COMPRESSED COMPUTATION UNDER L^4 LOSS IS LIKELY COMPUTATION IN SUPERPOSITION

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural networks are thought to represent concepts as directions in their activation space, and superposition lets them encode more concepts than they have dimensions. It is natural to ask whether they can also compute more functions than they have neurons, i.e., perform computation in superposition. In this regime many functions of sparse inputs are evaluated by a layer with fewer neurons than there are functions to compute. Representation in superposition is by now fairly well understood, but computation in superposition is not, and there are few toy models of it arising through training rather than being hand designed. As a toy model of computation in superposition we study the compressed-computation setup: a single-hidden-layer ReLU network with 50 neurons that must compute the ReLU of each of 100 sparse input features. We show that training it under an L^4 loss (the mean fourth power of the error), instead of the usual L^2, elicits a solution that appears to compute all features in superposition. We then reverse-engineer this solution. We find that the network assigns each feature a sparse binary codeword over neurons and decodes it with a pseudoinverse of the encoder. Given these codewords, a description with only three scalars recovers most of the network's performance, and we validate it by building equivalent networks from hand-designed codes.[†]

1 INTRODUCTION

Neural networks are widely assumed to make use of *superposition* to represent more concepts than they have dimensions (Elhage et al., 2022). A natural follow-up is whether they also *compute* in superposition, i.e., implement more non-linear functions than they have dimensions (Hänni et al., 2024; Adler & Shavit, 2024).

The picture of computation in superposition (CiS) we have in mind is that of a network simulating a larger, sparser network in which every feature has its own dedicated neuron (Elhage et al., 2022; Hänni et al., 2024; Bushnaq & Mendel, 2024; Linsefors & Bushnaq, 2025). The small network implements each neuron of the larger network as a *virtual neuron*: a group of its actual neurons, with each actual neuron shared among many virtual ones. Sharing is beneficial under sparsity: because few features are active at once, the virtual neurons rarely collide, and interference stays small. A network computes in superposition when it implements more virtual neurons than it has actual neurons.

While representation in superposition is increasingly well understood, CiS is not. In particular, until recently there have been no toy models of computation in superposition arising through training rather than being hand designed (see Section 2). Such toy models are desirable as they allow studying CiS at a scale where we can hope to fully reverse engineer the trained networks.

To this end, Braun et al. (2025) introduced the toy model of compressed computation, where a single-hidden-layer network must compute more ReLUs than its hidden width. However, Bhagat et al. (2025) showed that this toy model did not actually elicit computation in superposition.

In this work, we train a version of the compressed computation toy model under L^4 loss instead of L^2 and argue that the resulting network performs computation in superposition.

Our contributions are:

[†]Code: `https://anonymous.4open.science/r/cis-paper-code/`

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1. We train a toy model of compressed computation with L^4 loss and show that it meets an operational criterion for computation in superposition: the number of features it computes exceeds its neuron count (Section 4.4), across a wide range of sparsities and feature-to-neuron ratios.
2. We reverse-engineer the trained network: it assigns each feature a sparse binary codeword over neurons and decodes with a pseudoinverse of the encoder. Given these codewords, a 3-scalar parameterization (on-code value, off-code value, decoder scale) recovers $\approx 1.1\times$ its loss.
3. We validate this description by substituting hand-designed binary codes in the same 3-parameter family: the best designed codes yield networks at $\approx 1.1\times$ the trained model's loss.

2 Related Work

Comparable empirical work. Gibson (2026) is concurrent, independent work. The tasks studied differ: both networks take continuous sparse inputs, but ours is a regression task (compute the ReLU of each input), whereas Gibson's network must identify which inputs are active, outputting a fixed value at each active index regardless of its magnitude. Both settings nonetheless learn binary codes over neurons, with the same aspects left unexplained: non-uniform on-codeword values, a decoder deviating from a scaled pseudoinverse and a trained network outperforming designed alternatives.

Theoretical constructions for CiS. Gibson (2025) considers a task in which a continuous input passes through one of many small circuits sharing a single wider network, with exactly one circuit active per forward pass. They provide a hand-designed zero-error construction for computation in superposition via sparse memory blocks; we find a sparse-coding solution arising through training.

Random sparse binary codes. Hänni et al. (2024) proposes random sparse binary codes for computation in superposition. Our work appears to be an empirical instance of their constructions.

The compressed computation toy model. Braun et al. (2025) introduced the toy model of compressed computation, and Bhagat et al. (2025) showed that it does not elicit computation in superposition. We train a version of this model under L^4 loss and show that it does.

3 Methodology

Our network (Figure 1) is the compressed-computation toy model of Braun et al. (2025) with the residual connection removed (see below). The input is 100-dimensional, with each feature being zero with probability $1 - p$ and drawn uniformly from $(-1, 1)$ with probability p. We set $p = 0.02$, giving two active features in expectation. A single hidden layer of 50 ReLU neurons without bias acts directly on the input through trainable weights $W_{\text{in}} \in \mathbb{R}^{50\times 100}$ and $W_{\text{out}} \in \mathbb{R}^{100\times 50}$, and the task is to compute the elementwise ReLU of the input.

Braun et al. (2025) additionally embed the input as random near-orthogonal directions in a higher-dimensional (1000) space through a fixed random matrix W_E with unit-norm rows, read back out by the transpose $W_E^\top$, so that the features are not axis-aligned. This is a more realistic and challenging setting for the parameter-decomposition methods the model was built to test. That embedding turns out to be incidental to the computation-in-superposition solution, as we find that training with it recovers an identical solution. For clarity of exposition, we present the simpler axis-aligned model in the main text and defer the embedded variant to Appendix A.

Beyond setting aside the incidental embedding, we also remove the residual connection, motivated by Bhagat et al. (2025)'s analysis. They show that, in Braun et al.'s model, the residual connection is equivalent to a mixing matrix M_{mix} in the target, $y = \text{ReLU}(x) + M_{\text{mix}}x$, where M_{mix} couples input x_j to target y_i for $i \neq j$. It is M_{mix}, not superposition, that lets the model beat the naive baseline of ignoring half the features. Removing the residual connection sets $M_{\text{mix}} = 0$, so the target is exactly $y = \text{ReLU}(x)$ and any substantial improvement over the naive baseline must come from superposition. Bhagat et al. (2025) showed that removing the residual connection does not by itself

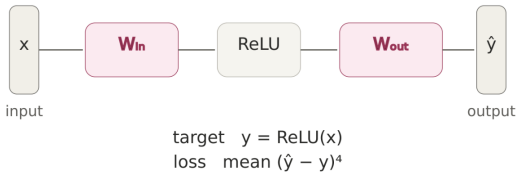


Figure 1: **Network architecture:** a linear encoder W_{in}, 50 ReLU neurons without bias, and a linear decoder W_{out}. The input and output are 100-dimensional; the task is to compute the elementwise ReLU of the input.

elicit computation in superposition, and we confirm it: trained under the usual L^2 loss our model still finds the naive solution (Section 4.1). Nor is it needed for superposition to arise: the original residual-kept model trained under L^4 also finds the superposition solution (Appendix A.4).

The change we make that elicits CiS is the loss. We train under L^4 loss, the average fourth power of the elementwise error, instead of the usual L^2. We train with Adam (learning rate 0.01, cosine annealing schedule), batch size $8{,}192$, for $100{,}000$ steps; W_{in} and W_{out} are initialized entrywise from $U(-0.1, 0.1)$ and $U(-0.15, 0.15)$. The intuition behind making this change is that L^4 penalizes outlier errors more than L^2, and hence should prioritize networks approximating all features roughly equally well (superposition) instead of computing the output for some features perfectly and for others not at all. In Section 4.1 we also sweep the loss exponent, including standard L^2.

4 Results

4.1 Evidence for computation in superposition

As evidence that L^4 training elicits superposition, we compare the per-feature L^4 loss of the L^2- and L^4-trained networks, evaluated on the training distribution itself: each feature active with probability $p = 0.02$, active values uniform on $(-1, 1)$ (Figure 2). Training under L^2 results in a "naive solution" that learns half the features perfectly and ignores the rest (red). Under L^4, the trained network instead learns all 100 features approximately equally well. We further compare against another no-superposition baseline, namely one that represents half the features exactly and emulates a bias on the rest by adding a constant offset to their outputs (following Bhagat et al. (2025)). We impose that this offset must be produced by the bias-free architecture itself, so this is the best *architecturally realizable* constant. At our sparsity the offset helps only marginally, with the emulate-bias baseline having loss $\sim 8\%$ lower than the naive baseline, still $\sim 26\times$ higher than the L^4 solution.

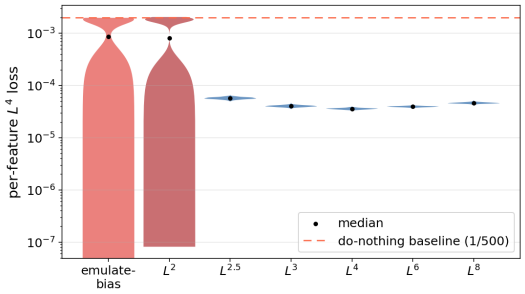


Figure 2: **Per-feature L^4 loss on the training distribution: the strongest non-superposition baseline and the trained networks at several loss exponents.** Each violin pools the per-feature L^4 loss, evaluated on the full training distribution over the 100 features and over 10 seeds for the trained networks. Black dots are medians. The emulate-bias baseline and L^2 (reds) are bimodal, with half the features computed and half near the do-nothing (output-zero) baseline $p\,\mathbb{E}[\text{ReLU}(x)^4 \mid \text{active}] = 0.02/10 = 1/500$ (dashed). In contrast, each exponent above 2 (blue) concentrates the error into a narrow band far below the baselines, spread evenly across all 100 features.

As there are more features than neurons, achieving low error on all features requires sharing neurons across features, i.e., superposition. Eliciting this solution is not unique to L^4, as Figure 2 shows: every exponent above 2 produces a similar concentrated distribution. To quantify this, we summarize each network by the coefficient of variation of its per-feature L^4 loss across the 100 features, i.e., the standard deviation divided by the mean. We use the coefficient of variation instead of the raw standard deviation because the mean per-feature loss varies greatly across exponents, so a scale-free measure is needed to compare them. This is small when the error is spread evenly across features (indication of superposition) and large when a few features carry most of the error (the naive solution). Every exponent above 2 that we tried $(2.5, 3, 4, 6, 8)$ gives a superposition-like solution, with coefficients of variation between 0.023 and 0.045. L^2 instead gives the naive solution, with a coefficient of variation of 1.00. L^4 is thus a representative choice. Across seeds the coefficient of variation varies by at most 0.013.

Note that the coefficient of variation, in and of itself, does not constitute evidence of superposition, as a network computing nothing at all would also score low by this metric. However, the conjunction of a low coefficient of variation and a loss below non-superposition baselines does constitute evidence of superposition. In Section 4.4 we turn this two-part comparison into an operational criterion, namely the number of features the network computes measured against its neuron count. We then sweep sparsity and the feature-to-neuron ratio to map out where the superposition solution arises.

Further, L^2 and L^4 appear to have different optima. Under mean squared error, the L^4-trained network (2.41×10^{-3}) is $1.45\times$ *worse* than the L^2-trained network (1.66×10^{-3}), while under L^4 it is $26\times$ better. This indicates that a different solution arising for each loss is not due to a failure of optimization, but instead due to the different losses preferring different types of solution. L^2 being the loss that fails to elicit superposition is in line with prior reports: mean squared error is known to be a poor loss for eliciting superposition (Hazelden & Linsefors, 2026).

4.2 The mechanism

We characterize the L^4-trained network's mechanism through three observations about its encoder $W_{\text{in}} \in \mathbb{R}^{N\times F}$, which maps each input feature to the neuron pre-activations it induces, and its decoder $W_{\text{out}} \in \mathbb{R}^{F\times N}$.

Figure 3 shows the distribution of the entries of the encoder W_{in}, pooled over all 100 feature columns. The entries fall into two well-separated groups (values reported as mean $\pm$ standard deviation): a large mass of small negative values (-0.017 ± 0.006) and a smaller mass of large positive values (0.34 ± 0.06). We call the support of the large entries the *codeword* of feature j, written $M_j \in \{0,1\}^N$. This is the subset of neurons that fire when feature j is active, with codeword length $K = |M_j|$. Stacking these gives a binary matrix $M \in \{0,1\}^{F\times N}$ whose pattern matches the shape of W_{in}: large positives where $M_{j,n} = 1$, small negatives where $M_{j,n} = 0$. In terms of the picture of CiS from the introduction, each codeword can be seen as a virtual neuron: the set of neurons that jointly play the role of feature j's dedicated neuron in the larger simulated network.

The codewords are quite regular (Figure 4): every feature uses $K = 5$-7 neurons (mean 5.5), and every neuron participates in 10-12 codewords (mean 10.9). The code is thus close to *biregular*: every codeword is nearly the same length, and every neuron is shared by nearly the same number of codewords. The decoding load is thus spread near-equally across neurons.

If the codewords serve as the network's internal identifier for each feature, then transplanting feature i's hidden activation values onto feature j's codeword neurons should make the decoder output feature j. We verify this directly: for every pair of features (i, j) of equal codeword length (2,546-3,528 pairs per seed), we transplant feature i's hidden magnitudes onto feature j's codeword neurons, and the decoder's top output is j in 99.97% of cases (mean over 10 seeds; worst seed 99.87%; Figure 5). As a control, we transplant the same magnitudes onto uniformly random neuron subsets of matched size. This makes the top output j in only 1.0% of trials, which is exactly what we would expect by chance with 100 features.

Finally, the decoder seems to do little more than invert the encoder: the decoder W_{out} has Frobenius cosine 0.996 with a scaled pseudoinverse of the encoder (the Frobenius cosine is the cosine similarity between the two flattened matrices). To confirm, we replace the trained decoder outright with a scaled pseudoinverse of the encoder and find the L^4 loss increases only slightly ($1.14\times$ the trained network,

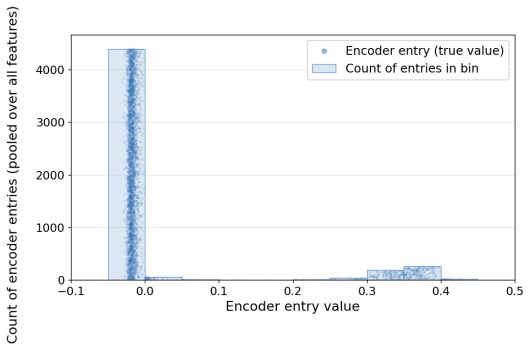


Figure 3: **Encoder entries are bimodal:** pooled histogram of all entries of the encoder W_{in} over the 100 feature columns. Bars are binned in steps of 0.05 with edges aligned to zero; bar height is the number of entries in each bin, with those entries scattered over the bar at their true value (vertical position is jitter, for visibility). The entries split into a large off-code cluster of small negatives (-0.017 ± 0.006) and a smaller on-code cluster of large positives (0.34 ± 0.06).

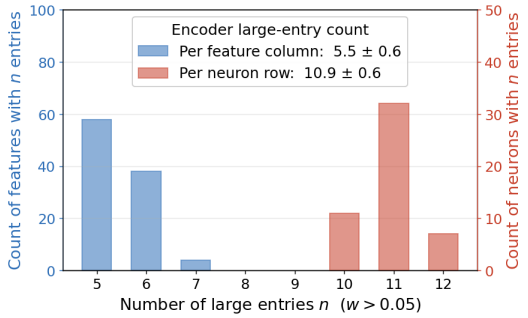


Figure 4: **The binary code is regular:** number of large encoder entries (those exceeding 0.05, i.e. on-code entries) per feature column (blue, left axis) and per neuron row (red, right axis). Every feature uses 5-7 neurons (mean 5.5) and every neuron is shared by 10-12 codewords (mean 10.9).

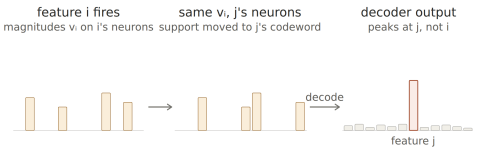


Figure 5: **Codeword swap test:** the hidden magnitudes produced by feature i, placed on feature j's codeword neurons, decode to j. The codeword support, not the magnitudes, determine the decoded feature.

seed standard deviation 0.001), so the decoder is well described as a pseudoinverse read-out of the codewords. Because the codewords have small pairwise overlap, this pseudoinverse is itself close to a scaled transpose of the encoder, so the decoder ends up looking much like the encoder: each feature is read out along essentially the same neurons that encode it.

Together, these findings paint the picture in Figure 6: a binary-code encoder followed by a pseudoinverse decoder. One feature of this picture is that the decoded output is *attenuated*: the network systematically under-shoots the true ReLU. This is a response to interference: codewords overlap, so an active feature leaks through its shared neurons into the outputs of the (far more numerous) inactive features, and shrinking every active output keeps that leakage small. The single-feature input–output response in Figure 13 shows this directly. We probe interference between co-active features directly

in Appendix C: with k features active at once, the loss grows roughly quadratically in k, consistent with pairwise codeword collisions, and below the naive solution's loss throughout the tested range.

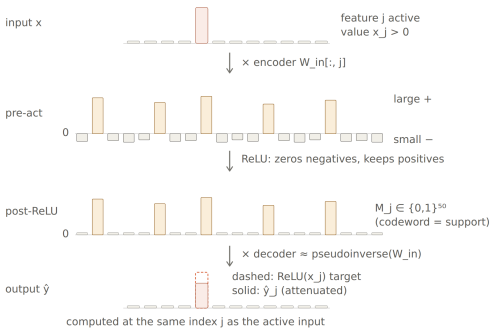


Figure 6: **Mechanism for a single active feature:** the encoder produces large positives on the feature's codeword neurons and small negatives elsewhere; ReLU zeros the negatives; the pseudoinverse decoder reads the codeword and outputs a peak at the same index.

4.3 A 3-PARAMETER DESCRIPTION GIVEN A BINARY CODE

The mechanism depicted in Figure 6 suggests a simple ansatz. We extract the binary codeword matrix M from the trained network by classifying each encoder entry as on- or off-code, then parameterize a new encoder with one value on the support and one off it, $W_{\text{in}} = a\,M^\top + b\,(1 - M^\top)$, and set the decoder to a scaled pseudoinverse of it, $W_{\text{out}} = c\,(W_{\text{in}})^+$. An entry is classified as on-code if it takes a value above 0.05. The classification is robust to this threshold: varying it tenfold, from 0.02 to 0.2, changes the mean codeword length only from 5.6 to 5.2 (moving from the default 0.05 to either extreme changes roughly 15% of codewords by a single borderline neuron), and the resulting ansatz loss, reported below, stays within a few percent ($1.16\times$ to $1.10\times$). This leaves three free scalars: the on-code value a, the off-code value b, and the decoder scale c.

Fitting these three scalars to the trained network's codewords gives an L^4 loss $1.13 \pm 0.01\times$ that of the trained network (mean $\pm$ std over 10 seeds), far below the non-superposition baselines (naive $\sim 28\times$, emulate-bias $\sim 26\times$, random network $\sim 56\times$).

The support carries little information. We test how much the specific support matters by replacing M with a randomly generated codeword matrix of a given codeword length K. We consider two families: *biregular* codes (every codeword the same length, every neuron shared by the same number of codewords) and random codes. We try each family both as generated and after applying random edge swaps that reduce pairwise codeword overlaps (Appendix D), expecting biregularity to balance load across neurons and uniform overlaps to minimize the worst-case interference that L^4 penalizes most.

Figure 7 shows the result. Overlap minimization matters: without it both families sit at 1.7-$2.0\times$ the trained loss, while with it the best designed code — biregular $K = 5$ — reaches $1.12 \pm 0.004\times$, matching the three-scalar version of the trained network's own code ($1.13 \pm 0.01\times$) within seed spread. Up to the choice of code family and codeword length K, then, the network's performance is captured by just three scalars, and the precise support M carries little extra information.

4.4 IN WHAT REGIME DOES SUPERPOSITION ARISE?

Reverse engineering is our strongest evidence for superposition, but it cannot be repeated at every operating point. Hence, to map out where superposition arises we use an operational criterion: the number of features the network computes, measured against its neuron count. We consider a feature computed when the network approximates its target $\mathrm{ReLU}(x_i)$ at least 10% better than every constant or linear solution of the same width, and than the affine part of its own input–output function. We choose these as references because they do not use the network's non-linear capacity (see Appendix E). We posit that solutions without superposition cannot compute more than N features

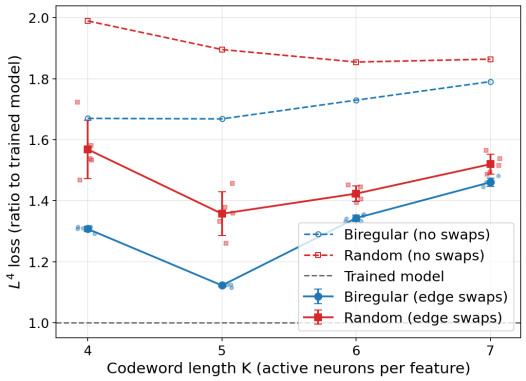


Figure 7: L^4 **loss of 3-parameter ansatz networks:** Loss of networks built from synthetic binary codes (biregular and random) as a function of codeword length K, normalized by the trained model's loss. Solid lines apply overlap-minimizing edge swaps (five seeds each; error bars are standard deviations, scatter points are individual seeds); dashed lines are the same codes without edge swaps. Biregular $K = 5$ with edge swaps is best at $\sim$1.12$\times$ the trained loss; without edge swaps both families stay within $\sim$2$\times$. No synthetic code beats the trained network.

under this criterion, so a count $C > N$ indicates computation in superposition. Under this criterion and at the operating point for which we presented the reverse engineering results, the trained network computes all 100 features with 50 neurons (see Appendix E for details and ablations).

We train the axis-aligned model over a grid of feature probabilities p from 0.005 to 1 and widths N from 200 down to 20 ($F = 100$ throughout), with the training methodology of Section 3 and 5 seeds per cell (Appendix E). Figure 8 shows the number of computed features per cell, averaged over seeds.

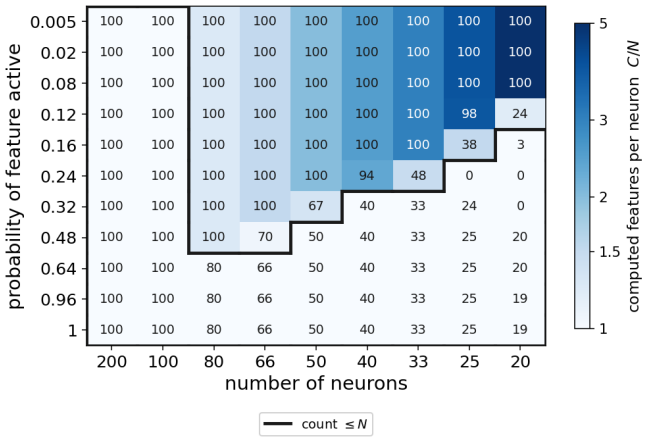


Figure 8: **Number of computed features across sparsity and width.** Each cell is an L^4-trained axis-aligned model ($F = 100$, 5 seeds), annotated with the number of computed features C (Appendix E) and colored by C/N: white where $C \leq N$ (no superposition; black outline), increasingly blue the further the count exceeds the neuron count.

The network computes all 100 features at every width up to 8 expected active features, four times the two of the main operating point, and superposition ($C > N$) persists up to 48 expected active features for the network with 80 neurons. This indicates L^4 loss leads to superposition broadly in this setup. As the input gets denser the number of computed features decays to each column's neuron count: for $p \geq 0.64$ every cell sits at (or just below) $C = N$. This convergence, together with the fact that the loss approaches the emulate-bias baseline's, suggests the network degrades by dropping features and implementing a progressively more naive-like solution. An exception to this are the networks in the two rightmost columns, close to the superposition boundary. These settle on solutions

that attempt to compute all features, but do so almost entirely in an affine way, which is why those cells show close to zero computed features.

We note that we have not re-verified the binary-code mechanism across the grid, so cells away from the main operating point are only indications of superposition.

5 Open Questions

The three-scalar ansatz built from the trained codewords reaches $1.13\times$ the trained loss, and the best hand-designed code (Figure 7) matches it within seed spread. Since the ansatz already uses the trained network's own support — so the codewords, including their lengths, match — what separates it from the trained network is just the encoder's specific per-entry values and the decoder. Table 1 decomposes how much each accounts for: relaxing the encoder or the decoder closes part of the gap, and the network reaches $1.00\times$ only when both are free. We understand each piece only partially.

Table 1: **What each ingredient accounts for.** For each construction we list the encoder, the decoder, and the resulting L^4 loss relative to the trained network ($1.00\times$), as mean $\pm$ std over seeds (5 for designed codes, 10 for trained-network rows). *Trained support*: the binary codeword pattern M, read off the trained encoder. *3 scalars*: the on-code/off-code/decoder-scale ansatz (a, b, c). *Free per-entry*: every encoder entry fit freely. *pinv*: decoder fixed to a scaled pseudoinverse of the encoder. Designed codes use codeword length $K = 5$; "overlap-min." applies the edge swaps of Appendix D (Figure 7).

Encoder	Decoder	L^4 loss
Designed code ($K = 5$), 3 scalars		
random, no overlap-min.	pinv	$1.90 \pm 0.03\times$
biregular, no overlap-min.	pinv	$1.67 \pm 0.02\times$
random, overlap-min.	pinv	$1.36 \pm 0.06\times$
biregular, overlap-min.	pinv	$1.12 \pm 0.004\times$
Trained support, 3 scalars	pinv	$1.13 \pm 0.01\times$
Trained support, free per-entry	pinv	$1.05 \pm 0.002\times$
Trained support, 3 scalars	trained	$1.05 \pm 0.006\times$
Full trained encoder	pinv	$1.14 \pm 0.001\times$
Trained (decoder tied to pinv)	pinv	$1.04 \pm 0.004\times$
Full trained encoder	trained	$1.00\times$

First, the per-entry encoder values. Replacing the two on/off scalars with the trained network's individual entries, while still decoding with a scaled pseudoinverse, lowers the loss to $1.05\times$; but we do not fully understand what these values encode. On-codeword values are not uniform, and enforcing uniformity harms performance. We observe that neurons used in more codewords have lower mean on-codeword values ($r \sim -0.82$), as one would expect if the network down-weights overloaded neurons; this accounts for $\sim 67\%$ of the variance in the neurons' mean on-codeword values, so it is most but not all of the story. Notably, the values still vary even in a designed biregular code, where every neuron has the same degree: with this load-balancing effect removed entirely, the fitted per-entry values vary about as much as in the trained network (coefficient of variation ~ 0.15 versus ~ 0.17).

Second, the decoder. The trained decoder is close to a scaled pseudoinverse of the encoder (Frobenius cosine 0.996). A network trained *from scratch* with its decoder tied to a scaled pseudoinverse of the encoder still reaches $1.04\times$, and a freely-trained decoder on the three-scalar ansatz reaches only $1.05\times$ (Table 1). The larger $1.14\times$ cost of swapping in a pseudoinverse *post hoc* is therefore mostly because the trained encoder was tuned for its own decoder, not because a pseudoinverse decoder is inherently worse.

Finally, it is unclear whether the trained network's binary support is special at all: in the three-scalar family it is not, as a designed biregular $K = 5$ code matches the three-scalar version of the trained code within seed spread.

Together, these results account for most of the trained network's performance and pin down its main mechanism, a sparse binary code decoded by a near-pseudoinverse, leaving the encoder's specific

per-entry values and the decoder's small deviation from a scaled pseudoinverse of the encoder as the pieces we have not fully explained.

6 Discussion and Future Work

Our reverse-engineered network looks like an empirical instance of the random sparse binary codes that Hänni et al. (2024) propose for computation in superposition: through training it independently arrives at the sparse-code-over-neurons motif their constructions predict. The match is suggestive, as their analysis targets Boolean U-AND while our network computes continuous ReLUs.

The original motivation for this toy model was a testbed for parameter-decomposition methods such as APD (Braun et al., 2025) and SPD (Bushnaq et al., 2025). Now that we have elicited superposition and reverse-engineered the network, we can and will use it as a ground-truth solution against which to test these and other interpretability methods.

More broadly, we would like to know how far this picture extends: to deeper and more realistic networks, and to more realistic kinds of computation than the elementwise ReLU studied here. Elementwise ReLU is a minimal form of computation which involves no feature interactions, with our result therefore demonstrating superposition of independent functions. Tasks with genuine interactions, such as the pairwise ANDs of Hänni et al. (2024), are a natural next step. Does superposition still arise there, can we still reverse-engineer the resulting solution, and does anything resembling the error-correction layers of Hänni et al. (2024) appear?

AI use statement

We used generative AI tools to generate the majority of the code used in our experiments. The code was verified for correctness by one of the authors. Generative AI tools were used for feedback and editing help on the manuscript. We did not otherwise use generative AI tools in this work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Reproducibility statement

Code to reproduce all experiments, together with the data behind every figure and table, is available at `https://anonymous.4open.science/r/cis-paper-code/`. Section 3 specifies the model, and all relevant training details. Appendix A describes the embedded variant of the model, Appendix D the edge-swap procedure used to optimize the synthetic codes, and Appendix E the criterion used to establish whether a feature is computed, as well as details of the sparsity–width sweep. All experiments ran on a single consumer GPU in under 12 GPU-hours.

References

Micah Adler and Nir Shavit. On the complexity of neural computation in superposition. *arXiv preprint arXiv:2409.15318*, 2024.

Jai Bhagat, Sara Molas-Medina, Giorgi Giglemiani, and Stefan Heimersheim. Compressed computation is (probably) not computation in superposition. In *Mechanistic Interpretability Workshop at NeurIPS 2025*, 2025.

Dan Braun, Lucius Bushnaq, Stefan Heimersheim, Jake Mendel, and Lee Sharkey. Interpretability in parameter space: Minimizing mechanistic description length with attribution-based parameter decomposition. *arXiv preprint arXiv:2501.14926*, 2025. URL `https://arxiv.org/abs/2501.14926`.

Lucius Bushnaq and Jake Mendel. Circuits in superposition: Compressing many small neural networks into one. `https://www.lesswrong.com/posts/roE7SHjFWEoMcGZKd/circuits-in-superposition-compressing-many-small-neural`, October 2024. LessWrong / AI Alignment Forum. Accessed 2026-08-25.

Lucius Bushnaq, Dan Braun, and Lee Sharkey. Stochastic parameter decomposition. *arXiv preprint arXiv:2506.20790*, 2025. URL https://arxiv.org/abs/2506.20790.

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

Alex Gibson. Ping pong computation in superposition. https://www.lesswrong.com/posts/g9uMJkcWj8jQDjybb/ping-pong-computation-in-superposition, 2025. LessWrong.

Alex Gibson. Neural networks learn bloom filters. https://www.lesswrong.com/posts/buxBdp8NtHGgBwabv/neural-networks-learn-bloom-filters, 2026. LessWrong.

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

Phil Hazelden and Linda Linsefors. MSE loss does not generate superposition. https://www.lesswrong.com/posts/cfwAK4Qvjne4RjB74/mse-loss-does-not-generate-superposition, 2026. LessWrong.

Linda Linsefors and Lucius Bushnaq. Circuits in superposition 2: Now with less wrong math. https://www.lesswrong.com/posts/FWkZYQceEzL84tNej/circuits-in-superposition-2-now-with-less-wrong-math, 2025. LessWrong.

A THE EMBEDDED MODEL

The main text studies a model in which the 50 ReLU neurons act directly on the 100 input features. Braun et al. (2025)'s original compressed-computation model instead embeds the 100-dimensional input into $\mathbb{R}^{1000}$ through a fixed random matrix W_E with unit-norm rows: the 50 neurons read and write this 1000-dimensional space through trainable $W_{\text{in}} \in \mathbb{R}^{50\times1000}$ and $W_{\text{out}} \in \mathbb{R}^{1000\times50}$, and the output is read back out by the transpose $W_E^\top$. Because $1000 > 100$ and W_E has full row rank, the embedding is information-lossless: the network is equivalent to one acting directly on the input features through the effective weights $\widetilde{W}_{\text{in}} = W_{\text{in}}W_E^\top$ and $\widetilde{W}_{\text{out}} = W_E W_{\text{out}}$, which play exactly the roles of the encoder W_{in} and decoder W_{out} analyzed in Sections 4.1 and 5. It should therefore make no difference to the learned solution, and we confirm below that it does not. Figure 9 shows this embedded architecture, with the fixed embedding W_E and unembedding $W_E^\top$ restored relative to the axis-aligned model of Figure 1.

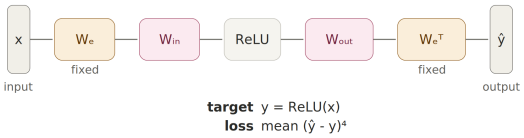


Figure 9: **Embedded architecture** (cf. Figure 1). The input is embedded as random near-orthogonal directions in a 1000-dimensional space through a fixed matrix W_E with unit-norm rows and read back out by the transpose $W_E^\top$; the trainable encoder W_{in}, 50 ReLU neurons, and decoder W_{out} are unchanged. The embedding (W_E, $W_E^\top$) is held fixed during training.

We retain the embedded variant because the compressed-computation toy model's original purpose was as a testbed for parameter-decomposition methods such as APD (Braun et al., 2025) and SPD (Bushnaq et al., 2025), for which the non-axis-aligned features of the embedded model are the more challenging and realistic setting. For the narrower claim that the trained network computes in superposition, the axis-aligned model in the main text is cleaner: with axis-aligned input features there is no random embedding that could, even in principle, be credited with the network's performance over the non-superposition baselines.

A.1 THE EMBEDDED MODEL REPRODUCES THE AXIS-ALIGNED SOLUTION

We verify directly that adding the embedding changes nothing of substance. We train the embedded model under the same loss, optimizer, and schedule, and recompute every quantity from Sections 4.1 and 5. Table 2 reports the comparison. Across all of them — the per-feature loss spread, the codeword-length distribution, the swap test, the decoder's agreement with a pseudoinverse, and the three-scalar description — the two models are quantitatively the same; even the three fitted ansatz scalars (a, b, c) agree to within 1%. We conclude that the random embedding is incidental to the computation-in-superposition solution. Figures 10 to 12 reproduce the main-text figures (Figures 2 to 4) for the embedded model.

Table 2: **The solution is unchanged by adding the random embedding.** Headline quantities from Sections 4.1 and 5 for the axis-aligned model (main text) and an otherwise identical model trained with Braun et al. (2025)'s random embedding. CV is the coefficient of variation of the per-feature L^4 loss, evaluated on the training distribution, across the 100 features. The ansatz ratio is the L^4 loss of the three-scalar description relative to the trained model.

Quantity	Axis-aligned	Embedded
Per-feature loss CV (L^4)	0.031	0.032
Per-feature loss CV (L^2)	1.00	1.00
Codeword length K	5-7	5-7
Swap-test pass rate	99.97%	99.99%
Decoder–pinv cosine	0.996	0.996
On/off-code value	$+0.34/-0.02$	$+0.34/-0.02$
3-scalar ansatz ratio	$1.13\times$	$1.12\times$

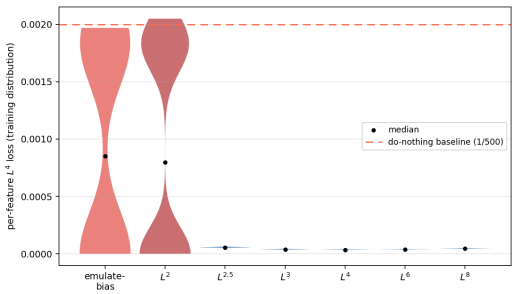


Figure 10: **Embedded model, per-feature L^4 loss on the training distribution** (cf. Figure 2). As in the axis-aligned model, the emulate-bias baseline and L^2 are bimodal (half the features handled, half left near the do-nothing baseline) while every exponent above 2 spreads the error evenly across all 100 features.

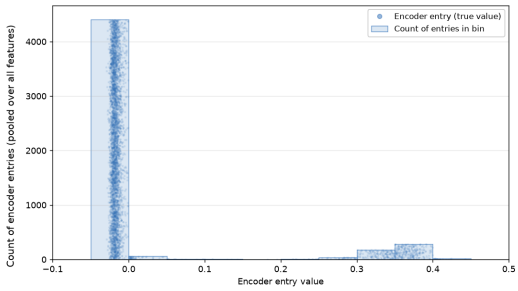


Figure 11: **Embedded model, effective-encoder entries** (cf. Figure 3). Pooled histogram of the entries of the effective encoder $\widetilde{W}_{\text{in}} = W_{\text{in}} W_E^\top$; the same bimodal split into a large off-code cluster of small negatives and a smaller on-code cluster of large positives appears.

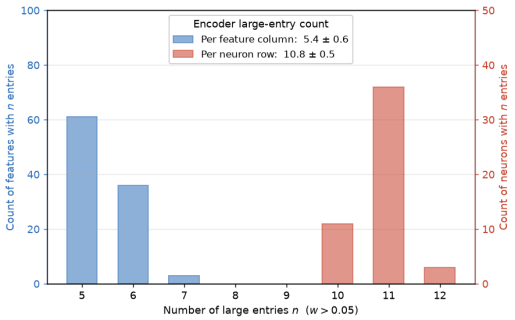


Figure 12: **Embedded model, codeword regularity** (cf. Figure 4). Large-entry counts per feature column and per neuron row for the effective encoder; the code is close to biregular, as in the axis-aligned model.

A.2 Transpose versus pseudoinverse unembedding

The embedded model reads its output back out with the transpose $W_E^\top$ of the embedding. Because W_E has unit-norm but not orthogonal rows, this transpose is only an approximate inverse, leaving a small residual cross-talk between the near-orthogonal feature directions. We retrain the embedded model with its unembedding set instead to the exact pseudoinverse W_E^+ — the left-inverse that removes this cross-talk — holding the loss, optimizer, schedule, embedding, and seed fixed. It reaches the same L^4 loss (within 1%) and the same per-feature error spread.

A.3 Other model sizes

To check that the solution is not specific to our $F = 100$, $N = 50$, $d = 1000$ choice, we vary the model size and measure the coefficient of variation of the per-feature L^4 loss on the training distribution (Section 4.1). Tightening the embedding to $d = 50$ (keeping $F = 100$, $N = 50$) still gives indications of superposition, though weaker: the coefficient of variation is 0.20, versus 0.03 at $d = 1000$. Some weakening is expected here: with $d = 50 < F = 100$ the embedding itself is lossy, so the task is strictly harder than in the invertible $d \geq F$ case. Scaling up an order of magnitude — $F = 1000$ features, a fixed random embedding into $d = 500$, and $N = 500$ neurons — gives a low coefficient of variation again (0.08). In both regimes the error is spread across all features whereas in the naive solution it is concentrated. We check only this loss statistic here, not the full binary-code mechanism, so we report these as indications of superposition, not confirmation.

A.4 Restoring the residual connection

The main text and the appendices above all remove Braun et al. (2025)'s residual connection — equivalently, the mixing term $M_{\text{mix}}x$ in the target (Section 3). We consider here the original model unchanged, i.e., the network with embedding Appendix A plus a residual connection, so that the target is $y = x + \text{ReLU}(x)$. We train under the same L^4 recipe, with 10 seeds. The superposition-like solution appears for this model as well: the coefficient of variation of the per-feature L^4 loss on the training distribution is 0.06 (range 0.055-0.065 across seeds). The effective encoder again shows binary codewords, though fuzzier than the main model's: codeword lengths 4-14 (mean 9.3) versus 5-7, and the decoder has Frobenius cosine 0.990 with a scaled pseudoinverse of the encoder. With the residual kept, the mixing term lets non-superposition solutions beat the naive baseline (Bhagat et al., 2025), so baseline comparisons are not informative here and we hence refrain from making them. As in Appendix A.3, we check these statistics and not the full mechanism, so we report this as an indication of superposition.

B Output attenuation

Following Bhagat et al. (2025), we examine the network's input–output response by driving a single feature j with a value v (all other features zero) and reading the output at index j, repeating this

for each of the 100 features (Figure 13). A network executing the task perfectly would output $y = \mathrm{ReLU}(v)$; the trained network instead outputs a scaled-down ReLU, with mean slope ≈ 0.81. To state the accuracy in absolute terms: when a feature is active with a positive value, the root-mean-square error at its output index is 0.148, while the target $\mathrm{ReLU}(x_j)$ has a root-mean-square of 0.577, meaning that the typical relative error is 26%. This error splits into two parts, plus a cross-term: the error the network makes on feature j when it is the only active feature (attenuation), and the error caused by the other active features (interference). In squared terms, the attenuation contributes 56% and the interference 20%, with the remaining 24% being the cross-term between the two. The per-feature responses are nearly identical — the 5–95% band across features is too narrow to see — consistent with the even, superposition-like solution in which every feature is handled the same way.

This attenuation is a response to interference. Because features are sparse (active with probability $p = 0.02$) and represented by overlapping codewords, an active feature's signal leaks through shared neurons into the outputs of the inactive features, pulling those outputs above the zero they should produce. Under L^4 this cross-talk is costly. The network compensates by attenuating every active output: shrinking the on-feature response reduces the leaked interference on the far more numerous inactive features, at the cost of a small under-shoot on the active one. The net effect is the systematic under-shoot in Figure 13.

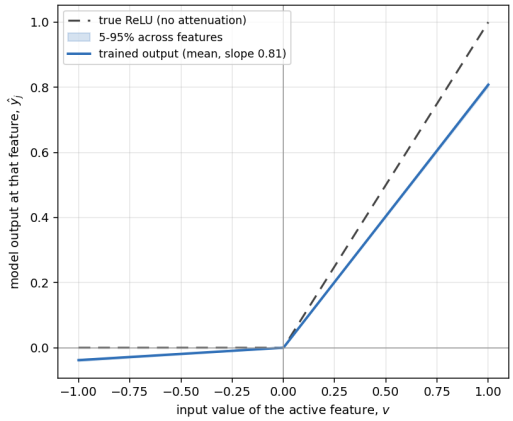


Figure 13: **Single-feature input–output response.** For each feature j in turn, feature j alone is driven with value v and we read the output at index j; the blue curve is the mean over all 100 features. The trained network traces a ReLU attenuated to mean slope ≈ 0.81, below the true ReLU (dashed). The 5–95% band across features is too narrow to be visible: every feature is attenuated almost identically, as expected for the even superposition solution.

C Interference between co-active features

The cost of superposition is interference: when multiple features are active simultaneously, their codewords collide. The interventions of Figures 5 and 13 test one feature at a time, so here we probe the regime with multiple active features. For each k we sample inputs with exactly k active features (values uniform on $(-1, 1)$) and measure the mean per-output L^4 loss of the 10 L^4-trained seeds, the 10 L^2-trained (naive) seeds, and the do-nothing baseline.

The L^4-trained network's loss grows approximately quadratically in k — doubling k roughly quadruples the loss — consistent with interference dominated by pairwise codeword collisions. The naive solution instead grows linearly, since it simply loses the active features it dropped. The trained network nonetheless stays below the naive solution over the whole tested range, up to $k = 32$ simultaneously active features — sixteen times the training regime's expected $pF = 2$ — and far below the do-nothing baseline everywhere. The superposition solution is hence not brittle, although the quadratic interference growth does mean the naive solution would eventually win at large enough k, consistent with the intuition that superposition does not survive in dense-enough regimes.

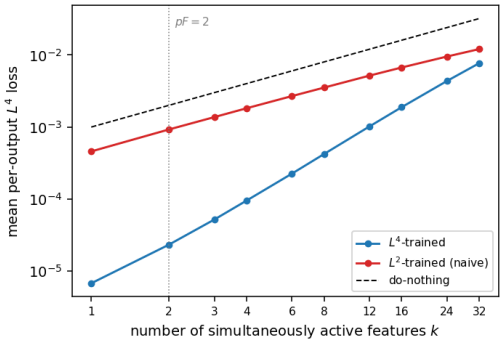


Figure 14: **Loss versus number of co-active features.** Mean per-output L^4 loss on inputs with exactly k active features, for the L^4-trained networks (blue), the L^2-trained naive networks (red; bands span the 10 seeds), and the do-nothing baseline (dashed). The dotted line marks the training regime's expected number of active features $pF = 2$. The L^4 solution's interference grows $\sim k^2$ but remains below the naive solution throughout $k \leq 32$.

D CODEWORD OVERLAP MINIMIZATION

When two features are active at once, their codewords interfere: neurons in both codewords receive contributions from both features, and the decoder must disentangle them. The size of this interference is set by the pairwise overlap $|M_i \cap M_j|$, and under L^4 the largest overlaps dominate the loss. Before fitting the ansatz to a synthetic code we therefore reduce its overlaps, and prefer codes whose overlaps are both small and uniform, which minimizes the worst-case interference. We do this with a simple edge-swap procedure.

Let the pairwise overlap matrix be $O \in \mathbb{N}^{F \times F}$ with $O_{ij} = |M_i \cap M_j|$. Our objective is the sum of off-diagonal squared overlaps, $\sum_{i \neq j} O_{ij}^2$.

At each iteration: sample two edges $(i, a), (j, b)$ uniformly at random from the edges in M; if $i \neq j$, $a \neq b$, $M_{ib} = 0$, and $M_{ja} = 0$, propose the swap removing (i, a) and (j, b) and adding (i, b) and (j, a); accept if the objective does not increase.

We run 8×10^5 iterations per code. Edge swaps preserve codeword length (and, for biregular codes, neuron degree), so the structural properties of the family are kept fixed while the overlap distribution is optimized.

E THE COMPUTED-FEATURES CRITERION: DETAILS AND ROBUSTNESS

In this appendix, we elaborate on the criterion for superposition introduced in Section 4.4.

Definition. For each feature i, let ℓ_i be the respective per-feature L^4 loss on the training distribution. As the reference $\ell_{\mathrm{ref},i}$ we take the strongest of three terms: the best constant predictor, $\min_c \mathbb{E}\left[(\mathrm{ReLU}(x) - c)^4\right]$; the best width-$N$ linear network, which we fit numerically (see below); and the affine part of the trained network's own input–output function (see below). We say a network computes a feature i when it eliminates at least a fraction θ of the reference loss, $r_i = 1 - \ell_i / \ell_{\mathrm{ref},i} > \theta$. We set $\theta = 0.1$ (our criterion is robust to this choice, see below), and write C for the number of computed features.

Why these references. We make this comparison because computation in superposition requires implementing more non-linear functions than there are non-linearities available, and constant, linear, and affine maps are the solutions that implement none. A feature that beats them is then one that received a share of the network's non-linear capacity.

ReLU networks can be linear end to end. $\mathrm{ReLU}(z) - \mathrm{ReLU}(-z) = z$ holds for every z. Take a pair of neurons with opposite encoder rows, v and $-v$, and give them opposite decoder columns, u and $-u$. The pair's joint contribution to the output is $u\left[\mathrm{ReLU}(v^\top x) - \mathrm{ReLU}(-v^\top x)\right] = u\,v^\top x$: the first neuron passes the positive part of $v^\top x$, the second the negative part, and their difference reconstructs it. Each pair therefore contributes the rank-one linear map $uv^\top$, and the nonlinearities cancel identically, which means that the network's input–output map is linear even though every neuron is a ReLU. With N neurons one can form $\lfloor N/2 \rfloor$ such pairs, and sums of $\lfloor N/2 \rfloor$ rank-one terms realize the linear maps of rank up to $\lfloor N/2 \rfloor$. This is what "the best linear network realizable at width N" means: the best linear map of rank at most $\lfloor N/2 \rfloor$.

Why the linear reference is needed. The target decomposes as $\mathrm{ReLU}(x) = x/2 + |x|/2$. A linear map can reproduce the linear part $x/2$ (up to the rank constraint), while a constant predictor cannot track the input at all. In the sparse regime this makes the linear reference better than the constant predictor. In fact, at the main operating point it does $3.4\times$ better than the best constant. Hence, considering only the best constant could in principle overstate the number of computed features.

Fitting the linear reference. For each cell (p, N) of the sweep we parameterize the linear map as $W = AB$ with $A \in \mathbb{R}^{F \times \lfloor N/2 \rfloor}$ and $B \in \mathbb{R}^{\lfloor N/2 \rfloor \times F}$, and fit it with Adam (learning rate 0.01, cosine annealing, $4{,}000$ steps, batch size $8{,}192$) under the same per-feature L^4 objective and input distribution as the trained networks, evaluating per-feature losses on 4×10^5 fresh samples. The per-feature losses are close to uniform, and we use the mean for this term of the reference.

Dense inputs let ReLU networks emulate affine maps. Write $\mathrm{ReLU}(z) = z/2 + |z|/2$. When inputs are dense, each preactivation $z_j = w_j^\top x$ sums contributions from many active features, so $|z_j|$ concentrates around its mean, and each neuron contributes approximately $z_j/2 + \text{const}$: a rank-one linear term plus an offset. The network as a whole then approximates an affine map $y = Wx + c$ with $\mathrm{rank}(W)$ up to N and an emulated offset, even though the architecture has no bias parameters. In fact, we observe that at $p = 1$ the trained networks are nearly affine. An affine fit $y \approx Ax + b$ to their input–output function captures 90–95% of the output variance and matches their task loss ($1.0\times$ at $N \in \{50, 33\}$), with a full-rank linear part and mean offset ≈ 0.3. Under a constant-and-linear-only reference these networks would count more than N features as computed, which is why we must enhance our reference class with affine references.

The affine part of the network. For each trained network we fit its *affinization* — the best affine approximation $y \approx Ax + b$ of its input–output function, by least squares on 4×10^4 on-distribution samples — and evaluate this affine map's per-feature loss on the task, on 4×10^5 fresh samples. A feature counts as computed only when the network also beats its own affine part by θ: otherwise the feature's performance is attributable to an affine map, which does not constitute evidence of the non-linear computation required by superposition. As expected, this term is relevant for the denser regimes we study and not for the sparse ones. For the squared-error re-scoring below we evaluate the same affinization under squared error.

Solutions without superposition score at most N. Constant, linear, and affine solutions score $C = 0$ by construction, since a map cannot beat its own affine part. The naive solution computes exactly its $\min(N, F)$ represented features: on those, exact ReLU outputs beat every reference, including the naive solution's own affinization, because ReLU is not affine; on the dropped features it outputs zero and beats nothing. The emulate-bias baseline likewise scores exactly its represented features, since its remaining outputs are constants. Consistently, the trained networks in the dense rows of Figure 8, which degrade toward naive-like solutions, score exactly $C = N$. We note that this is not exhaustive: there could in principle be other solutions without superposition that compute more than N features, but we have not found any.

Robustness at the main operating point. The count $C = 100$ at $(F, N, p) = (100, 50, 0.02)$ is robust to the choice of θ: any value up to 0.85 gives $C = 100$. The criterion scores how well features are computed with the same metric the network was trained under, L^4, but our conclusions hold under squared error as well: re-scoring ℓ_i and the references with squared error still gives $C = 100$ for any margin up to 0.14, while the L^2-trained naive solution scores exactly $C = 50$ at every θ.

Sweep details and robustness. Every cell of the sweep in Figure 8 is trained with the methodology of Section 3, with 5 seeds per cell. The counts shown are means over seeds. Varying θ from 0.05 to 0.2 moves the grid-average count by about ± 4 features around its value at $\theta = 0.1$, and flips the superposition verdict $C > N$ only inside the transition band $0.12 \leq p \leq 0.48$: the sparse region ($p \leq 0.08$) and the dense region ($p \geq 0.64$, none) are unchanged for every θ in this range. The θ-sensitivity of the transition band has a mechanistic reading: its narrowest cells settle on evenly-shared but nearly affine solutions — the affinization matches their loss to within 0–10%, and at $(p, N) = (0.32, 20)$ even beats it — so every feature hovers at the margin, and the counts there mean "marginal non-linear benefit spread over all features" rather than a specific set of computed features. In every cell with fewer neurons than features, the trained network beats the naive, emulate-bias, and do-nothing solutions in mean per-feature L^4, by $187\times$ in the sparsest, widest such cell down to $1.1\times$ in the densest cells. This is consistent with superposition bringing a large advantage when inputs are sparse and a vanishing one when they are not. With $N \geq F$ no compression is required and the naive solution is exact, so no advantage over it is possible there.