Identifying Sparsely Active Circuits Through Local Loss Landscape Decomposition

Anonymous Authors[1]

Abstract

1. Background

Mechanistic interpretability aims to uncover the internal mechanisms responsible for the so-called black box behavior or large models so that developers can better understand, intervene on, and align models (Bereska & Gavves, 2024). One goal of the field is to be able to decompose model behavior into subcomponents that are less complex and more human interpretable but that still fully express model behavior. The most popular method in this space are Sparse Autoencoders (SAEs) (Gao et al., 2024; Cunningham et al., 2023; Bricken et al., 2023) which have been successful in resolving compressed features from the activation space of a model. SAEs are designed to identify latent features by projecting a model's compressed activation space into a sparsely activated, overcomplete basis. These learned basis vectors represent distinct features, which can then be used to reconstruct the original activations.

1.1. Limitations of Activation Space-Based Methods

However, decomposing the activation space of a model has various limitations. First, current SAE paradigm assumes that model's represent features linearly in a single layer's activation space. Growing evidence suggests that even architectures designed to encourage linearity, such as the residual stream in a transformer, may in fact encode non-linear features. (Engels et al., 2024a;b). Additionally, activation-based approaches generally treat layers and blocks as independent computational modules, reconstructing activations from specific layers without capturing the possibility of cross-layer interactions and superposition (Merullo et al., 2024; Lindsey et al., 2024). Both of these limitations become even more pronounced in models with a less clearly defined read/write stream, such as diffusion models, recurrent networks, adversarial models, and RL models (Pascanu, 2013; Goodfellow et al., 2014; Ho et al., 2020; Mnih et al., 2015).

1.2. Loss Landscape Geometry

An alternative idea is to decompose model's through their loss landscapes. Parameters are the fundamental objects updated during training, capturing information from data in a way that persists beyond a single inference step. Singular learning theory describes how the structure of high-dimensional parameter space influences generalization (Watanabe, 2007; 2000; 2005; Bushnaq et al., 2024; Davies et al., 2023), and developmental interpretability extends the work of SLT by characterizing phases of the training process. (Wang et al., 2024; Hoogland et al., 2024).

A key insight of SLT is that large models are highly degenerate (Wei et al., 2022) in the parameter space. Not only do modern architectures have inherent isotropies, but they are also degenerate with respect to the training data distribution. Models can have many different parameter configurations that result in minimum loss. In fact, gradient descent tends to converge on configurations that are at the intersection of many of these degenerate directions, called singularities. Our work takes this hypothesis one step further: if in relation to the loss on the full training distribution, models are highly singular then in relation to a subset of the training data, they are probably even more singular.

1.3. Paired Attribution Methods

The other phenomenon our method relies on is that in high-dimensional landscapes, local attribution methods seem to be good approximations of global relationships between pairs of samples. For example, attribution patching can successfully change the output of a model by targetting specific activations, determined by the first order gradients of paired outputs (Nanda, 2023; Kramár et al., 2024; Syed et al., 2023) Similarly, steering vectors, which are derived from differences in activations from paired samples, can effectively guide models toward specific behaviors even when applied beyond the original magnitude of activation differences (Turner et al., 2023; Subramani et al., 2022). Local and global attribution methods are good approximations of each other, especially when considering the space between paired outputs.

1.4. Loss Landscape Decomposition

Our goal in this paper is to identify directions in parameter space that correspond to computationally relevant subnetworks.

To our knowledge, there have only been two prior methods attempting to decompose parameter space for interpretability.An earlier work(Matena & Raffel, 2023), decomposes parameter space by computing principal directions of a per-sample Fisher Information matrix to resolve meaningful features. A recent method, Attribution Parameter Decomposition (APD) (Braun et al., 2025) used a bi-optimization approach to identify subnetworks where (1) the sum of subnetwork weights is close to the original model parameters, and (2) for any given input, the sum of the topk-attributed networks has low behavioral loss when compared to the original model.

Both of the approaches work with behavioral loss - seeking to understand circuits that when perturbed or ablated, would change a sample's overall predictions. Our method uses a slightly different approach, constraining the problem in such a way that allows us to work in first-order gradient space. When decomposing a model, we deciededly are only interested in circuits that move a model's predictions within the original subspace the outputs existed in. This is a more constrained problem than identifying circuits that merely change model output - we wish to identify to find circuits that change model outputs in the direction of another theoretical sample within the training distribution.

Our object of interest is therefore the gradient (with respect to the model parameters) of the divergence between a sample's output and a reference output, sampled from our training distribution. We seek to identify low-rank directions in parameter space such that for any pair of samples, a small number of these directions can be used to reconstruct their gradient of output divergence. We call our decomposition method Local Loss Landscape Decomposition (L3D). In this work, we first describe the mathematical foundation of our approach. We then develop progressively more complex toy models to measure the efficacy of L3D and quantify its limitations.

Local loss landscape decomposition (L3D) aims to decompose a model's gradients of loss between output pairs with respect to parameter space 1. We seek to identify directions in parameter space such that (1) the set of directions can approximately reconstruct any gradient of paired sample divergences, and (2) for any given pair of samples, an even smaller subset of directions can be used in reconstructing this gradient. To reduce computational complexity, we learn low-rank representations of these parameter directions. In this work, we first describe the mathematical foundation of our approach. We then develop progressively more complex toy models to measure the efficacy of L3D and quantify its limitations.

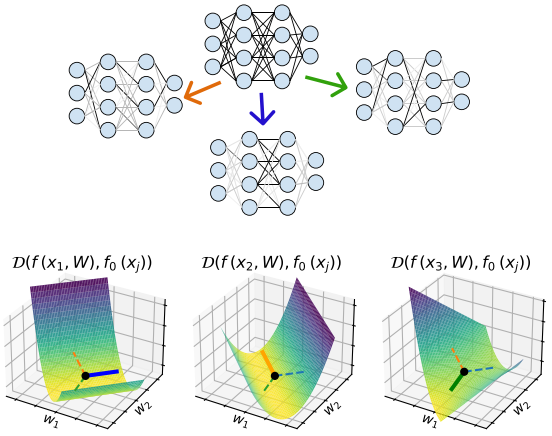


Figure 1: Decomposing a loss landscape into a set of parameter directions , where (1) the set of directions can approximately reconstruct any per-sample-pair gradient of divergence and (2) for any given sample, the curvature in most of the subnetworks directions is approximately zero.

2. Methodology

In the next sections, we will formally setup our decomposition problem (Sec. 2.1), define the criteria we will use for our subnetworks/parameter directions (Sec. 2.2), describe how to efficiently decompose parameters into these directions (Sec. 2.3), walk through our training algorithm (Sec. 2.4), and then explain how to use these decompositions to intervene on a model's behavior (Sec. 2.5).

The terms parameter directions, subnetworks, and circuits can all be used interchangeably, and will refer to the directions in parameter space that we wish to learn.

2.1. Set up

Consider a model f that takes an batch of inputs X and parameter values of W, and outputs a batch of outputs.

$$f(x, W) : R^{n_s \times n_f} \mapsto R^{n_s \times n_o} \tag{1}$$

where n_s is the number of samples in the batch, n_f is the length of each input vector, and n_o is the length of each output vector.

Our project rests on the following hypothesis:

1. For a given input, there are many components of a model that are not involved in inference. Changing

these parameters will not affect change the model's output.

2. For the same given input, there are a smaller set of components in a model that are highly involved in inference. Changing these parameters will change the model's output.

3. We are interested in parameter directions that meaningfully change a model's output. We'd like to identify directions that when perturbed, do not break the model's output by moving it completely outside of realm of realistic outputs, but rather turn decrease or enhance the power of a specific computational circuit.

2.2. Divergences of Paired Outputs

To put point (3) a different way, intervening in a certain parameter direction should not move our outputs too far away from a reference output distribution. Therefore, we will be decomposing gradients of divergences between pairs of outputs - with the aim of finding directions that move the model's output within the same subspace that the reference output distribution lies.

Divergence of a pair of outputs can be defined as:

$$\nabla_W D(f(x_i, W), f(x_r, W_0))|_{W=W_0} \quad \text{where } x_i, x_r \in X \tag{2}$$

Here, D is a divergence measuer, f is our model, x_i is our input of interest, x_r is a randomly selected reference input , W is a set of parameters and W_0 is model's parameters fixed. Our toy models, which are regression-type models, so we use MSE as divergence.

Wee will abbreviate $f(x_r, W_0)$ as $f(x_r)$ and the expression in Eq. 2 as $\nabla_W D$.

2.3. Sparse Principal Directions

We want to transform our gradient into directions in parameter space, where each sample's gradient can be written as a linear combination of a small set of these components. We will do this by learning transforms $V^{in} \in R^{n_v \times n_w}$ and $V^{out} \in R^{n_w \times n_v}$ where n_w is the number of parameters in the model, and n_v is the number of components (subnetworks) we wish to use to represent the parameter space.

V^{in} effectively transforms a gradient from the parameter space to the subnetwork space, so that:

$$\nabla_V D = V^{in} \nabla_W D \tag{3}$$

We want to find V^{in} and V_{out} such that for any given pair of samples a small subset of subnetworks can approximately reconstruct the gradient of divergence.

$$\nabla_W D \approx {V^{out}}_{:,\mathcal{K}} {V^{in}}_{\mathcal{K},:} \nabla_W D \quad \text{where } \mathcal{K} = \text{argTopFrac}_k \text{abs}(\nabla_{v_k} D) \tag{4}$$

argTopFrac relies on a hyperparameter that controls the number of components we wish to use to reconstruct each sample. In practice, we use a batchTopK (Bussmann et al., 2024) and a fraction rather than an absolute number. If we use a batchTopFrac of .1 it means that we select the top 10% of $\nabla_V D$ magnitudesover v_k and x_i to reconstruct our batch of gradients.

2.3.1. Low Rank parameter directions

Learning a set of full rank parameter parameter directions would be extremely expensive (we would be learning $n_w n_v$ values). We also expect that modular, sparsely active circuits would be lower rank that their full-model counterparts (). Therefore, we use low-rank representations of our V^{in} and V^{out}, and correspondingly learn low-rank circuits. We describe the procedure for doing this in B.1.

2.4. Training

We wish to learn the decomposition-related transforms V^{in} and V^{out} that minimize the topK reconstruction loss of our divergence gradient described above. We use a (normalized) L2 norm loss.

$$L = ||\nabla_W D - V^{out}_{:,\mathcal{K}} V^{in}_{\mathcal{K},:} \nabla_W D||_2 / \tag{5}$$

For each batch of samples, we randomly select a reference sample x_r to be paired with each sample x_i in the batch. We then compute the gradient of divergence of x_i and x_r at the model's current parameters W_0. We transform that gradient into the subnetwork space using V^{in}, and compute the topK components. We transform those components back into the original parameter space using V^{out}, and compute the loss between the reconstructed gradient and the original gradient. We apply a learning update to V^{in} and V^{out} with the goal of minizing this loss. We also normalize V^{out} to be a unit vector after each update.

2.5. Measuring and Intervening

Our learned subnetworks will just be the rows of V^{out}, reorganized into the same tensor structure as W. After identifying subnetworks, we may want to intervene on a specific circuit.

If we wish to intervene using a single subnetwork, we can update the parameters in by moving them in a scalar factor (δ) of that unit direction. To tune our model in the direction of subnetwork v_i and compute predictions on x, we evaluate:

Algorithm 1 Training Algorithm

for each epoch **do**
  for each X **do**
    for each $x_i \in X$ **do**
      Randomly select $x_r \in X$
      $\nabla_w D_i = \nabla_w D(f(x_i, W), f(x_r))|_{W=W_0}$
    end for
    $\nabla_v D = V^{in} \nabla_w$
    $\tau = \text{topK}(\text{abs}(\nabla_v D))$
    $\hat{\nabla}_w D = V^{out}(\nabla_v D \odot (\text{abs}(\nabla_v D) > \tau))$
    $L = ||\nabla_w D - \hat{\nabla}_w D||_2$
    L.backward()
    Update V^{in} and V^{out}
    Normalize V^{out} to be unit vectors.
  end for
end for

$$f(x, W + \delta v_i) \tag{6}$$

Going one step further, we may want to finetune our model, but constrain our finetuning to specific circuits for efficiency or performance reasons. While we do not use finetuning in this work, we describe what the process of finetuning would look like in Appendix B.2.

Additionally, we may want to identify which subnetworks were most relevant to the predictions of various samples, or identify which samples most relied on a given subnetwork. We describe this process in Appendix B.2.1

3. Results

Parameter space decomposition is extremely new, and there are not well-defined benchmarks by which to test our method. For this work, we focus on developing toy models that involve well-defined subnetworks, and evaluating L3D on decomposing these.

We designed several toy models to test the efficacy of L3D. Our toy models all consist of several well-characterized computations being performed by the same model, with an input space designed to rely on a single or small set of tasks. By designing our toy models this way, we have well characterized ground truth subnetworks by which to compare L3D's decompositions.

Our toy models progressively test more complex types of circuits. Table 1 describes our 4 toy models and the different attributes of circuitry the are designed to capture. The specific hyperparameters used to train our toy models is described in Appendix B.3, as well as the hyperparmaters used for each decomposition in Appendix B.4

3.1. Toy Model of Superposition

3.1.1. Setup

We start off by validating our algorithm on a well-studied toy problem, the toy model of superposition (TMS), with a small variation. TMS is an autoencoder with a single low-dimensional hidden layer in between and a relu activation at the output (Elhage et al., 2022). The model is trained on a dataset of samples where few features are active at a time, and "superimposes" these features in the hidden layer such that features embeddings in the hidden layer have minimal interference with each other S2. We train a toy model of superposition with 5 features and 2 hidden dimensions (with 1-sparsity=.05), and then place three such models in parallel, to test our method's ability to resolve superimposed features, as well as indpendent circuit modules.

3.1.2. Decomposition

We decompose the TMS model into 15 subnetworks, using rank-1 parameter tensors. The network decomposes into subnetworks as we would expect: each subnetwork represents the embedding and reconstruction of a single input element, involving only the weights connecting the relevant input and output, and the final bias. Fig. 2 shows the subnetworks corresponding to each of the first 5 features, and Fig. S3 shows the full decomposition. One thing to note is that parameter vectors do not have a preferred direction. L3D is equally likely to identify a parameter vector in the direction of θ as it is in the direction of $-\theta$. This is why, for example, the weights and biases in subnetwork 1 are in the opposite direction as the original network (Table 1).

This decomposition resulted in a reconstruction error of .19. 15 components of rank-1 can successfully express each individual $X_i : \hat{X}_i$ circuit, but does not capture the interference between features when multiple features are active in the same sample. We expect decompositions of higher dimensional networks to exhhibit less reconstruction error, as the amount of nearly orthogonal parameter vectors (non-interferring) that can be compressed into parameter space scales exponentially with dimension. We see this effect in later toy models.

3.1.3. Intervention

Parameter vectors learned by L3D can be used to intervene on model behavior. In principle, we could finetune a model with an adaptor that only changes the parameters of a network in the direction of a specified set of parameter vectors (See Appendix B.2). While we do not go the extent of finetuning a model, we explore the effect of perturbing a model's parameter space in the direction of a subnetwork (by an increment of δ). If subnetworks do in fact represent sparse computations, we hope that intervening on a

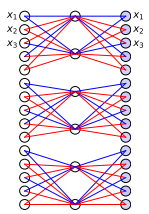
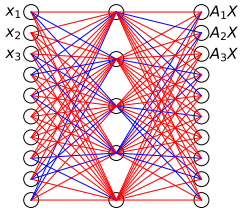
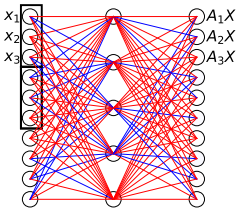
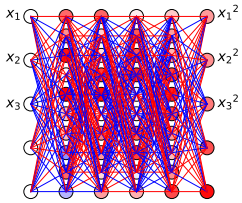
	Toy model of superposition	Circuit Superposition	Higher Rank Circuit Superposition	Complex Loss Landscape
				
	$X \mapsto X$	$X \mapsto AX$	$X \mapsto AX$	$X \mapsto X^2$
Feature Superposition	✓	✓	✓	✓
Circuit Superposition	×	✓	✓	✓
Circuits > rank-2	×	×	✓	probably ✓
Complex Loss Landscape	×	×	×	✓

Table 1: Toy models and their properties (transposed)

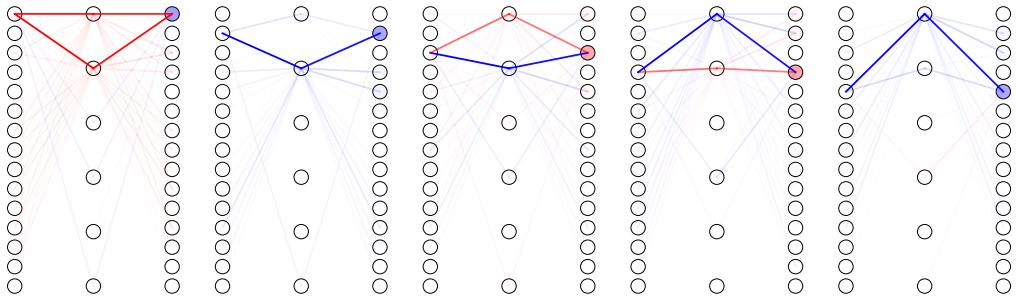


Figure 2: Selected 5 subnetworks that L3D decomposes the TMS-in-parallel model into

subnetwork has a strong effect on the predictions of relevant samples, and no effect on other samples. As shown in Fig. 3, moving the TMS-in-parallel model in the direction of a single subnetwork does in fact achieve just this. Perturbing in the direction of network 1 primarily affects samples where feature 1 was active, with a small effect on the inputs that had interference to feature 1's embeddings. In fact, for TMS-in-parallel, we can successfully fully "turn off" a computation by moving far enough in the direction of the subnetwork. (Although for models with more complex loss landscapes, turning "off" a computation is not as straightforward, as we will later discuss).

3.2. Toy Model of Circuit Superposition

3.2.1. Setup

TMS famously exhibits feature superposition - the input elements' low dimenional embeddings are non-orthogonal. However, the sparse circuits in TMS are noteably *not* in superposition - a given weight or parameter is only relevant for a single circuit and circuits. It seems highly unlikely that real world model circuits would decompose this way, since learning circuits composed of non-orthogonal parameter vectors would allow a model to put more expressivitiy into a compressed space. We therefore develop a toy model of *circuit superposition* (TMCS) in order to analyze L3D's ability to resolve such circuits. We define circuit superposition as a phenomenon by which subnetworks share parameter elements, and even more generally have non-orthogonal parameter vectors.

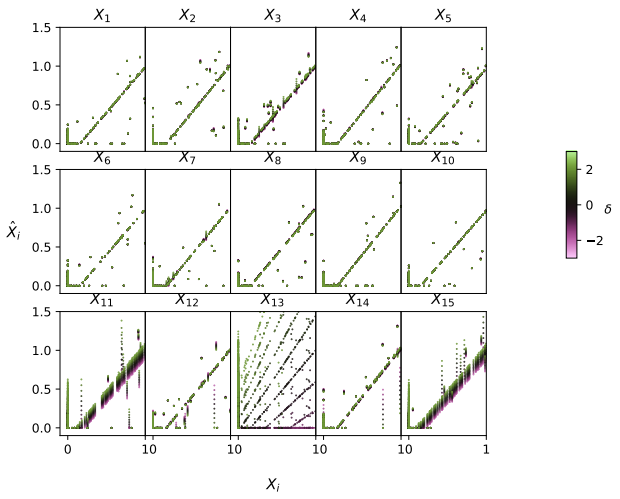


Figure 3: The effect of intervening on the TMS-in-parallel model in the direction of Network$_1$.

Our toy model of circuit superposition uses the same architecture and input data distribution as TMS, but is trained to predict linear combinations of the input features as its output ($X \mapsto AX$). We randomly select values of A between 0 and 3 to generate input-output pairs to train the toy model with. We use an input with 10 input features, and 10 output features (although such a model does not need to have the same number of input and outputs).

In this model, if we consider "subnetworks" to be parameter directions involved in inference of a small set of samples, then we would expect each circuit to be associated with a single input feature. If this is the case, then parameters would be involved in multiple circuits: ${W^{dec}}_{i,1}$ (the set of parameters connecting the hidden nodes to the first output node) will contain information about both $A_{1,1}, A_{2,1}, A_{3,1}$... Put another way, the circuits will interfere with each other - parameter directions associated with each circuit will be non-orthogonal.

3.2.2. Decomposition

We decompose TMCS into 10 subnetworks of rank-1 parameter tensors (4) with a reconstruction loss of .064. The subnetworks look like we expect, with each strongly corresponding to a single input feature.

Since each subnetwork theoretically corresponds to the computations involved with a single input feature, we should be able to reconstruct the original A values from each subnetwork. To derive A from each subnetwork, we (1) identify the which column in the subnetwork's W^{dec} direction has the largest norm and then (2) trace the weights of the network through that path. That is for subnetwork k:

$$j^* = argmax_j ||{W^{dec}}_{j_k}||_2 \tag{7}$$
$$\hat{a}_{i,j} = {W^{enc}}_{i,j_k} {W^{dec}}_{i,j_k}$$

The parameter vectors are normalized to be unit vectors so we expect them to be a scalar multiple of the true A values. As seen in Figure 4, our derived $\hat{\alpha}$ have a very high correlation to the original A values ($r^2 = 0.92$).

3.3. Higher Rank Circuits

3.3.1. Setup

Because each subnetwork in TMCS traces the path of a single input neuron, the underlying subnetworks should inherently have a rank of 1. In order to test the ability of L3D to learn higher rank circuits, we developed a toy model with inherently higher rank circuits. For this model, we use the same set up as TMCS, but we correlate the sparsities of sets of input features. We use 25 input features, and we filter our data to ensure that input features 1-5, 6-10, etc, are always active (> 0) or inactive (< 0) together. In this setup, circuits should always be associated with groups of 5 input features and so should have a rank of 5.

3.3.2. Decomposition

Although we expect the model to have 5 subcircuits, we use excess parameter tensors (n=10) in order to allow more flexibility in learning. We track the fraction of inputs for which a subnetwork was used in the topK reconstruction (P_{act}) to identify which were "dead circuits", and report the last epoch. Futhermore, although we expect the underlying subcircuits to be rank 5, we experiment with using different rank representations to see how well lower-rank parameter directions can represent the model. Interestingly, rank-1 representations of the parameter tensors are able to represent the model nearly as well as rank-5 representations (Fig. S5). In Fig. 6, we show the decomposition of a rank-3 decomposition. L3D successfully learns a subnetwork corresponding to each of the 5 sets of input features, as well as a number of "dead". The higher and lower rank decompositions also learn similar subnetworks (Figure S6). When we trained L3D without these additional subnetworks, the reconstruction loss would get caught in a local minima. Similar to training SAEs (Cunningham et al., 2023), having extra degrees of freedom allows for better learning, even if at the end of training the extra subnetworks are never active.

3.4. Complex Loss Landscape

3.4.1. Setup

The previous set of toy models all had relatively quadratic landscapes with respect to each element ofW^{enc} or W^{dec} -

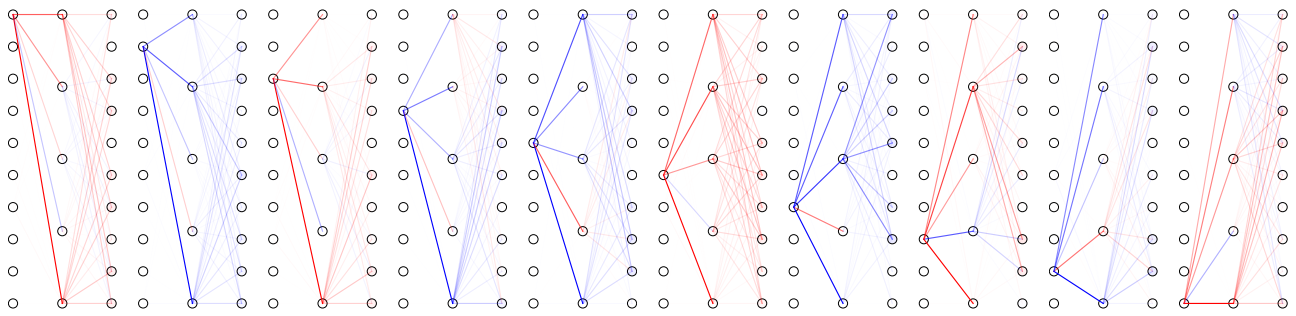


Figure 4: The learned subnetworks of TMCS

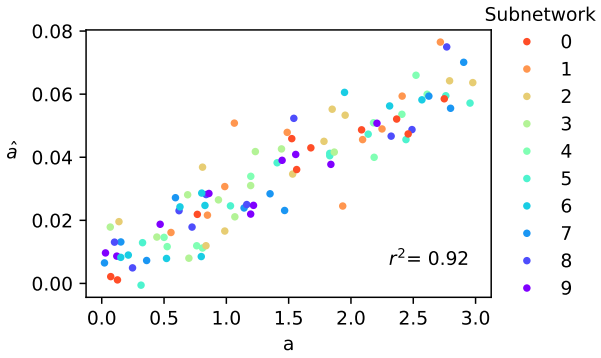


Figure 5: The coefficients derived from the subnetworks compared to actual coefficients

$dMSE/dW_i$ depends on W_i to the first order (or not at all, depending on whether the ReLU has fired). This suggests that the local loss landscape of the models is a good approximation for much of the global loss landscape, and we should expect intervening on a circuit to have well-behaved effects, even at large values of δ (as in Fig. 3).

However, we wanted to test the limitations of a L3D on a model with a more complex loss landscape, especially when it comes to intervening with a subnetwork.

We therefore trained a multi-layer model to predict multiple non-linear functions of input features at once. We train a GeLU network for $X_i \mapsto X_i^2$. We use a network with 4 hidden layers of 5 neurons each, and 5 input and output neurons. Once again, the input features are sparse, incentivizing the toy model to learn circuits in superposition whose interferences will cause minimal errors on the sparse input distribution.

We expect the model to have 5 subnetworks, one for each input feature. Although it is less clear what rank the tensors of the underlying circuits should be, we expect that with a non-linear model they will likely be high rank.

3.4.2. Decomposition

We decompose our model into 5 rank-2 parameter tensors in our decomposition and instead of varying rank, we experiment with using different numbers of subnetworks to represent our model. In the 5-subnetwork decomposition (Figure 7), we see that subnetworks tracing the path of $X_i \mapsto X_i^2$ for each input feature i. However, this decomposition has a relatively high reconstruction error of .32. Much of this is probably because we kept our topFrac constant throughout all our our models, using a topFrac of .1 for consistency. With only 5 subnetworks, this means that on average each sample's reconstruction will use ¡1 subnetwork - limiting the reconstruction error we can achieve.

We also experiment with holding rank constant (we drop to rank-1) and decompose the model into different numbers of subnetworks (3,5,10, and 15 subnetworks). In our 3-subnetwork decomposition, L3D still learned subnetworks corresponding to single input features, but can of course only represent 3 out of the 5 inputs. As we add more subnetworks, we are able to successfully learn more expressive decompositions of the model that reduce reconstruction error (Figure S11). Each decomposition continues to learn input-output pair specific subnetworks, with the larger decompositions resulting in a few more dead subnetworks as well (Fig. S10).

3.4.3. Intervention

Intervening on these circuits helps us understand how much local loss landscape is representative of global loss landscape, particularly when it comes to inactive subnetworks remaining inactive as we move through parameter space. If local loss landscape is truly representative of global loss landscape in this way, then intervening on on a single subnetwork should result in only a consistent small number of samples being affected, even if we move very far in that direction. Fig. 8 shows our results for these interventions. Even in this more complex toy model, local loss landscape is a relatively good approximation of the global loss landscape.

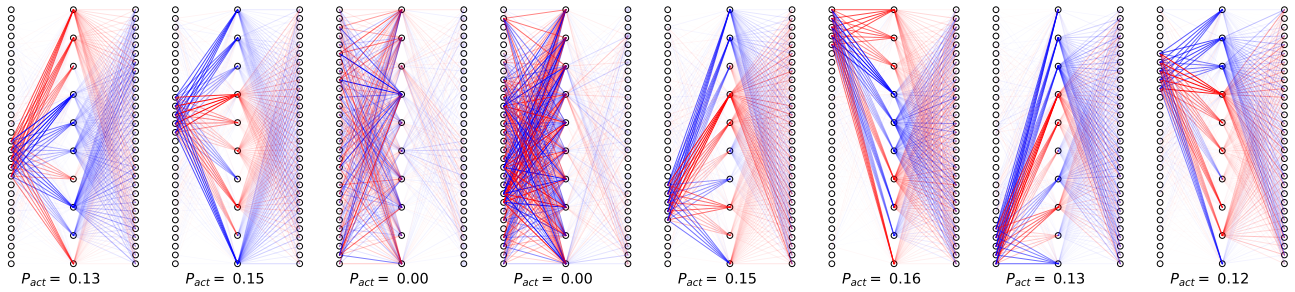


Figure 6: Parameter representations learned by L3D for the high rank circuit decomposition task.

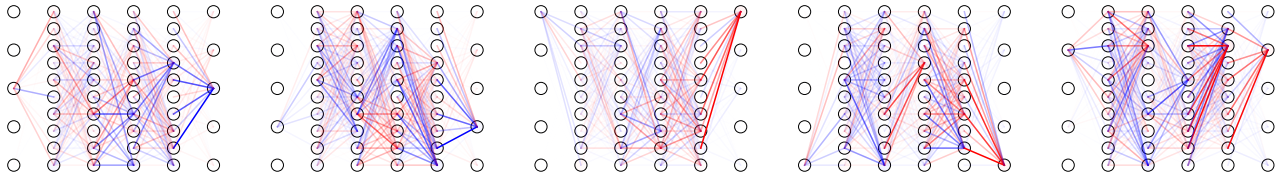


Figure 7: Subnetworks learned by L3D for the $X \mapsto X^2$ model, and effect of intervening on each subnetwork

We can perturb the parameters in a direction of interest and have a large impact on the predictions of that sample and a minor impact on others. If we perturb far enough (Fig. S7), we do begin to see effects on the predictions of other samples, but ratio of change in predictions to the relevant samples to those of the irrelevant samples is very high.

A careful reader may have noticed is that it would be very easy to identify parameter directions in the first or last layer of the $X \mapsto X^2$ model involved in a sparsely active circuit. The circuit for $X_i \mapsto X_i^2$ would just involve all weights connecting input node i to the first hidden layer, and all weights connecting the hidden node i to the output node i, as well as the bias of output node i. In order to make sure L3D is not just learning this trivial solution, we want to test if the hidden parameter directions it learns are also relevant circuits. Therefore, we perform the same intervention experiments but only interveining with the subnetwork components related to the model's hidden layers' weights and biases (S8). The results are very similar, where inventions on a parameter direction only affect the output of a subset of samples - showing that the circuits that L3D has learned perform relevant computation even in their middle layers.

8 shows changes in predictions as we move in a single direction in parameter space. We also wanted to undrestand how subcircuits might interact with each other as we move through parameter space. In S9 we perturb multiple subnetworks at once, and measure the new predictions. For the most part, the subnetworks have little inference with each other: the relevant output values for each subnetwork move relatively independently of each other. For some parameter directions, we do see some unexpected interactions, such as when we move in the direction of subnetwork 1 and 2, we see a small effect on the predictions of a few samples not associated with either subnetwork. With larger models, especially those with more modular circuits that are working together, we expect this kind of interaction to become more common. We discuss more in the discussion section.

4. Discussion

4.1. Simple Improvements

Hyperparameter Choice

Flexible Rank Choices

4.2. Challenges

Interpretation of a circuit

Local Loss Landscape

Relationship to overparameterized models

4.3. Extensions

Finetuning

Identifying Specific Circuits with Constrastive Pairs

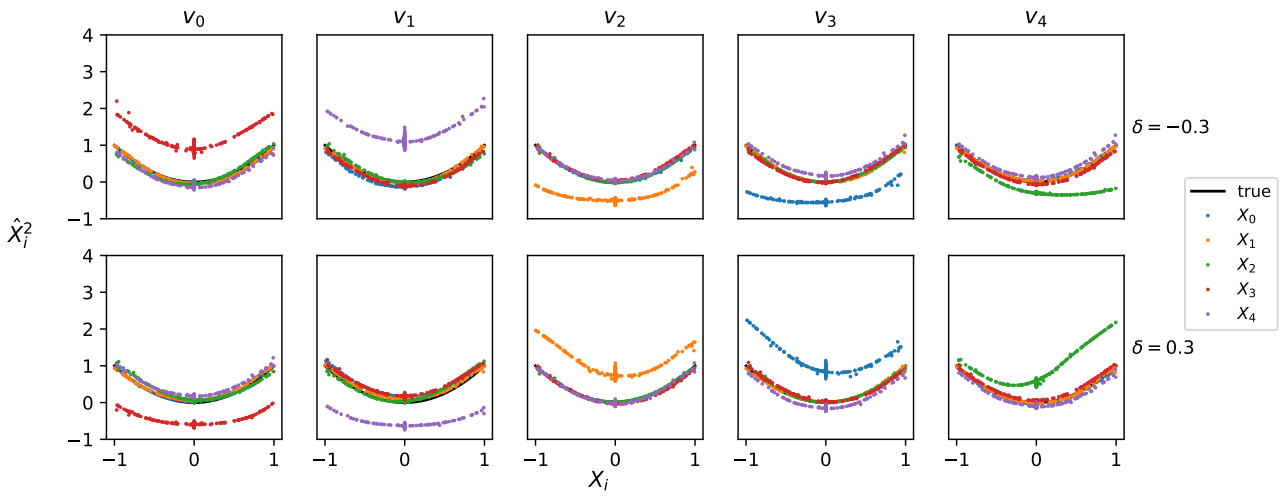


Figure 8: The effect of intervening on each subnetwork

5. Impact Statement

References

Bereska, L. and Gavves, E. Mechanistic interpretability for ai safety–a review. *arXiv preprint arXiv:2404.14082*, 2024.

Braun, D., Bushnaq, L., Heimersheim, S., Mendel, J., and Sharkey, L. Interpretability in parameter space: Minimizing mechanistic description length with attribution-based parameter decomposition. *arXiv preprint arXiv:2501.14926*, 2025.

Bricken, T., Templeton, A., Batson, J., Chen, B., Jermyn, A., Conerly, T., Turner, N., Tamkin, A., and Carter, S. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, 2023. Accessed: 2025-02-17.

Bushnaq, L., Mendel, J., Heimersheim, S., Braun, D., Goldowsky-Dill, N., Hänni, K., Wu, C., and Hobbhahn, M. Using degeneracy in the loss landscape for mechanistic interpretability. *arXiv preprint arXiv:2405.10927*, 2024.

Bussmann, B., Leask, P., and Nanda, N. Batchtopk sparse autoencoders. *arXiv preprint arXiv:2412.06410*, 2024.

Cunningham, H., Ewart, A., Riggs, L., Huben, R., and Sharkey, L. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

Davies, X., Langosco, L., and Krueger, D. Unifying grokking and double descent. *arXiv preprint arXiv:2303.06173*, 2023.

Elhage, N., Hume, T., Olsson, C., Schiefer, N., Henighan, T., Kravec, S., Hatfield-Dodds, Z., Lasenby, R., Drain, D., Chen, C., et al. Toy models of superposition. *arXiv preprint arXiv:2209.10652*, 2022.

Engels, J., Michaud, E. J., Liao, I., Gurnee, W., and Tegmark, M. Not all language model features are linear. *arXiv preprint arXiv:2405.14860*, 2024a.

Engels, J., Riggs, L., and Tegmark, M. Decomposing the dark matter of sparse autoencoders. *arXiv preprint arXiv:2410.14670*, 2024b.

Gao, L., la Tour, T. D., Tillman, H., Goh, G., Troll, R., Radford, A., Sutskever, I., Leike, J., and Wu, J. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2406.04093*, 2024.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Hoogland, J., Wang, G., Farrugia-Roberts, M., Carroll, L., Wei, S., and Murfet, D. The developmental landscape of in-context learning. *arXiv preprint arXiv:2402.02364*, 2024.

Kramár, J., Lieberum, T., Shah, R., and Nanda, N. Atp*: An efficient and scalable method for localizing llm behaviour to components. *arXiv preprint arXiv:2403.00745*, 2024.

Lindsey, J., Templeton, A., Marcus, J., Conerly, T., Batson, J., and Olah, C. Sparse crosscoders for cross-layer features and model diffing. *Transformer Circuits Thread*, 2024.

Matena, M. and Raffel, C. Npeff: Non-negative per-example fisher factorization. *arXiv preprint arXiv:2310.04649*, 2023.

Merullo, J., Eickhoff, C., and Pavlick, E. Talking heads: Understanding inter-layer communication in transformer language models. *arXiv preprint arXiv:2406.09519*, 2024.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Nanda, N. Attribution patching: Activation patching at industrial scale. *URL: https://www. neelnanda. io/mechanistic-interpretability/attribution-patching*, 2023.

Pascanu, R. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2013.

Subramani, N., Suresh, N., and Peters, M. E. Extracting latent steering vectors from pretrained language models. *arXiv preprint arXiv:2205.05124*, 2022.

Syed, A., Rager, C., and Conmy, A. Attribution patching outperforms automated circuit discovery. *arXiv preprint arXiv:2310.10348*, 2023.

Turner, A. M., Thiergart, L., Leech, G., Udell, D., Vazquez, J. J., Mini, U., and MacDiarmid, M. Steering language models with activation engineering. *arXiv preprint arXiv:2308.10248*, 2023.

Wang, G., Farrugia-Roberts, M., Hoogland, J., Carroll, L., Wei, S., and Murfet, D. Loss landscape geometry reveals stagewise development of transformers. In *High-dimensional Learning Dynamics 2024: The Emergence of Structure and Reasoning*, 2024.

Watanabe, S. Algebraic information geometry for learning machines with singularities. *Advances in neural information processing systems*, 13, 2000.

Watanabe, S. Algebraic geometry of singular learning machines and symmetry of generalization and training errors. *Neurocomputing*, 67:198–213, 2005.

Watanabe, S. Almost all learning machines are singular. In *2007 IEEE Symposium on Foundations of Computational Intelligence*, pp. 383–388. IEEE, 2007.

Wei, S., Murfet, D., Gong, M., Li, H., Gell-Redman, J., and Quella, T. Deep learning is singular, and that's good. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):10473–10486, 2022.

A. Definitions

A.0.1. DIMENSIONS

n_s: The number of samples in a batch of inputs

n_f: The dimensions of a single input vector to a model

n_o: The dimensions of a single output vector from a model

n_w: The number of parameters values in a model.

n_v: The number of subnetworks or parameter directions chosen to decompose a model.

A.0.2. MODEL SYNTAX

$X \in \mathbb{R}^{n_s \times n_f}, x \in \mathbb{R}^{n_f}$: Batch and individual input vectors to a model.

$W \in \mathbb{R}^{n_w}, w \in \mathbb{R}$: The set of and individual parameter values of a model

$\sqsupseteq$: The set of parameters corresponding to a specific tensor or block in a model.

$f : \mathbb{R}^{n_s \times n_f} \mapsto \mathbb{R}^{n_s \times n_o}$: A model mapping a set of input vectors to a set of output vectors.

$f(X, W)$: The output of model f with parameter values W on input X.

$f(X, W_0)$ or $f(X)$: The output of model f with fixed parameter values W_0. W_0 is the set of learned parameter values from model training.

D: Divergence metric between two vectors. Typical divergence metrics are mean-squared error for regression-type outputs, and KL-divergence for probability-type outputs.

A.0.3. DECOMPOSITION SYNTAX

$V(or V^{out}) \in \mathbb{R}^{n_v \times n_w}, v(or V^{out}) \in \mathbb{R}^{n_w}$: The set of or individual parameter directions that are used to decompose a model. V_out can be used to transform parameter directions in the subnetwork vector space back into the original parameter space of the model. The terms parameter direction, subnetwork, and circuit all have the same meaning.

$\mathcal{V}_{out}, \sqsubseteq_{out}$: Components of V^{out} or V^{out} related to a specific tensor or block in the original model.

$V^{in} \in \mathbb{R}^{n_w}, V^{in} \in \mathbb{R}$: Transforms the original parameter space of the model into the subnetwork vector space.

$\mathcal{V}_{in}, \sqsubseteq_{in}$: Components of V^{in} or V^{in} related to a specific tensor or block in the original model.

r: The rank of each component of the decomposition vectors corresponding to tensors in the original model.

A.0.4. TRAINING

$\mathcal{K}$: The subset of indexes ¿= τ where τ is the threshold computed by using the top k absolute values in a tensor.

L: The L2 reconstruction loss used to optimize V^{in} and V^{out}.

A.0.5. MEASURING AND INTERVENTION

$I(x_i, x_j, v_k), I(x_i, v_k)$: The impact of subnetwork v_k on the divergence between samples x_i and x_j, or averaged across many x_j reference samples.

δ: A scalar value to move W in a specific direction.

B. Additional Methods

B.1. Low-Rank Tensor Representation

We use low-rank representations of our V^{in} and V^{out}, and correspondingly learn low-rank circuits.

While W is a vector of all of the parameters in a model, typically model parameters are organized into tensors $W = \{w_i\}_i$.

If our parameters are organized into tensors $W = \{w_i\}_i$, each subnetwork or parameter component can be organized as $V_i^{in} = \{v^{in}{}_i\}_i, V_i^{out} = \{v^{out}{}_i\}_i$ where we have parts of our subnetworks corresponding to each tensor in the original model parameters. We wish each of these tensors to be low rank, and we express them using the canonical polyadic decomposition () (a way to write 3+ dimentinoal tensors in terms of low-rank components).

$$v^{in}{}_{i,j} = \sum_{r=1}^{R} a_{i,j,r} \times b_{i,j,r} \times c_{i,j,r} ... \tag{8}$$

Where R is the rank we wish to use to represent the parameter component, and the number of factors (a, b, c,...) in the factorization is equal to the number of dimensions in the tensor w_i.

B.2. Finetuning

To finetune a model on a specific set of parameter directions, we would freeze the current set of weights and learn an adaptor consisting of linear combinations of the subnetworks of choice. During fine tuning, we would only need to learn the coefficients of these linear combinations greatly reducing cost.

$$f(X, W_0 + W_{ft} V_{\mathcal{K}}^{out}) \tag{9}$$

B.2.1. Quantifying Impact

We can quantify the impact of a subnetwork in two ways. First, we can compute the impact of a subnetwork on a pair of samples x_i, x_j, identyfing the subnetwork that, if intervened upon, would most strongly impact the the divergence of x_i's outputs when compared to x_j. The impact of subnetwork v_k on such a pair of samples, $I(x_i, x_j, v_k)$ can be measured by:

$$I(x_i, x_j, v_k) = \left| V_{k,:}^{in} \nabla_w D(f(x_i, W), f(x_j)) \right| \tag{10}$$

Alternatively, we can average the impacts of a subnetwork v_k and an input x_i over many different reference samples to better quantify the impact of the subnetwork on a single sample's predictions overall. Although more computationally expensive, this can give a more robust measurement for the impact of a subnetwork on a specific sample.

$$I(x_i, v_k) = \frac{1}{n_j} \sum_{j=1}^{n_j} I(x_i, x_j, v_k) \tag{11}$$

B.3. Toy Model Training

For all of our toy models (except the $X \mapsto X^2$ model), we generate uniformly random inputs between 0 and 1. For $X \mapsto X^2$, we generate uniformly random inputs between -1 and 1. For all toy model data, we use a sparsity value of 1-sparsity=.05. We generate 10000 datapoints and train for 1000 epochs with batch sizes of 32. We use an AdamW optimizer with a learning rate of 0.001.

B.4. L3D Model Training

To train L3D, we use the same training distributions as in each toy models. Although optimal hyperparameter values probably depend on the model size, and the rank and number of parameter tensors, we use the same hyperparameters for all of our models. We generate only 1000 datapoints, with a batch size of 32, and train for 1000 epochs. We use an AdamW optimizer with a learning rate of 0.01, and a learning decay rate of .8 every 100 steps. We always use a topFrac value of .1. We include all of the model's parameter tensors, including biases, in the decomposition.

C. Supplemental Figures

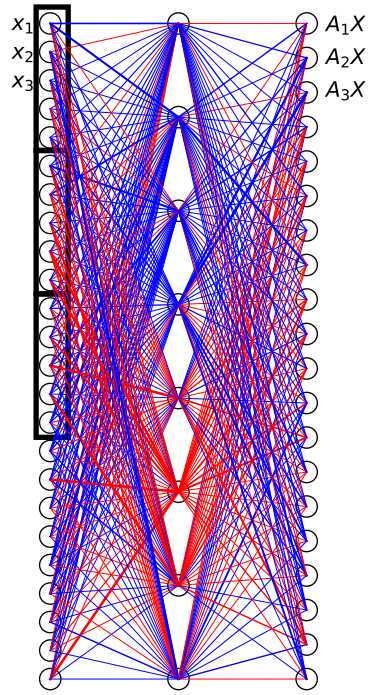


Figure S1: Full architecture of high rank circuit toy model.

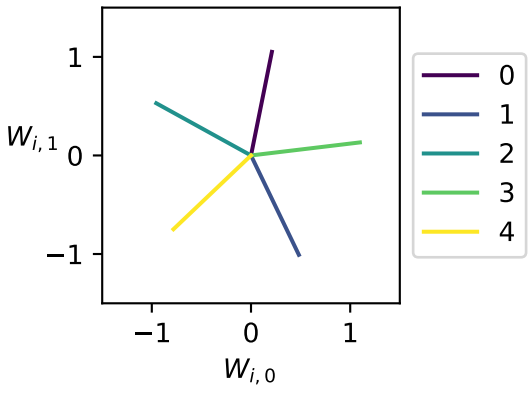


Figure S2: Encoder directions in TMS

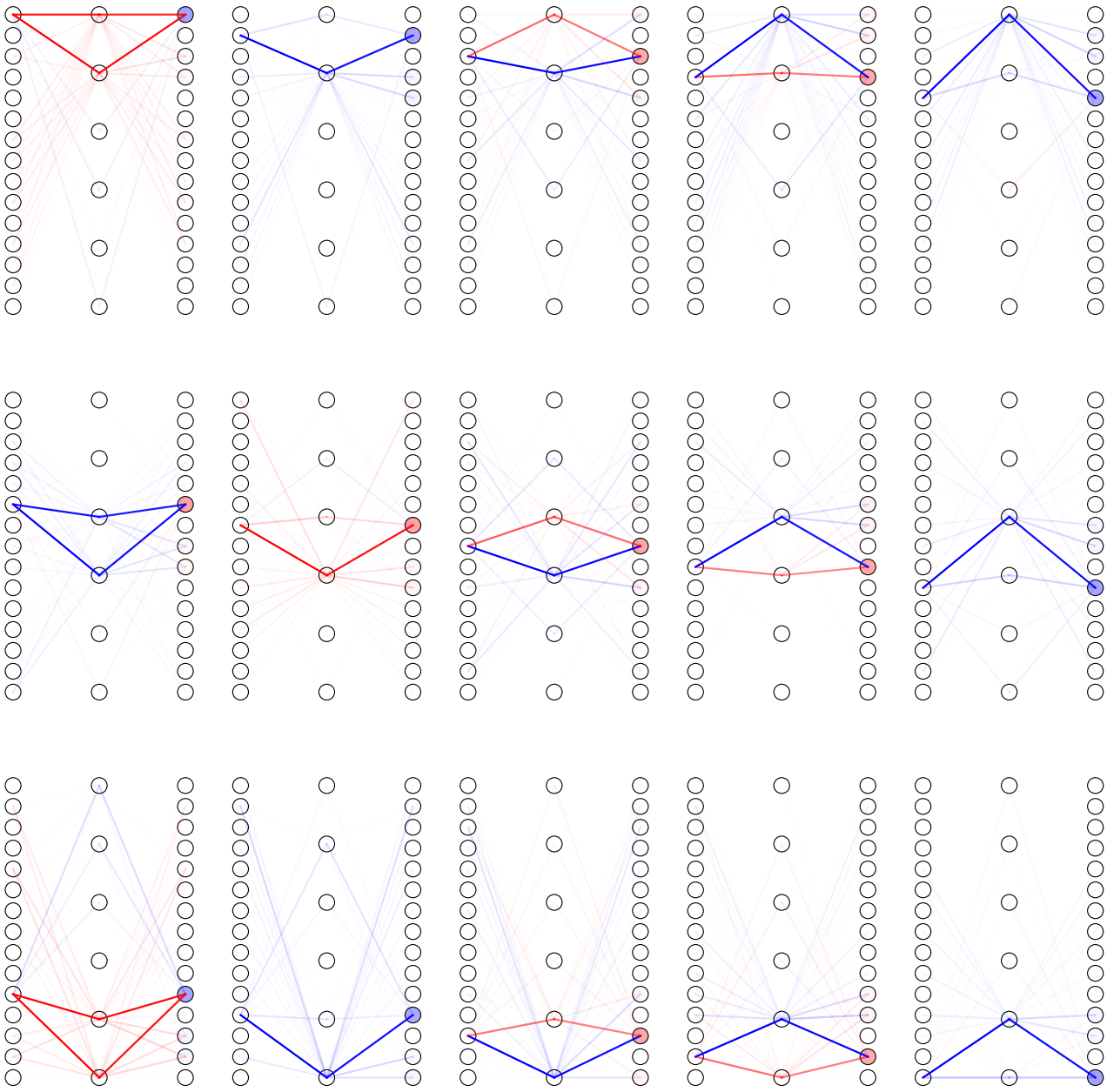
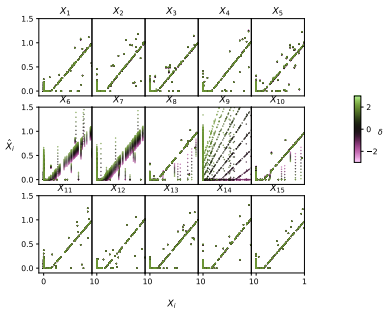
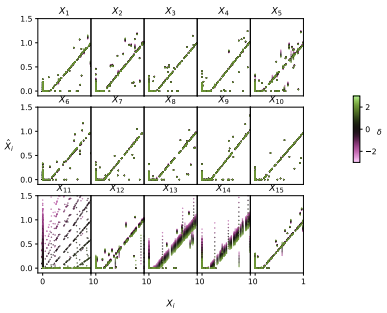


Figure S3: All subnetworks for the TMS decomposition

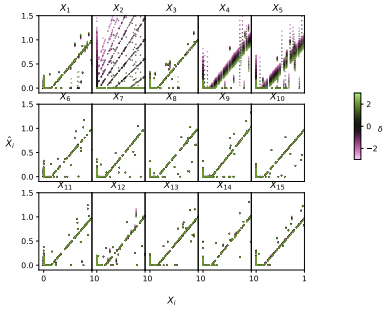
Figure S4: Effect of intervening on the first 5 subnetworks of the TMS-in-parallel model.



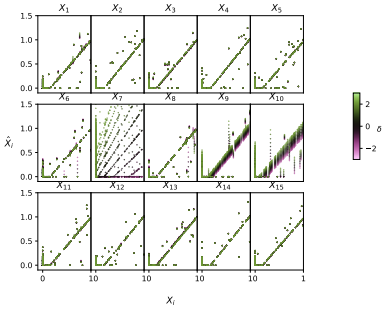
(a) Subnetwork 1



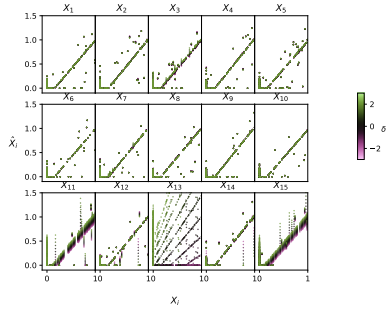
(b) Subnetwork 2



(c) Subnetwork 3



(d) Subnetwork 4



(e) Subnetwork 5

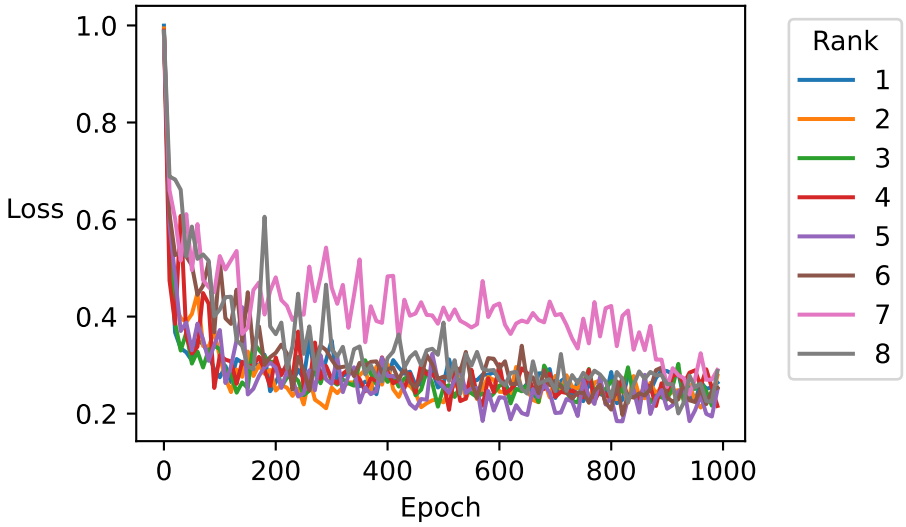
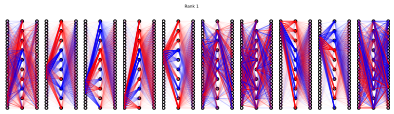
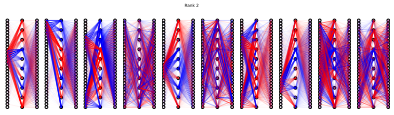


Figure S5: Loss vs Rank

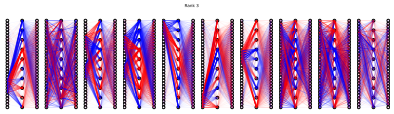
Figure S6: Decomposing the toy model of high rank circuits into different numbers of subnetworks



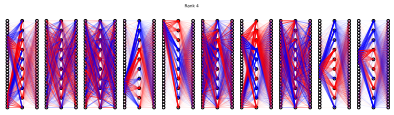
(a) Rank-1 Networks



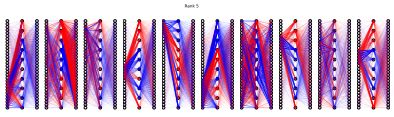
(b) Rank-2 Networks



(c) Rank-3 Networks



(d) Rank-4 Networks



(e) Rank-5 Networks

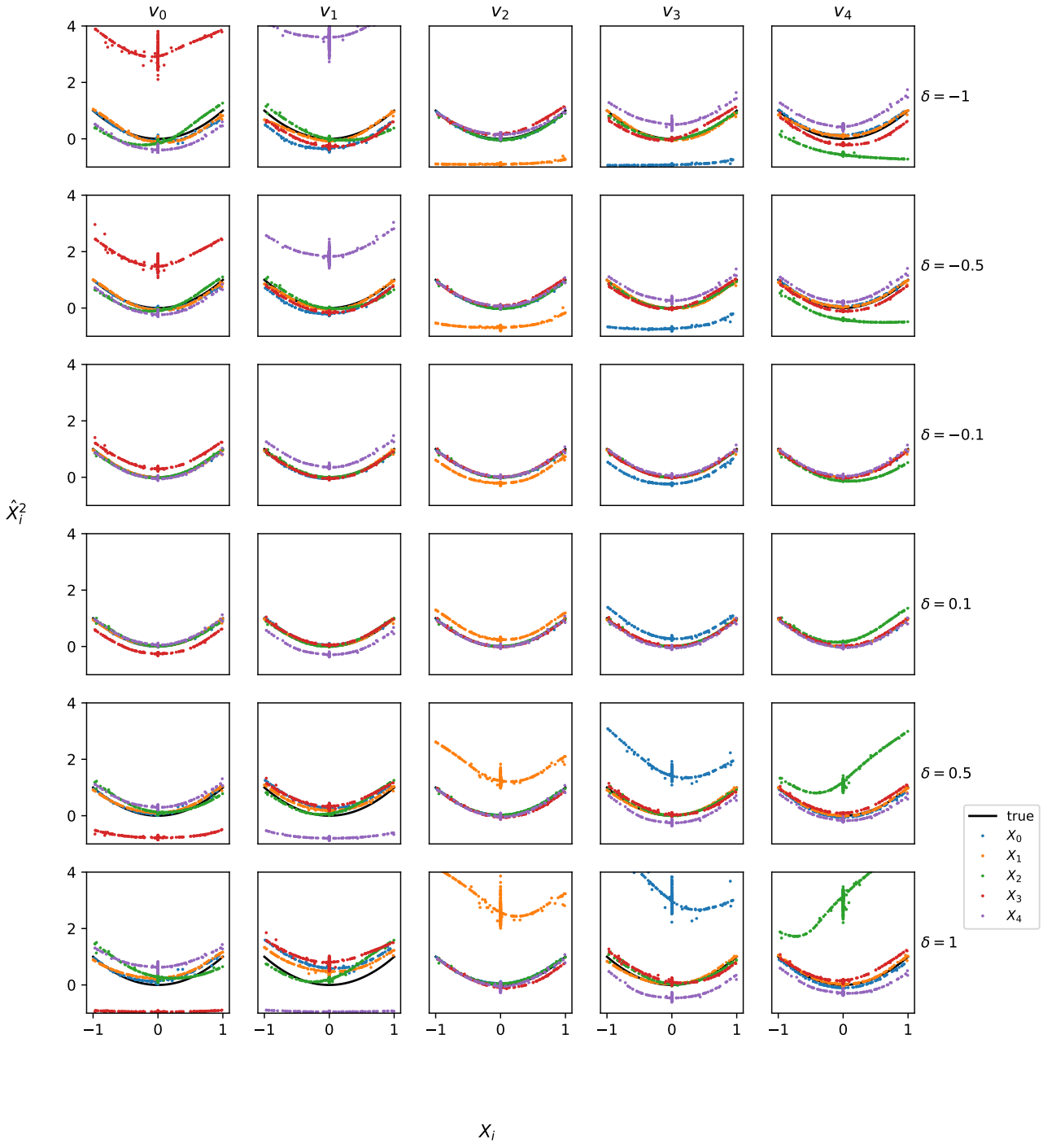


Figure S7: Effects of intervening on each of the 5 subnetworks of the $X \mapsto X^2$ model.

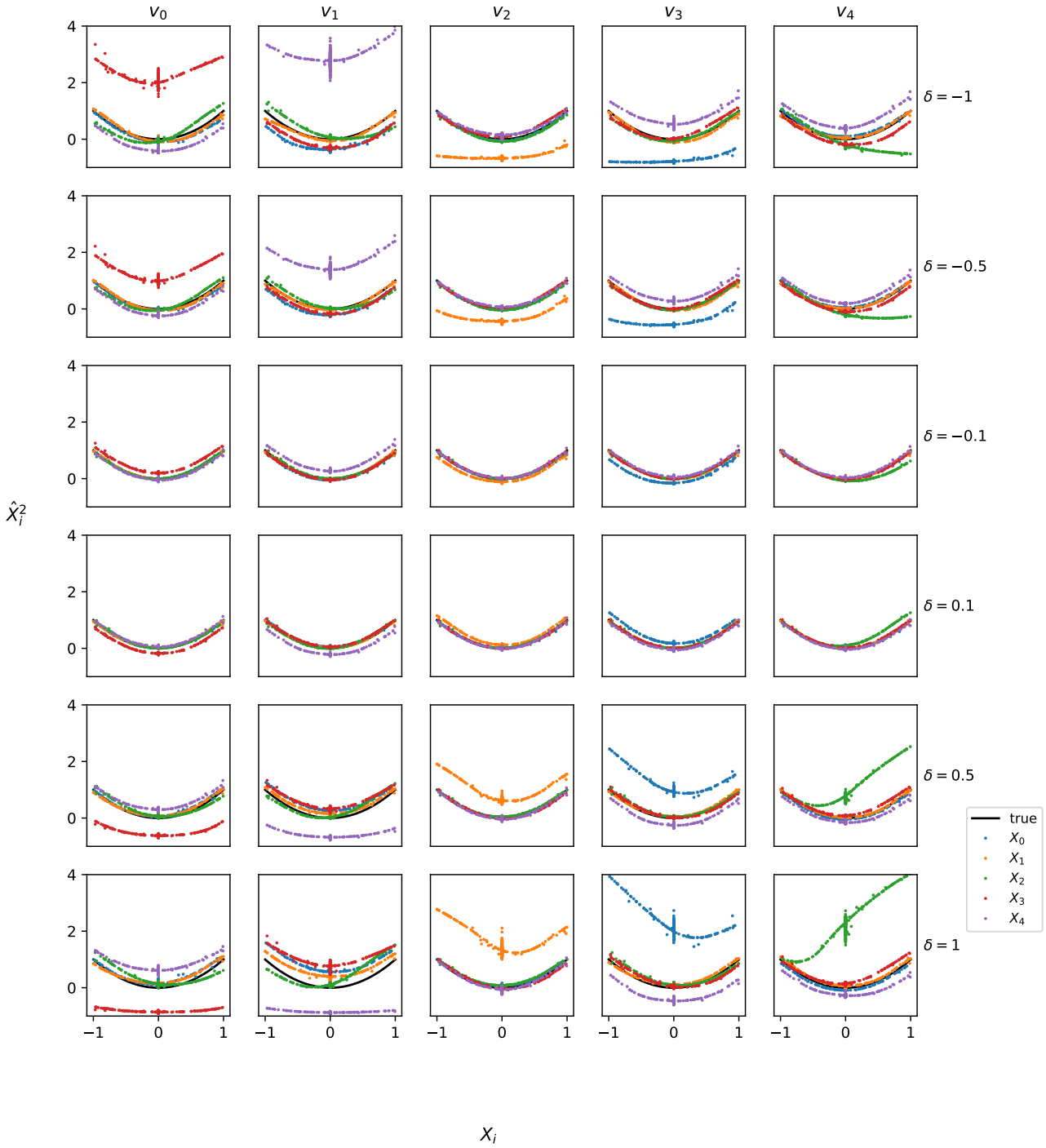


Figure S8: Intervening on just the weights and biases of the middle hidden layers in the $X \mapsto X^2$ network.

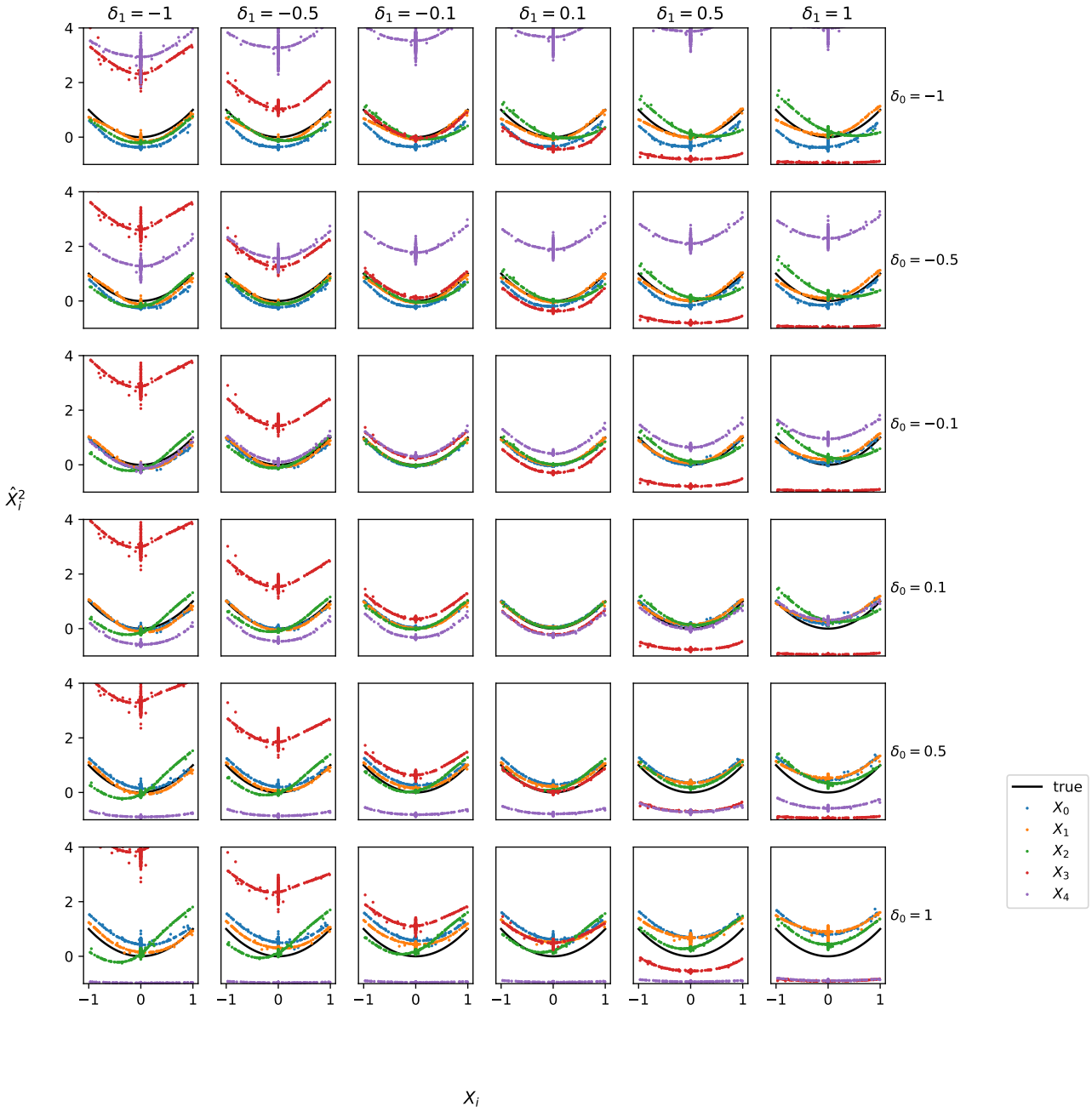
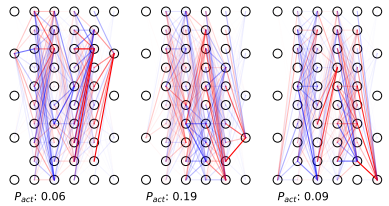
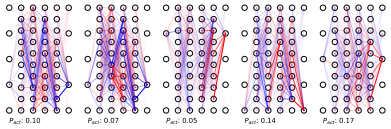


Figure S9: Intervening on multiple subnetworks at once.

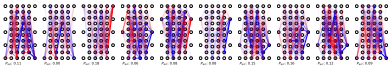
Figure S10: Decomposing the $X \mapsto X^2$ model into different numbers of subnetworks



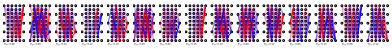
(a) 3 rank-1 Networks



(b) 5 rank-1 Networks



(c) 10 rank-1 Networks



(d) 15 rank-1 Networks

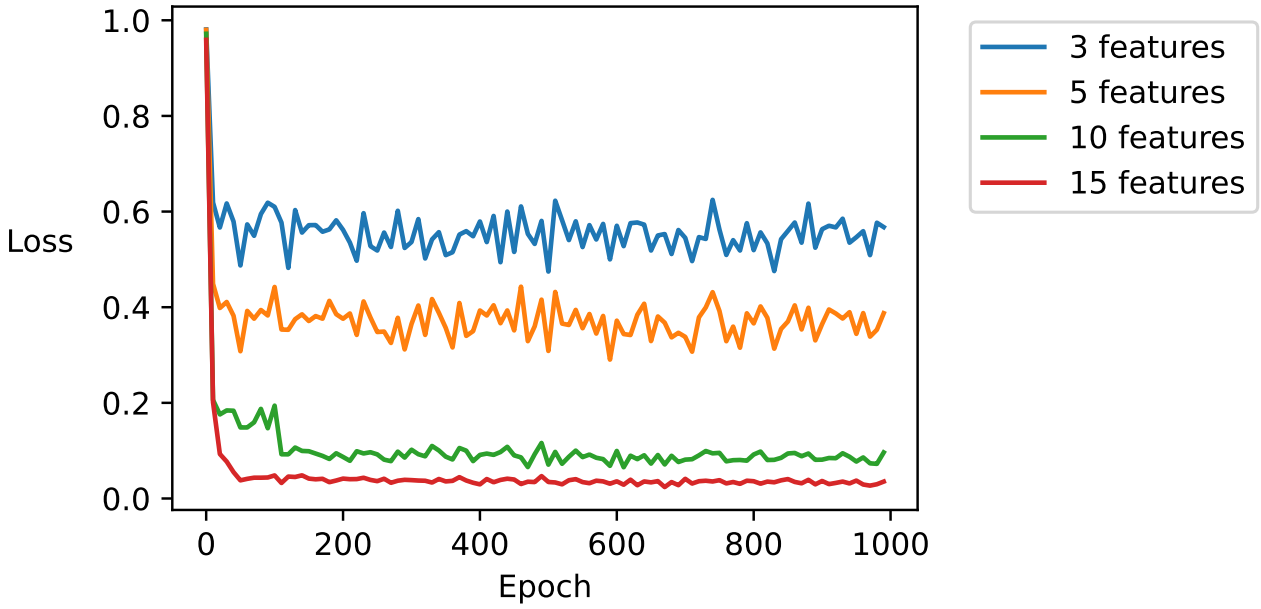


Figure S11: Training loss vs number of features for the $X \mapsto X^2$ model, and decompositions for a 3- and 5- subnetwork decomposition.