TOY MODELS OF COMBINATORIAL INTERPRETABILITY

Anonymous authors
Paper under double-blind review

1 INTRODUCTION

Recent research in neural network interpretability often views computation through high-dimensional activation spaces, analyzing directions corresponding to computed features. This geometric perspective has revealed phenomena such as polysemanticity, the superposition hypothesis (11), and strong empirical evidence of correlations between features and activation vector directions (2; 3; 4; 6; 9; 10; 12; 21; 22; 27; 28). While these techniques ingeniously capture feature representations and some interdependencies, they have yet to fully elucidate a complete dependency graph, the underlying functions being computed by the networks, or their precise computational mechanisms.

This paper introduces a combinatorial approach to interpretability, contrasting with the prevailing geometric view of activation spaces. Our combinatorial view focuses on the sign based categorization (positive, negative, or zero) of learned parameters, rather than their precise values. [1] While not mutually exclusive with the vector space perspective, this combinatorial approach offers insights into facets of interpretability—such as the logic of the circuits that compute features—that are challenging for vector space methods. We demonstrate, initially on small examples but with potential for future scalability, that our approach can directly extract features and their exact computations from the network's learned parameter structure. This extraction occurs without requiring auxiliary models like sparse autoencoders or transcoders (3; 9; 12; 22; 27). Essentially, our work aims to reveal the low-level computational mechanics—the how—going beyond merely identifying the when and where of feature relationships.

We focus on the computation of Boolean expressions—a meaningful class that facilitates initial exploration of such combinatorial representations. This allows us to propose a theory explaining how neural networks, at least for Boolean formulae, compute by representing features and their computations as combinatorial structures encoded in their weights and biases. Specifically, our theory explains phenomena like polysemanticity and superposition as arising from feature-associated codes.

Central to our combinatorial approach is the *feature channel coding hypothesis*. This hypothesis specifies how features are represented (analogous to the superposition hypothesis) and, crucially, how computation proceeds between them (a novel aspect of our theory). Under this hypothesis, a feature in a given layer is represented by a "code"—a specific subset of neurons at that layer. The "bits" of this code identify the neurons whose activation contribute to that feature being active. Minimal overlap between the codes of different features at the same layer results in nearly orthogonal activation vectors (when considering precise values), mirroring the superposition hypothesis.

The key differentiator of feature channel coding is its explanation of how computation works. To compute a feature f_i, all prior-layer input features necessary for its computation are mapped to f_i's code. The network then performs the computation for f_i on each neuron in this code. For example, Boolean $f_i = x_{i_1} \wedge x_{i_2}$ is computed as $\text{ReLU}(x_{i_1} + x_{i_2} - 1)$). Thus, when the required input features from the prior layer are active, each neuron in f_i's code activates via its individual computation, collectively activating feature f_i. If a prior-layer feature contributes to multiple current-layer features, it is mapped to each of their respective codes. Crucially, a feature's code is primarily a mechanism for routing and congregating its necessary inputs, rather

[1] One could categorize to more than three classes, but we are starting with the simplest, which as you will see, already exposes lots of structure.

than directly encoding the function itself. While code overlap can introduce noise, computations remain robust if this overlap is minimal, yielding a sufficiently good approximation to the target result. Identifying these codes provides a description of the network's computation directly from its weight matrices, bypassing the need to learn neuron directionality in activation space.

We focused on neural computation of Boolean formulae for several reasons. Theoretical work on computing Boolean functions in superposition (1; 23; 35) initially motivated our feature channel coding hypothesis. Specifically, (1) presents a provably correct feature channel coding algorithm for the AND function. This led us to investigate whether neural networks, through training, could converge to similar algorithms—a question this work answers affirmatively. More significantly, Boolean formulae serve as an ideal experimental setting due to their straightforward data generation, quantifiable dataset sizes, and measurable training challenges like overfitting and generalization. They also allow for for for precise control over feature count and interaction complexity. We use this precision to provide evidence of how scaling laws are explicitly linked to feature capacity.

Our findings provide strong evidence that the simple networks we trained to compute Boolean formulae use a soft Boolean logic (33). In this paradigm, real-valued variables x_i and x_j undergo operations like $x_i \wedge x_j$, computed as $\text{ReLU}(x_i + x_j - b)$, where b is an adjustable threshold. The inherent ReLUed dot products in neural computation naturally facilitate these differentiable, soft Boolean operations. More speculatively, we hypothesize that soft Boolean formulae are fundamental to many complex tasks where neural networks excel.

In summary, we show:

- A novel combinatorial approach to interpretability that looks at the weights rather than activations of networks, and contrasts with the prevailing geometric view.
- The *feature channel coding hypothesis* that specifies how features are represented and, crucially, how they are computed by neural networks.
- Strong evidence that the simple networks we trained compute Boolean formulae via feature channel coding of soft Boolean logic.
- Examples of how the combinatorial approach can be used to analyze feature emergence during training and to explain how scaling laws are a consequence of the ability to code features.

Before continuing, we wish to acknowledge that our work, like Anthropic's (11)), uses greatly simplified settings without showing how they extend to general LLMs. Metaphorically, as scientists we understand that most phenomena shown in peas (Pisum Sativum) do not extend to humans (Homo Sapiens), but as it turns out the peas' seemingly simplistic genetic principles do... The problem of fully interpreting neural computation is hard, which is why we are initially focusing on much simpler settings in which we can show full interpretation for the first time. The hope is that these can be used to establish principles that are applicable in the large.

2 Background and Related Work

Early research in neural network interpretability viewed neurons as geometric vectors in activation space, associating neurons with single semantic directions (25; 32). However, it became evident that neurons often encode multiple unrelated features, known as *polysemanticity* (7; 11; 11; 13; 31), complicating interpretation (14; 15). This phenomenon relates closely to *superposition* (11), where there are more features than neurons. Polysemantic neurons have been empirically observed in large architectures, such as multimodal models like CLIP, where neurons simultaneously respond to disparate concepts (13). This overlap represents an intentional strategy for efficient capacity allocation (11). This capacity allocation problem was isolated and studied in (31). Theoretical analyses of combinatorial coding highlight this efficiency through explicit constructions (1; 16; 23) and lower bounds (1).

Recent advances in the use of autoencoders for network interpretability have led to enhanced methods such as sparse autoencoders (SAEs) (7) and transcoders (9) for isolating feature vectors in activation space (2; 3; 4; 6; 10; 12; 21; 22; 27; 28). These techniques allow the creation of *attribution graphs* that map the dependencies among features for the computation of a specific input (3). Yet, these attribution graphs are best described as targeted approximations of the original model, rather than elucidating the underlying computational processes. They capture which features influence subsequent features *for a specific input*, but they do not capture (a) the overall feature dependence (across all inputs), (b) the function that is being computed to realize that dependence, and (c) the underlying mechanism for computing that function.

Furthermore, these SAE-like techniques require new trained components that inherently simplify or omit certain aspects of the complexity of the original model. While this helps in understanding the feature landscape, it is fundamentally a reconfiguration rather than a direct observation of the original model's internal representation. In addition, recent work has raised doubts concerning the generalization ability of such approaches (17). Consequently, while SAE-like techniques are valuable for gaining insights into model behavior, they do not capture all the details and interactions of the original model. In contrast, our combinatorial interpretabiliy approach, though at a much earlier phase, aims to directly interpret the computation from analyzing the structure of the model itself.

Previous interpretability work has reversed engineered the underlying computational mechanics of specific functions, although they do not reveal complete circuits as we do in this work. Research on 2SAT (26) showed that transformers solving these Boolean formulae learn internal circuits that resemble symbolic algorithms. Nanda et al. (24) observed a discrete Fourier transform-like algorithm in networks performing modular arithmetic. Furthermore, theoretical frameworks such as the Information Bottleneck (34) and statistical mechanics approaches (5) offer complementary insights into how neural representations encode data and manage correlations. Superposition is also connected to scaling laws, suggesting that larger models might reduce polysemanticity by developing more specialized neuron representations as resources increase (18; 19; 30).

3 Neural Networks Learn Feature Channel Coding

An open question from (1) is whether feature channel codes occur naturally in networks trained via gradient descent; in other words, do SGD-trained neural networks learn to utilize such coding? We demonstrate that these codes indeed emerge naturally from gradient descent training. To illustrate this we first consider a two-layer MLP configured with 16 inputs, 16 first-layer neurons (each with a bias), and a single second-layer neuron (without bias). The network was trained to learn various hidden Boolean functions. Initially we studied randomly selected functions from the set of all possible pairwise ANDs of 16 Boolean variables, where variables were paired such that each appears in exactly one AND. We here present a typical result, where the network was trained[2] on the formula described in Figure 1.

Figure 1 (upper right) displays the complete set of learned parameters from a representative training run: Layer 1 weights (W_1), Layer 1 bias, and Layer 2 weights (W_2, oriented upright for readability). Colors indicate weight sign (blue: positive, red: negative). Let us first examine W_1. In its raw, unaltered form (columns corresponding to input features x_i, rows to first-layer neurons), W_1 offers little immediate insight, beyond neuron 2 appearing largely inactive. However, Figure 1 (upper left) shows W_1 with its columns sorted by features, grouping variables that are conjoined by an AND (e.g., columns 3 and 11 for $(x_3 \wedge x_{11})$). This sorted view begins to reveal an interesting pattern: in many, though not all, positions, adjacent columns corresponding to these AND-pairs have similar values.

[2]Specifically, the network was trained using 30,000 random inputs, each equally probable as a True or False instance of the formula. Each input contained between two and six True variables, also chosen with equal probability. The training utilized the BCE loss function and the Adam optimizer with a learning rate of 0.01, conducted for 2000 epochs.

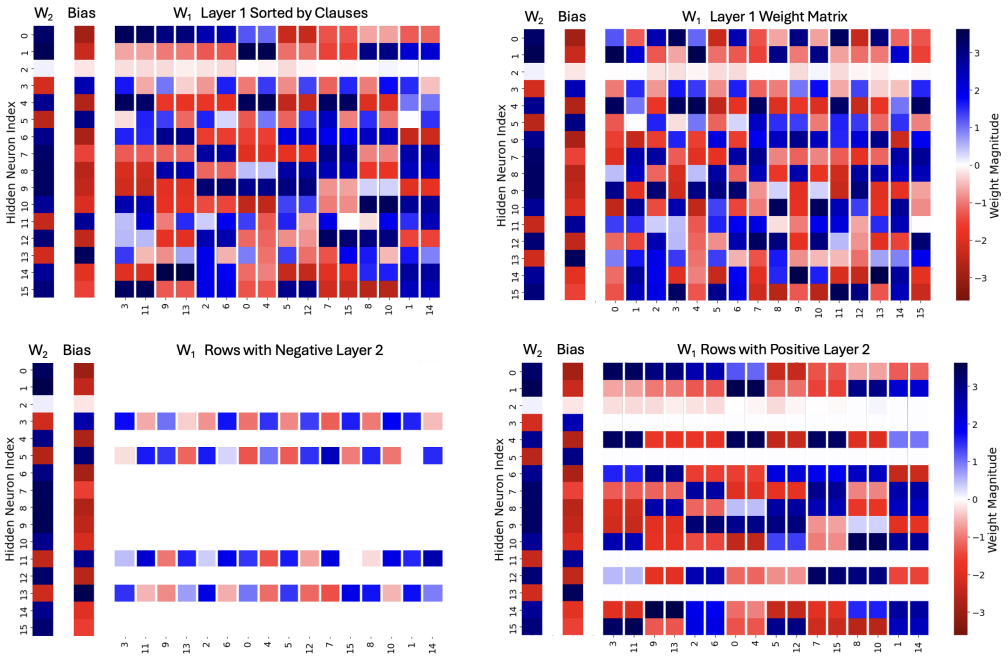

Figure 1: All weights and biases for a neural network trained on the Boolean formula $(x_3 \wedge x_{11}) \vee (x_9 \wedge x_{13}) \vee (x_2 \wedge x_6) \vee (x_0 \wedge x_4) \vee (x_5 \wedge x_{12}) \vee (x_7 \wedge x_{15}) \vee (x_8 \wedge x_{10}) \vee (x_1 \wedge x_{14})$

In Figure 1 (lower right), W_1's features are sorted identically, but we now filter to include only rows (neurons) corresponding to positive Layer 2 weights (this filtering is justified below). This visualization is striking: in every row, the columns for each AND pair are virtually identical. Furthermore, observing the positive (blue) weights reveals that every pair of input variables appearing in the same AND clause is represented by a feature channel code. The features computed by the first layer of this network are exactly the pairwise ANDs of the Boolean formula. The neurons comprising the codes for each of those features allows the AND features' computation to propagate to the next layer where the OR of those features is computed. For example, input features x_3 and x_{11}, used to compute $(x_3 \wedge x_{11})$, are each mapped to the feature channel code consisting of positive weights on neurons $\{0, 4, 6, 10, 12, 15\}$. Similarly, the feature $(x_9 \wedge x_{13})$ is represented by the code $\{0, 6, 8, 14\}$, and $(x_5 \wedge x_{12})$ by $\{6, 8, 9, 10, 15\}$. We can formalize this as follows (further formal modeling can be found in the appendix):

Definition Let $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ denote the set of features represented at a given layer of a neural network with n neurons. A *feature channel code* for a feature $f \in \mathcal{F}$ is a subset $C(f) \subseteq [n]$ of neuron indices, together with a Boolean (or soft Boolean) function φ_f defined on the activations of those neurons, such that the activation of f at that layer is given by

$$f(x) = \varphi_f\big(\{a_j(x) : j \in C(f)\}\big),$$

where $a_j(x)$ is the pre-activation or post-activation value of neuron j on input x. Different features f_i, f_j may have overlapping codes $C(f_i) \cap C(f_j) \neq \emptyset$, leading to polysemantic neurons. The collection $\{C(f) : f \in \mathcal{F}\}$ constitutes the *feature channel coding* of the layer, representing how the network routes subsets of input features onto shared groups of neurons for computation.

Figure 1 reveals that every neuron k computes $x_i \wedge x_j$ with soft Boolean logic via $a_k() = \text{ReLU}(x_i + x_j - b)$ for some b (regular Boolean variables would have $b = 1$). For every row (neuron) with a positive Layer 2 weight, the Layer 1 bias is significantly negative, representing this '$-b$' threshold. Thus, if two input features in the same AND clause (e.g., $(x_3 \wedge x_{11})$) are both True, their sum (calculated by the dot product of neuron weights with the input) will overcome the negative bias. This yields a positive value after the ReLU for all neurons in the pair's feature channel code, consequently leading to a "True" output from the network's sigmoid.

Note that the first layer of this network is polysemantic. For example, neuron 4 responds positively to four different AND features: $(x_3 \wedge x_{11})$, $(x_0 \wedge x_4)$, $(x_7 \wedge x_{15})$, and $(x_1 \wedge x_{14})$. Thus, neuron 4's activation will be similar when the input consists of 1s for only variables x_3 and x_{11} (a True instance for f) compared to 1s for only $x_3 = 1$ and $x_4 = 1$ (a False instance for f, as $(x_3 \wedge x_4)$ is not a target clause). Feature channel coding is crucial here. The outcome for a given AND output feature depends on all neurons in its specific code. These codes are learned to ensure that even if they overlap for some neurons, these overlaps are limited. For instance, the codes for input x_3 and input x_4 overlap in neurons 0 and 4. Consequently, if only x_3 and x_4 are 1s, these two neurons will produce some positive signal. However, this signal is substantially weaker than when $x_3 = 1$ and $x_{11} = 1$, because the feature channel code for the $(x_3 \wedge x_{11})$ pair activates six neurons, yielding a much stronger collective response.

However, since some positive signal does get through even with the False instance, we see that the negative rows play a role in the computation. We provide quantitative evidence of this below for various different problems; the degree of importance of the negative rows seems to depend on the specifics of the problem being computed. Here, we see that a strong negative signal is provided by neuron 3 in the case where the input is 1s only on variables x_3 and x_4. Neuron 3 activates because its W_1 weights for both x_3 and x_4 are positive. This activation, when multiplied by its negative W_2 weight, yields a strong negative input to the second layer, counteracting positive signals from neurons 0 and 4 that arise due to feature channel code overlap. However, when the input is 1s only on x_3 and x_{11}, that negative signal is greatly diminished by the fact that row 3 of the Layer 1 weight matrix has opposite signs in columns 3 and 11. This explains why the network has a strong learned preference for the inputs that appear in the same clause to have opposite Layer 1 weights for negative rows.

This observed mechanism motivates the classification of W_1 rows based on the sign of their corresponding W_2 weights (visualized in Figure 1, bottom panels). The sign of a W_2 weight dictates whether its associated Layer 1 neuron's activation contributes positively or negatively to the network's final output. We call Layer 1 neuron activations that are positively weighted by W_2 *positive witnesses*; they provide evidence supporting a True instance with a strong positive signal. Conversely, we call activations negatively weighted by W_2 *negative witnesses*, providing evidence for a False instance through a negative signal. The visualization (Figure 1, lower left) suggests that negative witnesses effectively compute an XOR of variables within a target clause: if one variable of the pair is present but its counterpart is absent, it signals a False instance.

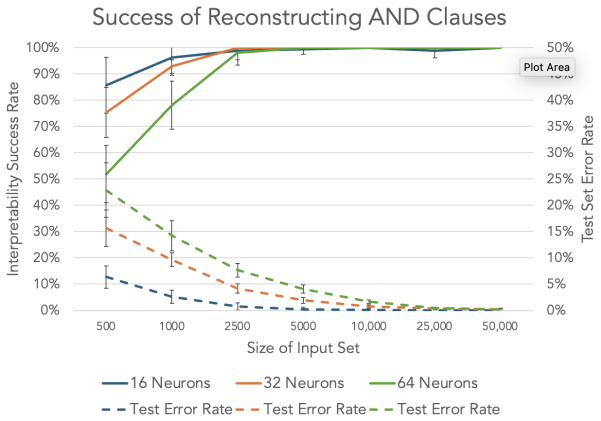

Figure 2: Formula recovery from Layer 1 weights

Our methods extend beyond structure, enabling recovery of the underlying Boolean formula directly from learned parameters. In the pairwise AND case, the

Layer 1 weight matrix suffices: take absolute values, then for each column i find the column j with highest correlation; if j is top-correlated with i, we infer the pair (i, j). (Results need not be symmetric.) Figure 2 illustrates performance when varying hidden size, formula size, and training set size. For each configuration we trained ten random formulas, measured test error on new inputs, and compared against the correlation algorithm's pairing accuracy (error bars show standard deviation). With sufficiently large training data, the algorithm almost always reconstructs the formula, and "sufficiently large" coincides with when the network itself achieves low test error. Appendix E.2 further demonstrates this approach for the four-consecutive-ones problem.

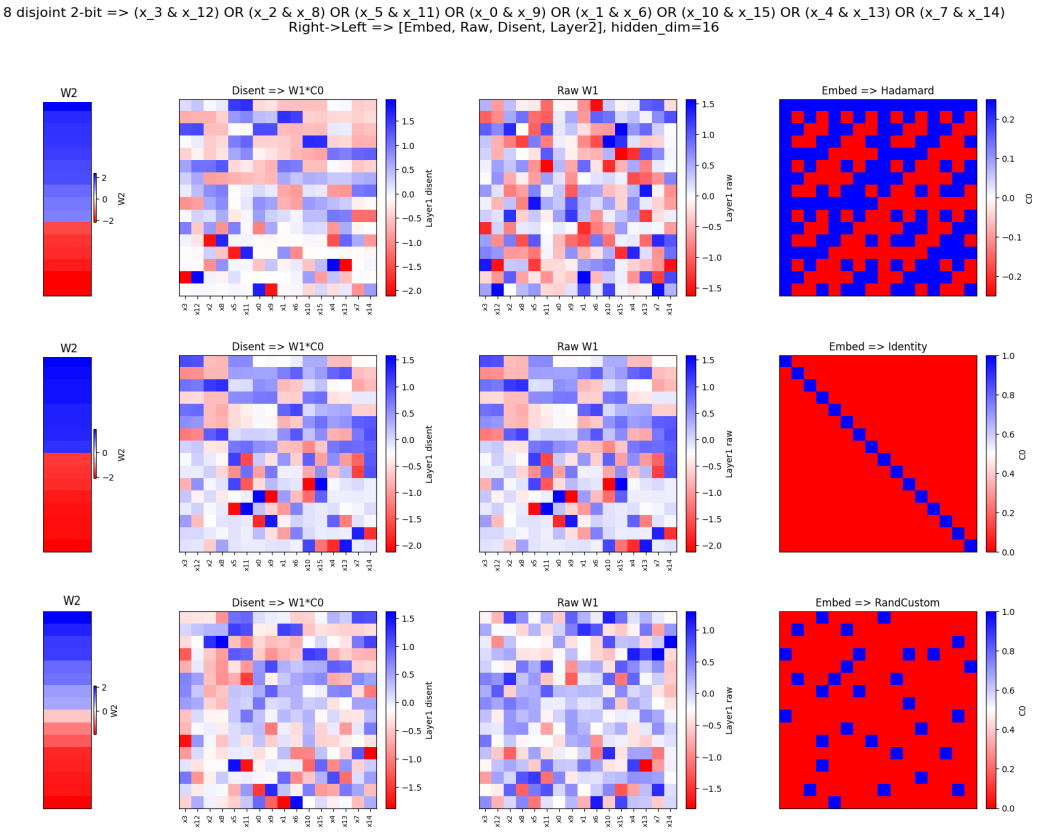

Figure 3: Disentangling to get code from a network with an initial input embedding: Hadamard embedding on top, the identity embedding in middle, and symmetric random at the bottom.

Finally, we demonstrate that that feature channel coding appears also if we add an embedding layer to the two layer network we described earlier in the paper as in real world networks. Specifically, Figure 9 depicts three example networks trained via gradient descent. For each, we show the layer 2 neuron W_2 on the left (with its weights sorted), the raw first layer weight matrix W_1 to its right, the disentangled weight matrix C_1 further to the right, and in the rightmost position the embedding being used: in the middle row, the embedding C_0 is the identity function (i.e., no embedding as in our earlier example in Figure 1), in the top row it is a Hadamard matrix, and in the bottom row it is a symmetric random embedding. The embedding forces the network to learn an entangled weight matrix, hence when it is not the identity as in the top and bottom rows, W_1 show no coding structure. However, as we explain in more theoretical detail in the

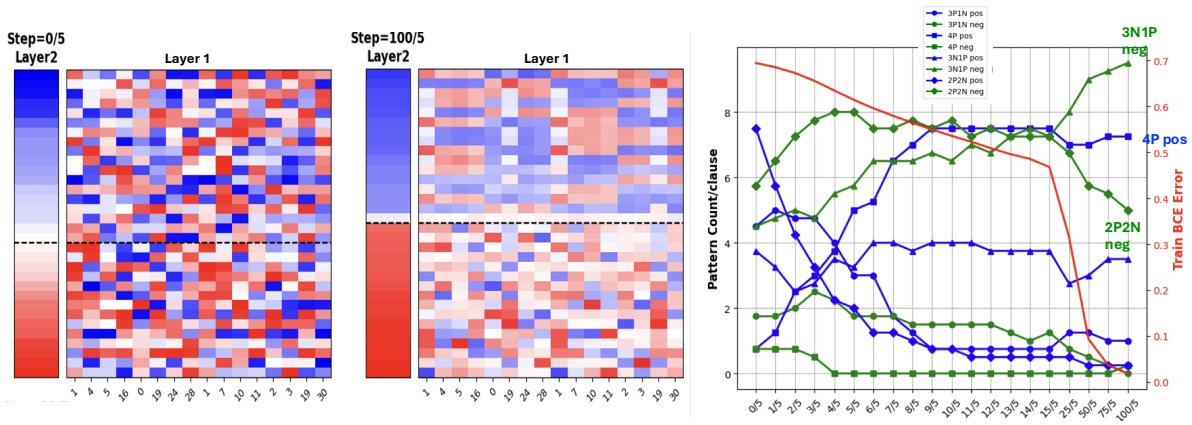

Figure 4: Emergence of codes and decrease in error during 20 epochs of training. Initial state left, final state middle, and the change in average code pattern per clause from initial to final on the right.

Appendix, W_1 can be viewed as $W_1 \approx C_1 C_0^*$, where C_0^* is the left inverse of C_0. The network is learning the mathematical equivalent of disentangling the embedding and then coding the features in the disentangled space of the C_1 weights. The disentanglement in W_1 can be undone to obtain C_1 by the product $W_1 C_0$ since $W_1 C_0 \approx C_1 C_0^* C_0 \approx C_1 C_0^* C_0 \approx C_1$. In our case, one can clearly see that the rows in C_1 corresponding to positive witnesses code ANDs using two positive weights per clause, and the negative witness rows code XORs as in our earlier example. In short, codes appear in disentangled space even if we cannot see them in the raw weight matrices, a phenomenon that has interesting implications on understanding neural network computation and has already been used based on this work to understand important phenomena like sparsity. We also believe that the sparse auto-encoders (27) in the literature are building approximations of C_{i-1}^* for arbitrary layers i of MLPs and transformers.

For lack of space, we delegate to the appendix more empirical proof showing that feature channel coding allows the neural network to compute in "superposition" (1). We show it appears when the number of features encoded is much larger than the number of neurons. We also demonstrate a number of other examples of feature channel coding, for DNF formulae with negated variables, for CNF formulae, and for a one-dimensional vision problem.

4 Feature Channel Codes Emerge During Training

One advantage of the combinatorial approach is the ability to detect patterns that would be hard to define and track for vectors in high dimensional geometric space. We used this advantage to see if and how feature channel coding emerges (Figure 4) during training with SGD. We study the alignment between Boolean formula clauses and specific patterns of positive and negative values within the Layer 1 weight matrix.

For clarity we show here one specific example execution though we conducted many that conform with what is seen here. We trained for 20 epochs a network with 32 neurons on a DNF with 4 clauses with 4 variables per clause, and looked at the emergence of patterns in the weights associated with the clauses as the network learns. The left side is the initial 0-epoch heat map of W_1, the middle is the final W_1 after 20 epochs, with the x-axes sorted by the variables of the 4-clauses. We look at the 4-weight patterns for each clause and categorize them into five types based on the number of positive (P) and negative (N) weights corresponding to a clause's four variables in a given row (neuron): 4P (4 positive), 4N (4 negative), 3P1N (3 positive, 1

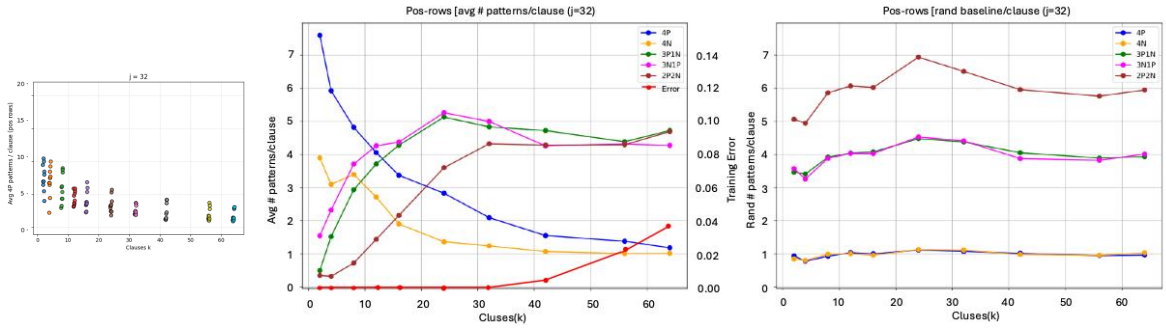

Figure 5: Prevalence of coding patterns in positive rows. The 4P pattern, in a blue line in middle plot, emerges as totally non-random. For better clarity on error, instead of error bars, the actual distribution of the ten 4P counts are given on the left. As can be seen, when there is sufficient room for multiple coding rows per clause, there is some variability on the size of the codes, but as code capacity becomes a constraint, the distribution becomes more concentrated.

negative), 3N1P (3 negative, 1 positive), and 2P2N (2 positive, 2 negative). For example, 3P1N signifies that a clause's four variables align with a weight matrix row containing three positive and one negative value. The graph on the right depicts the various pattern counts per clause, the error in red. The x-axis non-linearly shows the first 3 epochs in more detail in 1/5 epoch intervals. Initally, in the random initial state, 4P and 4N patterns are the least prevalent, as expected about 1 per clause, and 2P2N is the most prevalent, close to 8 per clause. During training, positive 4P patterns rapidly increased, while other positive patterns diminished. Similarly, negative patterns, except 3P1N and 2P2N, decreased. By epoch 2, the dominant 4P positive coding pattern that we already showed corresponds an AND stabilized. However, training (BCE) error remained around 0.55 until negative witness 2P2N patterns transitioned to 3N1P, which was critical for achieving final accuracy. The final pattern (middle heatmap where clear 4P positive and 3N1P negative patterns are visible) is similar to the results of the trained graph of Figure 5 which we describe shortly. Notably, the rapid rise of 4P positive patterns before the first epoch underscores their importance and ease of discovery by gradient descent. While positive patterns converged quickly, the subsequent refinement of negative row patterns (flipping 2P2N to 3N1P) was essential for solving the task, demonstrating a multi-stage learning process where different pattern types become critical at different phases of training. "Why and how gradient descent converges to feature channel codes" is a fascinating research topic that the combinatorial setting could enable.

Additional results are in the appendix. In particular Appendix B presents a formal model of combinatorial interpretability and initial steps for disentangling multi-layer superposed neural networks. Appendix C examines single layer networks with multiple outputs. Appendix D provides more details on the quantitative analysis from Section 5. Appendix E, offers more examples of feature channel coding and combinatorial network interpretation. Finally, we outline future research directions.

5 Combinatorial Analysis of a Scaling Law

This section employs our combinatorial interpretability framework to investigate scaling laws governing *how* models compute Boolean formulas and *why* they fail with increasing complexity.

We focus on DNF formulas composed of 4-variable clauses where all variables are positive (results for clauses with 3 positive and 1 negated variable are in Appendix D.4). As depicted in Figure 5, we achieve this by

progressively increasing the number of clauses in the formula, starting from a small number $k = 2$, until the model can no longer learn it accurately with $k = 64$ clauses). This modification increases the complexity of the underlying truth table of the Boolean formula, leaving the network architecture and input size unchanged. Specifically, all experiments use a network with the structure described above with 32 input variables, 32 neurons, and a single sigmoid output neuron. All results are averaged over 10 trials, each with a distinct random Boolean formula. Further training details and deeper insights into network learning are provided in Appendix D.

Figure 5 provides quantitative evidence for feature channel coding and illuminates the network's learning capabilities and limitations. As before, each pattern type is represented by a line in the figure, where the vertical axis indicates the average number of positive witness Layer 1 weight matrix rows (associated with positive Layer 2 weights) exhibiting that specific pattern for a given clause. The middle graph in the figure shows results for positive witness rows, indicating that with few clauses, 4P patterns are abundant (e.g., over 7 per clause for $k = 2$), but their frequency declines as k increases, with error (red) rising correspondingly. Error growth begins once the average 4P count per clause drops below $\sim$1.5–2, consistent with the feature coding hypothesis: below this threshold, clauses cannot be reliably coded. This decline stems from the weight matrix's limited capacity to both (a) allocate many 4P patterns across clauses and (b) preserve $\sim$50% negative weights per row. Increasing the fraction of positive weights would retain 4P patterns but compromise discrimination between four positives within one clause versus spread across clauses.

The middle graph of Figure 5 compares trained networks to those with random Layer 1 weights. Trained pattern frequencies clearly deviate from chance: patterns enriched relative to random aid learning, while depleted ones are uninformative. At low k, both 4P and 4N patterns are far more frequent than random; as k grows, 4N frequency converges to random around $k = 30$, while 4P frequency, though reduced from $> 7\times$ random, remains above random even at $k = 64$. This supports the feature coding hypothesis: 4P patterns encode clauses across multiple rows, while 4N patterns offset them, enabling discrimination between four positives within one clause versus across clauses. Additional evidence, including prevalence of 3N1P codes in negative witness rows, is provided in the appendix.

Next, in Figure 6, we demonstrate a similar capacity limit from a more traditional scaling law viewpoint: we keep the number of features/clauses fixed at $k = 64$ and increase the size of the hidden layer along the x-axis. We plot both the error and the frequency of 4P codes in positive rows (blue) and 3N1P codes in negative rows (green) versus the size of the hidden layer. We see that 4P and 3N1P code frequency (solid lines) is much higher than random (dotted lines) and that the decrease in error is tightly correlated to this increase in code frequency. We believe that this relationship is due to the capacity of the network to perform feature channel coding, where, as before, the network roughly achieves that limit.[3] This provides an easily explainable scaling law: the network's size implies a capacity to code the feature channels, with smaller networks being more limited in their coding ability, which explains the higher error rate.[4]

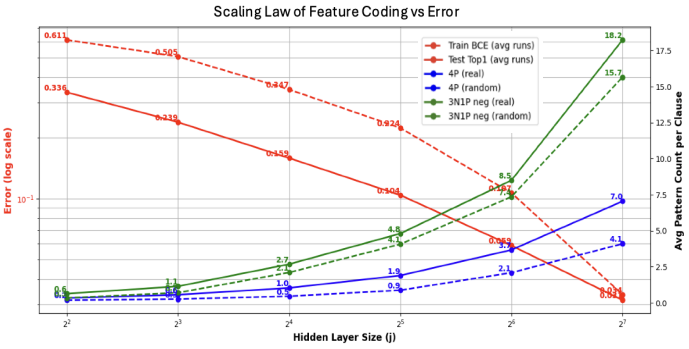

Figure 6: Feature channel coding provides combinatorial explanation for scaling as hidden layer size increases.

[3] Again, with a sufficiently large hidden layer, the network does not need to take full advantage of the space.

[4] Note that we trained here to half the epochs of prior examples and there is a larger error per model size.

REFERENCES

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

[2] Emmanuel Ameisen et al. Circuit tracing: Revealing computational graphs in language models. `https://transformer-circuits.pub/2025/attribution-graphs/methods.html`, 2025.

[3] Emmanuel Ameisen, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, Brian Chen, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits*, 2025.

[4] Thomas Bricken et al. Towards monosemanticity: Decomposing language models with dictionary learning. *arXiv preprint arXiv:2301.00001*, 2023.

[5] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4):045002, 2019.

[6] Hoagy Cunningham et al. Sparse autoencoders find highly interpretable features in language models. In *International Conference on Learning Representations*, 2024.

[7] Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

[8] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[9] Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable LLM feature circuits. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[10] Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable llm feature circuits. In *Advances in Neural Information Processing Systems*, 2024.

[11] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. `https://arxiv.org/abs/2209.10652`, 2022.

[12] Xuyang Ge, Fukang Zhu, Wentao Shu, Junxuan Wang, Zhengfu He, and Xipeng Qiu. Automatically identifying local and global circuits with linear computation graphs. *arXiv preprint arXiv:2405.13868*, 2024.

[13] Gabriel Goh, Nick Cammarata, Ludwig Schubert, Chris Olah, and Shan Carter. Multimodal neurons in artificial neural networks. *Distill*, 2021.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[15] Tal Haklay, Hadas Orgad, David Bau, Aaron Mueller, and Yonatan Belinkov. Position-aware automatic circuit discovery, 2025.

[16] Kaarel Hänni, Jake Mendel, Dmitry Vaintrob, and Lawrence Chan. Mathematical models of computation in superposition. *arXiv preprint arXiv:2408.05451*, 2024.

[17] Lovis Heindrich, Philip Torr, Fazl Barez, and Veronika Thost. Do sparse autoencoders generalize? a case study of answerability. *arXiv preprint arXiv:2502.19964*, 2025.

[18] Tom Henighan, Shan Carter, Tristan Hume, Nelson Elhage, Robert Lasenby, Stanislav Fort, Nicholas Schiefer, and Christopher Olah. Superposition, memorization, and double descent. `https://transformer-circuits.pub/2023/toy-double-descent/index.html`, 2023. Accessed: 2024-08-13.

[19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[20] Valentin Lecomte, Kanishka Thaman, Rylan Schaeffer, Noah Bashkansky, Tyrus Chow, and Sanmi Koyejo. What causes polysemanticity? an alternative origin story of mixed selectivity from incidental causes. In *ICLR 2024 Workshop on Representational Alignment*, May 2024.

[21] Aleksandar Makelov, George Lange, and Neel Nanda. Towards principled evaluations of sparse autoencoders for interpretability and control, 2024.

[22] Samuel Marks, Can Rager, Eric J Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

[23] Jake Mendel and Lucius Bushnaq. Circuits in superposition: Compressing many small neural networks into one. LessWrong, 2024. Online article.

[24] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

[25] Anh Nguyen, Jason Yosinski, and Jeff Clune. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. *arXiv preprint arXiv:1602.03616*, 2016.

[26] Nils Palumbo, Ravi Mangal, Zifan Wang, Saranya Vijayakumar, Corina S Pasareanu, and Somesh Jha. Mechanistically interpreting a transformer-based 2-sat solver: An axiomatic approach. *arXiv preprint arXiv:2407.13594*, 2024.

[27] Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Tom Lieberum, Vikrant Varma, János Kramár, Rohin Shah, and Neel Nanda. Improving sparse decomposition of language model activations with gated sparse autoencoders. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 775–818. Curran Associates, Inc., 2024.

[28] Senthooran Rajamanoharan et al. Improving dictionary learning with gated sparse autoencoders. In *Advances in Neural Information Processing Systems*, 2024.

[29] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[30] Jonathan S. Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[31] Adam Scherlis, Kshitij Sachan, Adam S. Jermyn, Joe Benton, and Buck Shlegeris. Polysemanticity and capacity in neural networks. *arXiv preprint arXiv:2205.00001*, 2022.

[32] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[33] Naresh K. Sinha and Madan M. Gupta. *Soft Computing and Intelligent Systems: Theory and Applications*. Academic Press, 2000.

[34] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW)*, pages 1–5. IEEE, 2015.

[35] Dmitry Vaintrob, Jake Mendel, and Kaarel Hänni. Toward a mathematical framework for computation in superposition. `https://www.alignmentforum.org/posts/2roZtSr5TGmLjXMnT/\toward-a-mathematical-framework-for-computation-in`, 2024. Accessed: 2024-06-04.

[36] Wikipedia contributors. Invertible matrix. `https://en.wikipedia.org/wiki/Invertible_matrix`, 2025.

Appendix

A LLM USAGE STATEMENT

We used LLMs to collect some but not all of the bibliographical references and to draft an initial version of the related work that was later revised. We also used them to format some of the references we found ourselves.

B MODELING COMPUTATION AND COMBINATORIAL DISENTANGLEMENT

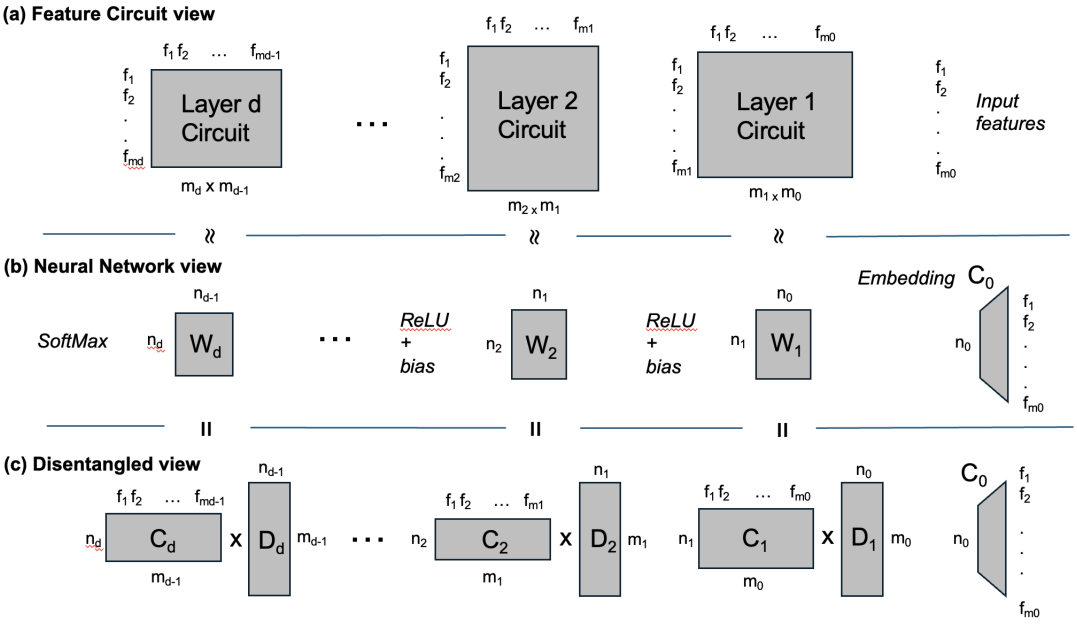

Figure 7: The feature circuit coding-decoding view of computation. Computation flows from right to left. The middle sequence (b) is the actual computation through the neural network. The top sequence (a) is the abstract computation mapping features to features in each layer being represented by the neural network. The lower sequence (c) is the feature coding view that bridges the two by thinking of the the weight matrix as implementing a decoding of the prior layer's features to serve as monosemantic inputs to a coding-and-computation of the features of the new one. $W_i = C_i D_i$ is a superposed $n_i \times n_{i-1}$ representation of the $m_i \times m_{i-1}$ i-th feature circuit layer.

Thus far we have illustrated the combinatorial view of a two-layer network. Real-world models—even plain multilayer perceptrons (MLPs)—are usually richer: they begin with an embedding layer and then stack several nonlinear layers. This section introduces a conceptual model for understanding MLP computation based on underlying "feature circuits." We then propose a novel perspective viewing MLP layers as implicitly performing a decode-compute-encode cycle, aiming to disentangle the network and reveal this circuit. Building on this, we outline a potential strategy, termed *cascading feature disentanglement*, for systematically uncovering the feature circuit layer by layer. We present initial steps demonstrating how the first layer

of computation can be revealed via disentanglement from an input embedding and also propose a general approach for deeper layers.

B.1 The Feature Circuit View of Neural Computation

Our modeling follows the intuitive ideas in (1; 11; 35). We aim to capture the fundamental computational structure implemented by an MLP. We model the computation at each layer i (for $i = 1, \ldots, d$) as producing a set of abstract output features $F_i = \{f_{i,1}, f_{i,2}, \ldots, f_{i,m_i}\}$. Each feature $f_{i,j} \in F_i$ is computed by applying a specific function to a subset of the features $F_{i-1} = \{f_{i-1,1}, \ldots, f_{i-1,m_{i-1}}\}$ from the preceding layer (where F_0 represents the initial input features, m_0 in number). The complete *feature circuit* of the network is the sequence of these computations across all layers, $F_1, F_2, \ldots, F_d$.

Figure 7(a) provides a visualization of such a multi-layer feature circuit. It takes m_0 input features and computes features layer by layer, where the computation of a feature $f_{i,j}$ depends only on a subset of features from F_{i-1}. For instance, in our earlier simple 2-layer network example (a fully connected layer followed by a single neuron computing a DNF of 2-ANDs), the feature circuit might resemble Figure 8. Note that in that figure, the rectangular portion of the matrix depicts the dependencies (using filled in rectangles), and the formulas to the left of the rectangle depict the actual functions being computed.

Figure 8: Two layer feature circuit (as an example of the top row of Figure 7) of the DNF $(x_3 \wedge x_{12}) \vee (x_2 \wedge x_8) \vee (x_5 \wedge x_{11}) \vee (x_0 \wedge x_9) \vee (x_1 \wedge x_6) \vee (x_{10} \wedge x_{15}) \vee (x_4 \wedge x_{13}) \vee (x_7 \wedge x_{14})$. Layer 1 defines the computation of the pairwise AND features and Layer 2 is the single OR feature over the results of the outputs from Layer 1. Together they define the abstract computation that the network in Figure 9 computes.

This abstract feature circuit finds a concrete implementation in a standard MLP, depicted in Figure 7(b). The m_0 input features are first mapped by an embedding matrix C_0 (size $n_0 \times m_0$) to an n_0-dimensional vector. Subsequent layers $i = 1, \ldots, d$ consist of a weight matrix W_i (size $n_i \times n_{i-1}$), a bias vector b_i, and a non-linearity (e.g., ReLU), transforming the n_{i-1} neuron activations from layer $i-1$ into n_i neuron activations for layer i. (While we use ReLU for illustration, other operations like batch normalization could be used instead). A final readout function (e.g., softmax) produces the network's output. Through training (e.g., via gradient descent), this MLP learns to approximate the underlying feature circuit computation.

We will say that a layer i of a neural network *computes in superposition* if $n_i < m_i$, that is, there are fewer neurons than output features in the circuit being computed by this layer. The network as a whole will be said to compute in superposition if its layers compute in superposition. Such networks are sometimes called *polysemantic* because each neuron participates in the computation of more than one output feature, as opposed to monosemantic ones, in which each neuron corresponds to a single output feature.

As (1; 35) show, a neural network can use a polysemantic representation with many fewer neurons than features to compute across multi-level neural networks. Part of our observations in this work suggest that polysemanticity can arise in training even when the network does not need to be in superposition, that is, $n_i \geq m_i$ (similar to results shown in (20)). In some neural networks, such as large language models or convolutional neural networks, the input to the neural network is provided in superposition. In these cases,

the input is encoded into superposition via a first layer that translates text (via tokens) or an image (via downsampling) into an embedding space.

B.2 The Disentangled Feature Circuit View

Our core proposal for interpreting MLPs involves viewing each layer's weight matrix W_i as implicitly factoring into two conceptual matrices, C_i and D_i, revealing an intermediate "disentangled" representation, shown in Figure 7(c). This perspective aims to bridge the gap between the concrete MLP and the abstract feature circuit.

Specifically, we hypothesize that the computation within layer i can be understood as:

1. **Decompression (D_i)**: An implicit matrix D_i (size $m_{i-1} \times n_{i-1}$) acts on the n_{i-1} neuron activations from layer $i-1$. Its role is to transform the m_{i-1} polysemantic features computed by the previous layer (F_{i-1}) into approximations of their monosemantic representations.
2. **Computation and Compression (C_i)**: An implicit matrix C_i (size $n_i \times m_{i-1}$), combined with the bias b_i and non-linearity σ_i (e.g., ReLU), acts on the monosemantic representations. It performs two roles simultaneously, using feature channel coding:
    - It computes the m_i new features for the current layer (F_i) based on the m_{i-1} input features.
    - It *compresses* these m_i computed features back into the n_i-dimensional polysemantic representations for the next layer.

Thus, the effective transformation by the weight matrix W_i in the standard MLP (Figure 7(b)) is conceptually equivalent to the combined operation $W_i \approx C_i D_i$ in our disentangled view (Figure 7(c)). The entire network can then be viewed as a sequence

$$C_d D_d \sigma_{d-1} \dots \sigma_1 C_1 D_1 (C_0 \cdot \text{input}),$$

where σ_i represents the bias and non-linearity applied after the $C_i D_i$ (or equivalently, W_i) multiplication. The actual MLP uses the compact $W_i = C_i D_i$ form, hiding the intermediate dimension of size equal to the number of features at layer i. The names D_i (Decompress) and C_i (Compress) reflect their roles in moving between the compact polysemantic representation and the potentially larger monosemantic representation.

This perspective is similar to concepts like sparse autoencoders and transcoders, where encode/decode operations learn similar transformations. Specifically, the D_i matrix resembles an encoder and the C_i matrix a decoder of a transcoder. However, a key distinction is that C_i and D_i are posited as *intrinsic* components of the *existing* trained weight matrix W_i, not externally added modules learned via a separate objective like in standard autoencoders or transcoders. Our goal is to *recover* this inherent $C_i D_i$ structure. Consequently, the placement of non-linearities and biases in our model follows the original network architecture, differing from typical autoencoder/transcoder setups.

Interestingly, if one ignores bias and ReLu, then notice that the feature circuit for any layer i could be given by $D_i * C_{i-1}$, a matrix of size $m_i \times m_{i-1}$ in which the codes of length n_i are matched between D_i and C_{i-1} yielding the layer i feature circuit matrix. So for example, as depicted in Figure 7, the Layer 1 feature circuit would be given by multiplying D_2 by C_1. This matrix is only an approximation because of the bias and ReLU, whereas $C_1 D_1$ is an accurate representation of the neural network matrix W_1.

B.3 How the Feature Channels Code Computation

Let us spend a moment revisiting how feature channel coding defines computation in this view of the neural network. We assume that $f_{i,j}$, a feature to be computed at layer i, is a function of a small set of k features $G = \{f_{i-1,1} \dots f_{i-1,k}\}$ from layer $i-1$. The features in G of the feature circuit view are each coded (n the

neural network view) by a set of neurons from layer $i-1$, and $f_{i,j}$ is coded by a set of neurons from layer i. To perform the computation for $f_{i,j}$, the weight matrix W_i maps each of the codes for features in G to the code for $f_{i,j}$. When a feature at layer $i-1$ is used for multiple features at layer i, it gets mapped to each of the codes for the features it is used in. Thus, prior to the non-linearity for the activation at layer i, each of the inputs to $f_{i,j}$ have been mapped to the same set of neurons at layer i. This allows the network to use the code for $f_{i,j}$ as a computational channel for computing f_0.

The entangled matrix W_i performs this mapping for the features used by $f_{i,j}$. The disentangled view makes this clear: D_i first maps those features to their monosemantic representation. C_i then brings them all together to the code for $f_{i,j}$. In a funny way, in terms of feature channel coding, D_i is a *decoding* matrix, moving codes into features, and C_i is a *coding* matrix, encoding them back into new collections of features, exactly the opposite of the naming convention in the autoencoding/transcoding literature (27; 9). The matrix C_i, combined with the bias and the non-linearity, performs the computation required for $f_{i,j}$. In the examples provided in earlier sections, the inputs were already in their monosemantic representation, so there was no need for a D matrix; the layer one weight matrix serves directly as the C matrix as well. Note that the channel's code is somewhat independent of the computation being performed - the goal of channel coding is to bring together the necessary inputs onto the same computational channel. Once they are on the same channel, a number of different functions can be performed on those incoming features. Thus, uncovering the codes for a given neural network would allow us to describe its feature circuit.

B.4 Cascading Feature Disentanglement

We propose the *cascading feature disentanglement* technique as a potential avenue for recovering the feature circuit from a trained MLP. The core hypothesis is that we can iteratively uncover the matrices C_i and D_i layer by layer.

Cascading Disentanglement Technique: For a d-layer MLP with weights $W_1, \ldots, W_d$ and input embedding C_0:

1. **Initialization (Layer 1):** Estimate C_1. Since there is typically no non-linearity between the input embedding C_0 (size $n_0 \times m_0$) and the first weight matrix W_1 (size $n_1 \times n_0$), the initial disentanglement is simply $C_1 \approx W_1 C_0$. The role of D_1 (size $m_0 \times n_0$) is to invert the input embedding, i.e., $C_0 = D_1^*$ (where D_1^* denotes an approximate right inverse of D_1, as D_1 is generally non-square (36)). If this holds, then since $W_1 \approx C_1 D_1$, we see that $W_1 C_0 \approx C_1 D_1 C_0 = C_1 D_1 D_1^* \approx C_1$.
2. **Iteration (Layer $i > 1$):** Assume we have successfully estimated C_{i-1}. Our goal is to estimate C_i from $W_i \approx C_i D_i$. Unlike Step 1, we cannot simply assume $D_i \approx C_{i-1}^*$ because a non-linearity and bias (σ_{i-1}) were applied after the computation involving C_{i-1}. The central hypothesis for this step is that by analyzing the estimated C_{i-1} (size $n_{i-1} \times m_{i-2}$), we can potentially deduce the m_{i-1} features computed by layer $i-1$ and how they are represented by the n_{i-1} neuron activations (after bias and ReLU). This understanding of the layer $i-1$ output features could allow us to construct an estimate for D_i (and hence its approximate inverse D_i^*). If we can find D_i^*, we can then estimate $C_i \approx W_i D_i^*$. Repeating this process for $i = 2, \ldots, d$ would, in principle, decode the entire network's feature circuit via $C_1, \ldots, C_d$.

Developing the precise methods to infer the layer $i-1$ feature vectors from C_{i-1} and construct the corresponding D_i (or D_i^*) remains a significant challenge and is a key direction for future research. This cascading approach, however, provides a conceptual roadmap for systematically disentangling deep MLPs.[5] We also

[5]Note that we are proposing an algorithmic framework without having the solution algorithm, apart from some initial heuristic attempts. Skeptics might criticize us for this. We note that this has been done before, for example, in the context

point out that an alternative approach would be to use a sparse autoencoder to determine the feature encoding being input to layer i, and use that to construct the pseudo-inverse D_i^*. In fact, the decode matrix of the sparse autoencoder may itself serve as the pseudo-inverse D_i^*, since it essentially performs the same function for layer i as an initial embedding matrix does for layer 1.

B.5 Cascading Feature Disentanglement with an Embedding Layer

We demonstrate that that Cascading Disentanglement works for a small example network where we add an embedding layer to the two layer networks we have described earlier in the paper. Specifically, Figure 9 depicts three example networks trained via gradient descent, where three different types of embeddings are added to the feature circuit depicted in Figure 8. Each row of matrices is a different type of embedding, where the rightmost column depicts the embedding being used: in the middle row, the embedding C_0 is the identity function (i.e., no embedding), in the top row it is a Hadamard matrix, and the bottom row it is a symmetric random embedding.

The middle column depicts the raw matrix W_1, i.e., prior to disentanglement. In the case of the identity embedding (middle row) feature channel coding is clear. In the case of the Hadamard and random embeddings, the W_1 matrices look random. The leftmost column depicts the result of disentanglement: the matrix obtained by the product W_1C_0. As described above, this gives us C_1 since $W_1C_0 \approx C_1D_1C_0 \approx C_1D_1D_1^* \approx C_1$. This reveals the feature channel encoding hidden within W_1 for both the Hadamard embedding and the random embedding. In other words, multiplying by C_0 undoes the implicit embedding that was learned by the network in W_1, and reveals the feature channel codes.

This is a good place to pause and contemplate the power of disentanglement. The relative magnitudes of the weights in the raw W_1 matrices, before disentanglement, might be quite misleading. For example, in the Hadamard embedding example, it would seem from W_1 that for Neuron 3, the weights in columns 2 and 15 are the most important (as they are of the largest magnitude), and yet we know from the disentangled matrix that these weights are just the result of the embedding space: the weights in these columns do not explain what neuron 3 is contributing to the computation. From the disentangled matrix C_1 we can tell that Neuron 3 is part of the coding for features $(x_2 \wedge x_{12})$ and $(x_0 \wedge x_9)$. This is the power of combinatorial disentanglement: if we want to decide which weights or neurons to prune, which to apply dropout to during training and so on, doing so in the disentangled space would give us the insights that are missing when using activation space approaches.

In the above experiments, we have used three different types of embeddings: identity (or no embedding), using a Hadamard matrix, and using a symmetric random embedding. In most of the statistics in Section 5 we trained on a network where the input had a Hadamard embedding layer C_0 which we then used to get $W_1 * C_0$, the coding matrix in which we counted the number of patterns (as a sanity check, we also ran the same training without the Hadamard embedding, getting similar results). In all other sections we used the identity embedding.

Finally, one can understand today's prevailing approach to interpreability via the training of an autoencoder (27) using our coding-decoding view of computation. The autoencoder approach is a way of learning an approximation of D_i for a given layer i. The autoencoder has a very large inner layer to which the output activations of C_{i-1} are mapped, along with a sparsity constraint to separate them so that the features of layer $i-1$ can be read. The difference in the combinatorial approach we propose here is to attempt to decode C_{i-1} and reconstruct D_i by analyzing the network weight matrices themselves rather then learning features from the activations using a separate trained network. As mentioned above, we think there is potential in a combined approach, for example where D_i^* is learned via an autoencoder (or other activation based learning

of public key cryptography (8) where Diffie and Hellman proposed a public key cryptography algorithm without the RSA algorithm (29) to implement it.

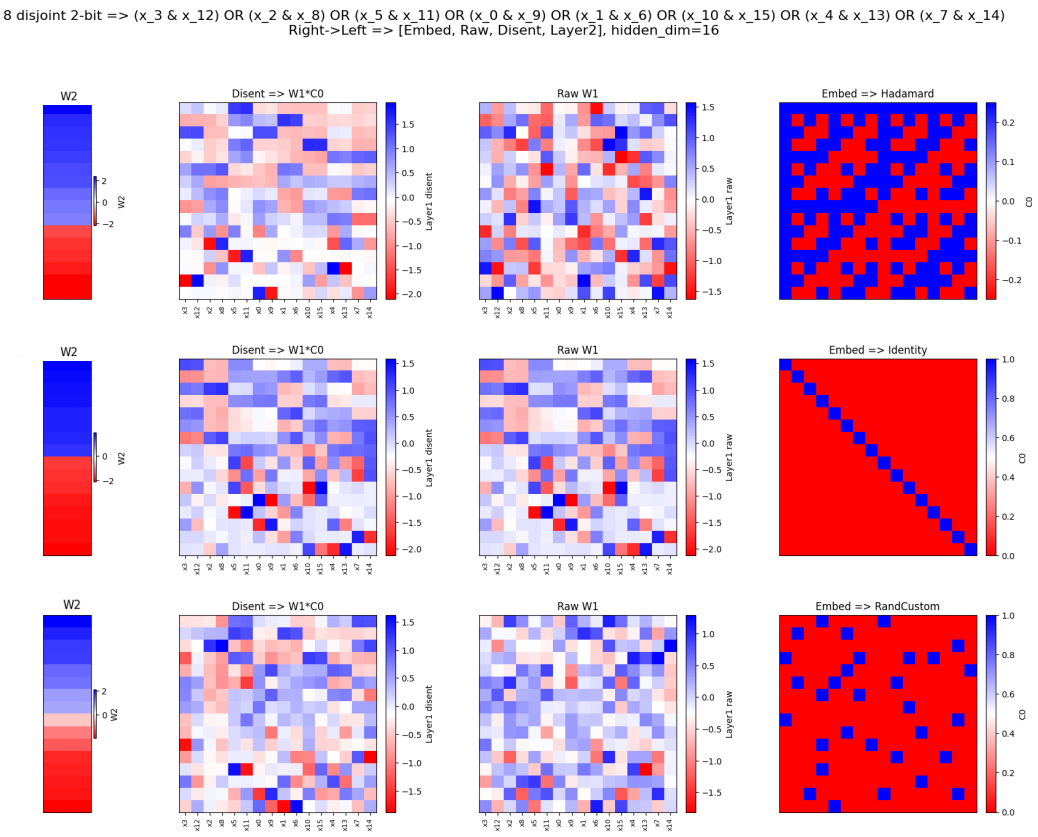

Figure 9: Disentangling of networks computing the feature circuit depicted in Figure 8. In top a 2-layer network with an initial Hadamard embedding layer C_0 that is trained as before on a DNF of 8 different 2-AND clauses. The raw trained W_1 matrix is shown immediately to the left of it. Then to its left we show the disentangled $C_1 = W_1 C_0$. Notice that C_1 has the usual expected structure with coding of 2P in the positive witness rows and coding for a XOR using 1N1P in the negative witness rows. Moreover, though having less 2P patterns and positive rows, it has a similar structure as the case where there is no embedding (the embedding is the identity matrix) as shown in the middle row of the figure. The bottom row shows how a symmetric random embedding is disentangled in a similar fashion.

approaches (9)), and then this is used to find and interpret C_i combinatorially. This would allow our approach to be applied to a single layer, without the need for cascading.

C Towards Deeper Networks

As a bit of a teaser as to the potential directions one can follow with the combinatorial interpretability approach, we examine what happens if there are multiple outputs - i.e., a classification problem with more than 2 possibilities. To do so, we add additional Boolean formulas to learn, and for each additional formula, there is a corresponding additional second layer output neuron responsible for that formula. The inputs and the first hidden layer are shared between the different outputs, and so the output neurons are computing

different formulas on the same set of Boolean variables and on the same hidden layer results. We still only consider the case where there is a single hidden layer, but we view increasing the size of the output layer as a small step towards having an additional hidden layer, as view computing multiple output features as a potential proxy for computing multiple features for use at the next hidden layer of the network.

We first consider the case with two output neurons. We tested a range of clause numbers, and found something similar to the single output case: the system is able to train fairly successfully, provided that the total number of clauses (in this case, summed across all outputs) is not larger than the hidden dimension. We here provide results for one representative example: in this case there are 32 Boolean variables and a hidden dimension of 32. The formulas are in DNF form with clauses of size 4 (4-AND), and each output is the OR of 8 clauses. The clauses for each output are non-overlapping in terms of variables, but the clauses for one output are chosen without taking into account the other output. Thus, each variable is used exactly once in each output but can be used in both outputs.

We generated 50000 inputs, each having between 8 and 10 variables set to true. The inputs were constructed independently, and each had equal likelihood of being each of the 4 output combinations (00, 01, 10, 11). Each output possibility was constructed in the same style as used in Section 5.

We provide the full neural network parameterization for the following pair of formulas:

$$f_0(x) = (x_3 \wedge x_4 \wedge x_7 \wedge x_{10}) \vee (x_2 \wedge x_6 \wedge x_{25} \wedge x_{27}) \vee (x_0 \wedge x_{13} \wedge x_{30} \wedge x_{31}) \vee (x_9 \wedge x_{16} \wedge x_{20} \wedge x_{21}) \vee (x_8 \wedge x_{12} \wedge x_{14} \wedge x_{28}) \vee (x_1 \wedge x_5 \wedge x_{17} \wedge x_{24}) \vee (x_{11} \wedge x_{22} \wedge x_{26} \wedge x_{29}) \vee (x_{15} \wedge x_{18} \wedge x_{19} \wedge x_{23})$$

$$f_1(x) = (x_{10} \wedge x_{15} \wedge x_{23} \wedge x_{26}) \vee (x_9 \wedge x_{13} \wedge x_{18} \wedge x_{19}) \vee (x_0 \wedge x_4 \wedge x_8 \wedge x_{14}) \vee (x_6 \wedge x_{20} \wedge x_{28} \wedge x_{31}) \vee (x_3 \wedge x_{21} \wedge x_{22} \wedge x_{30}) \vee (x_2 \wedge x_{11} \wedge x_{12} \wedge x_{24}) \vee (x_1 \wedge x_5 \wedge x_{25} \wedge x_{29}) \vee (x_7 \wedge x_{16} \wedge x_{17} \wedge x_{27})$$

This computation is depicted in Figure 10. This network achieved training loss of 3.42% after 1000 epochs, and at that time had overall accuracy (both bits correct) on the test set of 97.30%. Not only did the network learn this network very clearly, we see clear fature channel coding, despite having to do that coding for two different Boolean functions simultaneously.

A few observations about this parameterization:

- The biases are right in line with our expectations: if *either* output neuron has a positive weight to a hidden layer neuron, then the bias for the hidden layer neuron is negative. (Neurons 27 and 28 are exceptions and slightly positive.) All rows where both output layer weights are negative have a positive hidden layer bias.
- The positive row filtering for a given neuron sometimes codes exactly as expected, and other times it seems random. However, on closer examination, in every single case where it seems random, the weight from the other output neuron to that row is significantly larger. It is really coding for the other neuron in those rows, and simply looks random because we are sorting by the wrong set of clauses.
- The output neuron weights partition the hidden layer neurons into three categories: (1) positive and larger weight from output neuron 0, (2) positive and larger weight from output neuron 1, and (3) negative from both neurons. Neurons of type (1) code for output neuron 0 and function f_0. Neurons of type (2) code for neuron 1. Neurons of type (3) prove negative witnesses. This is visualized

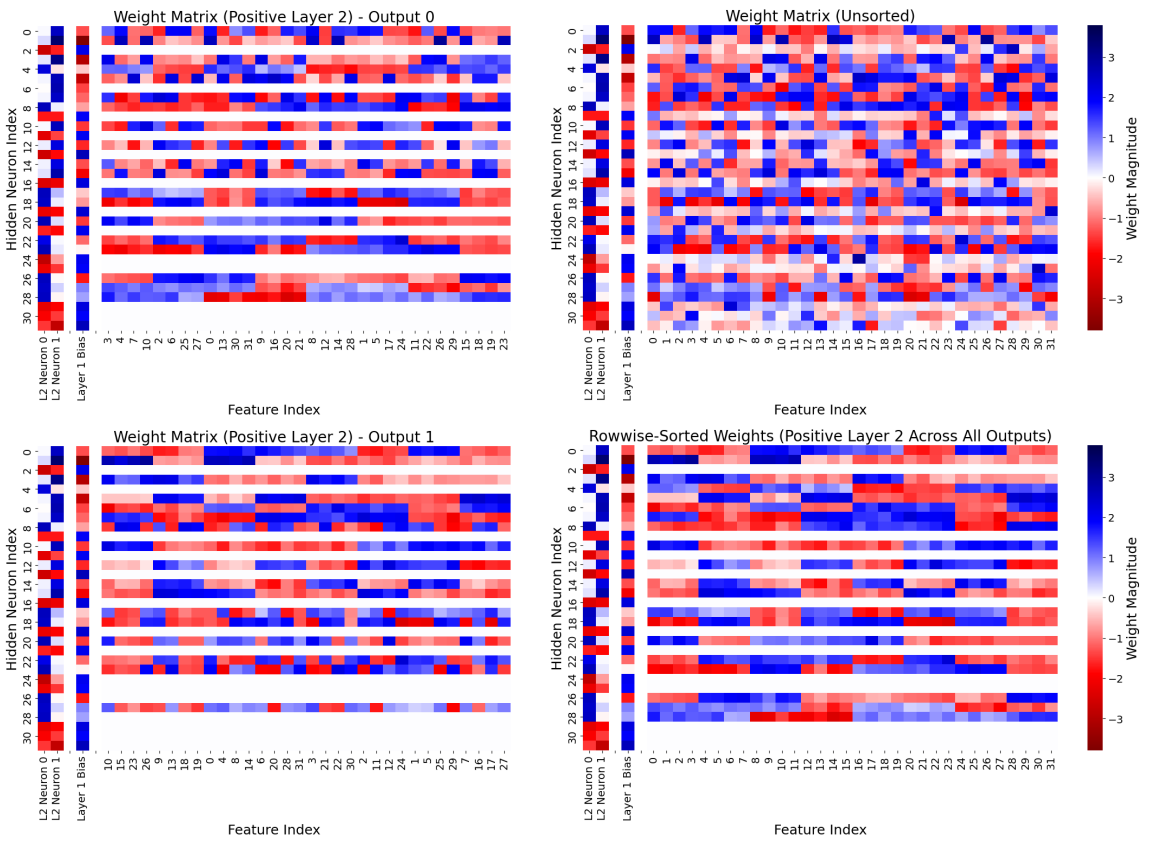

Figure 10: Trained neural network for f_0 and f_1. The top right is the unsorted version of the weight matrix, with all weights and biases. The top left is sorted by the clauses of f_0 (computed by output 0), and, as above, filtered to show only those neurons that have a positive edge to output neuron 0. The bottom left is the equivalent version for output neuron 1 and f_1. The bottom right has all rows with a positive edge to either output neuron, and each such row is sorted by the clauses corresponding to the output neuron with the larger weight to that row/hidden layer neuron.

clearly in the bottom right panel of Figure 10. There, each row is sorted according to which cell of the partition is in: for rows of type (1), we sort by the clauses of f_0, for rows of type (2), we sort by the clauses of f_1, and rows of type (3) are filtered out. We see that the result is nearly perfect coding.

- Once the system has chosen which output a row is coding for, it effectively ignores the other output for that row - the other weight has a much smaller magnitude weight, and that weight has a tendancy to be (slightly) positive.
- In Section 5, we saw that even as we increased the number of clauses, the network was not able to learn a weight distribution with more than roughly half positive rows. We here see a fairly even divide between the three cells of the partition, and so the number of negative rows is only 12. This is additional evidence that the system does not need as many negative rows as the solution that it is converging to has. For this case where there are 2 output neurons, one could also argue that it is seeking equality between the positive rows and the negative rows, and the negative rows are reused

between the two output neurons. However, (a) we do not see that same effect with 3 output neurons (below), and (b) even if that were the case, it does imply a limit on how much logic can be built into the negative rows (since they are the same between the two outputs, with relatively minor differences in how negative the output layer weight is).

We next turn to the case where we have 3 output neurons. The setup is the same otherwise, except that we now train with more data (200,000 inputs). The result was slightly worse performance for this more difficult case (since we now have 24 total clauses, instead of 16). Specifically, after 1000 epochs, we have training loss 5.86% and test set accuracy (all 3 outputs correct) of 95.05%. Figure 11 depicts the results for the following set of formulas:

$$g_0(x) = (x_5 \wedge x_{10} \wedge x_{11} \wedge x_{20}) \vee (x_4 \wedge x_7 \wedge x_{14} \wedge x_{27}) \vee (x_{12} \wedge x_{17} \wedge x_{21} \wedge x_{24}) \vee (x_{13} \wedge x_{15} \wedge x_{28} \wedge x_{30}) \vee (x_2 \wedge x_6 \wedge x_9 \wedge x_{22}) \vee (x_8 \wedge x_{23} \wedge x_{29} \wedge x_{31}) \vee (x_0 \wedge x_{16} \wedge x_{25} \wedge x_{26}) \vee (x_1 \wedge x_3 \wedge x_{18} \wedge x_{19})$$

$$g_1(x) = (x_0 \wedge x_{10} \wedge x_{19} \wedge x_{27}) \vee (x_{11} \wedge x_{12} \wedge x_{14} \wedge x_{24}) \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_7) \vee (x_6 \wedge x_8 \wedge x_9 \wedge x_{20}) \vee (x_{16} \wedge x_{22} \wedge x_{26} \wedge x_{30}) \vee (x_1 \wedge x_{21} \wedge x_{25} \wedge x_{29}) \vee (x_{17} \wedge x_{23} \wedge x_{28} \wedge x_{31}) \vee (x_5 \wedge x_{13} \wedge x_{15} \wedge x_{18})$$

$$g_2(x) = (x_1 \wedge x_5 \wedge x_{21} \wedge x_{30}) \vee (x_6 \wedge x_{12} \wedge x_{13} \wedge x_{20}) \vee (x_2 \wedge x_{17} \wedge x_{22} \wedge x_{29}) \vee (x_{10} \wedge x_{26} \wedge x_{28} \wedge x_{31}) \vee (x_4 \wedge x_{11} \wedge x_{15} \wedge x_{25}) \vee (x_0 \wedge x_7 \wedge x_{16} \wedge x_{18}) \vee (x_3 \wedge x_8 \wedge x_{23} \wedge x_{27}) \vee (x_9 \wedge x_{14} \wedge x_{19} \wedge x_{24})$$

This solution is very similar to the case with 2 output neurons, and as we can see in Figure 11, it is clearly using feature channel coding. However the structure of the network is not quite as clean. This is not surprising, given (a) the poorer performance, and (b) the fact that we are starting to approach the limit we discussed in Section 5, where the total number of clauses equals the size of the hidden dimension. Specifically, we see that the layer 1 biases are not as consistently negative for the positive rows, and that not every positive row chose a clear output to code for (see neuron 27, which just contributes a bit of positive value from all input variables. It is however, interesting to note that the number of negative rows is only 3 (or 4 if you count row 19), leaving each output about 9 rows of positive coding, again providing evidence that the single output solutions are over investing in negative rows.

We did also experiment with 4 output neurons, but our initial attempts at that did not yield particularly effective networks. In that case the number of clauses was equal to the size of the hidden layer; the trained networks tended to have error rates around 1/3. Still even in these poorly performing networks, it was clearly trying to use feature channel coding.

D Combinatorial Analysis of a Scaling Law

We here provide significantly more detail on the analysis found in Section 5. We first provide additional details on how the networks were trained. For each trial, we train for up to 100 epochs (with patience-based early stopping) on 20,000 samples. The samples are drawn from the following distribution: with 50% probability,

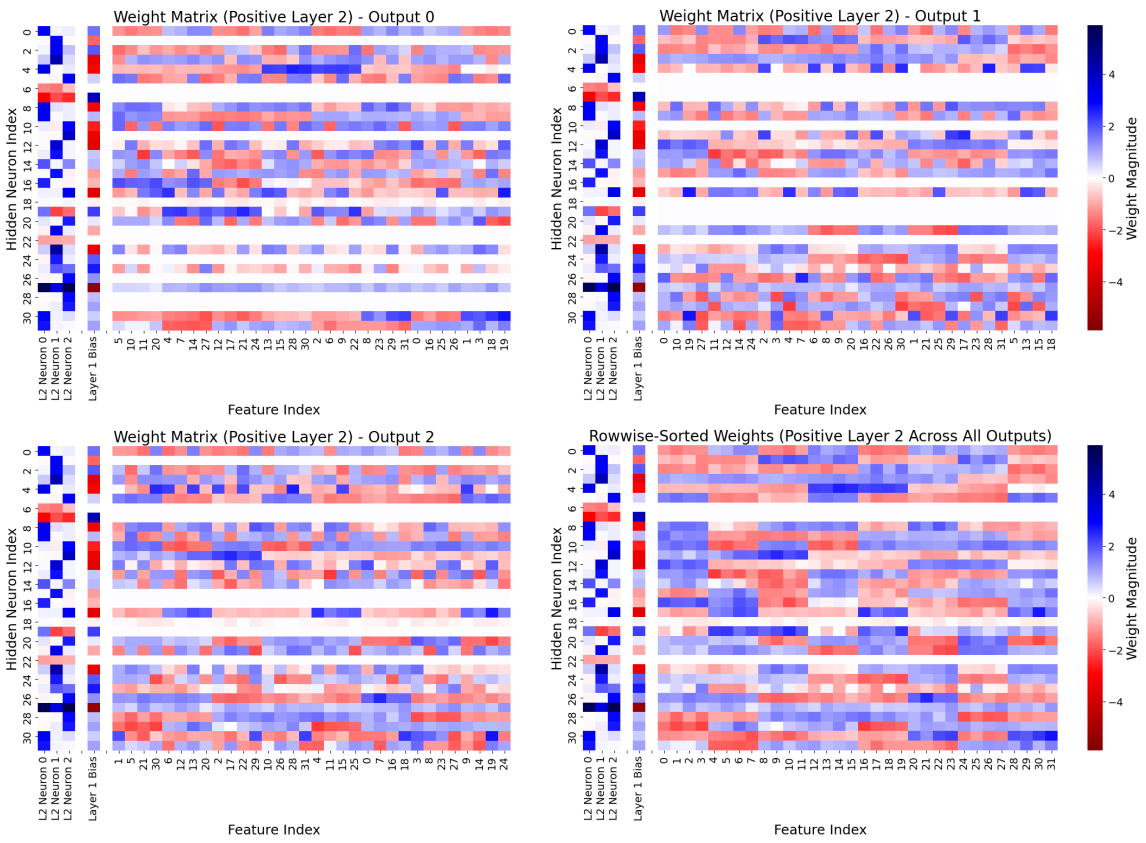

Figure 11: Trained neural network for g_0, g_1, and g_2. The top right is sorted by the clauses of g_0 (computed by output 0), and filtered to show only those neurons that have a positive edge to output neuron 0. The top left is the equivalent version for output neuron 1 and g_1, and the bottom left is the equivalent for output neuron 2 and g_2. The bottom right has all rows with a positive edge to any output neuron, and each such row is sorted by the clauses corresponding to the output neuron with the largest weight to that row/hidden layer neuron.

we assign a label of 1 (True), randomly select one of the formula's clauses, set its variables to satisfy that clause, and then choose between zero and two additional variables to set to 1 while preserving the clause's satisfiability. Otherwise (label 0, a value of False), we again pick a clause but now deliberately "break" it so that only three of the four variables are set to satisfy that clause. We also again choose additional variables to set to 1, so that the total number of set variables will be between four and six while the value will still be False. We use the Adam optimizer with a learning rate of 0.001, a batch size of 64, and a patience of 10 epochs for early stopping. We note that the distribution matters quite a bit for the overall numbers of patterns we report on, but not the general scaling trends that we describe.

In Figure 12, we examine training error for the three sizes of the hidden layer for the case of clauses of four non-negated variables, and see that it starts to accumulate when the number of clauses reaches the number of neurons in the hidden layer. Figure 13 provides all three versions of the scaling graphs presented in Figure 5. For ease of comparison, we provide the $j = 32$ case here again, along with the $j = 16$ and $j = 64$ cases not presented in Section 5. For an explanation of these graphs, see Section 5. Each point in Figure 13 is the

Figure 12: Onset of training error as number of clauses increases.

average of 10 different runs of model training. We also provide the full distribution of these samples for the 4P case in Figure 14. We there observe that for some values of j and k there is significant variability in how many coding rows appear. However, the more constrained the capacity of the Layer 1 weight matrix is, the tighter that variance is, and the more concentrated those values are around the upper limit of that capacity.

Next, in the three versions of Figure 15, we graph the percentage of hidden layer neurons that are connected to the output via positive weights. As we saw in Figure 1, the sign of that weight differentiates how the model uses that neuron, for positive or negative witnesses. We see that in these tests, the learned model does not stray far from having half of the neurons in positive rows and half in negative rows. In other experiments, not described in this paper, we see that the fraction of negative and positive rows is strongly influenced by the fraction of positive and negative examples in the training set. For example, if the training set has 90% positive examples instead of 50% positive examples, there will be very few negative rows. This is perhaps not that surprising if viewed through the lens of gradient descent: positive examples can only either increase or not change Layer 2 weights of our network, and negative examples can only decrease or not change Layer 2 weights. Fully understanding and quantifying this phenomenon is an interesting open problem, but for this work, we maintained a balanced distribution of positive and negative examples.

Figure 15 also depicts how many of the positive rows also have a negative bias (which is what feature channel coding would predict). Interestingly, we see that for the (simpler) functions with a smaller number of clauses, the system does indeed have all (or almost all) of the positive Layer 1 neurons associated with a negative bias. However, as the function becomes more complex, more and more of the second layer neurons have positive bias. And this does not line up with the point where the network training accuracy falls off - it happens around 15 to 20 clauses for all three values of j, and so seems independent of the size of the hidden layer. It is also past the point where accuracy falls off for 16 hidden neurons, but before it for the two cases of a larger hidden layer, and so it seems that having negative bias is helpful, but not crucial to learning the Boolean function. We will discuss below a possible reason for why the percentage of positive rows with negative bias decreases the way it does.

We next turn to the three versions of Figure 16. This depicts the fraction of entries of the hidden layer weight matrix that are positive and negative. We examine the overall number, as well as the fraction in positive rows that are in inputs used in at least one clause, positive rows that are in inputs not used by any clauses, and

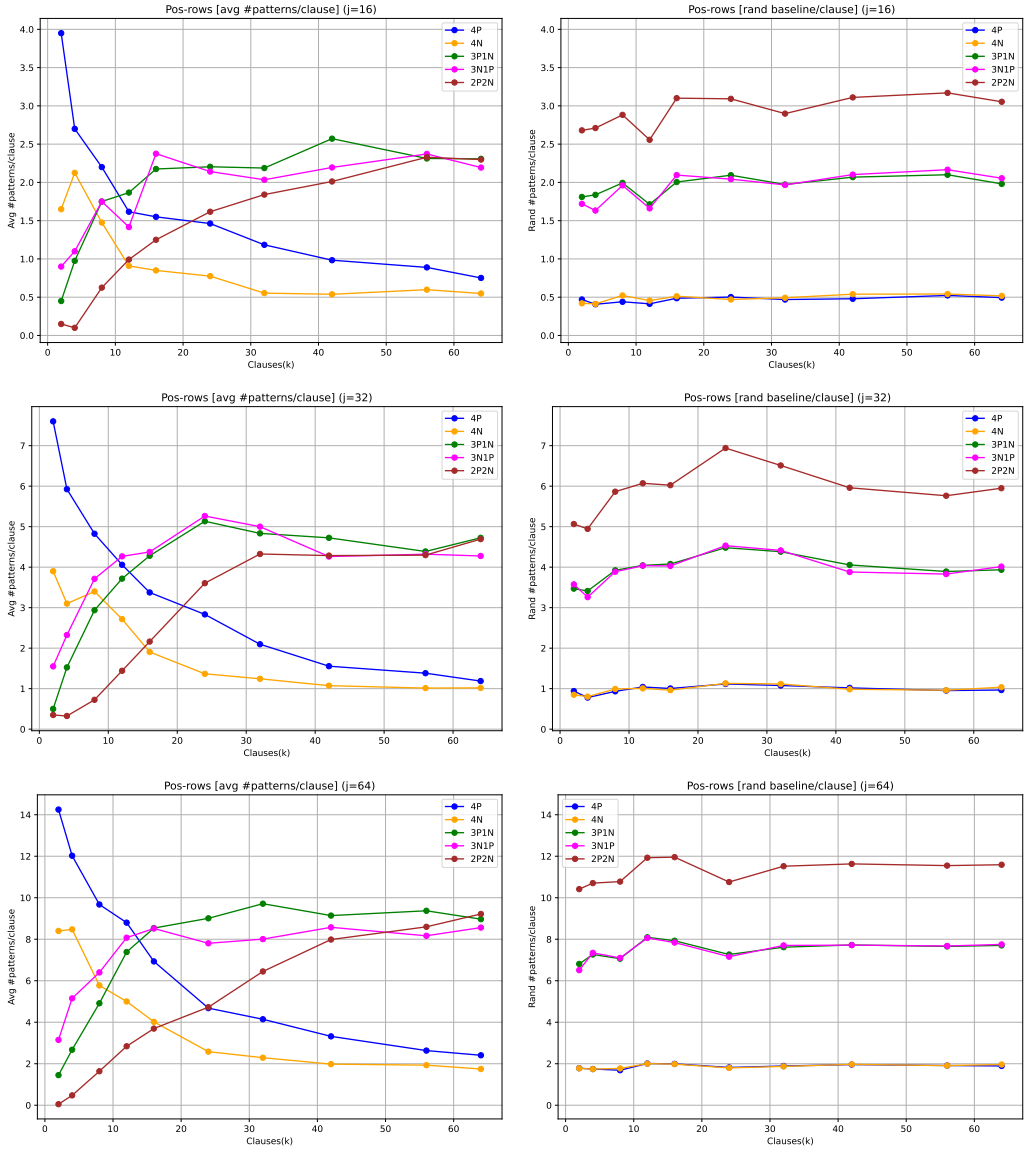

Figure 13: Prevalence of coding patterns in positive rows: trained networks versus random networks.

similarly for negative rows. We see that overall, very close to half of the matrix entries are positive, and that there is a small positive bias for the positive rows and a small negative bias for the negative rows. This small bias decreases as the number of clauses increases. We also see that the non-clause variables have the opposite bias of the clause variables, although closer examination (not depicted in the graph) reveals that the magnitude of these weights is much smaller than that of the clause variables, and so we do not believe this impacts how the neural network learns very much. Both the fact that there is so little bias in the clause variables and the fact that the three graphs are so similar (with no discernable difference where training error occurs) leads us

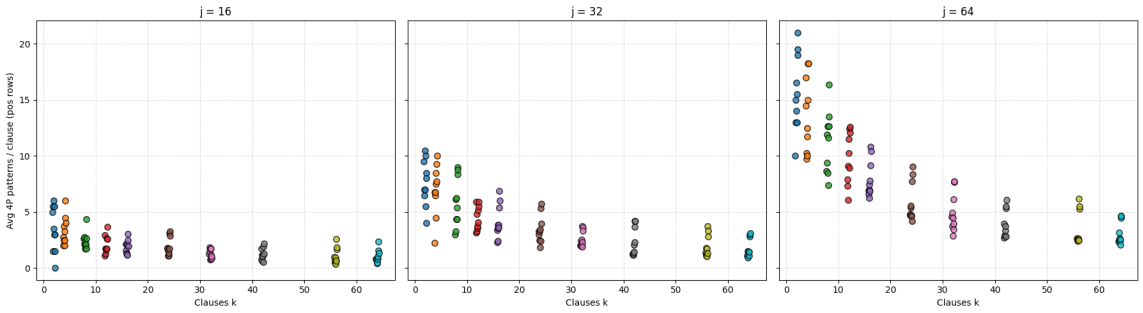

Figure 14: The distribution of 4P coding patterns observed in positive rows of the Layer 1 weight matrixes. Each point is one of 10 models for each value of j and k; their average value is depicted in Figure 13. As can be seen, when there is sufficient room for multiple coding rows per clause, there is some variability on the size of the codes, but as code capacity becomes a constraint, the distribution becomes more concentrated.

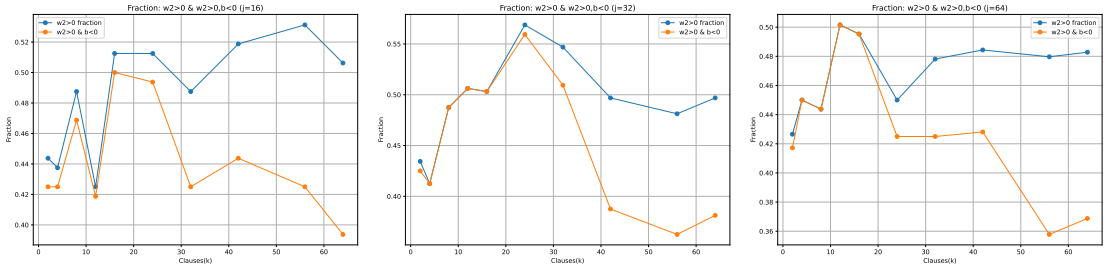

Figure 15: Fraction of positive Layer 2 weights and negative Layer 1 bias.

to believe that it is useful for the network to have close to an even mix of positive and negative values, but this measure does not explain training error.

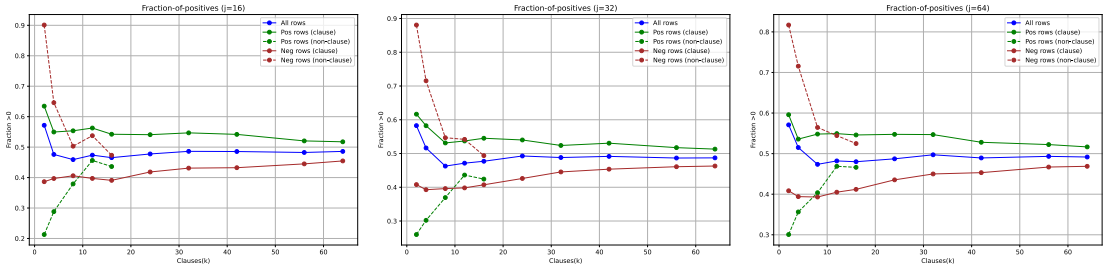

Figure 16: Bias of Layer 1 matrix weights, overall and by row and variable type

D.1 Theoretical Limits of the 4P Coding Pattern

We next dive deeper into the limit on how many 4P patterns can be packed into the weight matrix, described in Section 5. As pointed out there, the saturation of the 4P patterns seems to be a real limitation of the network. We first point out that the positive rows have roughly half positive (coding) values, and half of them are

negative. This is necessary to group the inputs contained in a feature together - if all inputs were positive it would not be possible to distinguish 4 inputs from the same AND clause being true from 4 inputs from different AND clauses being true. Thus, we assume here that if we have ℓ input variables, each row will have roughly $\ell/2$ positive values that can go towards 4P patterns. Let ρ be the fraction of positive rows. If there are j hidden neurons, then the total number of positive entries in positive rows will be roughly $\frac{j\ell\rho}{2}$.

Each 4P pattern requires 4 of those positive entries. There may be some overlap between the 4 positive entries of different 4P patterns, but since the clauses are chosen randomly, that overlap will be small, and we shall ignore that effect here (but we note that for clauses of size 2 that overlap is much more significant, and in fact we see that networks trained on such smaller clauses can handle a much larger number of clauses). As a result, the average number of 4P patterns per clause, when there are k clauses, will be $\frac{j\ell\rho}{8k}$. Since ρ is approximately $1/2$, this will be roughly $\frac{j\ell}{16k}$ This explains the saturation point we see in the above graphs. For example, when $j = k = \ell = 32$, according to this formula, we would not expect to see more than 2 occurrences of 4P per clause. Comparing this prediction to Figure 5, we see that our prediction is accurate: the value of the blue curve in the $j = 32$ graph of the right hand column, at the 32 clause point is almost exactly 2. In other words, the network in practice was able to achieve this maximum packing of 4P patterns, but not better.

If we increase k to 64 and hold the other variables fixed, our formula says there is not room for more than an average of one 4P per clause; again Figure 5 shows us the network is able to achieve that but not much higher (we suspect the slight overperformance is the result of increasing overlap of clauses). Conversely, when we decrease k, the network does not quite keep up with the theoretical maximum. But in those cases, the network does not have an incentive to keep up with that maximum: there is plenty of room for coding and so it does not have to saturate the network. We see a similar agreement with this packing limit for the $j = 16$ and $j = 64$ curves for 4P in Figure 5. In all cases, when the network is no longer able to achieve an average of approximately 2 coding rows per clause, it starts to accumulate error.

D.2 Codes as Negative Witnesses

We next turn our attention to the negative rows of the weight matrix, as depicted in Figure 17. These graphs are analogous to those in Figure 5, but focus on patterns emerging in negative rows, again using a random matrix baseline configured with the corresponding row counts and biases. For a small number of clauses, the 3N1P pattern exhibits the strongest signal in these negative rows. However, as the number of clauses increases, the dominant pattern transitions to 2P2N. Both patterns appear designed to detect instances where two or three positive variables, but not all four, are present in a clause. This detection generates a positive post-activation signal at Layer 1, which is then inverted to a negative signal by the negative Layer 2 weight. The 3N1P pattern achieves this using a positive Layer 1 bias, roughly equal negative Layer 1 weights, and ensuring the sum of the bias and the positive weight approximately offsets the sum of the three negative weights (e.g., weights: -1, -1, -1, +2; bias: +1). In this configuration, if the variable corresponding to the positive pattern column is 1, all three variables corresponding to negative columns must also be 1 to suppress a positive post-activation value. However, if any of the three negative weight variables are 0 (indicating the clause condition is not met), a positive Layer 1 activation results, leading to a negative final signal via Layer 2. As the clause count increases, there is no longer sufficient room to fully utilize the 3N1P pattern, and the mechanism shifts towards the 2P2N pattern. This coincides with a reduced reliance on positive bias in the negative rows. Additionally, the 3P1N, 4P, and 4N patterns are consistently suppressed in negative rows, indicating they are not effective for feature channel coding to be performed by these rows.

D.3 Code Interference

As described above, the number of 4P coding rows a clause coincides with seems to have a significant impact on the network's ability to learn the Boolean function. We take this a step further in Figure 18, where we

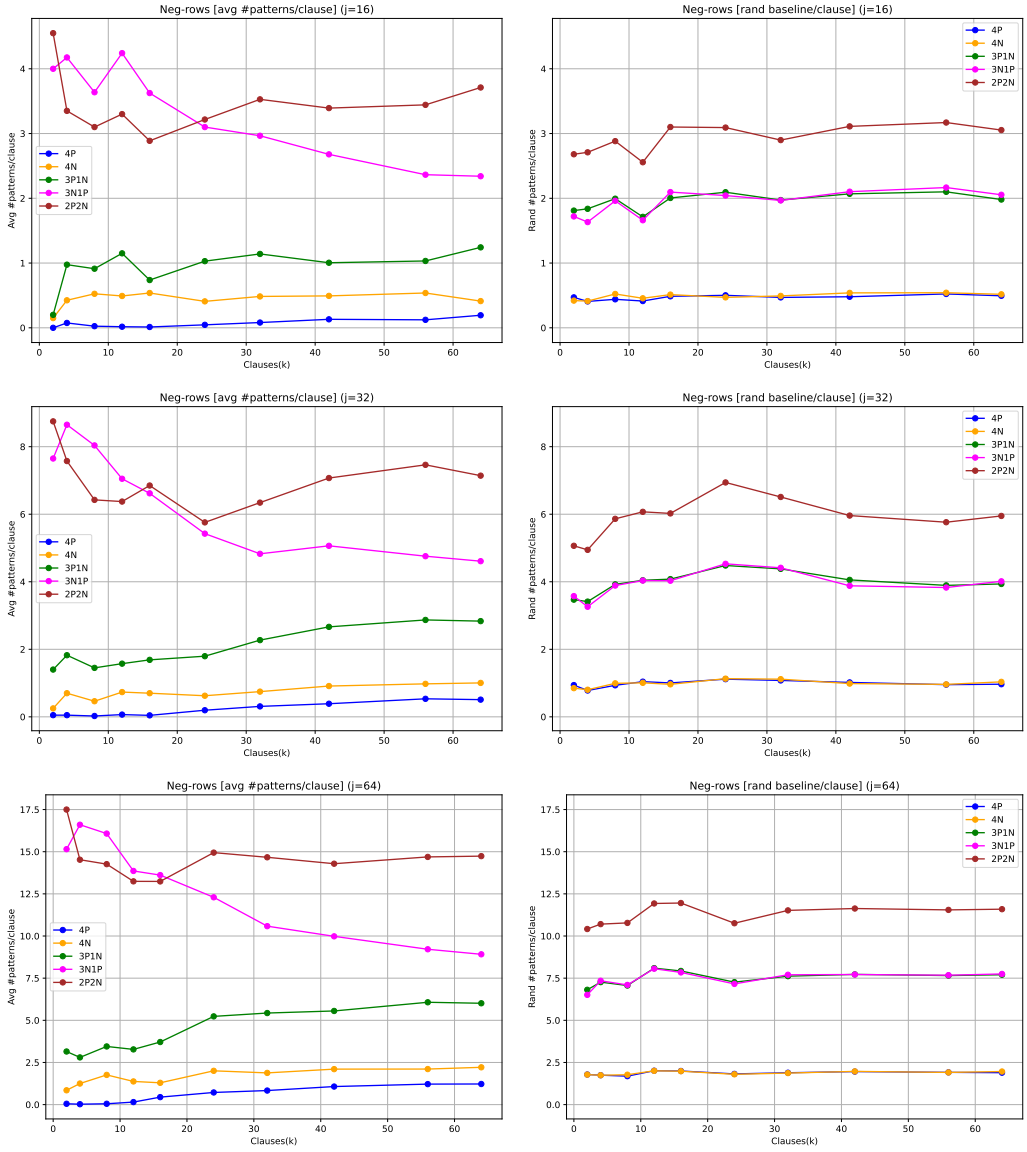

Figure 17: Prevalence of coding patterns in negative rows: trained networks versus random networks.

depict the number of clauses that do not coincide with any 4P rows. This is depicted on the right side - the left side is a duplicate of Figure 12, shown again here for convenience. We see here that there is, in fact, a close alignment between training error and the number of clauses with no 4P coding rows. However, it does seem like the network can tolerate a small amount of clauses without 4P coding rows.

Next, we examine the overlap between the codes that the network is using. As shown in (1), this is a central aspect of coding. Codes usually overlap and always will if there is sufficient saturation of the network by

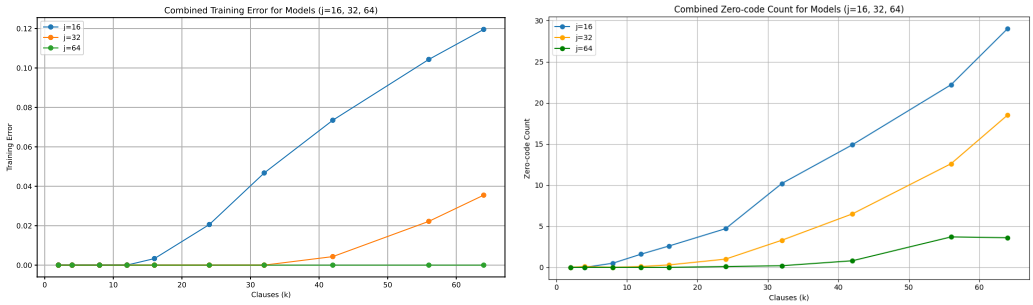

Figure 18: Scaling of training error and number of clauses without codes

the number of features being coded. However, as long as the overlap between pairs of codes is not too large, this is fine. One of the main advantages of using codes is that spreading out a signal across multiple neurons creates tolerance to noise that appears on some of those neurons. This is a core tenant of the feature channel coding hypothesis.

We show the code overlap measured from our experiments in Figure 19. We plot the average, over all pairs of clauses in each training run, of the overlap between the codes, where an overlap means that both codes use the same row as a 4P positive row. We also show the total number of 4P coding rows. For both values, we do not include clauses without coding rows in the count, so the blue curve (code size) will differ from that of Figure 5, which includes those clauses. These curves are almost identical (except for scaling) for the three hidden parameter sizes, and consistent with what we would expect from the feature channel coding hypothesis. Also, we do not see any indication that this overlap is responsible for the network not being able to train.

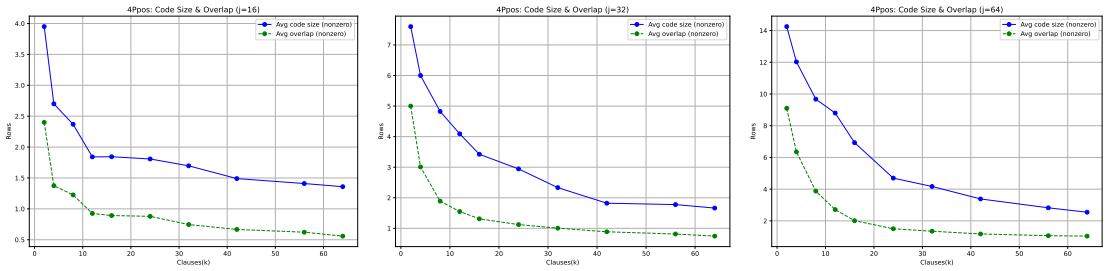

Figure 19: The average, over all pairs of clauses in each training run, of the overlap between the feature channel codes. Average code size is shown to give the reader a measure of the relative portion of the overlaps.

D.4 Clauses With a Negated Variable

So far we have shown how feature coding appears in DNFs that have all positive variables. In this section we show how features with negative variables are coded, by studying DNF clauses with 3 non-negated variables and 1 negated variable. We will see that with this slightly more complex formula, the overall coding behavior is very similar to the fully positive case. We also use this example to show how computation with feature channel codes proceeds.

We trained the same simple network with a single hidden layer as in our prior experiments, here with 32 input Boolean variables and 16 hidden neurons. In the top part of Figure 20, we show the result of training the

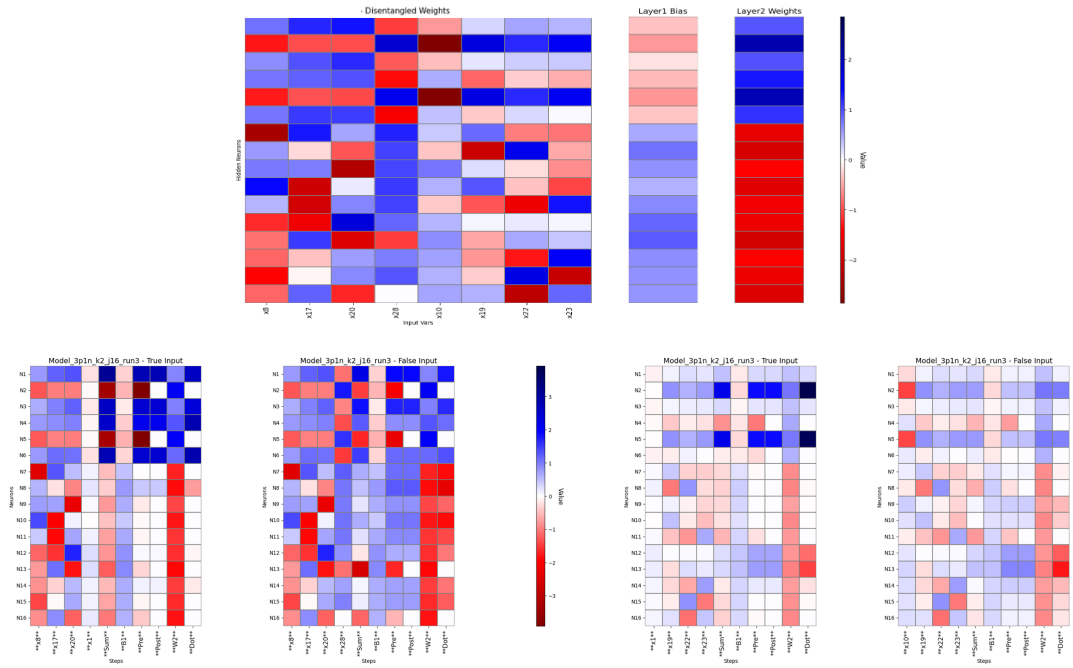

Figure 20: Execution traces show how channel coding in a model with clauses that have a negated variable each deliver the appropriate positive or negative response.

model with the formula $(x_8 \wedge x_{17} \wedge x_{20} \wedge \neg x_{28}) \vee (\neg x_{10} \wedge x_{19} \wedge x_{22} \wedge x_{23})$. For legibility we are only showing the columns corresponding to the 8 variables in the 2 clauses, not all 32 of them. The input features are sorted in the order of the clauses. A quick glace at the matrix shows us the feature channel coding patterns. As before, the network has converged to have a negative bias in Layer 1 neurons corresponding to positive witnesses (positive Layer 2 weights), and positive Layer 1 bias in neurons corresponding to negative witnesses (negative Layer 2 weights). Looking at the Layer 1 matrix entries for the clause $(x_8 \wedge x_{17} \wedge x_{20} \wedge \neg x_{28})$, we see clear positive coding rows emerge in neurons 1, 3, 4 and 6. For the clause $(\neg x_{10} \wedge x_{19} \wedge x_{22} \wedge x_{23})$, the positive coding rows are 1, 2, 3, and 5. Thus, the positive codes overlap in neurons 1 and 3, but the coding for the second clause has smaller magnitude weights in those overlapping rows (which is not a coincidence - see below.) The Boolean computation performed using these feature channels is as expected: the negated variables have corresponding columns with negative weights while the columns corresponding to positive variables have positive weights. As before we call this pattern 3P1N.

The bottom part of Figure 20 presents four execution traces for the model evaluating two clauses: $(x_8 \wedge x_{17} \wedge x_{20} \wedge \neg x_{28})$ (left two traces) and $(\neg x_{10} \wedge x_{19} \wedge x_{22} \wedge x_{23})$ (right two traces). Each trace details network activation over time (left-to-right), showing initial True variable inputs (first 4 columns), intermediate Layer 1 computations (dot product, bias, pre/post-ReLU values), Layer 2 weights, and the final Layer 1 neuron contributions to the Layer 2 output.

For the first clause, Trace 1 shows a satisfied case (x_8, x_{17}, x_{20}, x_1 True and x_{28} (not shown) False). Positive neurons (1, 3, 4, 6) activate according to x_8, x_{17}, x_{20}, while negative neurons (7-17) remain mostly inactive, resulting in a correct sigmoid output of 0.9999 (True). Trace 2 shows an unsatisfied case ($x_8, x_{17}, x_{20}, x_{28}$

True). Here, x_{28} inhibits positive neurons (1, 3, 4, 6) via negative weights and activates negative neurons (7-17) via positive weights, resulting in a correct sigmoid output of 0.0015 (False).

For the second clause, Trace 3 demonstrates that in a satisfied case $x_{19}, x_{22}, x_{23}, x_1$ True and x_{10} (not shown) False), positive activation occurs primarily through the set of neurons (2 and 5) that does not overlap with the activations in Trace 1 above. This indicates something impressive: the network has found feature channel coding that mostly separates clause representations. Trace 4 shows an unsatisfied case ($x_{10}, x_{19}, x_{22}, x_{23}$ True). Variable x_{10} suppresses the positive signal in neurons 2 and 5 via negative weights and activates negative witness neurons (9-16) via positive weights, leading to a False output.

As before, we could have hand constructed codes based on two monosemantic neurons, one for each of the clauses, and used large negative biases to provide a clean signal. However, the technique that gradient descent found involves coding using both positive and negative witnesses, where the negative witnesses might be partly playing the role of the bias in the final computation. In the final result, the coding is clear but uses varying weights, not clean binary values as in the hand crafted combinatorial feature channel codes presented in (1), most likely due to the pecularities of gradient descent. We believe the binary combinatorial setting may prove to be a good test ground for understanding this process better.

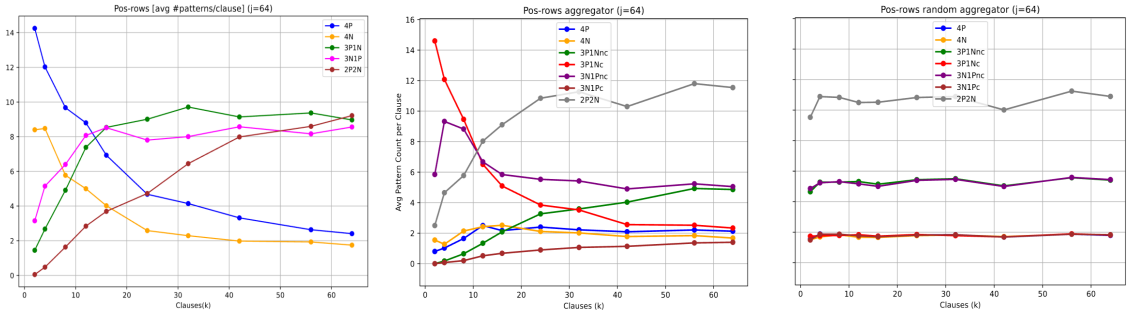

Figure 21: Left panel: a reminder of what clauses with 4 positive variables look like. Middle and right panel: Clauses with 3 positive and one negative variable. The middle panel shows the 3P1Nc coding in the positive rows that scales like the 4P pattern we saw for the 4-AND networks in the graph on the left, and very differently than the random distribution on the right.

We also analyzed the prevalence of coding patterns for this case of a DNF with three positive and one negative variable per clause as we did for networks that learn a DNF of clauses with four positive clauses. We ran the same set of tests and benchmarks. We found that the positive witnesses use a 3P1N pattern in which the negative weight in the pattern aligns with the negative variable in the clause. We will call this pattern 3P1Nc, where the c indicates "aligned with the clause," differentiating it from 3P1Nnc, the appearance of a 3P1N pattern where the negative weight is "not aligned with the clause." The left side of Figure 21 shows the 4-AND DNF positive rows coded by 4P for the case $j = 64$ that we already saw in Figure 5 (with all positive variables). The middle of Figure 21 shows the same type of plot for clauses that include a negative variable, and thus include the patterns 3P1Nc and 3P1Nnc. The right hand side shows the corresponding random pattern distribution. As can be seen, the neural network is using 3P1Nc as the clear non-random positive coding pattern for clauses with 3 positives and a negative, with a plot in red that looks very similar to the blue 4P pattern on the right, while the 4P patterns in this case look pretty much random, as we would expect.

We do not show the coding in the negative rows, but report that it uses 2P2N for lower k's but eventually saturates as k grows, looking similar to random.

E Further Examples of Feature Channel Coding

We next take a look at other examples of problems that the network used feature channel coding to solve: ORs, clauses in CNF form, and a toy vision problem. These help further clarify this techniques versatility and generality.

E.1 OR and Conjunctive Normal Form

We here study the difference in how feature channel coding works between computing an AND and computing an OR. Within the framework of soft Boolean logic, the difference between an AND and an OR is the bias. If we ignore the magnitude of the final result, then $x_1 \wedge x_2 = \text{ReLU}(x_1 + x_2 - 1)$, while $x_1 \vee x_2 = x_1 + x_2$. Thus, we expect gradient descent to find solutions where AND has significant negative bias, whereas OR have minimal bias. To study this, we first look at a very simple Boolean formula, where the output is simply the OR of all the variables in the system. We compare this to the type of formula presented in Figure 1. We studied the same network as in that scenario (hidden dimension equals input dimension equals 16, single neuron at the second layer), and we examined the Layer 1 bias for the neurons at Layer 1 with positive Layer 2 weights associated with them.

	Average Layer 1 bias (Mean $\pm$ SD)	Maximum absolute Layer 1 bias (Mean $\pm$ SD)
AND	-0.61 $\pm$ 0.13	1.54 $\pm$ 0.21
OR	0.0079 $\pm$ 0.023	0.0306 $\pm$ 0.085

Table 1: Comparison of AND vs OR Layer 1 bias for neurons with positive Layer 2 weight, including standard deviations (SD).

Table 1 summarizes the result of 10 random network trainings, each consisting of 10000 randomly generated inputs (settings of the variables). For AND, the inputs are as described for Figure 1. For OR, each input is a 1 independently with probability 0.043 and 0 otherwise; this provides approximately equal probability of a positive and negative instance. In this table, we see that the result are as expected: the AND function has a negative bias, which on average is large enough to have an impact, although as already discussed, the value of the bias tends to be a bit smaller than the theory would predict. The OR function, on the other hand, has a very small bias, with a negligible impact on the compuatation.

In an effort to further study the OR function, a seemingly good test bed would be to study formulas in Conjunctive Normal Form (CNF), which is the AND of a number of clauses, where each clause is the OR of a number of literals. The network solves the DNF formulas studied above by computing the AND clauses at the first layer, and then using the single neuron at the second layer to compute an OR of those results, and so our expectation was that the network would solve a CNF formula by computing the OR clauses at the first layer, and then use the second layer to compute the AND of those results. Our expectation was wrong, but we actually found something more interesting.

To study CNF, our neural network setup is the same as the above, with input dimension 16 and hidden dimension 16. We study CNF formulas where every clause is the OR of two (positive) variables, and each variable is used exactly once (the *feature influence* (1) of the network is 1). To train any given formula, we want to construct a roughly even split between True inputs and False inputs; we do this simply by choosing every variable to be True independently with probability 0.75. We studied a number of CNF formulas chosen

randomly from the set of such formulas, and saw very consistent results. We here present results for one such randomly chosen formula:

$$(x_3 \vee x_{11}) \wedge (x_9 \vee x_{12}) \wedge (x_2 \vee x_8) \wedge (x_0 \vee x_{10}) \wedge (x_{13} \vee x_{14}) \wedge (x_4 \vee x_7) \wedge (x_1 \vee x_5) \wedge (x_6 \vee x_{15}).$$

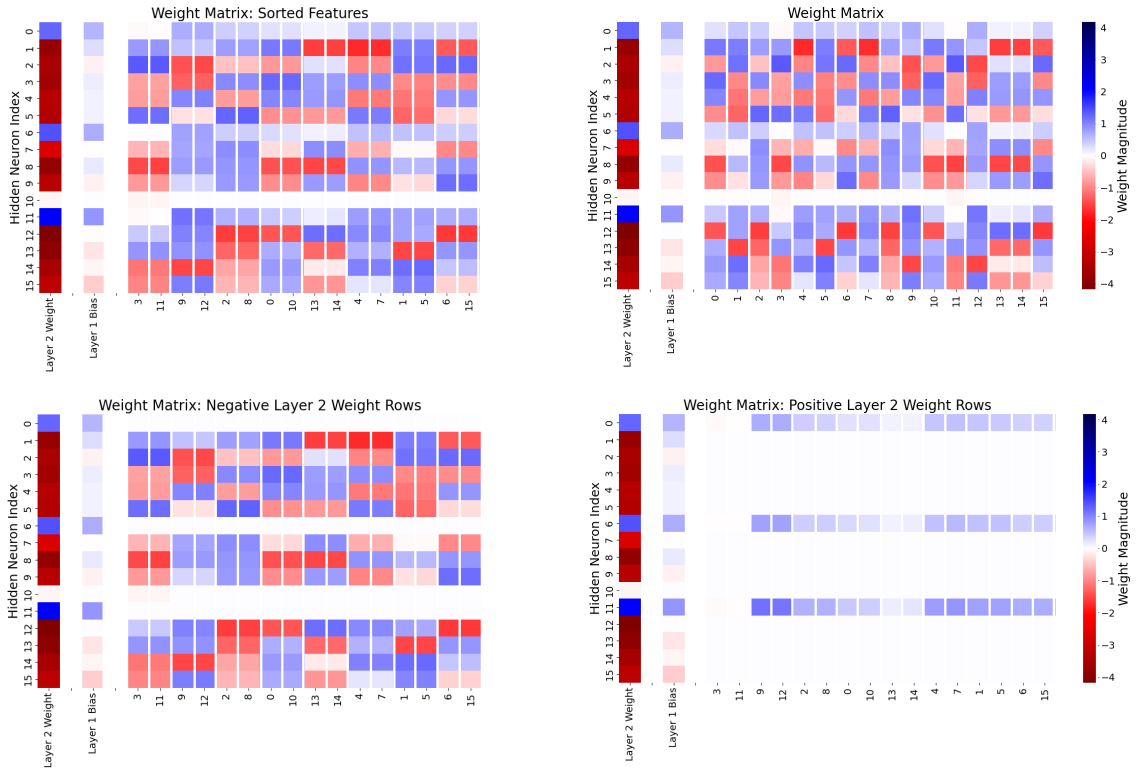

Figure 22: All weights and biases for a neural network trained on the Boolean formula $(x_3 \vee x_{11}) \wedge (x_9 \vee x_{12}) \wedge (x_2 \vee x_8) \wedge (x_0 \vee x_{10}) \wedge (x_{13} \vee x_{14}) \wedge (x_4 \vee x_7) \wedge (x_1 \vee x_5) \wedge (x_6 \vee x_{15})$

In Figure 22, we depict what the network learns for this formula with 40,000 randomly chosen inputs. We see in this figure that the network is not learning the AND as we had expected, and in fact, there is not much that is interesting happening on the small number of positive Layer 2 weight rows at all. On the other hand, the negative rows look much more interesting, and they take up the majority of the neurons. Furthermore, those rows do fit the pattern of feature channel coding, but they also do not look like they are computing the formula as described. On closer inspection, we see that the network has actually learned something clever: it has used a different representation of the same logical function, specifically:

$$\neg \Big(\begin{array}{r} (\neg x_3 \wedge \neg x_{11}) \ \vee \ (\neg x_9 \wedge \neg x_{12}) \ \vee \ (\neg x_2 \wedge \neg x_8) \ \vee \ (\neg x_0 \wedge \neg x_{10}) \vee \\ (\neg x_{13} \wedge \neg x_{14}) \ \vee \ (\neg x_4 \wedge \neg x_7) \ \vee \ (\neg x_1 \wedge \neg x_5) \ \vee \ (\neg x_6 \wedge \neg x_{15}) \end{array} \Big)$$

This can be seen to be equivalent to our original presentation of the formula by applying De Morgan's Rule at two different levels: once to the individual OR's of the clauses, and then again to the overall AND of the clauses. Of course, the network has no actual knowledge of De Morgan's Rule - it is simply given a subset of the truth table for the formula, and it learns some representation of that truth table. It is interesting to note that based on this example, the network does seem to have a propensity to learn DNF formulas instead of CNF formulas.

In a bit more detail, we see that the network is actually learning this formula as follows. The outer negation is realized through the utilization of the negative Layer 2 weights: if a positive signal comes through the channel codes, then that is converted into a significant negative value by these negative weights. The neuron at the second layer takes the OR of these negative weights: if any one of them evaluates to a (large enough) negative value, then the entire system evaluates to a negative value. If none of them are negative, then the small positive weights on the postive Layer 2 weight rows are enough to make the whole system positive. Each of the clauses is computed through a distinct feature channel code, and relies on the fact that $(\neg x_1 \wedge \neg x_2)$ can be computed in soft lagic as $\text{ReLU}(b - x_1 - x_2)$ where $b = 1$ in the binary case. Due to the negative values of the variables, the codes take on negative values (and so are depicted in red in this figure). We note that the Layer 1 bias is usually positive (corresponding to the +1 in that formula), but it is not usually as large as that formula dictates, which is analogous to what we see with DNF formulas, except with the opposite sign. Here though, some of the rows are negative, which is not analogous to the DNF formulas covered above, where we saw consistently negative bias in the positive weight rows. However, this is compensated for by positive values that appear elsewhere in the row. Due to the way the variables were set for the inputs, there is a high likelihood that every row will have an additional positive contribution, to make up for the slightly smaller bias.

E.2 Code Accuracy in a One Dimensional Vision Problem

We next turn our attention to a problem that extends beyond Boolean formulas, into a pattern matching problem that could be viewed as a one dimensional vision problem. We here consider the *consecutive four* problem: given an input string of binary values, are there 4 ones in a row? Thus the individual input variables are again binary. We consider this an extremely simple vision problem, in the sense that it deals with pattern matching with locality. With that locality in mind, we we will train and test our network with a restricted set of inputs to this problem: all inputs have exactly six ones in the string, and they must all be close to each other - within a region of eight consecutive bits of the input string. We want to see if neural networks trained to solve this problem have properties that are consistent with what we see with pure Boolean problems.

We point out that, like many pattern matching problems, this problem can be solved with a Boolean formula: the OR of all four ways ANDs of four consecutive bits. And in fact, even though the training process is not given this representation (only the ordered sets of inputs and their labels), using the Combinatorial Interpretability approach, we see that the networks we train to solve this problem do in fact use exactly that formula.

Our setup is as follows. We use a network with 128 input variables, 128 hidden neurons in the single hidden layer, and a single output neuron. We generate each training and test data input as follows:

- Flip a fair coin for each input to determine if it will be positive or negative.
- In either case, choose a sequence of eight consecutive variables uniformly at random.
- For a positive input, out of the set of eight chosen variables, pick a consecutive set of four variables uniformly at random. Also choose two other variables from the eight uniformly at random. These six variables are set to 1, and all others are set to 0.
- For a negative input, randomly pick a consecutive set of four variables out of the eight, but only set three of them to 1. Then pick another three of the eight variables uniformly at random, and set those

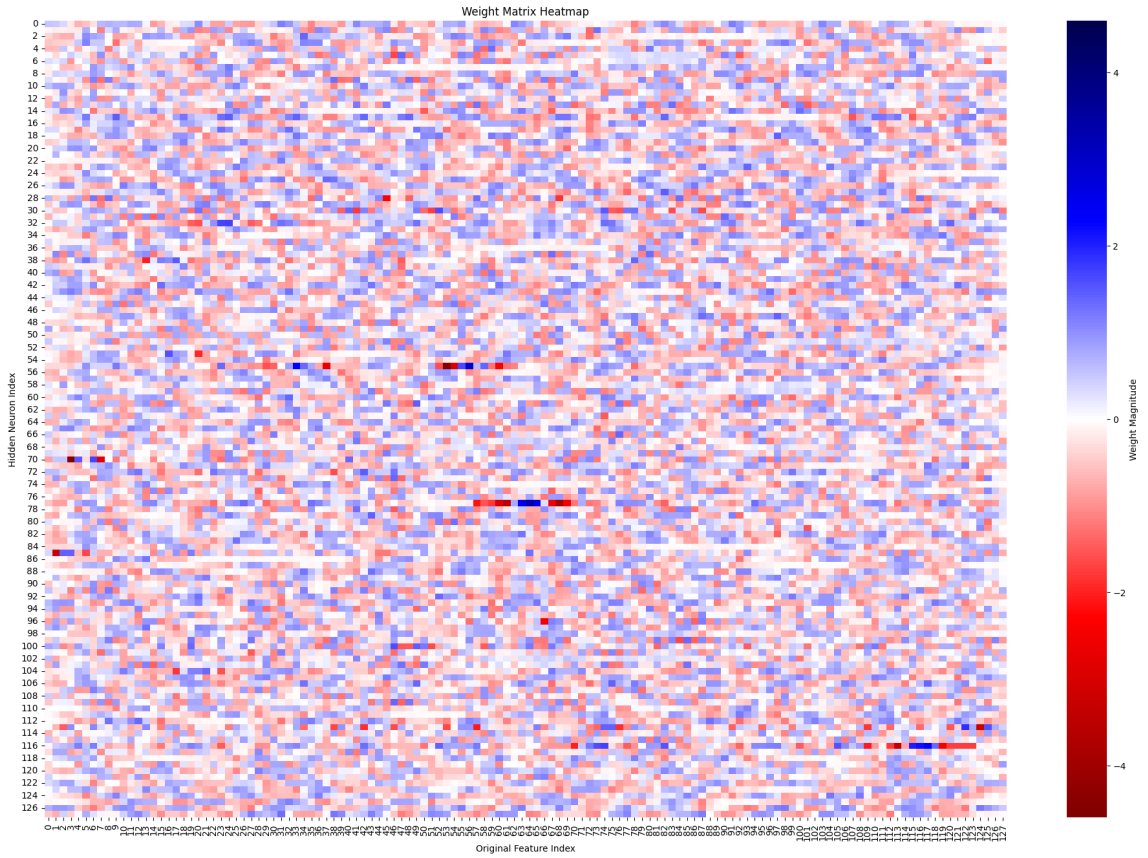

Figure 23: Layer 1 heat map for the connect four problem. The x-axis corresponds to the pixels of the one dimensional image we the network receives as input.

variables to 1 as well. Then test if the result has four consecutive variables set to 1. If it does, repeat until a negative input is generated, or 20 attempts have failed, in which case use the string of all zeros.

We generated 10,000 training inputs, and trained for 20 epochs, after which the network had 0 loss. Accuracy on the test set was 99%. The resulting weight matrix, shown in Figure 23, at first glance, might not seem to exhibit channel coding.

However, things become quite a bit clearer when we separate the rows into positive witnesses and negative witnesses, where (as before) a row is considered a positive witness if its Layer 2 weight is positive, and a negative witness if its Layer 2 weight is negative. The positive witness only heat map is depicted in Figure 24. This is a subset of the rows shown in Figure 23; no other changes have been made.

We can see visually that the positive witness rows do in fact demonstrate considerable feature channel coding. And in fact, we see that most of the rows of the (positive only) matrix consist of alternating sequences of four consecutive positive values, followed by a number of negative values. There are some sequences of ones that are longer than four, but most of them have length exactly four. This image alone provides convincing

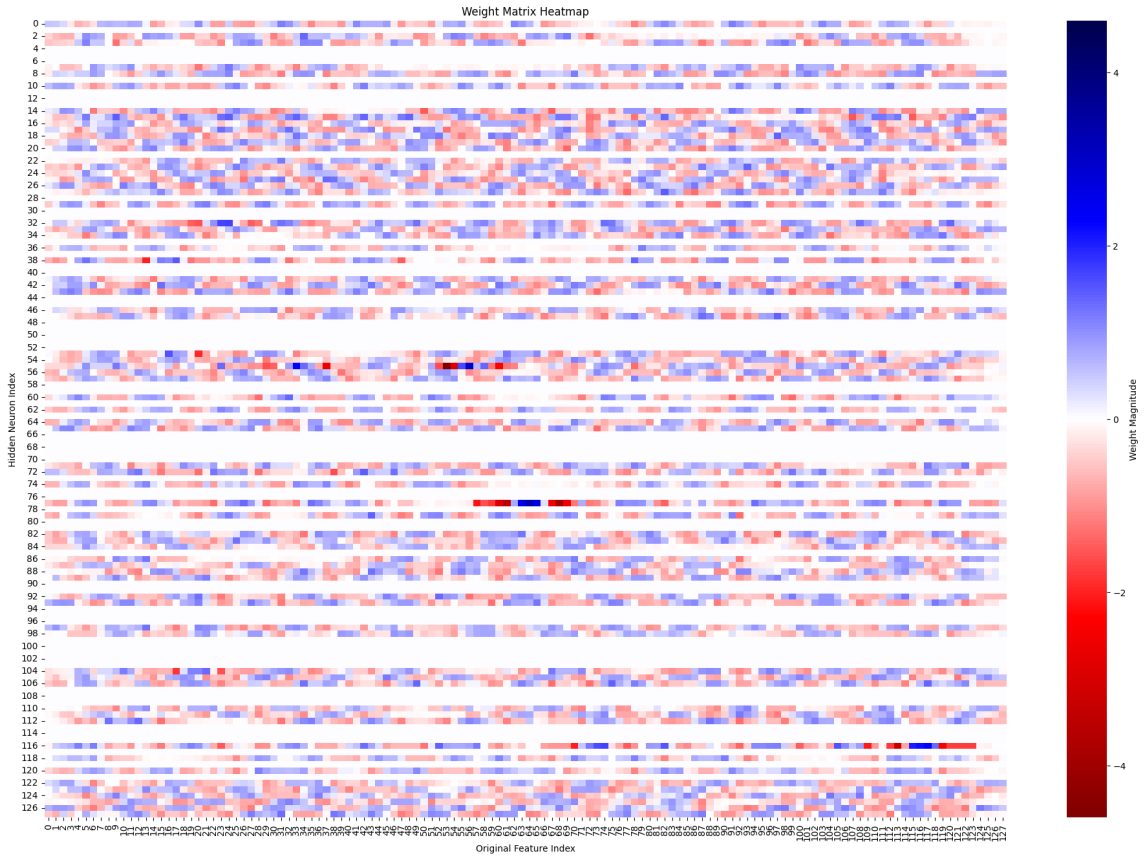

Figure 24: Layer 1 for connect four: positive witnesses.

evidence that this network uses feature channel coding, but we will demonstrate this with metrics as well. In this context, we consider a row to be participating in the code for a sequence of four consecutive ones starting at position i, if that row (a) corresponds to a positive layer 2 weight, and (b) it has four consecutive positive values starting at position i. The resulting statistics are summarized in Table 2.

These statistics clearly indicate that coding is being used here: an average of 12.77 rows code each possible positive example, and there is only an average overlap of 2.35 between rows. Furthermore, all coding channels clearly express a compution closely resembling an AND of the four bits that are being coded for: every single coding row has a negative bias (even though the biases are initialized uniformly and independently at random between -1 and 1).

The negative witnesses are depicted in Figure 25. We see here that the negative witnesses consist of alternating sequences of one positive value and one negative value, essentially computing an XOR.

We next show that we can extract and perform classification using the codes that are depicted in Figure 24, and measured in Table 2. We can view this process as a combinatorial replacement (for this specific network) of a sparse autoencoder: these codes correspond to how each feature is represented by the neural network by its positive coding rows. A feature here is any set of four consecutive ones in the input string. Extracting

Figure 25: Layer 1 for connect four: negative witnesses.

the codes is straightforward: we simply sweep across the different starting positions, and for each starting position extract the codes for that position per the definition of coding rows.

To use these extracted codes for classification, we construct a decoding algorithm that maps the Layer 1 post-activation values in positive rows of any input to a set of features that are active. In other words, the algorithm attempts to make a decision as to where there are four consecutive ones in the input string, not based on the entire neural network, but instead only on the Layer 1 post-activation values, and of those values, only those that are associated with positive witnesses. This serves three purposes: (1) it further demonstrates how to combinatorially interpret the trained model, (2) it demonstrates how well we can do with only that subset of information (and specifically without using any of the negative witnesses, and (3) it also demonstrates that the network is really learning the actual underlying features, as opposed to the summary (binary) classification problem it was asked to solve.

The algorithm proceeds by computing the Layer 1 post-activations of the network, and then comparing these values to the codes. Any feature is considered to be present if its code is entirely present. A code is entirely present if all of its rows have a post-activation value that is at least the sum of its four positive elements, minus its bias, minus 1.9 times the maximum value within its sliding window of eight consecutive bits. We test this algorithm on a set of inputs drawn from the same distribution as both the training set and the test set of inputs,

Statistic	Average	Std Dev
Codingness Statistics		
Total Number of Unique Coding Rows	70	-
Fraction of Contributions from Coding Rows	63.93%	10.29%
Number of Coding Rows	12.77	2.73
Weighted Sum of Coding Rows	9.0139	2.1991
Overlap Statistics		
Number of Overlapping Coding Rows	2.35	1.64
Weighted Overlapping Coding Rows	18.24%	12.71%
ANDness Statistics		
Percentage of Negative L1 Bias Coding Rows	100.00%	0.00%

Table 2: Summary of Codingness, Overlap, and ANDness

and find that this very simple decoding algorithm does a surprisingly good job of not just differentiating 0 inputs from 1 inputs, but also determining which features are actually present. The results from a single training run (and accompanying extraction of codes), tested on a set of 40 random test inputs is provided in Table 3. The actual matches are found simply by scanning the input for four consecutive ones; the code matches are from our extracted codes.

Test	Label	Actual Matches	Code Matches
0	1	[35]	[35]
1	0	—	—
2	1	[88, 89]	[88, 89]
3	1	[116, 117]	[116, 117]
4	1	[11, 12]	[12]
5	0	—	—
6	0	—	[116]
7	1	[18, 19, 20]	[18, 19, 20]
8	0	—	—
9	1	[40]	—
10	1	[15, 16]	[16]
11	1	[14, 15, 16]	[14, 15, 16]
12	0	—	—
13	0	—	—
14	1	[47, 48, 49]	[47, 48, 49]
15	0	—	—
16	0	—	—
17	0	—	—
18	1	[97]	—
19	1	[62, 63]	[62, 63]

Test	Label	Actual Matches	Code Matches
20	1	[4]	[4]
21	1	[106, 107]	[106, 107]
22	0	—	—
23	1	[29, 30, 31]	[29, 30]
24	0	—	—
25	0	—	—
26	0	—	—
27	1	[8, 9, 10]	[8, 9, 10]
28	0	—	—
29	1	[24, 25, 26]	[24, 25, 26]
30	1	[69]	[69]
31	0	—	—
32	0	—	—
33	0	—	—
34	1	[80, 81, 82]	[80, 81, 82]
35	0	—	—
36	0	—	—
37	1	[62]	[61]
38	0	—	—
39	0	—	—

Table 3: Sample output from testing: actual matches versus code matches.

We test this more thoroughly by running 50 different training runs, and for each training run testing the accuracy of the algorithm's predictions on a test set of 2000 inputs, drawn from the same distribution as the training set. We measure success by two criteria: how well did the algorithm predict whether or not there is a matching pattern of four consecutive ones, and how well did the algorithm predict the exact set of matches (which is the empty set if there is no match). Our results are summarized and compared to the original trained network in Table 4.

	FPR NN	FNR NN	FPR CODES	FNR CODES	FULLY CORRECT
Average	0.71%	1.65%	6.82%	7.66%	82.63%
Std Dev	0.33%	0.37%	1.82%	0.97%	1.80%

Table 4: Performance of Coding Interpretability Compared to Original Neural Network.

We note that while the results of our algorithm are noticably worse than the original network, it is still over 92% accurate in terms of the decision problem, and over 82% accurate in terms of picking out the exact set of matches (which the original network does not do). These results are important for a number of reasons:

- They further demonstrate how central the codes are to the computation.
- Even though the algorithm described above only uses the positive witnesses, they are able to achieve over 92% accuracy on the decision problem of whether there is a match or not.
- Even though the neural network was trained just to answer the decision problem, looking at the network through the lens of the resulting codes yields over 82% accuracy on precisely classifying where all the consecutive four sequences are. As we see from Table 3, often when it is wrong, it is very close to providing the correct answer (i.e., a sequence of 5 consecutive ones should return two matches, but the coding-based algorithm only returns the first one.)

We next turn to another phenomena we observed in studying this problem. This is not directly relevant to our central hypothesis of networks using feature channel coding, but it does demonstrate what can be discovered by using our combinatorial interpretability approach. To see this phenomena, we looked at all columns of the layer one weight matrix, and we computed all pairwise correlations between them. This is depicted in Figure 26. The strong self correlation on the diagonal is of course expected, and we see an expected pattern in the values close to that diagonal. We did not, however, expect to see such a strong and regular pattern in the upper and lower diagonal regions of this matrix, even for columns that represent inputs that are very far removed from each other.

Our best guess for the cause of this pattern is that there is a strong preference for a new set of four consecutive ones to appear on a different neuron after the current one has finished any existing pattern. This would cause regions of alternating sequences of "more likely to code" and "less likely to code", each of length four, which is consistent with the pattern. This preference presumably is fairly strong for it to not perceptibly fade throughout the entire matrix.

E.3 Polysemanticity and Superposition

Let us touch a bit more on the notions of polysemanticity and superposition (1; 11; 16; 18). When one looks at the set of neurons of a feature channel code as in our example above, we see that there is overlap. The same neuron is polysemantic and fires in response to multiple features. The accepted idea for how superposition is implemented is that neurons represent features using polysemanticity. To quote (11): "in the superposition hypothesis, features are represented as almost-orthogonal directions in the vector space of neuron outputs. Since the features are only almost-orthogonal, one feature activating looks like other features slightly activating." This is true, and yet our theory of feature channel coding suggests that superposition can also be explained by the combinatorial property of using feature channel codes that overlap. The noise of one feature slightly lighting up others is explained not just by alignment in vector space, but also by the use of multiple rows in the feature channel coding so the relative importance of a given neuron is lowered (1). Moreover, we don't need to decipher directionality in activation space or geometric representations in order to map the features captured by a collection of neurons: the feature channel coding in the weight matrices allows us to do it combinatorially without any activation information. Furthermore, as we saw in Section 5, using the combinatorial interpretation also allows us to analyze the code pattern distributions of networks to explain

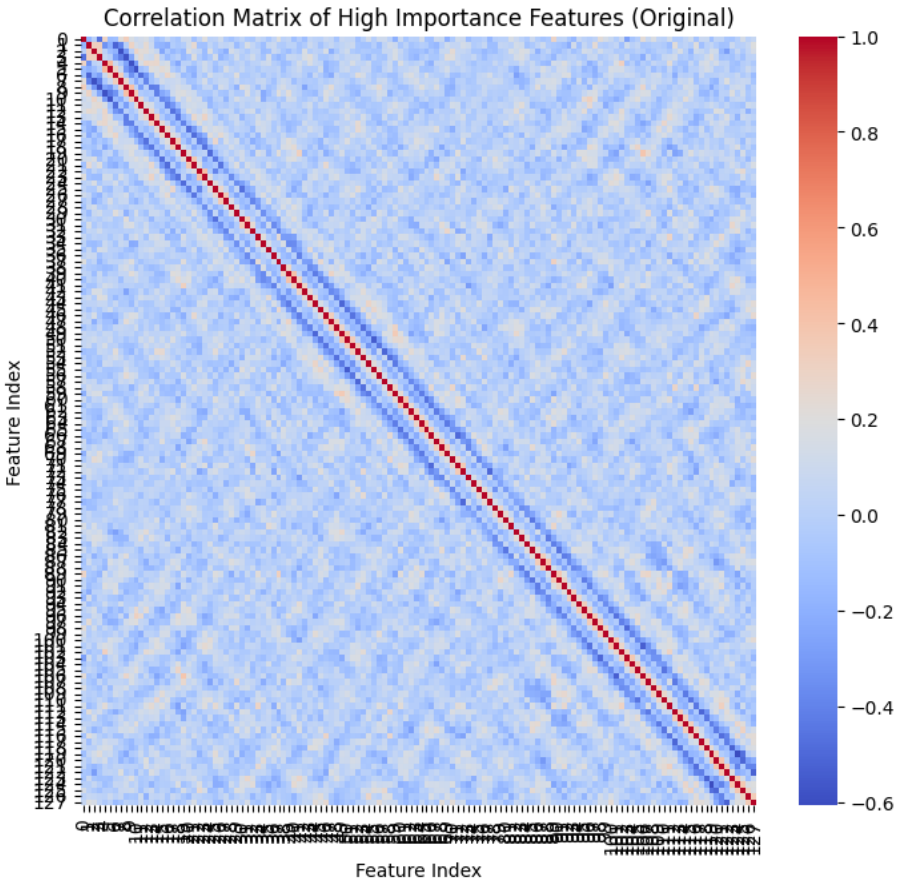

Figure 26: Correlation Matrix for columns of layer 1 weight matrix.

scaling laws in ways that seem impossible to capture by looking at features through their directionality in activation space.

Getting back to our example network described in Section 3, if overlapping codes introduce noise, given that our network was trained on 8 clauses using 16 neurons in the first layer, the training could have found a coding that had one code row per clause: one monosemantic neuron per feature. In this way, there would be no overlaps among codes, and thus no cross-feature noise. However, as in the matrices resulting from our training, we see that the training settled on multiple overlapping codes and the neurons are polysemantic. In other words, polysemanticity appears even when there are fewer features than neurons.

We believe that there is a combinatorial explanation for this. Gradient descent searches the state space for local minima. There are many more encodings of the desired computation using overlapping codes than using non-overlapping monosemantic ones, and thus likely many more local minima involving feature channel coding using multiple overlapping polysemantic neurons. Thus, gradient descent is more likely to find such an encoding than the monosemantic one, and then not be able to move from that local minimum. This is somewhat reminiscent to the findings of (20). As witnessed in our example, coding (and with it polysemanticity) does appear in cases where one does not need to compute in superposition. It would be

interesting to obtain more evidence or a proof of this hypothesis, which could be done using an approach similar to the one we take in the work that produced Figure 4.

F FURTHER RESEARCH

We believe the combinatorial approach and its first application through feature channel coding, can have implications even before we have the ability to apply it to large real world networks. In particular, in the same way we used it to understand scaling laws, one can attempt to understand other important neural computation aspects such as sparsity, quantization, and the linear representation hypothesis, to name a few. Understanding to what extent computation in real production neural networks contains Boolean features and tehir codes is also of great interest. Finally, it has not escaped our notice that the specific coding we have postulated immediately suggests a possible computation mechanism for neural tissue.