter. However, in this work, for simplicity we treat them as their own parameter component that is active for every input, and leave their decomposition for future work.

Summary: In total, we use three losses:

1. A faithfulness loss ($\mathcal{L}_{\text{faithfulness}}$), which trains the sum of the parameter components to approximate the parameters of the target network.
2. A minimality loss ($\mathcal{L}_{\text{minimality}}$), which trains the top-$k$ most attributed parameter components on any given input to produce the same output (and some of the same hidden activations) as the target network, thereby increasing their attributions on those inputs.
3. A simplicity loss ($\mathcal{L}_{\text{simplicity}}$), which penalizes parameter components that span more ranks or more layers than necessary.

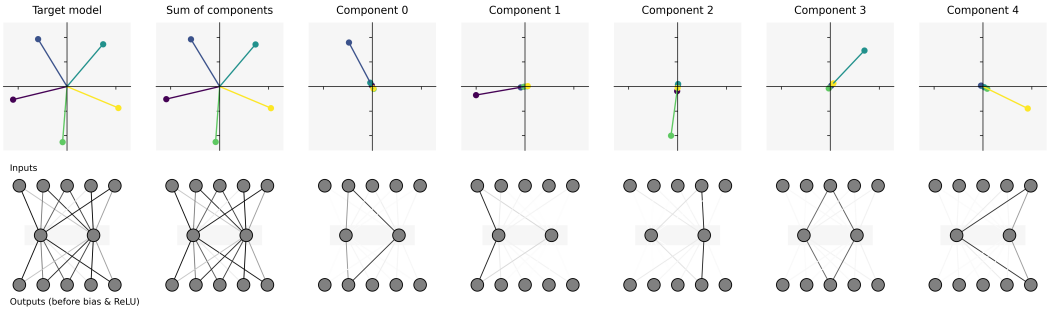

Figure 3: Results of running APD on TMS. **Top row:** Plot of the columns of the weight matrix of the target model, the sum of the APD parameter components, and each individual parameter component. Each parameter component corresponds to one mechanism, which in this model each correspond to one 'feature' in activation space [Elhage et al., 2022]. **Bottom row:** Depiction of the corresponding parametrized networks.

3 Experiments: Decomposing neural networks into mechanisms using APD

In this section, we demonstrate that APD succeeds at finding faithful, minimal, and simple parameter components in three toy settings with known 'ground truth mechanisms'. These are

1. Elhage et al. [2022]'s toy model of superposition (Section 3.1);
2. A novel toy model of compressed computation, which is a model that computes more nonlinear functions than it has neurons (Section 3.2);
3. A novel toy model of cross-layer distributed representations (Section 3.3).

In all three cases, APD successfully identifies the ground truth mechanisms up to a small error. The target models are trained using AdamW [Loshchilov and Hutter, 2019], though we also study a handcoded model in Appendix B.1. Additional figures and training logs can be found here. All experiments were run using github.com/ApolloResearch/apd. Training details and hyperparameters can be found in Appendix D.

3.1 Toy Model of Superposition

Our first model is Elhage et al. [2022]'s toy model of superposition (TMS), which can be written as $\hat{x} = \text{ReLU}(W^\top W x + b)$, with weight matrix $W \in \mathbb{R}^{m_1 \times m_2}$. The model is trained to reconstruct its inputs, which are sparse sums of one-hot m_2-dimensional input features, scaled to a random uniform distribution $[0, 1]$. Typically, $m_1 < m_2$, so the model is forced to 'squeeze' representations through a m_1-dimensional bottleneck. When the model is trained on sufficiently sparse data distributions, it can learn to represent features in superposition in this bottleneck. For certain values of m_1 and m_2, the columns of the W matrix often form regular polygons in the m_1-dimensional hidden activation space (Figure 3 leftmost panel).

What are the 'ground truth mechanisms' in this toy model? Let us define a set of matrices $\{Z^{(c)}\}$ that are zero everywhere except in the c^{th} column, where they take the values $W_{:,c}$:

$$Z^{(c)}_{:,j} = \begin{cases} W_{:,c} & \text{if } j = c, \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{1}$$

The data are sparse, so only some of the model's weights W are used on any given datapoint. Suppose we have a datapoint where only dataset feature c is active. On this datapoint, we can replace W with $Z^{(c)}$ and the model outputs would be almost identical (since the interference terms from inactive features should be small and below the learned ReLU threshold). Intuitively, a column of W is only 'used' if the corresponding data feature is active. This makes the matrices $\{Z^{(c)}\}$ good candidates for optimal 'minimality'. The matrices are also 'faithful', since $\sum_c Z^{(c)} = W$. They are also very simple because each matrix is rank 1, consisting of the outer product of the column of $W_{:,c}$ and the one-hot vector $e_c \in \mathbb{R}^{m_2}$ that indexes the nonzero column c:

$$Z^{(c)} = W_{:,c} e_c^\top \tag{2}$$

The matrices $\{Z^{(c)}\}$ are therefore reasonable candidates for the ground truth 'mechanisms' of this model [4]. We would therefore like APD to learn parameter components that correspond to them.

APD Results: TMS

We find that APD can successfully learn parameter components $\{P_c\}$ that closely correspond to the matrices $\{Z^{(c)}\}$ (Figure 3). We observe that the sum of the components is equal to W in the target network.

For illustrative purposes, we have focused on the setting with 5 input features ($m_2 = 5$) and a hidden dimension of 2 ($m_1 = 2$). However, training an APD model (and to a lesser extent, a target model) in this setting is very brittle and less effective than a higher-dimensional setting. Indeed, for the 2-dimensional hidden space setting, the results presented in this section required an adjustment of using attributions taken from the APD model rather than the target model (an adjustment that proved not to be beneficial in other settings). We expect that this brittleness has to do with the large amount of interference noise between the input features when projected onto the small 2-dimensional space. We thus also analyze a setting with 40 input features and 10 hidden dimensions. We use TMS_{5-2} to denote the setting with 5 input features and 2 hidden dimensions, and TMS_{40-10} to denote the setting with 40 input features and 10 hidden dimensions.

To show how close the learned parameter components are to the columns of W in the target model, we measure the angle between each column of W and the corresponding column in the component it lines up best with. We also measure how close their magnitudes are. To quantify the angles, we calculate the mean max cosine similarity (MMCS) [Sharkey et al., 2022]

$$\text{MMCS}(W, \{P_c\}) = \frac{1}{m_2} \sum_{j=1}^{m_2} \max_c \left(\frac{P_{c,:,j} \cdot W_{:,j}}{||P_{c,:,j}||_2 ||W_{:,j}||_2} \right), \tag{3}$$

where $c \in C$ are parameter component indices and $j \in [1, m_2]$ are input feature indices. A value of 1 for MMCS indicates that, for all input feature directions in the target model, there exists a parameter component whose corresponding column points in the same direction. To quantify how close their magnitudes are, we calculate the mean L2 Ratio (ML2R) between the Euclidean norm of the columns of W and the Euclidean norm of the columns of the parameter components P_c with which they have the highest cosine similarity

$$\text{ML2R}(W, \{P_c\}) = \frac{1}{m_2} \sum_{j=1}^{m_2} \frac{||P_{\text{mcs}(j),:,j}||_2}{||W_{:,j}||_2}, \tag{4}$$

where $\text{mcs}(j)$ is the index of the component that has maximum cosine similarity with weight column j of the target model. A value close to 1 for the ML2R indicates that the magnitude of each parameter component is close to that of its corresponding target model column.

The MMCS and ML2R for both TMS_{5-2} and TMS_{40-10} are shown in Table 1. We see in both settings that the MMCS values are ≈ 1. This indicates that the parameter components are close representations of the target model geometrically.

However, the ML2R is close to 0.9, implying there is some amount of 'shrinkage', reminiscent of feature shrinkage in SAEs [Jermyn et al., 2024, Wright and Sharkey, 2024]. We speculate that shrinkage in APD is caused by a forced trade-off between top-k reconstruction $\mathcal{L}_{\text{minimal}}$ and the Schatten norm penalty $\mathcal{L}_{\text{simplicity}}$. In this specific case, we suspect it might be due to noise in the target model output due to high interference between the input features. The parameter components are incentivised by $\mathcal{L}_{\text{minimal}}$ to reconstruct this noise such that each learns small amounts of different ground truth mechanisms $\{Z^{(c)}\}$. Additional visualizations of the TMS APD models can be found in a WandB report here.

It is worth reflecting on the differences between the APD solution and the decompositions that other commonly used matrix decomposition methods would yield, such as singular value decomposition [Millidge and Black, 2022, Meller and Berkouk, 2023] or non-negative matrix factorization [Petrov

[4] The c^{th} 'mechanism' in this model technically corresponds to $Z^{(c)}$ *and* the c^{th} element of the bias. For simplicity, we do not decompose biases in our current implementation and treat all biases as one component that is always active.

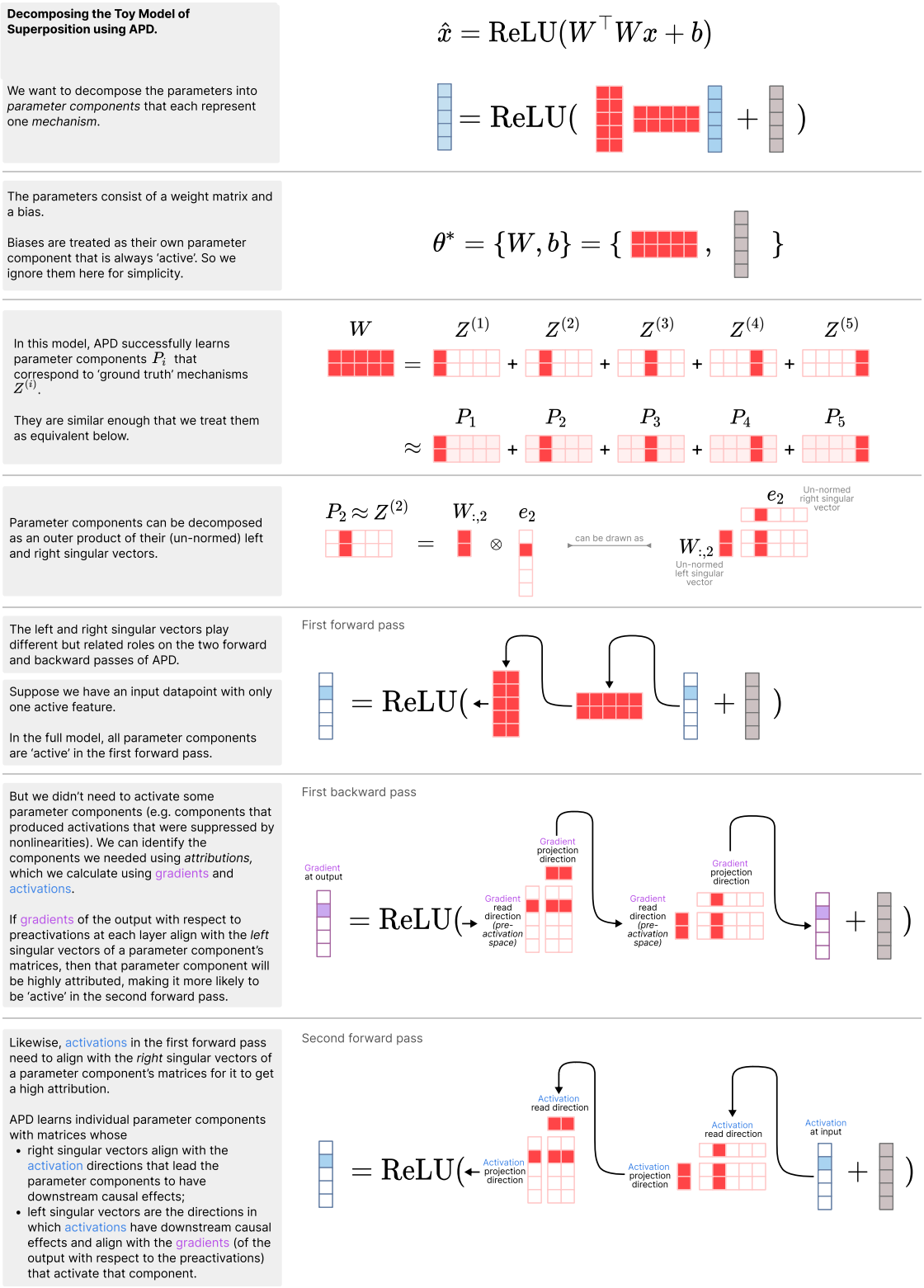

Figure 4: Decomposing TMS with APD.

	MMCS	ML2R
TMS_{5-2}	0.998 ± 0.000	0.893 ± 0.004
TMS_{40-10}	0.996 ± 0.003	0.935 ± 0.001

Table 1: Mean max cosine similarity (MMCS) and mean L2 ratio (ML2R) with their standard deviations (to 3 decimal places) between learned parameter components and target model weights for TMS. The MMCS is very close to 1.0, indicating that every column in the target model has a corresponding column in one of the components that points in almost the same direction. The ML2R is below 1.0, indicating some amount of shrinkage in the components compared to the original model.

et al., 2021, Voss et al., 2021]. Those methods can find at most $\text{rank}(W) = \min(m_1, m_2)$ components, and therefore could not decompose W into its ground truth mechanisms even in principle.

The toy model studied in this section was initially developed in order to demonstrate that neural networks can represent variables 'in superposition' using an overcomplete basis of the bottleneck activation space. However, our work decomposes the model, not activation space. Nevertheless, the mechanisms identified by our method *imply* an overcomplete basis in the activation space: The rank 1 mechanisms $Z^{(c)}$ can be expressed as an outer product of their (un-normed) left and right singular vectors $W_{:,c}e_c^\top$. The left singular vectors (corresponding to the columns of W) are an overcomplete basis of the m_1-dimensional hidden activation space[5]. Parameter vectors can thus imply overcomplete bases for the activation spaces that they interact with, even though they do not form an overcomplete basis for parameter space.

The structure of this matrix decomposition is also revealing: We can think of P_c as 'reading' from the $e_c^\top$ direction in the input space and projecting to the $W_{:,c}$ direction in the bottleneck activation space (Figure 4). Since we use W and $W^\top$ in this model, in the next layer we can also think of this parameter component 'reading' from the $W_{:,c}^\top$ in the bottleneck activation space and projecting to the e_c direction in the pre-ReLU activation space. In the backward pass, the roles are reversed: Directions that were 'reading' directions for activations become 'projecting' directions for gradients, and vice versa. In general, networks will learn parameter components consisting of matrices whose right singular vectors align with the hidden activations on the forward pass and whose left singular vectors align with gradients of the output with respect to the preactivations on the backward pass. Thus, they will ignore directions along which there are no activations, as well as directions that have no downstream causal effects.

3.2 Toy Model of Compressed Computation

While the previous example (TMS) analyzed APD on a model that stored more features than dimensions, here we examine APD on a model performing more computations than it has neurons — a phenomenon that we term *compressed computation*. We chose this model because neural networks trained on realistic tasks may often perform more computations than they have neurons. Compressed computation is very similar to the "Computation in Superposition" toy model introduced by Elhage et al. [2022], but our architecture and task differ. A key characteristic of representation in superposition [Elhage et al., 2022] and computation in superposition [Bushnaq and Mendel, 2024] is a dependence on input sparsity. We suspect our model's solutions to this task might not depend on the sparsity of inputs as much as would be expected, potentially making 'compressed computation' and 'computation in superposition' subtly distinct phenomena. But we could not conclusively establish that distinction, since experiments investigating it transpired to be more complicated than they initially appeared. We leave a more detailed study of this distinction for future work. To avoid potential confusion, we opted for a distinct term.

We train a target network to approximate a function of sparsely activating inputs $x_i \in [-1, 1]$, using a Mean Squared Error (MSE) loss between the model output and the labels. The labels we train the model to predict are produced by the function $y_i = x_i + \text{ReLU}(x_i)$. Crucially, the task involves learning to compute more ReLU functions than the network has neurons.

[5]Equivalently, since we use W and $W^\top$ for the matrices in this model, the right hand components of $e_c W_{:,c}^\top$ also imply an overcomplete basis of the m_1-dimensional hidden activation space.

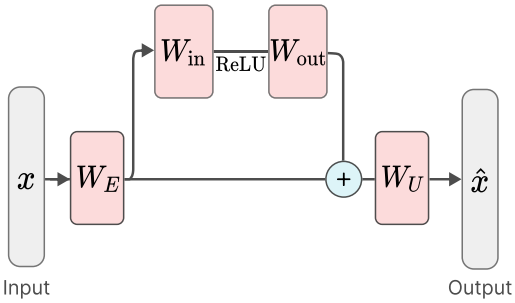

Figure 5: The architecture of our Toy Model of Compressed Computation using a 1-layer residual MLP. We fix W_E to be a randomly generated matrix with unit norm rows, and $W_U = {W_E}^\top$.

The target network is a residual MLP, consisting of a residual stream width of $d_{\text{resid}} = 1000$, a single MLP layer of width $d_{\text{mlp}} = 50$, a fixed, random embedding matrix with unit norm rows W_E, an unembedding matrix $W_U = W_E^\top$, and 100 input features. See Figure 5 for an illustration of the network architecture.

A naive solution to this task is to dedicate one neuron each to the computation of the first d_{mlp} functions, and to ignore the rest. This monosemantic baseline solution would perform perfectly for inputs that contained active features in only the first d_{mlp} input feature indices but poorly for all other inputs.

The large residual stream $d_{\text{resid}} = 1000$ was chosen as the trained target network performed better than the naive monosemantic baseline in this setting (small values of d_{resid} lead to higher interference and thus worse model performance). We chose fixed, instead of trained, embedding matrices to make it simpler to calculate the optimal monosemantic baseline and to simplify training.

To understand how each neuron participates in computing the output for a given input feature, we measure what we call the neuron's *contribution* to each input feature computation. For each neuron, this contribution is calculated by multiplying two terms:

1. How strongly the neuron reads from input feature i (given by $W_{\text{IN}} W_{E\,[:,i]}$).
2. How strongly the neuron's output influences the model's output for index i (given by $W_{U\,[i,:]} W_{\text{OUT}}$).

Mathematically, we compute neuron contributions for each input feature computation $i \in [0, 99]$ by $(W_{U\,[i,:]} W_{\text{OUT}}) \odot (W_{\text{IN}} W_{E\,[:,i]})$, where $\odot$ denotes element-wise multiplication. A large positive contribution indicates that the neuron plays an important role in computing the output for input feature i. Figure 6 (top) shows the neurons involved in the computation of the first 10 input features of the target model and their corresponding contribution values. We analyze this target model in more detail in Appendix C.1.

The goal for APD in this setting is to learn parameter components that correspond to the computation of each input feature in the target model, despite these computations involving neurons that are used to compute multiple input features. For simplicity, we only decompose the MLP weights and do not decompose the target model's embedding matrix, unembedding matrix, or biases.

We found that parameter components often 'die' during training, such that no input from the training dataset can activate them. For this reason, we train with 130 parameter components. This gives APD a better chance of learning all 100 of the desired parameter components corresponding to unique input feature computations. More APD training details are given in Appendix D.

APD Results: Toy Model of Compressed Computation

Despite the target model computing more functions (100) than it has neurons (50), we find that APD can indeed learn parameter components that each implement $y_i = x_i + \text{ReLU}(x_i)$ for unique input dimensions $i \in \{0, \cdots, 99\}$. Figure 6 provides a visual representation of a set of learned parameter components. It shows how the computation that occurs for each input feature in the target network (top) is well replicated by individual parameter components in the APD model (bottom). We see that,

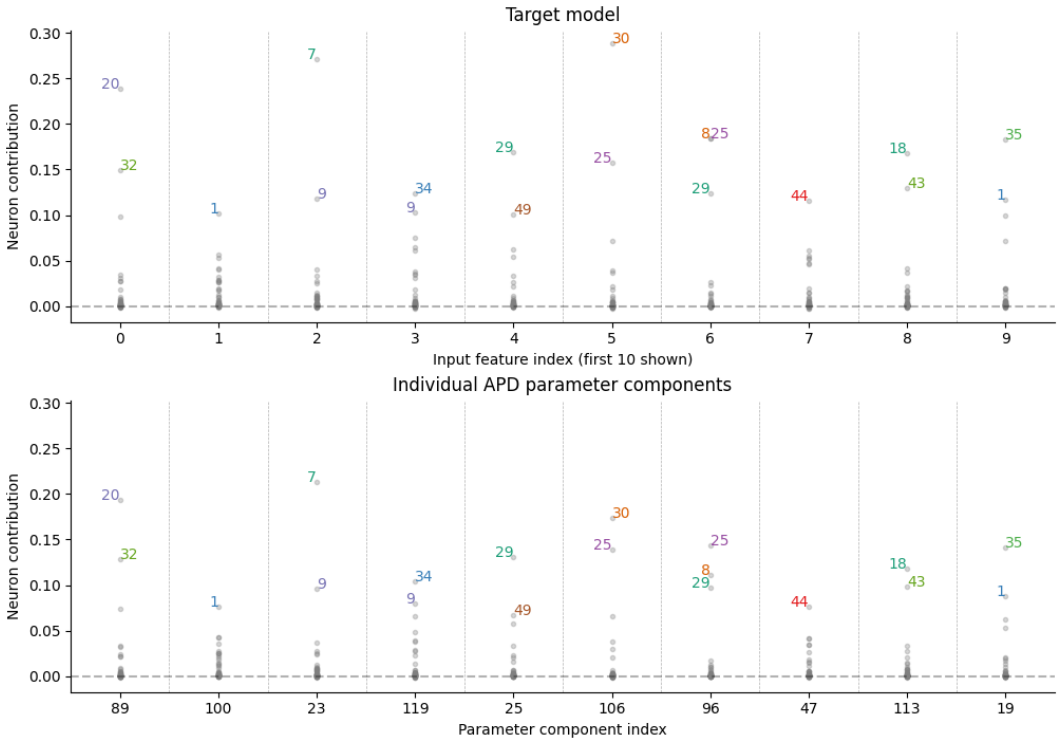

Figure 6: Similarity between target model weights and APD model components for the first 10 (out of 100) input feature dimensions. **Top**: Neuron contributions measured by $(W_{U[i,:]}W_{\text{OUT}}) \odot (W_{\text{IN}}W_{E[:,i]})$ for each input feature index $i \in [0, 9]$, where $\odot$ is an element-wise product. **Bottom**: Neuron contributions for the predominant parameter components, measured by $\max_k[(W_{U[i,:]}W_{\text{OUT}k}) \odot (W_{\text{IN}k}W_{E[:,i]})]$ for each feature index $i \in [0, 9]$. The neurons are numbered from 0 to 49 based on their raw position in the MLP layer. An extended version of this figure showing all input features and parameter components can be found here.

for each input feature, there is a corresponding parameter component that uses the same neurons to compute the function as the target model does. Note that while we do not see a perfect match between the target model and the APD model, a perfect match would not actually be expected nor desirable: the neuron contribution scores of the target model can contain interference terms from the overlapping mechanisms of other features, which a single APD parameter component is likely to filter out. However, there is some 'shrinkage', similar to what we observe in the results on the TMS model (see Section 3.1). Here, much of the shrinkage is due to batch top-k forcing APD on some batches to activate more components than there are features in the input, thereby spreading out input feature computations across multiple components. We discuss some of the trade-offs when setting batch top-k in Appendix C.2.

Next, we investigate whether individual APD components have minimal influence on forward passes where their corresponding input feature is not active using a Causal Scrubbing-inspired experiment [Chan et al., 2022]: When performing a forward pass we ablate half of the APD model's parameter components, excluding the ones that correspond to the currently active inputs (the 'scrubbed' run). We compare this to ablating half of the parameter components *including* those that correspond to currently active inputs ('anti-scrubbed'). Figure 7 gives a visual illustration of the output of multiple 'scrubbed' and 'anti-scrubbed' runs for a one-hot input $x_{42} = 1$. We see that ablating unrelated components perturbs the output only slightly, barely affecting the overall shape.

To investigate whether this holds true for all components and on the training data distribution, we collect MSE losses into a histogram (Figure 8). We find that the 'scrubbed' runs (i.e. ablating unrelated parameter components – pink histogram) does not cause a large increase in the MSE loss with respect to target network outputs. On the other hand, the anti-scrubbed runs (i.e. ablating parameter components that are deemed to be responsible for the computation – green histogram) does cause a large increase in MSE. This suggests that parameter components have mostly specialized to implement the computations for particular input features.

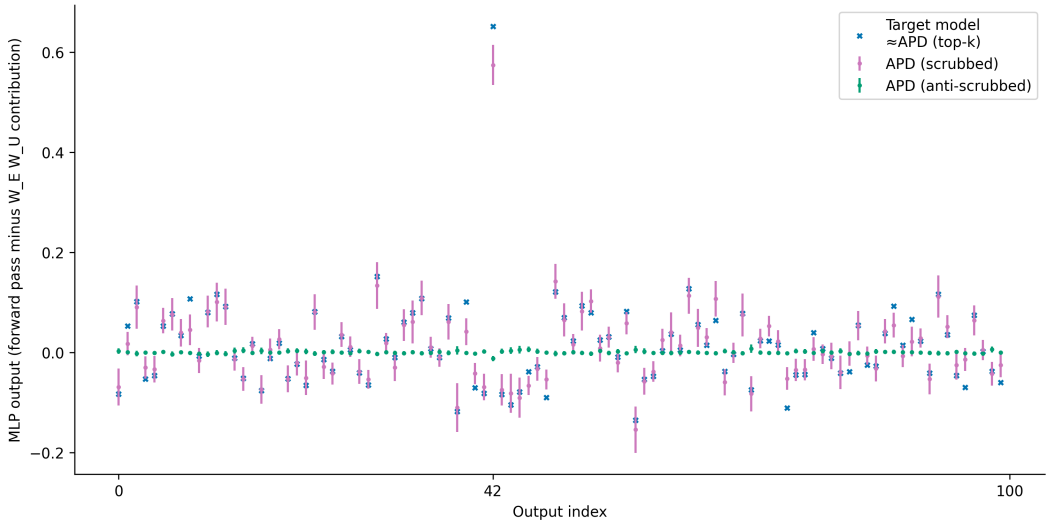

Figure 7: Output of multiple APD forward passes with one-hot input $x_{42} = 1$ over 10k samples, where half of the parameter components are ablated in each run. Purple lines show 'scrubbed' runs (parameter component corresponding to input index 42 is preserved), while green lines show 'anti-scrubbed' runs (component 42 is among those ablated). The target model output is shown in blue, which is almost identical to the output on the APD sparse forward pass (i.e. APD (top-k)). In this plot we only show the MLP output for clearer visualization. The embedding matrices are not decomposed and thus the residual stream contribution does not depend on APD components.

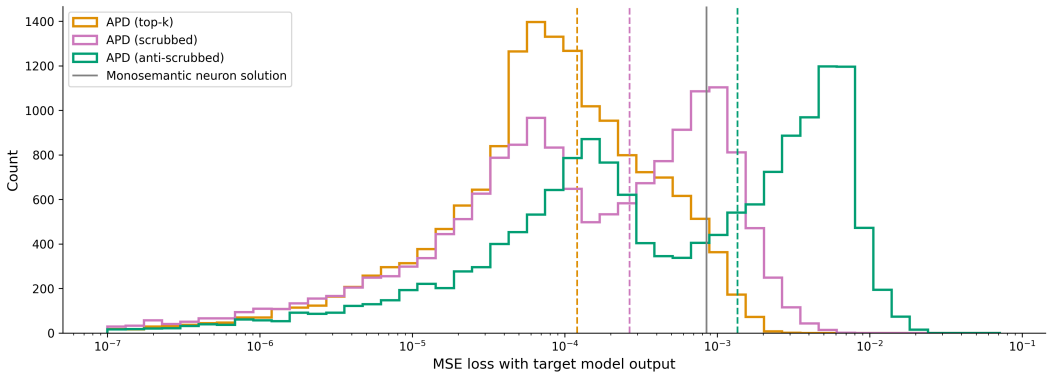

Figure 8: MSE losses of the APD model on the sparse forward pass ("top-k") and the APD model when ablating half (50) of its parameter components ("scrubbed" when none of the components responsible for the active inputs are ablated and "anti-scrubbed" when they are ablated). The gray line indicates the loss for a model that uses one monosemantic neuron per input feature. The dashed colored lines are the mean MSE losses for each type of run.

However, some parameter components appear to partially represent secondary input feature computations. This causes the visibly bimodal distributions of the scrubbed runs that can be seen in the figure: When these components are ablated, the loss of the model may be high when the secondary input feature is active. These components have the opposite effect on the loss when they are not ablated in the anti-scrubbed runs, making both scrubbed and anti-scrubbed losses bimodal. Preliminary work suggests that this can be improved with better hyperparameter settings or with adjustments to the training process, such as using alternative loss functions (Appendix A.2.3) or enforcing the APD components to be rank-1. We further quantify the extent to which some components partially represent secondary input feature computations in Appendix C.2.

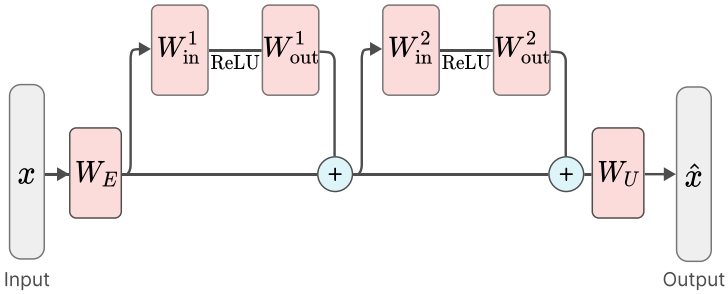

Figure 9: The architecture of our Toy model of Cross-Layer Distributed representations using a 2-layer residual MLP. We fix W_E to be a randomly generated matrix with unit norm rows, and $W_U = {W_E}^T$.

3.3 Toy Model of Cross-Layer Distributed Representations

We have seen how APD can learn parameter components that represent computations on individual input features, even when those computations involve neurons that contribute to the computations of multiple input features (Section 3.2). However, those computations take place in a single MLP layer. But realistic neural networks seem to exhibit cross-layer distributed representations [Yun et al., 2021, Lindsay et al., 2024]. In this section, we show how APD naturally generalizes to learn parameter components that represent computations that are distributed across multiple MLP layers[6].

We extend the model and task used in the previous section by adding an additional residual MLP layer (Figure 9). This model still performs compressed computation, but now with representations that are distributed across multiple layers. In this model, W_E is again a fixed, randomly generated embedding matrix with unit norm rows and $W_U = W_E^T$. We keep the residual stream width of $d_{\text{resid}} = 1000$ and 100 input features, but our 50 MLP neurons are now split across layers, with 25 in each of the two MLPs. We train APD on this model with 200 parameter components (allowing for 100 to die during training).

APD Results: Toy Model of Cross-Layer Distributed Representations

APD finds qualitatively similar results to the 1-layer toy model of compressed computation presented in Section 3.2. We see that the APD model learns parameter components that use neurons with large contribution values in both MLP layers (Figure 10, bottom). Again, we find that the computations occurring in each parameter component closely correspond to individual input feature computations in the target model (Figure 10, top versus bottom). For confirmation that the target model and APD model in the 2-layer distributed computation setting yield results that closely matching those observed in the 1-layer scenario, see Appendix C.3 and Appendix C.4, respectively.

However, the results exhibit a larger number of imperfections compared to the 1-layer case. In particular, more components represent two input feature computations rather than one. As in the 1-layer case, we again notice that batch top-k can cause some parameter components to not fully represent the computation of an input feature, and instead rely on activating multiple components for some input features (see Appendix C.4)[7].

4 Discussion

We propose APD, a method for directly decomposing neural network parameters into mechanistic components that are faithful, minimal, and simple. This takes a '*parameters-first*' approach to mechanistic interpretability. This contrasts with previous work that typically takes an '*activations-first*' approach, which decomposes networks into directions in activation space and then attempts to construct circuits (or 'mechanisms') using those directions as building blocks [Olah et al., 2020b, Cammarata et al., 2020, Cunningham et al., 2023, Bricken et al., 2023, Marks et al., 2024].

[6]For further validation that APD can identify cross-layer distributed representations, we apply it to a hand-coded network that implements a gated trigonometric function (Appendix B.1).

[7]Further analysis can be found in the "Toy Model of Cross-layer Distributed Representations (2 layers)" section of this WandB report.

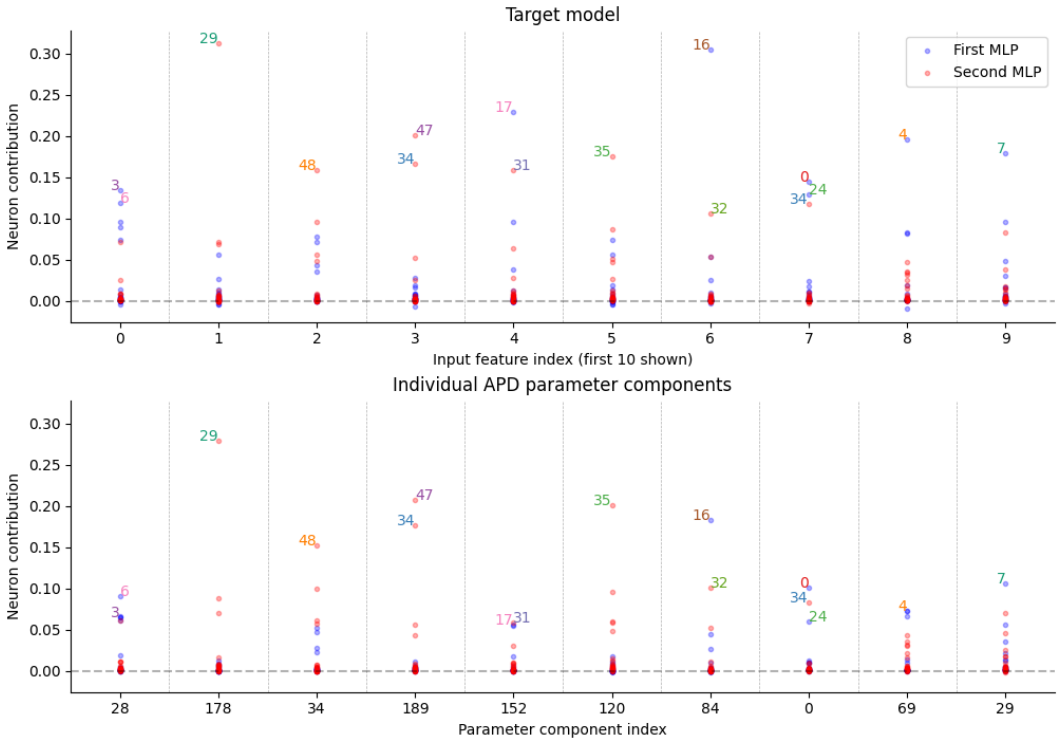

Figure 10: Similarity between target model weights and APD model components for the first 10 input feature dimensions in a 2-layer residual MLP. **Top**: Neuron contributions measured by $(W_E W_{\text{IN}}) \odot (W_{\text{OUT}} W_U)$ where $\odot$ is an element-wise product and W_{IN} and W_{OUT} are the MLP input and output matrices in each layer concatenated together. **Bottom**: Neuron contributions for the learned parameter components, measured by $\max_k[(W_{U\,[i,:]} W_{\text{OUT}\,k}) \odot (W_{\text{IN}\,k} W_{E\,[:,i]})]$ for each feature index $i \in [0, 9]$. The neurons are numbered based on their raw position in the network, with neurons 0 to 24 in the first layer and neurons 25 to 49 in the second layer. An extended version of this figure showing all input features and parameter components can be found here.

A parameters-first approach has several benefits. The new lens it provides suggests straightforward solutions to many of the of the challenges presented by the activations-first approach to mechanistic interpretability. Nevertheless, it also brings novel challenges that will need to be overcome. In this section, we discuss both the potential solutions and challenges suggested by this new approach and suggest potential directions for future research.

4.1 Addressing issues that seem challenging from an activations-first perspective

The activations-first paradigm struggles to identify minimal circuits in superposition, while APD achieves this directly. APD optimizes a set of parameter components that are maximally simple while requiring as few as possible to explain the output activations of any given datapoint. It is possible to think about these parameter components as circuits, since they describe transformations between activation spaces that perform specific functional roles.

Identifying a method to obtain minimal circuits by building on sparse dictionary learning (SDL) – which is an activations-first approach – has proven difficult for several reasons. One reason is that even though SDL might identify sparsely activating latent directions, there is no reason to expect the connections between them to be sparse. This might result in dense interactions between latents in consecutive layers, which may be difficult to understand compared with latent directions that were optimized to interact sparsely. Another reason that SDL has struggled to identify concise descriptions of neural network parameters is the phenomenon of feature splitting [Bricken et al., 2023], where it is possible to identify an ever larger number of latents using ever larger sparse dictionaries. However, more latents means more connections between them, even if the transformation implemented by this layer is very simple! Descriptions of the connections may include an ever growing amount of

redundant information. As a result, even very simple layers that transform latents in superposition may require very long description lengths.

A parameters-first approach suggests a conceptual foundation for 'features'. A central object in the activations-first paradigm of mechanistic interpretability is a 'feature'. Despite being a central object, a precise definition remains elusive. Definitions that have been considered include [Elhage et al., 2022]:

1. *'Features as arbitrary functions'*, but this fails to distinguish between features that appear to be fundamental abstractions (e.g. a 'cat feature') and those that don't (e.g. a 'cat+car' feature).
2. *'Features as interpretable properties'*, but this precludes features for concepts that humans don't yet understand.
3. *'Features as properties of the input which a sufficiently large network will reliably dedicate a neuron to representing'*. This definition is somewhat circular, since it defines object in neural networks using objects in other neural networks, and may not account for multidimensional features [Engels et al., 2024a, Olah, 2024b] .

In our work, we decompose neural networks into parameter components that minimize mechanistic description length, which we define as the network's 'mechanisms'. A network's mechanisms are not equivalent to its 'features'. However, defining a network's features as '*properties of the input that activate particular mechanisms*' seems to overcome the definitional issues above. In particular, it overcomes the issues in Definition 1 because we expect mechanisms, which are simple, minimally active components, to prefer to learn 'cat components' rather than 'cat+car components', since the latter is neither simpler (It requires machinery that can detect both cats and cars) nor minimal (A cat+car detector will activate more often than either a cat or car detector). Additionally, it does not rely on a notion of human interpretability, thus overcoming the issue with Definition 2. It also seems to overcome the issues of Definition 3, since the definition is not circular and should also be able to identify multidimensional mechanisms (and hence multidimensional features that activate them), although we leave this for future work.

The definition also overcomes issues caused by 'feature splitting', a phenomenon observed in SDL where larger dictionaries identify sets of different features depending on dictionary size, with larger dictionaries finding more sparsely activating, finer-grained features than smaller dictionaries. This happens because SDL methods can freely add more features to the dictionary to increase sparsity even if those features were not fundamental building blocks of computation used by the original network. APD components also need to be faithful to the target network's parameters when they are added together, meaning it cannot simply add more components in order to increase component activation sparsity or simplicity. To see this, consider a neural network that has a hidden layer that implements a d-dimensional linear map on language model data. A transcoder could learn ever more sparsely activating, ever more fine-grained latents to minimize its reconstruction and sparsity losses and represent this transformation. By contrast, the APD losses would be minimized by learning a single d-dimensional component that performs the linear map. The losses cannot be further reduced by adding more components, because that would prevent the components from summing up to the original network weights.

Incidentally, this thought experiment not only sheds light on feature splitting, but also sheds light on the difference between parameter components and 'features' as they are usually conceived. Parameter components are better thought of as "steps in the neural network's algorithm", rather than "representations of properties of the input". The network may nevertheless have learned mechanisms that specifically activate for particular properties of the input, which may be called 'features'.

A parameters-first approach suggests an approach to better understanding 'feature geometry'. Bussman et al. [2024] showed that the Einstein SAE latent has a similar direction to other SAE latents that were German-related, physics-related, and famous people-related. This suggests that the latents that SDL identify lie on an underlying semantic manifold. Understanding what gives this manifold its structure should suggest more concise descriptions of neural networks. But SDL treats SDL latents as fundamental computational units that can be studied in isolation, thus ignoring this underlying semantic manifold [Mendel, 2024]. We contend that the reason the Einstein latent points in the 'physics direction' (along with other physics-related latents) is because the network

applies 'physics-related mechanisms' to activations along that direction. Therefore, by decomposing parameter space directly, we expect interpretability in parameter space to shed light on computational structure that gives rise to a network's SAE 'feature geometry'.

Interpretability in parameter space suggests an architecture-agnostic method to resolving superposition. Neural network representations appear not to neatly map to individual architectural components such as individual neurons, attention heads, or layers. Representations often appear to be spread across various architectural components, as in attention head superposition [Jermyn et al., 2023] or cross layer distributed representations [Yun et al., 2021, Lindsay et al., 2024]. It is unclear how best to adapt SDL to each of these settings in order to tell concise mechanistic stories [Mathwin et al., 2024, Wynroe and Sharkey, 2024, Lindsay et al., 2024]. A more general approach that requires no adaptation would be preferred. Interpretability in parameter suggests a way to overcome this problem in general, since any architecture can in theory be decomposed into directions in parameter space without the method requiring adaptation.

4.2 Next steps: Where our work fits in an overall interpretability research and safety agenda

We had two main goals for this work. Our first goal was to resolve conceptual confusions arising in the activations-first, sparse dictionary learning-based paradigm of mechanistic interpretability. Our other main goal was to develop a method that builds on these conceptual foundations that can be applied to real-world models. However, APD is currently only appropriate for studying toy models because of its computational cost and hyperparameter sensitivity. We see two main paths forward:

1. **Path 1:** Develop APD-like methods that are more robust and scalable.
2. **Path 2:** Use the principles behind our approach to design more intrinsically decomposable architectures[8].

We are excited about pursuing both of these paths. In the rest of this section, we focus on Path 1, leaving Path 2 to future work.

We will outline what we see as the main challenges and exciting future research directions for building improved methods for identifying minimal mechanistic descriptions of neural networks in parameter space. To become more practical, APD must be improved in several ways. In particular, it should have a lower computational cost; be less sensitive to hyperparameters; and more accurate attributions. We may also need to fix outlying conceptual issues, such as the extent to which APD privileges layers. We also discuss several safety-oriented and scientific research directions that we think may become easier when taking a parameter-first approach to interpretability.

4.2.1 Improving computational cost

While developing this initial version of APD, we focused on conceptual progress over computational efficiency. At a glance, our method involves decomposing neural networks into parameter components that each have a memory cost similar to the target network. That would make it very expensive, scaling in the very worst case as something like $\mathcal{O}(N^2)$ where N is the parameter count of the original model. However, there are several reasons to think this might not be as large an issue as it initially appears:

- **More efficient versions of APD are likely possible.** We think there exist paths toward versions of APD that are more computationally efficient. For the experiments in this paper, we often permitted the parameter components to be full rank. But theories of computation in superposition suggest that for a network to have many non-interfering components, they need to be low rank or localized to a small number of layers [Bushnaq and Mendel, 2024]. A version of APD that constrains components to be lower rank and located in fewer layers would reduce their memory cost. Even if there are many high rank mechanisms, we think it may be possible to identify principles that let us stitch together many low rank components if the ground truth mechanisms are high rank, or let us use hierarchical representations of

[8] In particular, we are excited about research that explores how to pre-decompose models using mixtures-of-experts with many experts, where the experts may span multiple layers, like the parameter components in our work.

components. It may be possible to apply our method to models one layer at a time, like transcoders, which may save on memory costs of having to decompose every parameter at the same time (discussed further in Section 6.2).

- **Alternative approaches, such as SDL, may be even more expensive.** To use sparse dictionary learning to decompose a single layer's activation space may be relatively cheap compared with training an entire network. But even if it were possible to reverse engineer neural networks using sparse dictionaries (which is unclear), we would need to train sparse dictionaries on every layer in a network in order to reverse engineer it, which may be very expensive. It becomes even more expensive considering the need to identify or learn the connections between sparse dictionary latents in subsequent layers. At present, there is no reason to expect that it costs less to train sparse dictionaries on every layer than to perform APD. It may indeed cost much more to use SDL, since we do not know in advance what size of dictionary we need to use and how much feature splitting to permit. We suspect that a reasonably efficient version of APD, which aims to identify minimal mechanistic descriptions, to reverse engineer networks will fare favorably compared to using SDL to achieve similar feats, if SDL can be used for that purpose at all.
- **Our method suggests clear paths to achieving interpretability goals that other approaches have struggled to achieve.** Even if our method were more expensive than SDL-based approaches, our approach confers significant advantages (discussed in Section 4.1) that might make the computational cost worth it.

4.2.2 Improving robustness to hyperparameters

A practical issue at present is that the method is sensitive to hyperparameters. Extensive hyperparameter tuning was often required for APD to find the correct solution. Making the method more robust to hyperparameters is a high priority for future work. It is worth noting that we encountered fewer hyperparameter sensitivity issues when scaling up the method to larger toy models. It is likely that hyperparameter sensitivity was exacerbated due to the amount of interference noise between input feature computations in our experiments in small dimensions, and this may resolve itself when scaling up.

4.2.3 Improving attributions

One of the reasons that the method might not be robust is that the method currently uses gradient attributions, which are only a first order approximation of causal ablations [Mozer and Smolensky, 1988, Molchanov et al., 2017]. Previous work, such as AtP [Nanda, 2022a] and AtP* [Kramár et al., 2024] indicates that using gradients as first-order approximations to causal ablations work reasonably well, but become unreliable when gradients with respect to parameters become small due to e.g. a saturated softmax [Kramár et al., 2024]. This problem could potentially be alleviated by using integrated gradients [Sundararajan et al., 2017], learning masks for each parameter component on each datapoint during training [Caples et al., 2025], or other attribution methods instead.

4.2.4 Improving layer non-privileging

We want to find components in the structure of the learned network algorithm, rather than the network architecture. Thus, we would prefer our formalism to be entirely indifferent to changes of network parameterization that do not affect the underlying algorithm. However, APD is currently not indifferent to changes of network parameterization that mix network layers. Layers are therefore still slightly privileged by APD. This is because we optimize parameter components to be simple by penalizing them for being high rank, and the rank of weight matrices cannot be defined without reference to the network layers. Thus, if two components in different neural networks implement essentially the same computation, one in a single layer, the other in cross-layer superposition, the latter component may be assigned a higher rank. Therefore, while we think that APD can still find components that stretch over many layers, it may struggle to do so more than for components that stretch over fewer layers. We would need to find layer-invariant quantities that more accurately track the simplicity of components independent of their parametrization than effective rank. Speculatively, some variation of the weight-refined local learning coefficient [Wang et al., 2024] might fulfill this requirement.

4.2.5 Promising applications of interpretability in parameter space

If we can overcome these practical hurdles, we think that interpretability in parameter space may make achieving some of the safety goals of interpretability easier than with activations-first methods. For instance, if we have indeed identified a way to decompose neural networks into their underlying mechanisms, it will be readily possible to investigate mechanistic anomaly detection for monitoring purposes [Christiano, 2022]. Interpretability in parameter space may also be easier to perform precise model editing or unlearning of particular mechanisms [e.g. Meng et al. [2023a,b]], since model descriptions are given in terms of parameter vectors, which are the objects that we would directly modify.

We are also excited about applications of APD that might help answer important scientific questions. For instance, we suspect that APD can shed light on the mechanisms of memorizing vs. generalizing models [Henigan et al., 2023, Zhang et al., 2017, Arpit et al., 2017]; the mechanisms of noise robustness [Morcos et al., 2018]; or leveraging the fact that APD is architecture agnostic in order to explore potentially universal of mechanistic structures that are learned independent of architecture [Li et al., 2015, Olah et al., 2020b], such as convolutional networks, transformers, state space models, and more.

5 Conclusion

This work introduces Attribution-based Parameter Decomposition (APD) as a fundamental shift in mechanistic interpretability: instead of analysing neural networks through their activations, we demonstrate that directly decomposing parameter space can reveal interpretable mechanisms that are faithful, minimal, and simple. Our approach suggests solutions to long-standing problems in mechanistic interpretability, including identifying minimal circuits in superposition, providing a conceptual foundation for 'features', enabling better understanding of 'feature geometry', and serving as an architecture-agnostic approach to neural network decomposition.

Our experiments demonstrate that APD can successfully identify ground truth mechanisms in multiple toy models: recovering features from superposition, separating compressed computations, and discovering cross-layer distributed representations.

Although our results are encouraging, several challenges remain before APD can be applied to real-world models. These include improving computational efficiency, increasing robustness to hyperparameters, and incorporating more robust attribution methods.

By decomposing neural networks into their constituent mechanisms, this work brings us closer to reverse engineering increasingly capable AI systems, helping to open the door toward a variety of exciting scientific and safety-oriented applications.

6 Related Work

Our approach draws on several ideas from prior work in mechanistic interpretability and other fields.

6.1 Sparse Autoencoders

Sparse Autoencoders (SAEs) are a sparse dictionary learning (SDL) method that is commonly used in mechanistic interpretability. Although APD is not a SDL method, it has many connections.

SAEs can be used to identify an overcomplete basis for activation space consisting of sparsely activating directions [Lee et al., 2007, Yun et al., 2021, Sharkey et al., 2022, Cunningham et al., 2023, Bricken et al., 2023]. Similarly, APD finds a set of sparsely used vectors (parameter components) in parameter space. However, we do not expect this set of vectors to form an overcomplete basis for parameter space, since the number of mechanisms a network can implement is upper-bounded by its parameter count [Bushnaq and Mendel, 2024].

Networks that have sparsely activating latent directions might sometimes dedicate a mechanism to operate on one of these directions. See Section 3.1 for an example. However, parameter components are more general than directions in activation space. They may also operate on multidimensional activation subspaces, stretch over multiple layers, and operate on input latents that are not sparse.

The training process used in our work, notably the second forward pass that uses only the (batch) top-k most attributed parameter components, resembles the training process of top-k and batch top-k sparse autoencoders [Makhzani and Frey, 2013, Gao et al., 2024, Bussmann et al., 2024]. An alternative APD procedure that uses an L_1 or L_p penalty on attributions is also possible (Appendix A.2.3), which would be analogous to classic SAE training methods with an L_1 or L_p penalty on latent activations.

Crosscoders are a variant of SAEs that take as input and reconstruct activations at multiple layers simultaneously. They can identify representations that span multiple layers [Lindsay et al., 2024, Yun et al., 2021]. Similarly, our work identifies mechanisms that span multiple layers. However, like other SDL approaches, crosscoders decompose the activations, which are the *result* of a network's mechanisms, and do not immediately suggest a way to decompose the network's mechanisms themselves.

Braun et al. [2024] train end-to-end SAEs to identify latent directions for which as few as possible are necessary to reconstruct the output activations and hidden activations. APD similarly identifies parameter components for which as few as possible are necessary to reconstruct the output activations (and sometimes the hidden activations).

SAEs can be considered as lossy compression algorithms that optimize for compressed descriptions of activation datasets [Ayonrinde et al., 2024]. Similarly, APD optimizes for compressed descriptions of the causal processes that a network applies to activations over the course of computing its output. A somewhat related perspective on neural network decomposition and interpretation is proposed in Gross et al. [2024], which argues that the amount of mechanistic understanding about a neural network can be meaningfully quantified by how much it compresses proofs about the network's behavior.

6.2 Transcoders

Transcoders are an SDL method that is similar to SAEs but is trained to reconstruct the output activations of a layer given its input activations [Dunefsky et al., 2024, Mathwin et al., 2024, Wynroe and Sharkey, 2024]. Although transcoders decompose activations, they can also be considered to decompose the transformation implemented by that layer. This makes them related to APD, though APD decomposes the transformation in parameter space.

However, the 'activation' of parameter components are disanalogous to the activation of transcoder latents. In a transcoder, the activation of a latent is determined by whether an activation on the forward pass is above a threshold. This nonlinear threshold filters out the 'interference terms' from non-orthogonal latents represented in superposition and prevents them from being forward-propagated. In APD, the 'activation' of a parameter component is determined by whether it had a causal influence downstream (i.e. whether it was attributed). In APD, attributions, rather than nonlinearities, filter the interference terms from non-orthogonal parameter components and prevent their effects from being forward-propagated.

Although our work decomposed all layers of our toy models at once, it is likely possible to apply APD to only one layer at a time. This approach may be useful for keeping computational costs low (see Section 4.2.1 for further discussion). Similarly, transcoders could in theory be trained on every layer at the same time, though this may be even more expensive than APD, especially since there is no upper limit on the number of latents that transcoders should have.

With any finite number of dictionary elements, transcoders will always involve an irreducible activation reconstruction error term, even in the limit of using infinite latents [Engels et al., 2024b]. This is because transcoders attempt to use linear combinations of activation directions to approximate the transformations implemented by a layer, such as an MLP layer [Dunefsky et al., 2024] or attention block [Mathwin et al., 2024, Wynroe and Sharkey, 2024] (among other reasons). But those may use very different activation functions, such as a softmax nonlinearity or GeLUs and may be difficult to approximate with any finite number of dictionary elements. APD therefore has the additional benefit of being equally applicable to arbitrary neural architectures, such as attention blocks, recurrent neural networks, convolutional networks, or state space models, without requiring specific adaptation to each architecture.

6.3 Weight masking and pruning

Some previous work identifies masks for neural network parameters in order to isolate particular functionality. This is similar to APD, since activating a parameter component is functionally equivalent to leaving that component unmasked while masking others. Our approach is therefore similar to learning many different parameter masks, one for each parameter component.

Mozer and Smolensky [1988] is an early work that identifies relevant units in a neural network and prunes irrelevant ones, thus implicitly masking the parameters that connect to those units. Similar to our work, it uses gradient-based attributions to identify relevant components. Similarly, LeCun et al. [1989] ablates individual parameters, but uses second-order approximations of causal perturbation in contrast to our first-order approximations. Later work identifies explicit or implicit binary masks (or masks whose elements take values in $[0, 1]$) over units or parameters [Csordás et al., 2021, Cao et al., 2021b, Zhang et al., 2021, Cao et al., 2021a, Patil et al., 2023, Lepori et al., 2023, Mondorf et al., 2024]. However, masks that are constrained to take values in $[0, 1]$ privilege components aligned with the neuron- or parameter-bases. But gradient descent does not necessarily privilege these bases. Our implicit masks may take values outside $[0, 1]$ so that they do not privilege components that align with the neuron- or parameter-bases[9].

Csordás et al. [2021] learned differentiable binary masks over weights to identify components responsible for specific functions. However, this required collecting a dataset to define a task distribution that defines which components would be used. This is unlike our work, which optimizes components for minimum description length on any given datapoint, which implicitly defines task distributions on which particular components are used in an unsupervised way.

6.4 Circuit discovery and causal mediation analysis

Circuit discovery methods are a broad class of approaches that typically use causal interventions [Chan et al., 2022, Wang et al., 2022, Conmy et al., 2024] (or an approximation of them [Nanda, 2022a, Syed et al., 2023, Kramár et al., 2024]) on activations to find simple circuits that are sufficient for computations required for performance on particular tasks if the remainder of the model is ablated.

These approaches and APD have many similarities, but also crucial differences. They both involve ablating components of models with the aim of finding core computational mechanisms. However, previous circuit dicsovery methods have typically assumed a particular decomposition of networks into neurons, MLPs, attention heads, while APD makes much fewer assumptions about components. Later work uses sparse dictionary learning to determine the components that are causally intervened upon [Cunningham et al., 2023, Bricken et al., 2023, Marks et al., 2024, Geiger et al., 2024]. However, these approaches assumed components in activation space that are localized in single layers. APD learns components in parameter space.

Circuit discovery methods operate on a computational graph, representing all components of a network and their interactions. But the way in which components interact is abstracted away; typically every connection is simply assigned a scalar interaction strength. APD on the other hand operates on the parameters, with each parameter component containing all the parameters needed to explain its function. Circuit discovery methods also measure complexity as the number of nodes or edges in the computational graph, while APD uses an approximation of the sum of the rank of the weight matrices in the parameter component.

6.5 Interpreting parameters

Although most prior work aims to understand activations primarily, there exists a small amount of work that tackles weights more directly.

Some early machine learning [Rumelhart et al., 1986, McClelland and Rumelhart, 1985, Srivastava et al., 2014] and mechanistic interpretability [Olah et al., 2020a] interprets the raw parameters, thus assuming a decomposition that aligns with the unit basis in parameter space.

[9] It may be somewhat counterintuitive to suppose that we can decompose network parameters into components that take positive or negative values when a given parameter is e.g. positive. But other parameter decomposition methods (such as singular value decompositions of matrices) also permit this while nevertheless being meaningful functional decompositions.

Other work decomposes the parameters of the network using matrix decomposition methods. Millidge and Black [2022], Meller and Berkouk [2023] and Gross et al. [2024] use SVD to decompose parameters. Although these works found interpretable structure, SVD is not capable of identifying mechanisms in superposition and is limited to single layers, while mechanisms may span multiple layers. Voss et al. [2021] and Petrov et al. [2021] both use NMF to decompose weights of image classifier models into factors and visualize the results. Voss et al. [2021] go further and study expanded weights, which are the result of multiplying weights of adjacent layers which, in practice, uses gradients. However, the factorization approaches used in this work do not decompose parameters into components that are minimal and simple, even if they may be faithful (see also Section 3.1 for further comparison between APD and NMF).

Static analysis of parameters studies the computational structure of a network without needing to run forward or backward passes. One example is Dar et al. [2023], who use the logit lens [nostalgebraist, 2020] to project model parameters into vocabulary space to help interpret them. Other work interprets decompositions of the weights of bilinear layers, since their simple mathematical form facilitates decomposition despite their nonlinearity [Pearce et al., 2024b,a].

6.6 Quanta identification

Michaud et al. [2024] identify 'quanta', where a quantum is defined as 'An indivisible computational module that, for example, retrieves a fact, implements an algorithm, or more generally corresponds to some basic skill possessed by a model'. This definition reflects aspects of what our work defines as a 'mechanism', though our definition is formal. To identify quanta in language models, they cluster the gradients of the loss with respect to the parameters. This shares the intuition with our work that different gradients reflect different active mechanisms. However, clustering gradients does not decompose them. By training parameter components to be sparsely attributed, our method implicitly trains parts of our parameter components to align sparsely with components of gradients (Section 3.1).

6.7 Mixture of experts

Mixture of expert (MoE) architectures leverage the notion that only a subset of network's mechanisms are useful on any given forward pass [Jacobs et al., 1991]. Each expert is used only on a subset of the training dataset, and tends to specialize to that subdistribution, similar to the mechanisms learned in our method [Fedus et al., 2022]. However, MoE architectures usually also have a gating function that determines which experts are used during the forward pass. Our approach, by contrast, can only identify which experts are active using attribution methods, which require an initial forward pass. Another difference is that our mechanisms may span multiple layers, whereas experts in MoE architectures typically exist in one layer only (though see Park et al. [2024]). The principles uncovered by our method may be useful to develop new mixture-of-experts approaches that are more easily decomposed and interpreted.

6.8 Loss landscape dimensionality and degeneracy

Our work is directly inspired by research on the intrinsic dimensionality of and degeneracy in the loss landscape. Both our work and Li et al. [2018] parametrize a network using linear combinations of basis vectors in parameter space. However, Li et al. [2018] used a fixed, random basis for a subspace of parameter space and trained only the coefficients. Our work, by contrast, trained the bases such that they are simple and uses as few basis elements as possible for any given datapoint.

Singular learning theory [Watanabe, 2009] quantifies degeneracy in network weights present over the entire data distribution using the learning coefficient. APD finds vectors in parameter space along which parameters can be ablated on some network inputs, so they are degenerate directions over a subset of the data. Likewise, Wang et al. [2024] defined the data-refined learning coefficient, which measures degeneracy in neural network weights over a subset of the distribution. Unlike Wang et al. [2024], APD finds subsets of the data over which some directions are degenerate in an unsupervised manner rather than starting with a subset of the distribution and quantifying its degeneracy.

Author contributions

Research iteration Our method underwent significant iteration throughout development, changing many times in response to experimental results. DB, SH, LB, JM, and LS were responsible for driving forward various steps in the iteration cycle that led to the current paper.

Conceptualisation The core ideas for APD were developed through close collaboration between LS and LB. LS originated the idea to decompose the network parameters into sparsely used directions in parameter space and made initial suggestions for the faithfulness, minimality, and simplicity losses. SH and JM red-teamed the idea that directions in parameter space correspond to individual computations in networks with polysemanticity or computation in superposition. LB, with input from JM, developed a better minimality loss based on attributions and developed the MDL framing and its formalism. LS developed the idea for the top-k version of the method (versus the original L_p penalty version). The idea for the Schatten-norm-based simplicity loss was developed independently by LB and LS, based on experiments from DB and SH. LB developed the idea for an efficient implementation of that loss. SH, JM, LS, and LB identified various problems with previous versions of the method during the research iteration cycle.

Target models and task designs LB and LS independently identified the Toy Model of Superposition as a good toy model for APD. LB, SH, and DB developed the Toy Model of Compressed Computation, with implementation and training by DB and SH. JM and LB developed the idea for the handcoded model in the appendix, and JM handcoded it.

Experiments and analysis DB ran most of the experiments throughout the various research iteration cycles. DB and SH ran most of the experiments for Toy Model of Compressed Computation and Toy Model of Cross Layer Distributed Representations. DB and SH analyzed most of the experiments, with significant input from LB and some input from LS. LB helped analyse experiments through the research iteration cycle. SH and JM ran most of the experiments for the hardcoded model throughout the iteration cycle; DB and SH ran the experiments on the version that is in the appendix.

Writing LS, LB, DB, and SH contributed to writing and editing the paper. LS wrote most of the Introduction, Discussion, Conclusion, and Related Work. LS and LB wrote the methods section. LB wrote most of the more detailed description of the APD method in the appendix. DB wrote most of the results section and appendix results, with significant contributions from SH and LS.

Figures and illustrations LS designed and made the illustrations of the method and target models' architectures. SH and DB designed and generated the results figures.

Acknowledgements

Our work benefited from valuable discussions with and feedback from many colleagues and collaborators. We owe particular gratitude to Bilal Chughtai, Nicholas Goldowsky-Dill, Kaarel Hänni, and James Fox. We also owe thanks for their helpful feedback to Adrià Garriga-Alonso, Garret Baker, Joshua Batson, Brianna Chrisman, Jason Gross, Gurkenglas, Leo Gao, Marius Hobbhahn, Linda Linsefors, Daniel Murfet, Neel Nanda, Michael Pearce, Dmitry Vaintrob, John Wentworth, Jeffrey Wu, and likely many others who we are regrettably forgetting.

References

Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. A closer look at memorization in deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 233–242. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/arpit17a.html.

Kola Ayonrinde, Michael T. Pearce, and Lee Sharkey. Interpretability as compression: Reconsidering sae explanations of neural activations with mdl-saes, 2024. URL https://arxiv.org/abs/2410.11179.

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. URL https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Lucius Bushnaq and Jake Mendel. Circuits in superposition: Compressing many small neural networks into one, Oct 2024. URL https://www.alignmentforum.org/posts/roE7SHjFWEoMcGZKd/circuits-in-superposition-compressing-many-small-neural.

Bart Bussman, Michael Pearce, Patrick Leask, Joseph Bloom, Lee Sharkey, and Neel Nanda. Showing sae latents are not atomic using meta-saes, Aug 2024. URL https://www.lesswrong.com/posts/TMAmHh4DdMr4nCSr5/showing-sae-latents-are-not-atomic-using-meta-saes.

Bart Bussmann, Patrick Leask, and Neel Nanda. Batchtopk sparse autoencoders, 2024. URL https://arxiv.org/abs/2412.06410.

Nick Cammarata, Gabriel Goh, Shan Carter, Ludwig Schubert, Michael Petrov, and Chris Olah. Curve detectors. *Distill*, 2020. doi: 10.23915/distill.00024.003. https://distill.pub/2020/circuits/curve-detectors.

Nicola De Cao, Leon Schmid, Dieuwke Hupkes, and Ivan Titov. Sparse interventions in language models with differentiable masking, 2021a. URL https://arxiv.org/abs/2112.06837.

Steven Cao, Victor Sanh, and Alexander M. Rush. Low-complexity probing via finding subnetworks, 2021b. URL https://arxiv.org/abs/2104.03514.

Deigo Caples, Jatin Nainani, Callum McDougall, and Rob Neuhaus. Scaling sparse feature circuit finding to gemma 9b, Jan 2025. URL https://www.lesswrong.com/posts/PkeB4TLxgaNnSmddg/scaling-sparse-feature-circuit-finding-to-gemma-9b.

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

David Chanin, James Wilken-Smith, Tomáš Dulka, Hardik Bhatnagar, and Joseph Bloom. A is for absorption: Studying feature splitting and absorption in sparse autoencoders, 2024. URL https://arxiv.org/abs/2409.14507.

Paul Christiano. Mechanistic anomaly detection and elk, 11 2022. URL https://www.lesswrong.com/posts/vwt3wKXWaCvqZyF74/mechanistic-anomaly-detection-and-elk.

Mark M. Churchland and Krishna V. Shenoy. Temporal complexity and heterogeneity of single-neuron activity in premotor and motor cortex. *Journal of Neurophysiology*, 97(6): 4235–4257, 6 2007. doi: 10.1152/jn.00095.2007. URL https://journals.physiology.org/doi/full/10.1152/jn.00095.2007. PubMed ID: 17376854.

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36, 2024.

Róbert Csordás, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Are neural nets modular? inspecting functional modularity through differentiable weight masks, 2021. URL https://arxiv.org/abs/2010.02066.

Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023. URL https://arxiv.org/abs/2309.08600.

Guy Dar, Mor Geva, Ankit Gupta, and Jonathan Berant. Analyzing transformers in embedding space, 2023. URL https://arxiv.org/abs/2209.02535.

Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. Attention is not all you need: Pure attention loses rank doubly exponentially with depth, 2023.

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

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition, 2022.

Joshua Engels, Isaac Liao, Eric J. Michaud, Wes Gurnee, and Max Tegmark. Not all language model features are linear, 2024a. URL https://arxiv.org/abs/2405.14860.

Joshua Engels, Logan Riggs, and Max Tegmark. Decomposing the dark matter of sparse autoencoders, 2024b. URL https://arxiv.org/abs/2410.14670.

William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning, 2022. URL https://arxiv.org/abs/2209.01667.

Matthew Finlayson, Aaron Mueller, Sebastian Gehrmann, Stuart Shieber, Tal Linzen, and Yonatan Belinkov. Causal analysis of syntactic agreement mechanisms in neural language models, 2021. URL https://arxiv.org/abs/2106.06087.

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.

Atticus Geiger, Zhengxuan Wu, Christopher Potts, Thomas Icard, and Noah D. Goodman. Finding alignments between interpretable causal variables and distributed neural representations, 2024. URL https://arxiv.org/abs/2303.02536.

Paris Giampouras, René Vidal, Athanasios Rontogiannis, and Benjamin Haeffele. A novel variational form of the schatten-p quasi-norm, 2020. URL https://arxiv.org/abs/2010.13927.

Jason Gross, Rajashree Agrawal, Thomas Kwa, Euan Ong, Chun Hei Yip, Alex Gibson, Soufiane Noubir, and Lawrence Chan. Compact proofs of model performance via mechanistic interpretability, 2024. URL https://arxiv.org/abs/2406.11779.

Tom Henigan, Shan Carter, Tristan Hume, Nelson Elhage, Robert Lasenby, Stanislav Fort, Nicholas Schiefer, and Christopher Olah. Superposition, memorization, and double descent, Jan 2023. URL https://transformer-circuits.pub/2023/toy-double-descent/index.html.

Geoffrey F. Hinton. Shape representation in parallel systems. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 1088–1096, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

Kaarel Hänni, Jake Mendel, Dmitry Vaintrob, and Lawrence Chan. Mathematical models of computation in superposition, 2024. URL https://arxiv.org/abs/2408.05451.

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991. doi: 10.1162/neco.1991.3.1.79.

Jett Janiak, Chris Mathwin, and Stefan Heimersheim. Polysemantic attention head in a 4-layer transformer, Nov 2023. URL https://www.lesswrong.com/posts/nuJFTS5iiJKT5G5yh/polysemantic-attention-head-in-a-4-layer-transformer.

Adam Jermyn, Chris Olah, and Tom Henigan. Attention head superposition, May 2023. URL https://transformer-circuits.pub/2023/may-update/index.html#attention-superposition.

Adam Jermyn, Adly Templeton, Joshua Batson, and Trenton Bricken. Tanh penalty in dictionary learning. https://transformer-circuits.pub/2024/feb-update/index.html#:~:text=handle%20dying%20neurons.-,Tanh%20Penalty%20in%20Dictionary%20Learning,-Adam%20Jermyn%2C%20Adly, 2024.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

János Kramár, Tom Lieberum, Rohin Shah, and Neel Nanda. Atp*: An efficient and scalable method for localizing llm behaviour to components, 2024. URL https://arxiv.org/abs/2403.00745.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

Honglak Lee, Chaitanya Ekanadham, and Andrew Ng. Sparse deep belief net model for visual area v2. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007. URL https://proceedings.neurips.cc/paper_files/paper/2007/file/4daa3db355ef2b0e64b472968cb70f0d-Paper.pdf.

Michael A. Lepori, Thomas Serre, and Ellie Pavlick. Break it down: Evidence for structural compositionality in neural networks, 2023. URL https://arxiv.org/abs/2301.10884.

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes, 2018.

Yixuan Li, Jason Yosinski, Jeff Clune, Hod Lipson, and John Hopcroft. Convergent learning: Do different neural networks learn the same representations?, 2015. URL https://arxiv.org/abs/1511.07543.

Jack Lindsay, Adly Templeton, Jonathan Marcus, Thomas Conerly, Joshua Batson, and Christopher Olah. Sparse crosscoders for cross-layer features and model diffing, October 2024. URL https://transformer-circuits.pub/2024/crosscoders/index.html.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Aleksandar Makelov, George Lange, and Neel Nanda. Towards principled evaluations of sparse autoencoders for interpretability and control. https://openreview.net/forum?id=MHIX9H8aYF, 2024.

Alireza Makhzani and Brendan Frey. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

Samuel Marks, Can Rager, Eric J Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. *arXiv preprint arXiv:2403.19647*, 2024.

Chris Mathwin, Dennis Akar, and Lee Sharkey. Gated attention blocks: Preliminary progress toward removing attention head superposition. https://www.lesswrong.com/posts/kzc3qNMsP2xJcxhGn/gated-attention-blocks-preliminary-progress-toward-removing-1, 2024.

James L. McClelland and David E. Rumelhart. Distributed memory and the representation of general and specific information. *Journal of experimental psychology. General*, 114 2:159–97, 1985. URL https://api.semanticscholar.org/CorpusID:7745106.

Dan Meller and Nicolas Berkouk. Singular value representation: A new graph perspective on neural networks. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent, editors, *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 3353–3369. PMLR, 25–27 Apr 2023. URL https://proceedings.mlr.press/v206/meller23a.html.

Jake Mendel. Sae feature geometry is outside the superposition hypothesis, Sep 2024. URL https://www.alignmentforum.org/posts/MFBTjb2qf3ziWmzz6/sae-feature-geometry-is-outside-the-superposition-hypothesis.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt, 2023a.

Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer, 2023b. URL https://arxiv.org/abs/2210.07229.

Eric J. Michaud, Ziming Liu, Uzay Girit, and Max Tegmark. The quantization model of neural scaling, 2024. URL https://arxiv.org/abs/2303.13506.

Beren Millidge and Sid Black. The singular value decompositions of transformer weight matrices are highly interpretable, Nov 2022. URL https://www.lesswrong.com/posts/mkbGjzxD8d8XqKHzA/the-singular-value-decompositions-of-transformer-weight.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference, 2017. URL https://arxiv.org/abs/1611.06440.

Philipp Mondorf, Sondre Wold, and Barbara Plank. Circuit compositions: Exploring modular structures in transformer-based language models, 2024. URL https://arxiv.org/abs/2410.01434.

Ari S. Morcos, David G. T. Barrett, Neil C. Rabinowitz, and Matthew Botvinick. On the importance of single directions for generalization, 2018. URL https://arxiv.org/abs/1803.06959.

Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988. URL https://proceedings.neurips.cc/paper_files/paper/1988/file/07e1cd7dca89a1678042477183b7ac3f-Paper.pdf.

Neel Nanda. Attribution patching: Activation patching at industrial scale. https://www.neelnanda.io/mechanistic-interpretability/attribution-patching, 2022a.

Neel Nanda. Attribution patching: Activation patching at industrial scale. *neelnanda.io*, 2022b. URL https://www.neelnanda.io/mechanistic-interpretability/attribution-patching.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks, 2016. URL https://arxiv.org/abs/1602.03616.

nostalgebraist. interpreting gpt: the logit lens, August 2020. URL https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

Chris Olah. Weight superposition, May 2023. URL https://transformer-circuits.pub/2023/may-update/index.html#weight-superposition.

Chris Olah. The next five hurdles, July 2024a. URL https://transformer-circuits.pub/2024/july-update/index.html#hurdles.

Chris Olah. What is a linear representation? what is a multidimensional feature?, July 2024b. URL https://transformer-circuits.pub/2024/july-update/index.html#linear-representations.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. An overview of early vision in inceptionv1. *Distill*, 2020a. doi: 10.23915/distill.00024.002. https://distill.pub/2020/circuits/early-vision.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 5(3):e00024–001, 2020b.

Jungwoo Park, Young Jin Ahn, Kee-Eung Kim, and Jaewoo Kang. Monet: Mixture of monosemantic experts for transformers, 2024. URL https://arxiv.org/abs/2412.04139.

Shreyas Malakarjun Patil, Loizos Michael, and Constantine Dovrolis. Neural sculpting: Uncovering hierarchically modular task structure in neural networks through pruning and network analysis, 2023. URL https://arxiv.org/abs/2305.18402.

Michael T. Pearce, Thomas Dooms, and Alice Rigg. Weight-based decomposition: A case for bilinear mlps, 2024a. URL https://arxiv.org/abs/2406.03947.

Michael T. Pearce, Thomas Dooms, Alice Rigg, Jose M. Oramas, and Lee Sharkey. Bilinear mlps enable weight-based mechanistic interpretability, 2024b. URL https://arxiv.org/abs/2410.08417.

Michael Petrov, Chelsea Voss, Ludwig Schubert, Nick Cammarata, Gabriel Goh, and Chris Olah. Weight banding. *Distill*, 2021. doi: 10.23915/distill.00024.009. https://distill.pub/2020/circuits/weight-banding.

Lawrence Phillips, Garrett Goh, and Nathan Hodas. Explanatory masks for neural network interpretability, 2019. URL https://arxiv.org/abs/1911.06876.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. URL https://api.semanticscholar.org/CorpusID:205001834.

Lee Sharkey. Sparsify: A mechanistic interpretability research agenda, April 2024. URL https://www.alignmentforum.org/posts/64MizJXzyvrYpeKqm/sparsify-a-mechanistic-interpretability-research-agenda.

Lee Sharkey, Dan Braun, and Beren Millidge. Taking features out of superposition with sparse autoencoders, Dec 2022. URL https://www.alignmentforum.org/posts/z6QQJbtpkEAX3Aojj/interim-research-report-taking-features-out-of-superposition.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks, 2017.

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

Demian Till. Do sparse autoencoders find "true features"?, February 2024. URL https://www.lesswrong.com/posts/QoR8noAB3Mp2KBA4B/do-sparse-autoencoders-find-true-features.

Dmitry Vaintrob, Jake Mendel, and Kaarel Hänni. Toward a mathematical framework for computation in superposition, Jan 2024. URL https://www.alignmentforum.org/posts/2roZtSr5TGmLjXMnT/toward-a-mathematical-framework-for-computation-in.

Andreas Veit, Michael Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks, 2016.

Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Simas Sakenis, Jason Huang, Yaron Singer, and Stuart Shieber. Causal mediation analysis for interpreting neural nlp: The case of gender bias, 2020.

Chelsea Voss, Nick Cammarata, Gabriel Goh, Michael Petrov, Ludwig Schubert, Ben Egan, Swee Kiat Lim, and Chris Olah. Visualizing weights. *Distill*, 2021. doi: 10.23915/distill.00024.007. https://distill.pub/2020/circuits/visualizing-weights.

George Wang, Jesse Hoogland, Stan van Wingerden, Zach Furman, and Daniel Murfet. Differentiation and specialization of attention heads via the refined local learning coefficient, 2024. URL https://arxiv.org/abs/2410.02984.

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. *arXiv preprint arXiv:2211.00593*, 2022.

Sumio Watanabe. *Algebraic geometry and statistical learning theory*, volume 25. Cambridge university press, 2009.

Benjamin Wright and Lee Sharkey. Addressing feature suppression in saes, Feb 2024. URL https://www.alignmentforum.org/posts/3JuSjTZyMzaSeTxKk/addressing-feature-suppression-in-saes.

Keith Wynroe and Lee Sharkey. Decomposing the qk circuit with bilinear sparse dictionary learning, July 2024. URL https://www.lesswrong.com/posts/2ep6FGjTQoGDRnhrq/decomposing-the-qk-circuit-with-bilinear-sparse-dictionary.

Zeyu Yun, Yubei Chen, Bruno A Olshausen, and Yann LeCun. Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors, 2021. URL https://arxiv.org/abs/2103.15949.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization, 2017. URL https://arxiv.org/abs/1611.03530.

Dinghuai Zhang, Kartik Ahuja, Yilun Xu, Yisen Wang, and Aaron Courville. Can subnetwork structure be the key to out-of-distribution generalization?, 2021. URL https://arxiv.org/abs/2106.02890.

Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. Moefication: Transformer feed-forward layers are mixtures of experts, 2022.

A More detailed description of the APD method

Suppose we have a trained neural network $f(x, \theta^*)$, mapping network inputs x to network outputs $y = f(x, \theta^*)$, with parameters $\theta^* \in \mathbb{R}^N$.

We want to decompose into the 'mechanisms' that the network uses to compute its behavior. The network's parameters implement these mechanisms. We would therefore like some way to decompose a network's parameters into its constituent mechanisms.

We first define a set of parameter components

$$P = \{P_1, \ldots, P_C\}, \quad P_c \in \mathbb{R}^N, \quad \forall c \in \text{range}(1, \ldots, C), \quad C \in \mathbb{N}. \tag{5}$$

We want to train these parameter components such that they correspond to a network's mechanisms. We think that it is reasonable to define a network's mechanisms as a set of components that minimizes the total description length of the network's behavior, per data point, over the training dataset. In particular, we want to identify components that are maximally faithful to the target network, maximally simple, and where as few as possible are used to replicate the network's behavior on any given datapoint.

In Section A.1, we will more carefully define what we mean by a parameter vector P_c being 'used' to replicate the network's behavior, and how to approximately measure this in practice using attribution techniques. In Section A.2, we will first discuss in what sense we want the average description length per datapoint to be minimised by deriving an idealized loss for APD. Then, we will use some approximations to find a proxy for this idealized loss that is more tractable to optimise in practice.

A.1 Component attributions

We want most parameter components $P_c \in \mathbb{R}^N$ to not be 'used' on any one network input x, in the sense that we can ablate all but a few of them without changing the outputs of the network.

In other words, if we initialize a new network with a parameter vector $\kappa(x)$ composed of only a few of the most 'used' parameter components, we should have:

$$f(x|\kappa(x)) \approx f(x|\theta^*) \tag{6}$$

where

$$\kappa(x) := \sum_{c=1}^{C} s_c(x) P_c, \quad s_c(x) \in \{0, 1\}, \quad \sum_{c=1}^{C} s_c(x) \ll C. \tag{7}$$

Since the 'inactive' components are supposed to play no meaningful role in the computation of the output, we should also be able to ablate them (including partial ablation - see the stricter definition of 'inactive' below), and still get the same result.

Directly checking that this condition is satisfied would be computationally intractable. Instead, we try to estimate whether the condition is approximately satisfied by calculating attributions of the output to each component. Currently, we do this using gradient attributions [Finlayson et al., 2021, Molchanov et al., 2017, Nanda, 2022b]. This estimates the effect of ablating P_c as:

$$A_c(x) := \sqrt{\frac{1}{d_L} \sum_{o=1}^{d_L} \left(\sum_{l,i,j} \frac{\partial f_o(x, \theta^*)}{\partial \theta_{l,i,j}} P_{c,l,i,j} \right)^2}. \tag{8}$$

We take the average square of this term over all output indices o, where the final output layer has width d_L.

Previous work, such as [Nanda, 2022a] and [Kramár et al., 2024], indicates that gradient-based first-order attribution methods can be somewhat accurate in many circumstances, but not always. For example, a saturated softmax in an attention head would render them inaccurate. Therefore, we expect that we might need to move to more sophisticated attribution methods in the future, such as integrated gradients [Sundararajan et al., 2017].

A stricter definition of 'inactive' parameter components In general, we should get the same network output for any parameter configuration everywhere along any monotonic 'ablation curve' $\gamma_c(x,t)$ where $t \in [0,1]$:

$$\gamma_c(x,0) = 1, \quad \gamma_c(x,1) = s_c(x), \quad \kappa(x,t) := \sum_{c=1}^{C} \gamma_c(x,t) P_c$$
$$f(x|\kappa(x,t)) \approx f(x|\theta^*)\,. \tag{9}$$

This is a stricter definition of 'inactive' that seeks to exclude cases like components P_c and $P_{c'}$, both being 'active' and important but canceling each other out. Without this stricter condition, we could have pathological solutions to the optimisation problem. For example, if $P'_1, \ldots, P'_C$ are a large set of parameter vectors for expert networks specialized to particular tasks the target network is capable of, completely unrelated to the target network θ^*, we could set

$$P_c = P'_c + \frac{1}{C}\left(\theta^* - \sum_{c'} P'_{c'}\right)\,. \tag{10}$$

The resulting components would add up to the target network, $\sum_c P_c = \theta^*$. And a single P_c would always be sufficient to get the same performance as the target network. However, the components P_c could be completely unrelated to the mechanistic structure of the target network. Requiring that the parameter components can be ablated part of the way and, in any combination, excludes counterexamples like this.

The assumption of parameter linearity Equations 5 and 9 together define what we mean when we say that we want to decompose a network's parameter vector into a sum of other parameter vectors that correspond to distinct mechanisms of the network. The idea this definition expresses is that different mechanisms combine *linearly* in parameter space to form the whole network. If neural networks do consist of a unique set of mechanisms in a meaningful sense, the ability of APD to recover that set of mechanisms relies on the assumption that the mechanisms are encoded in the network parameters in this linear manner, at least up to some approximation. We call this the assumption of parameter linearity.

The assumption of parameter linearity approximately holds for all the neural networks we study in this paper. Figure 7 shows a test of the assumption on our compressed computation model, by checking whether inactive components in the APD decomposition can be ablated in random combinations without substantially affecting the end result.

Whether the assumption of parameter linearity is satisfied by all the non-toy neural networks that we might want to decompose is ultimately an empirical question. Current theoretical frameworks for computing arbitrary circuits in superposition [Vaintrob et al., 2024, Bushnaq and Mendel, 2024] do seem to satisfy this assumption. They linearly superpose mechanisms in parameter space to perform more computations than the model has neurons[10]. This tentatively suggests that real models relying on superposition for computation do the same.

A.2 Deriving the losses used in APD from the Minimum Description Length Principle

A.2.1 Idealised loss: Minimum description length loss, $\mathcal{L}_{\text{MDL}}$

We suspect that many models of interest only use a fraction of the mechanisms they have learned to process any particular input. Thus, we want a decomposition of our models into components such that the total complexity of the components used on any given forward pass (measured in bits) is minimized.

Motivating case: Parameter components that are rank 1 and localized in one layer. Suppose the elementary components in our model were all very simple, with each of them being implemented by a rank 1 weight matrix $P_{c,l,i,j} = U_{c,l,i}V_{c,l,j}$ in some layer l of the network.[11] If we wanted to

[10] In such a manner that equation 9 should be satisfied up to terms scaling as ca. $\mathcal{O}(\epsilon)$, where ϵ is the noise level in the outputs of the target model due to superposition

[11] We think that the total number of components C here seems in theory capped to stay below the total number of network parameters $C = \mathcal{O}(N)$. See Bushnaq and Mendel [2024] for discussion.

minimize the complexity used to describe the model's behavior on a given data point x, then we should minimize the number of components that have a non-zero causal influence on the output on that data point. In other words, we want to optimise the component attributions $A(x)$ to be sparse.

With a dense code, the attributions $A_c(x)$ on a given input x would cost $\sum_{c=1}^{C} \alpha = C\alpha$ bits to specify, where α is the number of bits of precision we use for a single $A_c(x)$.

However, with a sparse code, we would instead need $\sum_{c=1}^{C} ||A_c(x)||_0 \, (\alpha + \log_2(C))$ bits, where $||A_c(x)||_0$ is the L_0 'norm' of $A_c(x)$. If the parameter component attributions $A_c(x)$ are sparse enough, this can be a lot lower than $C\alpha$. This leverages the fact that we can list only the indices and attributions of the subnets with non-zero $A_c(x)$. This requires $\log_2(C)$ bits for the index and α bits for the attribution.

General case: Parameter components that have arbitrary rank and may be distributed across layers. We do not expect all the parameter components of models to always be rank 1 matrices; they may be arbitrary rank and span multiple layers. [12]

We can treat this similar to the motivating case above, but where a parameter component that consists of a rank 2 matrix can be represented as two rank 1 matrices that always co-activate. If two rank 1 matrices almost always coactivate, then we can describe their attributions in two ways:

1. If we consider them as **two separate components**, then we would need $2\log_2(C) + 2\alpha$ bits to describe their attributions for each data point they activate on ($\log_2(C)$ for the index and 2α for the two attributions).
2. However, if we consider them as **one separate component**, then we only need one index to identify both of them, and therefore only need $\log_2(C) + 2\alpha$ bits

This means that we may be able to achieve shorter description lengths using a mixed coding scheme that allows for both dense and sparse codes. Thus, if we use a mixed coding scheme that allows rank 1 parameter components to be aggregated into higher dimensional components, it gives us a description length of

$$\mathcal{L}_{\text{MDL}}(x) = \sum_{c=1}^{C} ||A_c(x)||_0 \left(\alpha \sum_l \text{rank}(P_{c,l}) + \log_2(C) \right) \tag{11}$$

$$= \log_2(C) \left(\sum_{c=1}^{C} ||A_c(x)||_0 \right) + \alpha \left(\sum_{c=1}^{C} ||A_c(x)||_0 \sum_l \text{rank}(P_{c,l}) \right) \tag{12}$$

$$=: \log_2(C) \mathcal{L}^{\text{idealized}}_{\text{minimality}}(x) + \alpha \mathcal{L}^{\text{idealized}}_{\text{simplicity}}(x) \tag{13}$$

where $\sum_l \text{rank}(P_{c,l})$ is the total rank of component P summed over the weight matrices in all the components of the network.

Optimizing our components P_c to minimise $\mathcal{L}_{\text{MDL}}(x)$ would then yield a decomposition of the network that uses only small values for the total number of active components and the total rank of the active components on a particular forward pass.

The prefactor α in this equation then sets the point at which two lower-rank components coactivate frequently enough that merging them into a single higher-rank component lowers the overall loss. Thus, α is effectively a hyperparameter controlling the resolution of our decomposition. As α increases, the threshold for merging components rises, with all components becoming rank 1 in the limit $\alpha \to \infty$. If we set $\alpha = 0$, all components would merge, so our decomposition would simply return the target network's parameter vector.

Full idealised MDL loss For our the loss term that we use to train our parameter components, we want a decomposition that approximately sums to the target parameters and minimises description

[12] Nevertheless, current hypotheses for how models might implement computations in superposition suggest that components would tend to be low-rank. [Bushnaq and Mendel, 2024]. Otherwise, there would just not be enough spare description length to fit all the (high rank) parameter components that are necessary to do the computation in superposition into the network.

length. We can accomplish this by adding a faithfulness loss

$$\mathcal{L}_{\text{faithfulness}} = \sum_{l,i,j} \left(\theta^*_{l,i,j} - \sum_{c=1}^{C} P_{c,l,i,j} \right)^2 , \tag{14}$$

to our minimum description length loss. Our full loss is then:[13]

$$\mathcal{L}_{\text{faithfulness}} + \mathcal{L}_{\text{MDL}}(x) = \mathcal{L}_{\text{faithfulness}} + \beta \mathcal{L}^{\text{idealized}}_{\text{minimality}}(x) + \alpha \mathcal{L}^{\text{idealized}}_{\text{simplicity}}(x) \tag{15}$$

However, this idealized loss would be difficult to optimize since the L_0 'norm' $||A_c(x)||_0$ and $\text{rank}(P_c)$ are both non-differentiable. We therefore must optimize a differentiable proxy of this loss instead.

We have devised two different proxy losses for this, leading to two different implementations of APD. The first uses a **top-k formulation** (Section A.2.2), whereas the second assigns an L_p penalty to attributions A.2.3. We primarily use the top-k formulation in our work. But we include the L_p version for explanatory purposes.

A.2.2 Practical loss: Top-k formulation of APD

Approximating $\mathcal{L}^{\text{idealized}}_{\text{minimality}}$ in the top-k formulation We can approximate optimizing for the loss in equation 15 with a top-k approach: We run the network once on data point x and collect attributions $A_c(x)$ for each parameter component P_c. Then, we select the parameter components with the top-k largest attributions and perform a forward pass using only those components.

$$\begin{aligned} s_c(x) &\in \{0, 1\} \\ s_c(x) &= \text{top-k}(\{A_c(x)\}) \\ \kappa(x) &:= \sum_{c=1}^{C} s_c(x) P_c \end{aligned} \tag{16}$$

This 'sparse' forward pass should ideally only involve the structure in the network that is actually used on this specific input, so it should give the same result as a forward pass using all P_c. We can optimise for this using a loss

$$\mathcal{L}_{\text{minimality}}(P|\theta^*, X) = D\left(f(x|\theta^*), f(x|\kappa(x))\right) . \tag{17}$$

where D is some distance measure between network outputs, e.g. MSE loss or KL-divergence. Minimising $\mathcal{L}_{\text{minimality}}$ for a small k then approximately minimises $\sum_{c=1}^{C} ||A_c(x)||_0$ in the ideal loss.

Reconstructing hidden activations It is possible that reconstructing the network outputs on the sparse forward pass is not a strong enough condition to ensure that the components we find correspond to the mechanisms of the network, particularly since our attributions are imperfect. To alleviate this, we can additionally require some of the model's hidden activations on the sparse forward pass to reconstruct the target model's hidden activations. This can also aid training dynamics in deeper models, as APD can match the target model layer by layer instead of needing to re-learn everything from scratch. However, theories of computation in superposition predict that unused components still contribute noise to the model's hidden preactivations before non-linearities, which is then filtered out [Hänni et al., 2024, Bushnaq and Mendel, 2024]. So we do not necessarily want to match the hidden activations of the target model everywhere in the network. Finding a principled balance in this case is still an open problem. We use a hidden activation reconstruction loss for our single and multilayer models of compressed computation in Section 3.2 and Section 3.3.

Approximating $\mathcal{L}^{\text{idealized}}_{\text{simplicity}}$ in the top-k formulation To approximate $\mathcal{L}^{\text{idealized}}_{\text{simplicity}}$, we need some tractable objective function that approximately minimises $\text{rank}(P_c)$. We use the *Schatten norm*: The rank of a matrix M can be approximately minimised by minimising $||M||_p$ [Giampouras et al., 2020] with $p \in (0, 1)$:

$$||M||_p := \left(\sum_m |\lambda_m|^p \right)^{\frac{1}{p}} \tag{18}$$

[13] Here, we've absorbed $\log(C)$ from $\mathcal{L}_{\text{MDL}}$ in the previous section into β.

Here, λ_m are the singular values of M. So, we can approximate rank(P_c) in the loss with

$$\sum_{c=1}^{C} ||P_c||_p^p = \sum_{c=1}^{C} \sum_{l,m} |\lambda_{c,l,m}|^p \,, \tag{19}$$

where $\lambda_{c,l,m}$ is singular value m of component c in layer l.

Performing a singular value decomposition for every component at every layer every update step would be cumbersome. We can circumvent this by parametrizing our components in factorized form, as a sum of outer products of vectors U, V:

$$P_{c,l,i,j} := \sum_k U_{c,l,m,i} V_{c,l,m,j} \tag{20}$$

If we now replace $\lambda_{c,l,m}$ with

$$\lambda_{c,l,m} \to \left(\sum_{i,j} U_{c,l,m,i}^2 V_{c,l,m,j}^2 \right)^{\frac{1}{2}} \tag{21}$$

then V_c and U_c will be incentivised to effectively become proportional the right and left singular vectors for subnet P_c.

The Schatten norm of P_c can then be written in factorised form as:

$$\mathcal{L}_{\text{simplicity}}(x) = \sum_{c=1}^{C} s_c(x) \sum_{l,m} \left(\sum_{i,j} U_{c,l,m,i}^2 V_{c,l,m,j}^2 \right)^{\frac{p_2}{2}} . \tag{22}$$

Full set of loss functions in the top-k formulation To summarise, our full loss function is

$$\begin{aligned}
\mathcal{L}(x) &= \mathcal{L}_{\text{faithfulness}} + \beta \mathcal{L}_{\text{minimality}}(x) + \alpha \mathcal{L}_{\text{simplicity}}(x) \\
\mathcal{L}_{\text{faithfulness}} &= \sum_{l,i,j} \left(\theta^*_{l,i,j} - \sum_{c=1}^{C} P_{c,l,i,j} \right)^2 \\
\mathcal{L}_{\text{minimality}}(x) &= D\left(f(x|\theta^*), f(x| \sum_{c=1}^{C} s_c(x) P_c) \right) \\
\mathcal{L}_{\text{simplicity}}(x) &= \sum_{c=1}^{C} s_c(x) \sum_{l} ||P_c||_p^p \,.
\end{aligned} \tag{23}$$

The components P_c are parametrised as

$$P_{c,l,i,j} := \sum_k U_{c,l,m,i} V_{c,l,m,j} \,. \tag{24}$$

The top-k coefficients $s_c(x)$ are chosen as

$$\begin{aligned}
s_c(x) &\in \{0, 1\} \\
s_c(x) &= \text{top-k}(\{A_c(x)\})
\end{aligned} \tag{25}$$

where $A_c(x)$ are attributions quantifying the effect of components P_c on the network, computed with attribution patching as in equation 8, or with some other attribution method. Finally, $||P_c||_p$ denotes the Schatten norm, and $p \leq 1.0$ is a hyperparameter.

$\mathcal{L}_{\text{minimality}}(x)$ may include additional terms penalizing the distance D between some of the hidden activations of the target model θ^*, and the sparse forward pass using parameters $\sum_{c=1}^{C} s_c(x) P_c$.

We use batch top-k instead of top-k [Bussmann et al., 2024], picking the components with the largest attributions over a batch of datapoints instead of single inputs.

A.2.3 Alternative practical loss: APD formulation that uses an L_p penalty on attributions

As an alternative to the top-k loss, we can also approximately optimize for loss 15 with an L_p approach. Optimizing the L_p norm with $p \leq 1$ will tend to yield solutions with small L_0 'norm', while still being differentiable. So we can replace $||A_c(x)||_0$ in the loss with $|A_c(x)|^p$. Our losses would then be

$$\mathcal{L}^{L_p}_{\text{minimality}}(x) = \sum_{c=1}^{C} |A_c(x)|^{p_1}$$
$$\mathcal{L}^{L_p}_{\text{simplicity}}(x) = \sum_{c=1}^{C} \sum_{l} |A_c(x)|^{p_1} \left(\sum_{i,j} U^2_{c,l,m,i} V^2_{c,l,m,j} \right)^{\frac{p_2}{2}}, \tag{26}$$

where $p_1, p_2 \leq 1.0$ are the norm of the attributions and the Schatten norm of the matrices respectively.

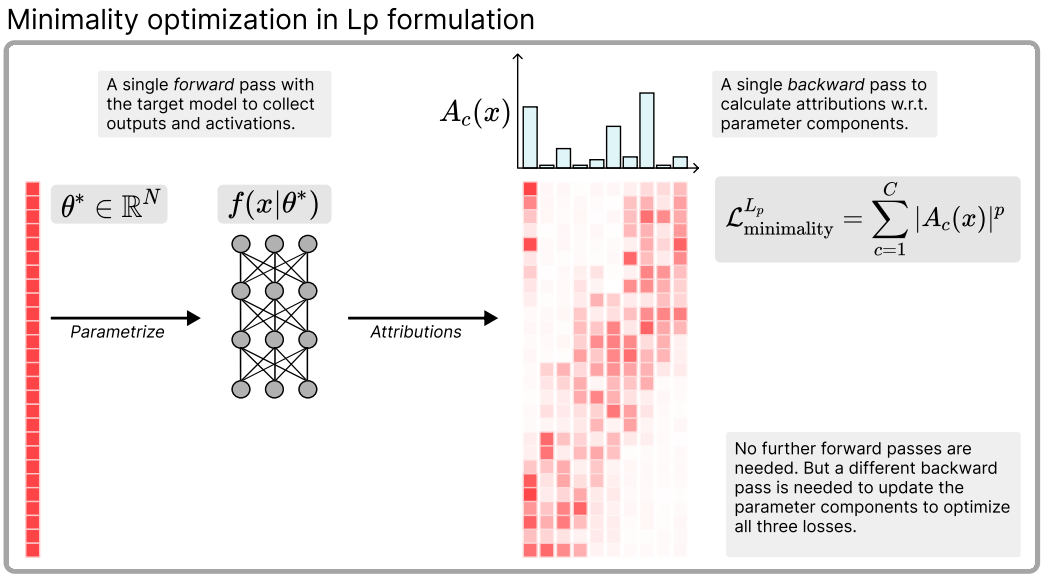

Figure 11: Optimizing $\mathcal{L}^{L_p}_{\text{minimality}}$

We did not thoroughly explore this implementation because our early explorations that used the L_p approach did not work as well as our top-k implementation for unknown reasons. We may revisit this approach in the future.

B Further experiments

B.1 Hand-coded gated function model: Another cross-layer distributed representation setting

B.1.1 Setup

In this task, we hand-code a target network to give an approximation to the sum of a set of trigonometric functions, governed by a set of control bits. The functions being approximated are of the form $F_i(x) = a_i \cos(b_i x + c_i) + d_i \sin(e_i x + f_i) + h_i$ with randomly generated coefficients $\{a_i, b_i, c_i, d_i, e_i, f_i, g_i\}$ drawn from uniform distributions (Table 2) for each unique function i.

The input to the network is a vector, $(x, \alpha_0, \cdots, \alpha_{n-1})$, whose entries are a scalar $x \in [0, 5]$ and a set of n binary control bits $\alpha_i \in \{0, 1\}$. The control bits α_i are sparse, taking a value of 1 with probability $p = 0.05$ and 0 otherwise. A function is only "active" (i.e. it should be summed in the output of the network) when its corresponding control bit is on.

Similar to our model of cross-layer distributed representations in Section 3.3, we use 2-layer residual MLP network with ReLU activations. This model is hand-crafted to have n clearly separable

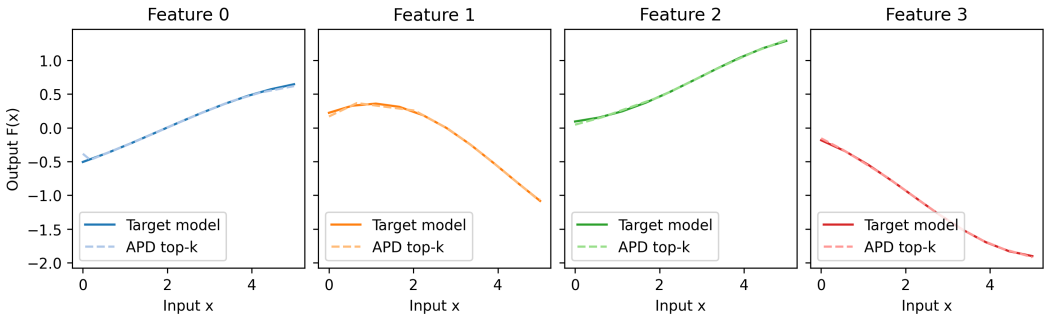

Figure 12: Hand-coded gated function model: The four functions $f_i(x)$ implemented by the hand-coded gated function model (solid lines), and the outputs of the top-k forward pass of the APD-decomposed model (dashed lines). The APD model almost perfectly matches the hand-coded network.

mechanisms that each approximate a unique trigonometric function. Notably, each function is computed by a unique set of neurons.

The output of the target model is a piecewise approximation of $y(x) = \sum_i \alpha_i F_i(x)$ with n functions $y_i(x)$.

Coefficient	Range
a	$\mathcal{U}(-1, 1)$
b	$\mathcal{U}(0.1, 1)$
c	$\mathcal{U}(-\pi, \pi)$
d	$\mathcal{U}(-1, 1)$
e	$\mathcal{U}(0.1, 1)$
f	$\mathcal{U}(-\pi, \pi)$
g	$\mathcal{U}(-1, 1)$

Table 2: Ranges of coefficients sampled from uniform distributions for the functions used in the hand-coded gated function model.

In our experiments, we use a total of $n = 4$ unique functions, with each function using $m = 10$ neurons to piecewise-approximate the functions $F_i(x)$. We show these approximated functions in Figure 12 (solid lines). The 5 inputs of our network (x, α_0, α_1, α_2, α_3) are stored in the first 5 dimensions of the residual stream, alongside a dimension that we read off as the output of the network ($\hat{y}(x)$). To hand-code the piecewise approximation of the individual functions y_i we randomly select m neurons from the MLPs, typically distributed across layers. This also means that the value of $\hat{y}_i$ is not represented in the intermediate layers, but only in the final layer.

We show the weights of the hand-coded target network in the leftmost panel of Figure 13. The graph shows the residual MLP network, with weights shown as lines. Each neuron is monosemantic, that is, it is used to approximate one of the $F_i(x)$ functions. Each neuron connects to the respective control bit α_i as well as the x input. All neurons write to the output activation, which is the last dimension in the residual stream. The line color in Figure 13 indicates which task (i.e. which function F_i) the weight implements; the line width indicates the magnitude of the weight.

When applied to this network, APD should partition the network weights θ^* into $C = n$ parameter components P_c, each corresponding to the weights for one approximated function $F_i(x)$ (i.e. of one colour).

B.1.2 Results

We find that APD can decompose this network into approximately correct parameter components. However, APD is particularly difficult to train in this setting, with only minor changes in hyperparameters causing large divergences. We hypothesize that this may be due to the fact that the ground truth network is itself hand-coded, not trained. We show a cherry-picked example (out of many runs that vary the number of MLP layers and number of functions) in Figure 13.

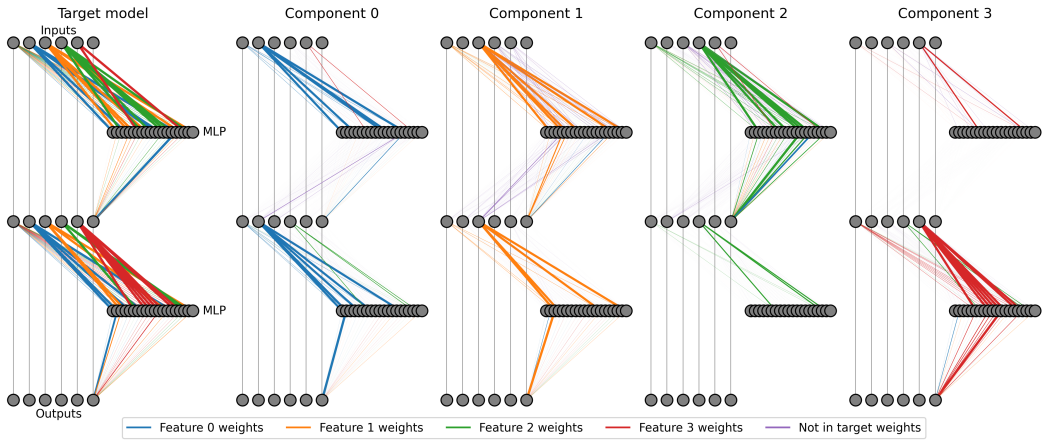

Figure 13: The parameters of the hand-coded gated function model decomposed into parameter components. **Leftmost panel:** The hand-coded network parameters, colored by the unique functions $F_i(x)$. **Other panels:** The parameter components identified by APD, coloured by the function they correspond to in the target model, or purple if the weight is zero in the target model.

Figure 13 shows the target network weights (leftmost column), and their decomposition into the four APD-generated components (remaining columns). We color the weights by which feature they correspond to in the target model, or purple if the weight is not present in the target model. We observe that the components mostly capture one function each (most weights within a parameter component are the same color).

However, the solution is not perfect. Some weights that are not present in the target network are nevertheless nonzero in some of the parameter components. Additionally, the W^1_{out} weights of parameter component 2 and W^0_{out} weights of parameter component 3 seem to be absorbed into other parameter components. This may be due to the difficulty in training APD on a handcoded model as mentioned earlier, or may be a symptom of the simplicity loss $\mathcal{L}_{\text{simplicity}}$ not being fully layer-independent, causing an over-penalization of weights being in a layer on their own (see Appendix 4.2.4 for a discussion on layer-privileging).

C Further analyses

C.1 Analysis of the compressed computation target model

In this section we provide more details about the performance of the target residual MLP model that is used to train APD, as discussed in Section 3.2.

Recall that we train the target network to approximate $y_i = x_i + \text{ReLU}(x_i)$. Note that the model output can be written as

$$\mathbf{y} = W_U W_E \mathbf{x} + W_U W_{\text{out}} \text{ReLU}(W_{\text{in}} W_E \mathbf{x}).$$

Since W_E consists of random unit vectors and is not trained. Also, $W_U = W_E^T$. As a result, the first summand already approximates a noisy identity and the second summand mostly approximates the ReLU function.

Figure 14 (left) shows the output of the model for an arbitrary one-hot input ($x_{42} = 1$). We see that the output $\hat{x}_{42} \approx 1.6$ is close to the target value of 2.0, and the remaining outputs $\hat{x}_{i \neq 42}$ are close to 1.0. We checked whether the noise in the $\hat{x}_{i \neq 42}$ outputs comes from the $W_U W_E$ or MLP term, and found that it is dominated by the MLP term.[14] We confirm that $\hat{x}_{42}$ indeed approximates a ReLU function for $\hat{x} \in [-1, 1]$ in Figure 14 (right panel), though not perfectly. It appears to systematically undershoot the labels. We expect that this is due to the MSE loss: Although the model could scale the outputs (by scaling e.g. W_{out}) to match $y_{42} = 2.0$, it would also increase the loss overall.

[14]This is not the case for small embedding sizes, such as $d_{\text{resid}} = 100$. This is why we chose a large embedding size to focus on the MLP noise.

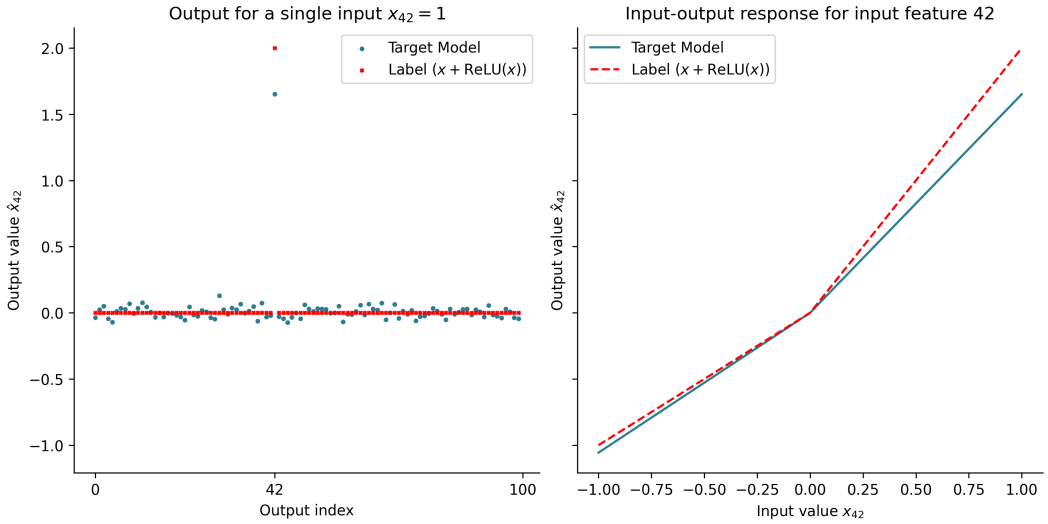

Figure 14: Output of the 1-layer residual MLP target model compared to true labels for a single active input. **Left**: Output at all output indices for single one-hot input $x_{42} = 1$. **Right**: Output at index $j = 42$ for inputs with $x_{42} \in [0, 1]$ and $x_j = 0$ for $j \neq 42$.

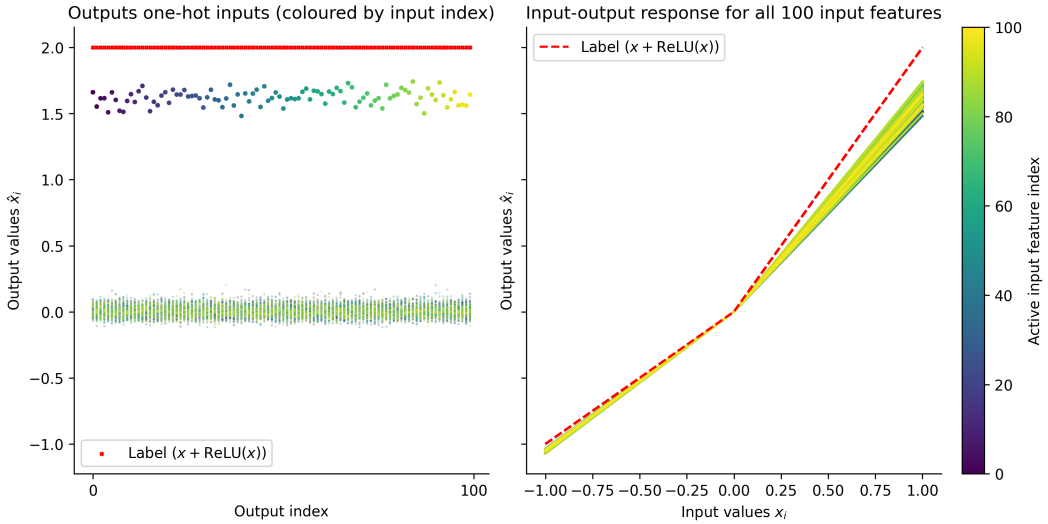

Figure 15: Output of the 1-layer residual MLP target model compared to true labels for the full set of 100 one-hot inputs. **Left**: Output at all output indices over the set of inputs. The point color indicates the active input feature, and label values are in red. **Right**: Output at index i for inputs with $x_i \in [0, 1]$ and $x_j = 0$ for $j \neq i$. Line colors indicate the input feature index.

So far we have focused on the arbitrary input index 42. Figure 15 repeats the same experiment but overlaying the results of all 100 input features (lines color indicating the input feature index). We can see that the model treats all input features approximately the same.

C.2 Analysis of the compressed computation APD model

For a setting like the compressed computation task, where the dataset consists of input features activating independently with probability p, a natural choice for the batch top-k hyperparameter is a value close to p multiplied by the number of input features. In our experiments, this would be $0.01 \times 100 = 1$. For this value of batch top-k (and similar), there are batches in which APD must activate more parameter components than there are active features, and likewise, batches in which APD must activate fewer parameter components than there are active features. In our 1-layer and

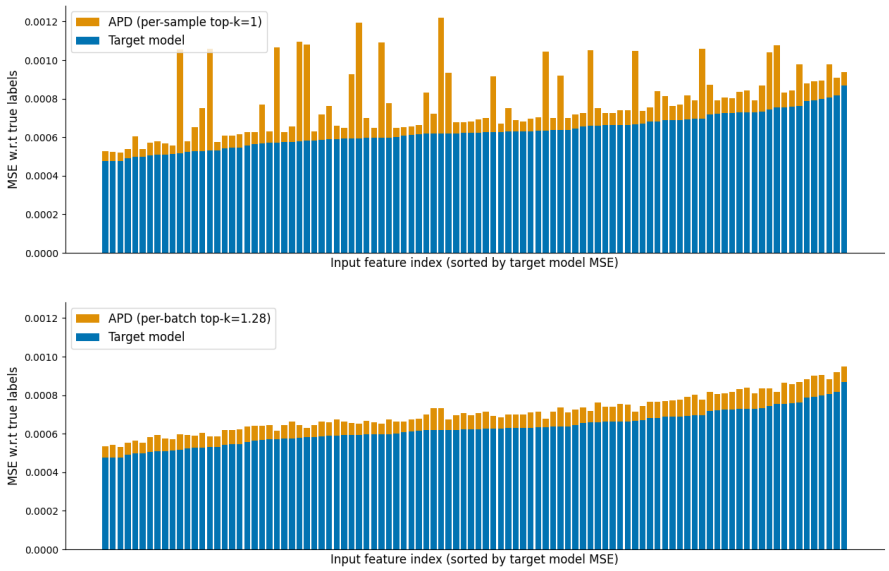

Figure 16: MSE for APD trained with batch top-$k = 1.28$ in the 1-layer residual MLP setting for samples with a single active input feature (i.e. one-hot), averaged over 100k samples. **Top:** Comparison of the target model with the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). **Bottom:** Comparison of the target model with the APD model using batch top-$k = 1.28$. The batch top-k mask is applied to the original training distribution and then samples without exactly one active input feature are filtered out.

2-layer residual MLP experiments in Section 3.2 and Section 3.3, respectively, we chose the value of batch top-$k = 1.28$ to be such that in almost no batches would there be more active input features than active components (we use a batch size of 256). The benefits of choosing this large batch top-k value are:

1. APD can learn to handle rarer samples with many active input features.
2. Since there are very rarely more active input features than active components, the components are not encouraged to represent the computations of multiple input features.

However, since there are extra active parameter components in most batches, APD exhibits a behavior where, for a subset of input features, it represents part of its computation in multiple parameter components. This phenomenon is illustrated in Figure 16, where the APD model achieves a low loss across all input features when using its trained batch top-$k = 1.28$ setting (bottom). However, when constrained to activate only a single parameter component per sample, the model exhibits large losses for a non-negligible subset of the input features (top). These results are based on samples where only one input feature is active. This behavior is further characterized in Figure 18. Samples with higher MSE loss under single-component activation tend to require more parameter components on the training distribution with batch top-$k = 1.28$.

As shown in Figure 17, we see that training with a reduced batch top-k value of 1 (rather than 1.28) reduces the number of input features that have a large MSE loss when only activating a single parameter component. However, the downside of using a smaller top-k value is that we end up with more components that fully represent two different input feature computations, rather than one. See figures in the "Toy Model of Compressed Computation (1 layer) with batch top-k= 1" section here for details. This should not be surprising; when top-k is smaller, there are more batches in which the number of active input features is larger than the number of active components. APD is then incentivized to represent multiple input feature computations in a single parameter component to achieve a smaller $\mathcal{L}_{\text{minimality}}$ (though, at the cost of a larger $\mathcal{L}_{\text{simplicity}}$).

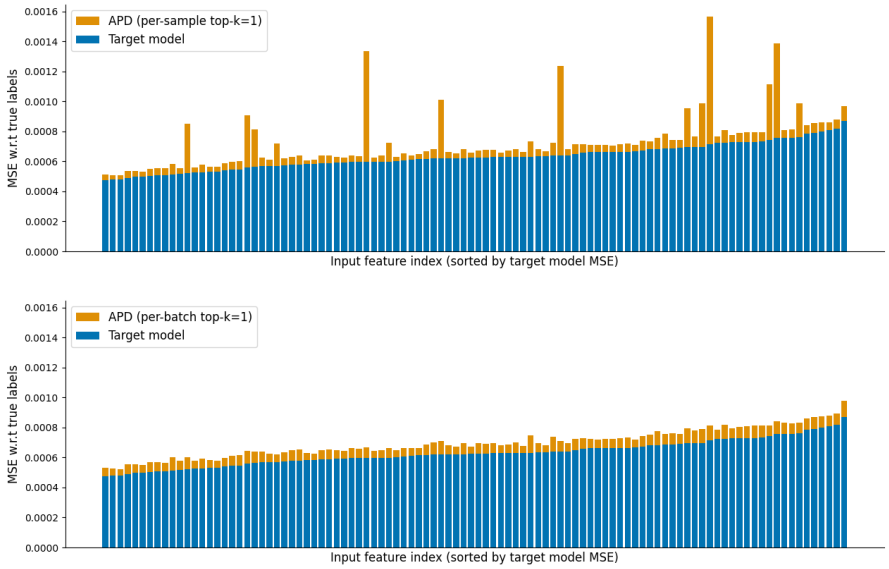

Figure 17: MSE for APD trained with batch top-$k = 1$ in the 1-layer residual MLP setting for samples with a single active input feature (i.e. one-hot), averaged over 100k samples. **Top:** Comparison of the target model with the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). **Bottom:** Comparison of the target model with the APD model using batch top-$k = 1$. The batch top-k mask is applied to the original training distribution and then samples without exactly one active input feature are filtered out.

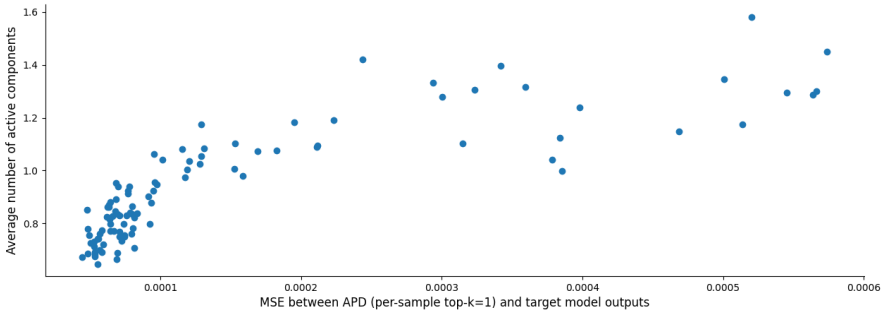

Figure 18: Relationship in the 1-layer residual MLP setting between: (y-axis) the average number of active APD parameter components when using batch top-$k = 1.28$, and (x-axis) the MSE between the target model outputs and the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). MSE is measured only on samples with a single active input feature.

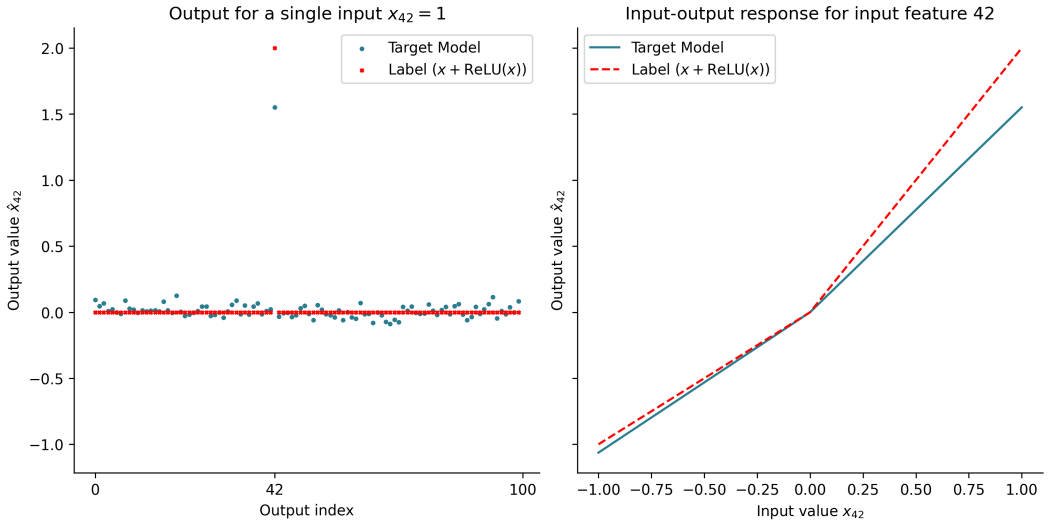

Figure 19: Output of the 2-layer residual MLP target model compared to true labels for a single active input. **Left**: Output at all output indices for single one-hot input $x_{42} = 1$. **Right**: Output at index $j = 42$ for inputs with $x_{42} \in [0, 1]$ and $x_j = 0$ for $j \neq 42$.

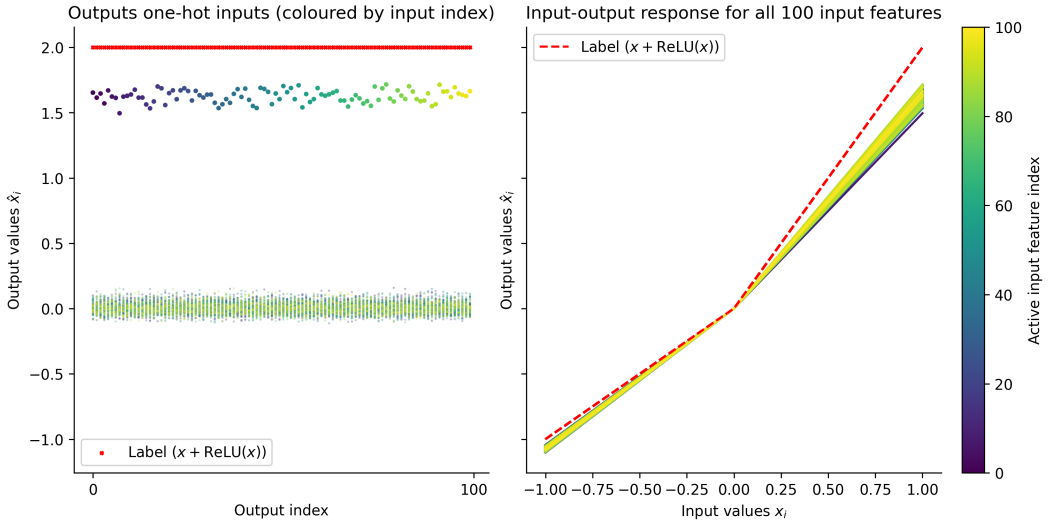

Figure 20: Output of the 2-layer residual MLP target model compared to true labels for the full set of 100 one-hot inputs. **Left**: Output at all output indices over the set of inputs. The point color indicates the active input feature, and label values are in red. **Right**: Output at index i for inputs with $x_i \in [0, 1]$ and $x_j = 0$ for $j \neq i$. Line colors indicate the input feature index.

C.3 Analysis of the cross-layer distributed representations target model

In Figures 19 and 20, we show that the trained target model for the cross-layer distributed representations setting in Section 3.3 (i.e. 2-layer residual MLP) is qualitatively similar to the target model in the compressed computation setting (i.e. 1-layer residual MLP) we analyzed in Appendix C.1.

C.4 Analysis of the cross-layer distributed representations APD model

Here, we show that the APD model for the cross-layer distributed representations setting in Section 3.3 (i.e. 2-layer residual MLP) is qualitatively similar to the APD model in the compressed computation setting (i.e. 2-layer residual MLP) we analyzed in Section 3.2.

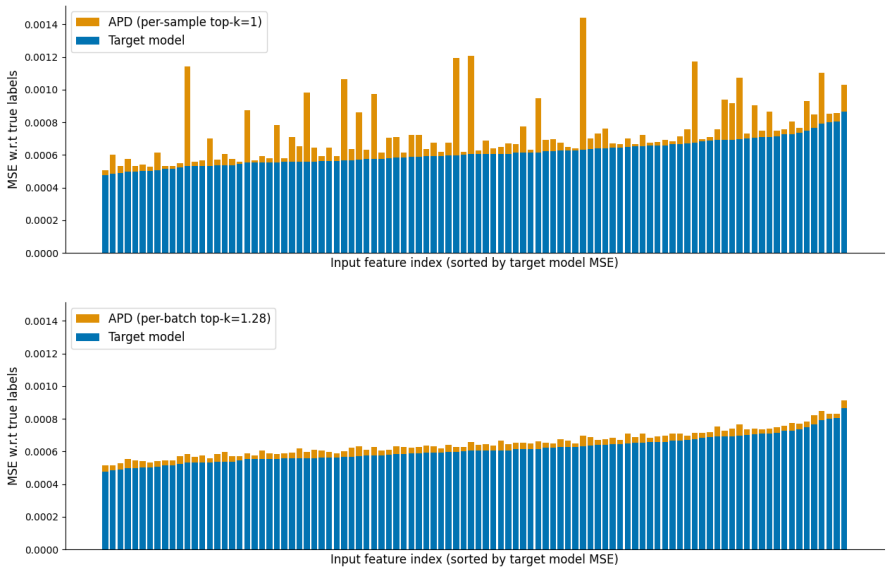

Figure 21: MSE for APD trained with batch top-$k = 1.28$ in the 2-layer residual MLP setting for samples with a single active input feature (i.e. one-hot), averaged over 100k samples. **Top:** Comparison of the target model with the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). **Bottom:** Comparison of the target model with the APD model using batch top-$k = 1.28$. The batch top-k mask is applied to the original training distribution and then samples without exactly one active input feature are filtered out.

When running APD with batch top-$k = 1.28$ in the 2-layer residual MLP setting, we observe the same phenomenon previously described in Appendix C.2 for the 1-layer case: certain input feature computations are not fully captured by individual parameter components (Figures 21 and 23). As in the 1-layer setting, training with a reduced batch top-k value of 1.28 helps address this issue (Figure 22), though we again end up with more components that fully represent multiple input feature computations (see figures in the "Toy Model of Compressed Computation (2 layer) with batch top-k= 1" section here for details).

It is worth noting that, if we instead enforce a rank-1 constraint on the parameter components in each network layer, we are able to get the best of both worlds. That is, APD does not learn parameter components that fully represent multiple input feature computations (it is unable to do this since this would require matrices with rank> 1), and one is able to reduce the batch top-k value to avoid having partial representations of an input feature computation across multiple components (in fact, leaving batch top-$k = 1.28$ almost completely rectifies this issue in the rank-1). See the "Rank-1 Toy Model of Cross-layer Distributed Representations (2 layers)" section here for details.

To further show that APD is indifferent to computations occurring in multiple layers, we replicate the 1-layer figures (7 and 8) for the 2-layer setting in Figures 24 and 25, respectively. The qualitatively similar results indicate that despite the learned parameter components representing computation occurring across multiple layers, the components have minimal influence on forward passes when their corresponding input feature is not active.

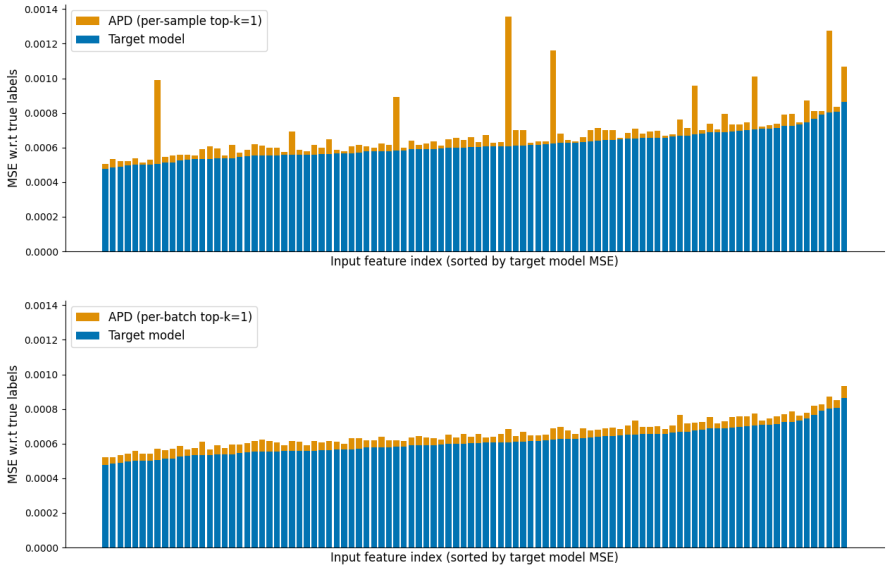

Figure 22: MSE for APD trained with batch top-$k = 1$ in the 2-layer residual MLP setting for samples with a single active input feature (i.e. one-hot), averaged over 100k samples. **Top:** Comparison of the target model with the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). **Bottom:** Comparison of the target model with the APD model using batch top-$k = 1$. The batch top-k mask is applied to the original training distribution and then samples without exactly one active input feature are filtered out.

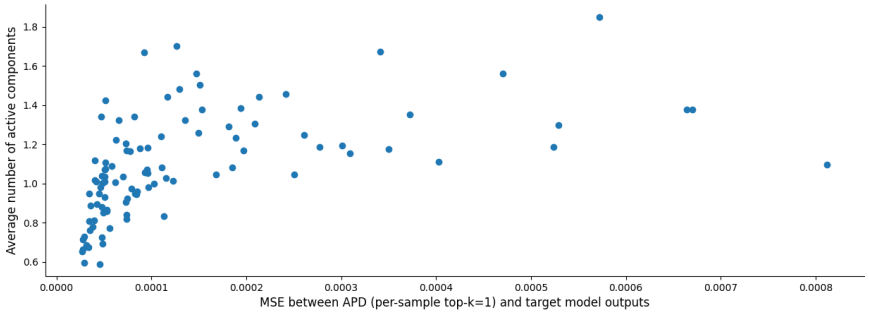

Figure 23: Relationship in the 2-layer residual MLP setting between: (y-axis) the average number of active APD parameter components when using batch top-$k = 1.28$, and (x-axis) the MSE between the target model outputs and the APD model when activating exactly one parameter component in each sample (i.e. top-$k = 1$). MSE is measured only on samples with a single active input feature.

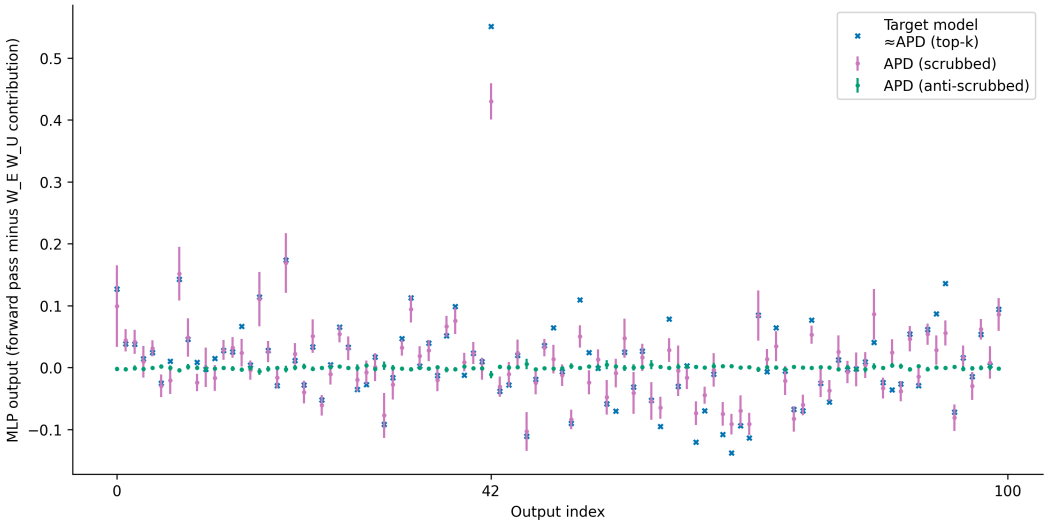

Figure 24: Output of multiple 2-layer residual MLP APD forward passes with one-hot input $x_{42} = 1$ over 10k samples, where half of the parameter components are ablated in each run. Purple lines show "scrubbed" runs (parameter component corresponding to input index 42 is preserved), while green lines show "anti-scrubbed" runs (component 42 is among those ablated). The target model output is shown in blue, which is almost identical to the output on the APD sparse forward pass (i.e. APD (top-k)).

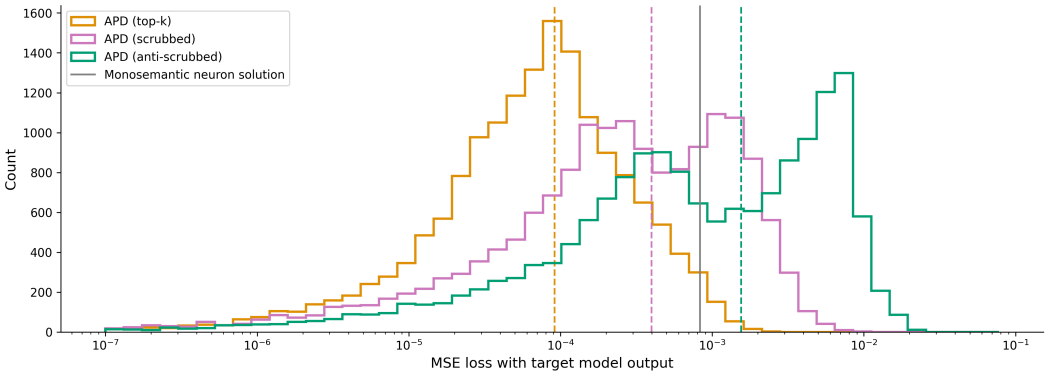

Figure 25: MSE losses of the 2-layer residual MLP APD model on the sparse forward pass ("top-k") and the APD model when ablating half (50) of its parameter components ("scrubbed" when none of the components responsible for the active inputs are ablated and "anti-scrubbed" when they are ablated). The gray line indicates the loss for a model which uses one monosemantic neuron per input feature.

D Training details and hyperparameters

D.1 Toy models of superposition (TMS)

D.1.1 TMS with 5 input features and hidden size of 2

The target model was trained for 5k steps with a batch size of 1024. We use the AdamW optimizer [Loshchilov and Hutter, 2019] with a weight decay of 0.01 and a constant learning rate of 0.005. Our datasets consists of samples with each of the 5 input features taking values in the range $[0, 1]$ (uniformly) with probability 0.05 and 0 otherwise.

To train the APD model for TMS, we use the Adam optimizer [Kingma and Ba, 2017] with a constant learning rate of 0.03 with a linear warmup over the first 5% steps. We use the same data distribution as for training the target model (feature probability 0.05). We train for 20k steps with a batch size of 2048, and a batch top-k value of 0.211, indicating that $0.211 \times 2048 = 432$ parameter components are active in each batch. The coefficients for the loss functions are set to 1 for $\mathcal{L}_{\text{faithfulness}}$, 1 for $\mathcal{L}_{\text{minimality}}$, and 0.7 for $\mathcal{L}_{\text{simplicity}}$ with a L_p norm of 1.

D.1.2 TMS with 40 input features and hidden size of 10

The target model was trained for 2k steps with a batch size of 2048 (we expect we would have achieved the same results with 5k steps and batch size 1024, as we used for the smaller TMS setting). We use AdamW with a weight decay of 0.01 and learning rate constant learning rate of 0.005. Our datasets consists of samples with each of the 5 input features taking values in the range $[0, 1]$ (uniformly) with probability 0.05 and 0 otherwise.

To train the APD model, we use Adam with a max learning rate of 0.001 that decays with a cosine schedule and has a linear warmup over the first 5% steps. We use 50 components, allowing for 10 to 'die' throughout training. We use the same data distribution as for training the target model (feature probability 0.05). We train for 20k steps with a batch size of 2048, and a batch top-k value of 1, indicating that an average of $1 \times 2048 = 2048$ parameter components are active in each batch. The coefficients for the loss functions are set to 1 for $\mathcal{L}_{\text{faithfulness}}$, 10 for $\mathcal{L}_{\text{minimality}}$, and 20 for $\mathcal{L}_{\text{simplicity}}$ with a L_p norm of 0.9.

D.2 Compressed computation and cross-layer distributed representation

Recall that the 1-layer residual MLP (Section 3.2) and 2-layer residual MLP (Section 3.3) both have 100 input features, an embedding dimension of 1000, and 50 MLP neurons (25 in each MLP layer for the 2-layer case). Both target models were trained using AdamW with a weight decay of 0.01, a max learning rate of 0.003 with cosine decay, batch size of 2048. The datasets consist of samples with each of the 100 input features taking values in the range $[-1, 1]$ (uniformly) with probability 0.01 and 0 otherwise.

Both 1-layer and 2-layer APD models were trained with the Adam optimizer with a max learning rate of 0.001 which had a linear warmup for the first 1% of steps and a cosine decay thereafter. The models were trained with a batch size of 2048, and a batch top-k value of 1.28, indicating that $1.28 \times 2048 = 2621$ parameter components are active in each batch. Both models have a coefficient set to 1 for $\mathcal{L}_{\text{faithfulness}}$, and 1 for a loss which reconstructs the activations after the non-linearity in the MLP layers.

The 1-layer model starts with 130 parameter components, trains for 40k steps, has a coefficient of 1 for $\mathcal{L}_{\text{minimality}}$ and 10 for $\mathcal{L}_{\text{simplicity}}$ with a L_p norm of 0.9. We also apply a normalization to the factorized form of the parameter components. Specifically, we normalize U in Equation 20 every training step so that it has unit norm in the in_dim dimension (labeled i in the equation). We expect that it's possible to achieve equivalent performance and stability without this normalization with a different set of hyperparameters.

The 2-layer model starts with 200 parameter components, trains for 10k steps, has a coefficient of 2 for $\mathcal{L}_{\text{minimality}}$ and 7 for $\mathcal{L}_{\text{simplicity}}$ with a L_p norm of 0.9.

We note that many of the inconsistencies in hyperparameters between different experiments are not due to rigorous ablation studies, and we expect to obtain similar results with more consolidated settings. In particular, changes to learning rate configurations (warmup, decay), training steps, L_p

norm for $\mathcal{L}_{\text{simplicity}}$, and batch size, did not tend to have a large influence on the results. Other hyperparameters such as the coefficients of the loss terms, and, to a lesser extent, the batch top-k value, do have a significant influence on the outcome of APD runs.

D.3 Hand-coded gated function model

Recall that our experiments use 4 unique functions, with each function using $m = 10$ neurons to piecewise-approximate each of the 4 functions.

For APD training, we use Adam with a max learning rate of 0.003 that decays with a cosine schedule and has a linear warmup over the first 0.4% steps. Our dataset consists of one input variable whose entries are drawn uniformly from $[0, 5]$ and four control bits taking a value of 1 with probability 0.05 and 0 otherwise. We train for 200k steps with a batch size of 10000, and a batch top-k value of 0.2217, indicating that $0.2217 \times 10000 = 2217$ parameter components are active in each batch. The coefficients for the loss functions are set to 0.1 for $\mathcal{L}_{\text{faithfulness}}$, 5 for $\mathcal{L}_{\text{minimality}}$, and 5 for $\mathcal{L}_{\text{simplicity}}$ with a L_p norm of 1.