The most important thing to pay attention to is how ***there's a shift from monosemantic to polysemantic neurons as sparsity increases***. Monosemantic neurons do exist in some regimes! Polysemantic neurons exist in others. And they can both exist in the same model! Moreover, while it's not quite clear how to formalize this, it looks a great deal like there's a neuron-level phase change, mirroring the feature phase changes we saw earlier.

It's also interesting to examine the structure of the polysemantic solutions, which turn out to be surprisingly structured and neuron-aligned. Features typically correspond to *sets of neurons* (monosemantic neurons might be seen as the special case where features only correspond to singleton sets). There's also structure in how polysemantic neurons are. They transition from monosemantic, to only representing a few features, to gradually representing more. However, it's unclear how much of this is generalizable to real models.

LIMITATIONS OF THE RELU HIDDEN LAYER TOY MODEL SIMULATING IDENTITY

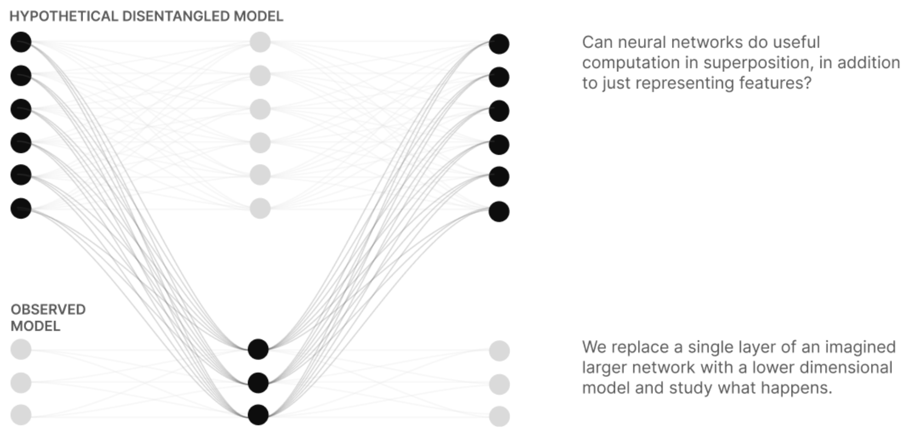
Unfortunately, the toy model described in this section has a significant weakness, which limits the regimes in which it shows interesting results. The issue is that the model doesn't benefit from the ReLU hidden layer – it has no role except limiting how the model can encode information. If given any chance, the model will circumvent it. For example, given a hidden layer bias, the model will set all the biases to be positive, shifting the neurons into a positive regime where they behave linearly. If one removes the bias, but gives the model enough features, it will simulate a bias by averaging over many features. The model will only use the ReLU activation function if absolutely forced, which is a significant mark against studying this toy model.

We'll introduce a model without this issue in the next section, but wanted to study this model as a simpler case study.

Computation in Superposition

So far, we've shown that neural networks can store sparse features in superposition and then recover them. But we actually believe superposition is more powerful than this – we think that neural networks can *perform computation entirely in superposition* rather than just using it as storage. This model will also give us a more principled way to study a *privileged basis* where features align with basis dimensions.

To explore this, we consider a new setup where we imagine our input and output layer to be the layers of our hypothetical disentangled model, but have our hidden layer be a smaller layer we're imagining to be the observed model which might use superposition. We'll then try to compute a simple non-linear function and explore whether it can use superposition to do this. Since the model will have (and need to use) the hidden layer non-linearity, we'll also see features align with a privileged basis.


HYPOTHETICAL DISENTANGLED MODEL
Can neural networks do useful computation in superposition, in addition to just representing features?
OBSERVED MODEL
We replace a single layer of an imagined larger network with a lower dimensional model and study what happens.

Specifically, we'll have the model compute $y = \mathrm{abs}(x)$. Absolute value is an appealing function to study because there's a very simple way to compute it with ReLU neurons: $\mathrm{abs}(x) = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$. This simple structure will make it easy for us to study the geometry of how the hidden layer is leveraged to do computation.

Since this model *needs* ReLU to compute absolute value, it doesn't have the issues the model in the previous section had with trying to avoid the activation function.

Experiment Setup

The input feature vector, x, is still sparse, with each feature x_i having probability S_i of being 0. However, since we want to have the model compute absolute value, we need to allow it to take on non-positive values for this to be a non-trivial task. As a result, if it is non-zero, its value is now sampled uniformly from $[-1, 1]$. The target output y is $y = \mathrm{abs}(x)$.

Following the previous section, we'll consider the "ReLU hidden layer" toy model variant, but no longer tie the two weights to be identical:

$$h = \mathrm{ReLU}(W_1 x)$$

$$y' = \mathrm{ReLU}(W_2 h + b)$$

The loss is still the mean squared error weighted by feature importances I_i as before.

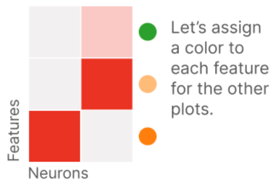
Basic Results

With this model, it's a bit less straightforward to study how individual features get embedded; because of the ReLU on the hidden layer, we can't just study $W_2^T W_1$. And because W_2 and W_1 are now learned independently, we can't just study columns of W_1. We believe that with some manipulation we could recover much of the simplicity of the earlier model by considering "positive features" and "negative features" independently, but we're going to focus on another perspective instead.

As we saw in the previous section, having a hidden layer activation function means that it makes sense to visualize the weights in terms of neurons. We can visualize W directly or as a neuron stack plot as we did before. We can also visualize it as a graph, which can sometimes be helpful for understanding computation.

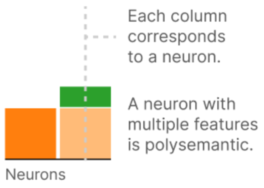
W as Matrix

Since the hidden layer now has a priviliged basis can visualize the raw weight matrix.



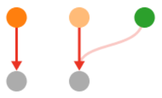
W as Stack Plot

Instead of showing a matrix, we can map features to colors and stack the weights per neuron.


Each column corresponds to a neuron.
A neuron with multiple features is polysemantic.
Neurons

W as Graph

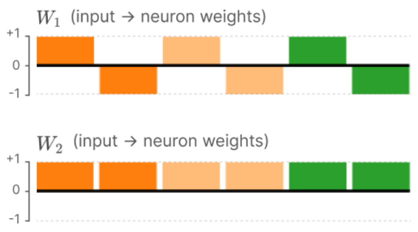
We can also visualize weights as the edges of a graph, as is often done for neural nets.



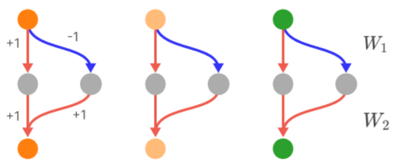
Let's look at what happens when we train a model with $n = 3$ features to perform absolute value on $m = 6$ hidden layer neurons. Without superposition, the model needs two hidden layer neurons to implement absolute value on one feature.

The resulting model – modulo a subtle issue about rescaling input and output weights[15] – performs absolute value exactly as one might expect. For each input feature x_i, it constructs a "positive side" neuron $\mathrm{ReLU}(x_i)$ and a "negative side" neuron $\mathrm{ReLU}(-x_i)$. It then adds these together to compute absolute value:

Raw Weights Visualization



Graph Visualization



Superposition vs Sparsity

We've seen that – as expected – our toy model can learn to implement absolute value. But can it use superposition to compute absolute value for more features? To test this, we train models with $n = 100$ features and $m = 40$ neurons and a feature importance curve $I_i = 0.8^i$, varying feature sparsity.[16]

A couple of notes on visualization: Since we're primarily interested in understanding superposition and polysemantic neurons, we'll show a stacked weight plot of the absolute values of weights. The features are colored by superposition. To make the diagrams easier to read, neurons are faintly colored based on how polysemantic they are (as judged by eye based on the plots). Neuron order is sorted by the importance of the largest feature.