On the Complexity of Neural Computation in Superposition

Micah Adler
MIT
micah@csail.mit.edu

Nir Shavit
MIT & Neural Magic
shanir@mit.edu

Abstract

Recent advances in the understanding of neural networks suggest that superposition, the ability of a single neuron to represent multiple features simultaneously, is a key mechanism underlying the computational efficiency of large-scale networks. This paper explores the theoretical foundations of computing in superposition, focusing on explicit, provably correct algorithms and their efficiency.

We present the first lower bounds showing that for a broad class of problems, including permutations and pairwise logical operations, a neural network computing in superposition requires at least $\Omega(m' \log m')$ parameters and $\Omega(\sqrt{m' \log m'})$ neurons, where m' is the number of output features being computed. This implies that any "lottery ticket" sparse sub-network must have at least $\Omega(m' \log m')$ parameters no matter what the initial dense network size. Conversely, we show a nearly tight upper bound: logical operations like pairwise AND can be computed using $O(\sqrt{m'} \log m')$ neurons and $O(m' \log^2 m')$ parameters. There is thus an exponential gap between computing in superposition, the subject of this work, and representing features in superposition, which can require as little as $O(\log m')$ neurons based on the Johnson-Lindenstrauss Lemma.

Our hope is that our results open a path for using complexity theoretic techniques in neural network interpretability research.

1 Introduction

A collection of groundbreaking publications [5, 8, 19, 23] by researchers at Anthropic present compelling evidence that features, functions applied to the input to recognize specific properties, are a fundamental computational unit of neural networks. It is also believed that superposition [3, 7], the computational phenomenon allowing a single neuron to fire as part of a polysemantic representation [23, 25] of many different features, is key to how neural networks compute [8]. This is in contrast to monosemantic representations, where each neuron only represents a single feature. There has recently been exciting work on the problem of taking a trained neural network, and extracting the monosemantic features from the superposed representations of the network [5, 8, 23]. This has been primarily driven by safety considerations, since being able to understand the features being computed by a neural network is crucial in order to understand the logic being used.

It has been suggested that because the set of features that are active at any given time is very small relative to the total number of features, superposition allows the network to represent the current state of a computation much more efficiently [8]. Superposition can be thought of as a compressed representation of the current state, and this more succinct representation allows the neural network to represent many more features than it has neurons. Furthermore, by doing superposition the right way, the neural network performs computation in that superposed representation. It is conjectured that computing in superposition is important from an efficiency perspective since the work on extracting features seems to indicate that large neural networks use sets of at least hundreds of millions of features and quite likely orders of magnitude more than that [23], and so inference in the non-compressed form would require tremendous computational resources [8, 19]. The superposed representation of the neural network provides compression, much greater computational efficiency and also lends itself well to all of the GPU optimizations that have made the AI revolution possible.

In this work, we address the question of whether it is possible to design explicit and provably correct algorithms for computing in superposition, and how efficient can such algorithms be? Our primary measure of efficiency is the level of superposition achievable: given a set of features to be computed, what is the minimum number neurons required to perform that computation in superposition, and how many parameters do those neurons require? Our work here is inspired by the important work of Vaintrob, Mendel, and Hänni in [24], that suggests the problem and shows a way of computing a k-AND circuit using a single layer of a neural network in partial superposition. We will elaborate on how we build and expand on this work in the related work section.

The superposition question is central to understanding how real trained neural networks compute. It also enables us to prove upper and lower bounds on the efficiency of neural networks. In fact, our lower bounds apply independently of representation. Furthermore, addressing this question allows us to separate the process of determining a feature set and a feature circuit (a description of what is being computed) from the process of mapping these to a superposed representation. We believe that during training, neural networks are implicitly doing both processes simultaneously, and that this disentangling (pun intended) of the computation in superposition from the process of finding the features and feature circuit opens up a much deeper understanding of neural networks, specifically what they are computing and how they work.

Finally, perhaps a bit further in the future, solving this problem could lead to a powerful way of designing neural networks by breaking the design problem up into two subproblems: (1) determine the feature set and feature circuit for a given problem domain, and then (2) map those using canonical techniques to a neural network in superposition. Subproblem (1) could be done either through feature circuit extraction from existing networks, or training an initial neural network to be monosemantic, and then using that as the feature circuit. The whole process can then be viewed as an algorithmically driven form of distillation [14].

1.1 Our Results

In this paper, we present nearly matching upper and lower bounds on computing in superposition. Essentially, we demonstrate that when computing in superposition, we can compress the representation of the features down to roughly the square root of the number of output features, but no further. These bounds and the techniques used to derive them open a number of further interesting questions concerning the design of neural networks, mechanistic interpretability, and further understanding the complexity of neural networks. We discuss these questions in the Conclusion.

We define our models below, but for now, let m be the number of input features to a monosemantic neural network description, and m' be the number of output features. Let n be the number of neurons being used in the superposed computation.

Lower Bounds

Our lower bounds apply to a very general model of computation that includes neural networks, and is independent of many implementation details, such as network structure, activation functions, etc. In this model, we introduce a lower bound technique that applies both to neural networks that must always be correct, as well as when the network, as in the real world, is allowed to make mistakes sometimes. This technique relies on what we call the *expressibility* of a neural network: how many different functions the neural network implementation (structure, activation functions, etc) is able to compute through different settings of the parameter weights of the model. We provide a Kolmogorov Complexity based proof that shows that if a neural network has high expressibility, the parameters of the final model must have a large description length, even if the model is allowed to make mistakes sometimes.

We use this technique to show that the minimum description of the parameters of the neural network must be at least

$$\Omega(m' \log m') \text{ bits},$$

for a broad class of problems that includes computing a permutation of the input features and computing m' pairwise ANDs of the m input features. For neural networks using square matrices and a constant number of bits per parameter, this implies that

$$n = \Omega(\sqrt{m' \log m'}).$$

This has several interesting implications for neural network compression, a topic of great interest given the memory limitations of today's GPUs [15]. The typical representation of the parameters of a neural network is as 8 or 16 bit values, and so asymptotically such neural networks will require $\Omega(m' \log m')$ parameters. Moreover, for *quantization* [15], the process of cutting the number of bits in parameter representations to improve performance, this bound suggests that reducing the number of bits beyond a certain point will require increasing the number of parameters.

Another form of compression of neural network representations is sparsity [10, 15]. It has been conjectured that typical neural networks contain a sparse sub-network with sometimes 90% or more of the original parameters removable while preserving accuracy. If this kind of sparsification is possible in most cases, then the sub-network has similar expressibility as the original network, and so our lower bounds imply that parameter sparsity is bound by the number of features the network is computing, and any "lottery ticket" sparse sub-network [10] must use at least $\Omega(m' \log m')$ bits to describe its parameters no matter the size of the original network it is derived from.

A final form of compression is knowledge distillation [14], where a large dense neural network is used to train a much smaller dense network preserving the original "knowledge" and hence the accuracy. Again, if this retains the expressibility of the original neural network, then there is a limit to how small a network can be distilled from a larger one without loosing accuracy.

To the best of our knowledge, these are the first general lower bounds on the number of parameters or the number of neurons required for neural network computation, i.e. without any structural assumptions [11] on the specific network architecture.

Several authors have pointed out that superposition is an application of the Johnson-Lindenstrauss Lemma [16]. It can also be viewed as a type of Bloom filter [4, 6]. In both cases, the previous work has focused on representation: how efficiently can the current state be represented? Our focus is on computation. In fact, it is not hard to show with either of these previous techniques that when there is only a constant number of active features, they can be represented together using $O(\log m')$ neurons. Since we demonstrate that $\Omega(\sqrt{m' \log m'})$ neurons are necessary for computation, our results imply an exponential gap between representation and computation.

Upper Bounds

According to [7, 12, 19] the ability of neural networks to compute in superposition is the result of *feature sparsity*: it is observed that in any layer only a small subset of input features are activated. Under that assumption, we provide an algorithm that takes a description of any m' pairwise ANDs of m inputs, and produces a neural network that correctly computes those ANDs using

$$n = O\left(\sqrt{m'} \log m'\right) \text{ neurons.}$$

This neural network does not make errors, and multiple layers of this network can be chained together. Our lower bound also applies to problems with feature sparsity, and so the upper bound is within a factor of $\sqrt{\log m'}$ to asymptotically optimal in terms of neurons. To the best of our knowledge, this is the first provably correct algorithm for computing a non-trivial function wholly in superposition.

We also introduce a concept closely related to feature sparsity, hinted at in [7], which we call *feature influence*: for a layer of the network, how many output features are impacted by a given input feature? We find that feature influence has a significant impact on what types of techniques are effective in computing in superposition, and for some functions, determines the ability to compute them in superposition at all. For the pairwise AND function, we show we can compute efficiently in superposition for any level of feature influence, but our main algorithm actually consists of three different techniques, and the pairwise ANDs are partitioned to divide the problem into pieces with similar feature influence.

One of the techniques we present applies to the low feature influence subset of ANDs being computed, specifically when no input influences more than ${m'}^{1/4}$ features. We consider this an important special case of the problem, since we expect that many real world models have low feature influence. Furthermore, the algorithm for this case introduces a general technique for computing in superposition, where inputs are routed to "computational channels" for each output, that are represented in superposition, and then computation is performed on those superposed channels. This technique may be of independent interest, and in fact, something like this may be happening in real neural networks.

Finally, we demonstrate extensions of the upper bound results in several directions:

- We demonstrate how to utilize the algorithms for multi layered networks.
- We show that they can be used to compute k-ways AND functions.
- These results can be extended to arbitrary Boolean functions. However, these extensions are beyond the scope of this paper, and will be presented in a subsequent manuscript. We do, however, here demonstrate that if the maximum feature influence of a 2-OR problem is too high, then it is not possible to compute that problem in superposition.

1.2 Related Work

Our work here is inspired by the groundbreaking work of Vaintrob, Mendel, and Hänni in [24], that lays out the problem of computing in superposition and shows a way of computing a k-AND circuit using a single layer of a neural network partly in superposition. Their work paves the way for ours, but falls short in several fundamental ways which we aim to improve and expand here.

First and foremost, we say "partly in superposition," because in their problem setup the neurons are in superposition, but neither the inputs nor the outputs are in superposition, which significantly simplifies the problem and avoids some its main challenges. In our work the computation as a whole is in superposition: the inputs, the neurons, and the outputs.

Secondly, their technique allows them to compute with a single neural network layer but does not extend to multiple layers due to error blowup. The new approach we propose here allows us to remove the error for an arbitrary depth of layers.

Another powerful aspect of our work relative to [24] is that we show you can use (and/or think about) much larger matrices in the process of constructing the layers of the algorithm and then multiply these matrices together for the construction, getting matrices and vectors of size dependent only on the number of neurons. The combined techniques allow us to implement arbitrary width AND expressions and arbitrary depth circuits.

Lastly, we introduce here a general framework for reasoning about superposition in neural networks. We separate the circuit being implemented from the neural network implementing it, with the observation that superposition is about the relation between the two. Unlike [24], this allows us to introduce the study of the complexity of a neural network algorithm implementing a circuit with a given level of superposition, capturing this complexity via not only the number of neurons but also the number of parameters in the network. In particular, unlike [24], we present not only upper bounds, but also for the first time lower bounds on the complexity of computing in superposition.

Another paper that studies the impact of superposition on neural network computation is [21]. However, they look at a very different question from us: the problem of allocating the capacity afforded by superposition to each feature in order to minimize a loss function, which becomes a constrained optimization problem. They do not address the algorithmic questions we study here.

Our work is also inspired by various papers from the reserach team at Anthropic [7, 8, 19]. Apart from their work influencing our general modelling approach, their findings also inspired our definitions of feature sparsity and feature influence. We also recognize that there is a body of work [11, 13] on lower bounds for the depth and computational complexity of specific network constructions such as ones with a single hidden layer, or the number of additional neurons needed if one reduces the number of layers of a ReLU neural network [2]. Our work here aims at a complexity interpretation relating the amount of superposition and the number of parameters in the neural network to the underlying features it detects, a recent development in mechanistic interpretability research [5, 7, 8, 19, 23].

2 Modeling Neural Computation

Our goal is the following: for a given set $\mathcal{F}$ of computational problems and set of possible inputs X to those problems, design an algorithm for converting a description of any $F \in \mathcal{F}$ into a neural network $N(F)$ that computes F in superposition for any input $x \in X$, possibly incorrectly for some of the $x \in X$. In the example of this we study in depth here, $\mathcal{F}$ is the set of all possible mappings of m Boolean input variables to m' Boolean output variables, where each output variable is the pairwise AND of two input variables. $F \in \mathcal{F}$ is a specific mapping of pairwise ANDs, and X is the set of all settings of the m inputs where only a small number are simultaneously active. Thus, there are two layers of algorithms here: there is the algorithm that converts any $F \in \mathcal{F}$ into $N(F)$, and then $N(F)$ is itself an algorithm for computing the result of F

on any input $x \in X$.

We here use two different models of computation for the neural network $N(F)$. For our lower bounds, we consider a general model for computation, called parameterized algorithms. This model includes any neural network algorithm, and so our lower bounds apply to neural networks but also more broadly. Our upper bounds utilize a specific type of parameterized algorithm, based on a widely used type of neural network. Both of these are described below. We also provide a framework for describing the computational problem F, called a feature circuit. This is also described below.

2.1 Parameterized algorithms

We use the term *parameterized algorithm* to be any algorithm that computes on an input based on a set of parameters and produces an output. The function that the algorithm computes can depend on the parameters. In our main lower bounds, we do not require any further assumptions on how computation proceeds. We can think of the parameterized algorithm as a black box that computes different functions $F \in \mathcal{F}$ based on different parameters. As a result, our lower bounds apply to a neural network with any structure, any activation function, or any other aspect of the computation, or even some other structure that has parameters but would not be considered a neural network. Any such structure can be a specific parameterized algorithm, and it will have a set of different functions $\mathcal{F}$ that it can compute based on its parameters. We use this model to prove lower bounds on the description length of the parameters required, based on the expressibility of the parameterized algorithm. Essentially, we show that if $\mathcal{F}$ is large then the number of parameters must also be large, both in the case where the parameterized algorithm is always correct, and when it can make mistakes on some of the inputs $x \in X$.

A neural network could be described as a parameterized algorithm in two different ways: the parameters can describe the structure of the neural network, as well as the weights associated with that structure, or the parameters can represent just the weights, in which case the structure of the network is considered part of the algorithm itself. Once we have a lower bound on parameters, we use this latter approach. Specifically, we assume a network structure that relies on the type of square $n \times n$ matrices we use in our upper bound model described below. With that structure, any lower bound of B on the number of parameters directly implies a lower bound of $\sqrt{B}$ on the number of neurons. Note that in this case, the lower bound does not count the description of the network itself, making the lower bounds even more compelling than if they included the description of the network layout and computation.

In the past, the notion of parameterized algorithms [1, 9] was used to refine known algorithms by introducing a small set of input dependent parameters, in addition to inputs and outputs, as a complexity measure. This was useful for tackling NP-hard problems by isolating the complexity to a parameter that is small in practical instances so as to provide efficient solutions for otherwise intractable problems. Here, we re-introduce this notion, but with a twist: the understanding that in this new variant of parameterized algorithms (1) the parameters define the algorithm and (2) the cardinality of the parameter set is a complexity measure in itself.

The study of this new type of parameterized algorithms is important today because of a fundamental shift in the focus of software development. Until recently, the bulk of software in the world was dedicated to databases and analysis problems, where the description of the algorithms is usually small relative to the input size, and where the complexity of algorithms is first and foremost a function of the input size.[1] Deep learning is quickly changing this balance, introducing a new form of software in which the size of the algorithm's description can be large relative to the input, and the complexity of the algorithm is often dominated by this size.

In our upper bounds, we will study a parameterized algorithm computed by a specific type of neural network: a standard multi-layer perceptron architecture [17]. Each neuron in this neural network performs a dot product of its inputs and its weights (parameters), then adds a bias followed by a ReLU non-linearity. A layer of this network has n neurons. If the inputs and outputs of that layer are represented in superposition, then the layer will also have n inputs and n outputs. We can view the resulting computation as follows. The input is an n-vector. The dot products of all neurons are computed by multiplying that vector by an $n \times n$ matrix and the bias is done by adding an n-vector. The ReLU is performed on each entry of the resulting vector independently, producing the output n-vector. The parameters of this computation are the entries of the $n \times n$ matrix and the bias vector. We will use this representation of the computation in our description of the upper bound.

The network can have any number of layers d. In practice though, d is typically much smaller than n. For simplicity, we do not include batch norm or max pooling computations and stick to ReLU as the source of non-linearity. Since our lower bounds apply to any activation function, and our upper bounds nearly match our lower bounds, computing with other nonlinearities (for example, quadratic activations as suggested in [24]) does not provide much benefit for the class of problems we consider here. In fact, it is possible that other non-linearities are not able to match our upper bound results. As we noted above, we also restrict the neural network to inputs and outputs that are Boolean. We suspect this does not result in any loss of generality, since Boolean variables can represent more fine grained values to arbitrary precision. We leave as an open question whether there are problems where removing these two restrictions (ReLU and Boolean), results in significantly greater computational power in the models we consider here.

[1]We note that Boolean circuits are a historically well studied form of parameterized algorithm that applies to hardware and is used in complexity theory, but accounts for a small fraction of real world software.

2.2 The Feature Circuit View of Neural Computation

We next provide a framework for describing F, the computational problem our parameterized algorithm $N(F)$ is intended to solve. We follow the intuitive definition in [8, 24] of capturing neural network computation as a set of output features that are the result of applying a circuit to a set of input features.

More formally, as depicted in Figure 1 below, a feature set is a set of functions $f_1, f_2, ..., f_m$ applied to an original input (an image, text etc). A feature circuit is an abstract computation that receives an input vector of m inputs, the results of the feature set computation on the original input, and computes a vector of m' outputs that are features $f'_1, f'_2, ..., f'_{m'}$ defined by the input feature set and the feature circuit applied to the inputs. Though the inputs and outputs of this circuit could be in any range, we choose to simplify it as depicted in Figure 1: the f_i and f'_i are Boolean indicators of whether a given feature appears or not in the input (and output, respectively). The feature circuit is thus a Boolean function from a Boolean input vector, indicating which features from the feature set appeared in the input, to a Boolean output vector indicating which output features appeared in the original input. For example, if the input features include detecting 4 legs, a tail, and a collar, then output features for cat and dog might be valid outputs of the feature circuit computation.

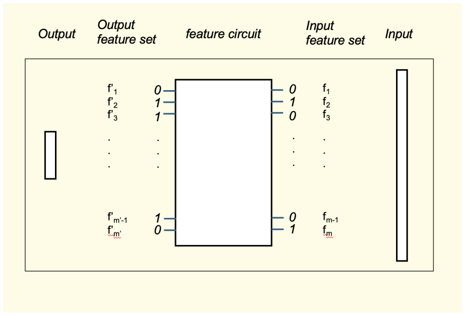

Figure 1: The feature circuit representation of neural network computation.

The circuit can be described using a neural network [20], a Boolean circuit, or any other algorithmic description of a Turing computable function. We note that one can use a single feature circuit to describe the computation of a neural network as a whole, or use a cascade of feature circuits, as depicted in Figure 2, to describe the details of the computation of the layers of such a network, where each layer's input feature set is a function of the previous ones. The final output can be a readout of the final layers output features, the equivalent of a softmax or some more complex function that might require another layer of computation. We do not touch on this exercise as it has little effect on the complexity measures we study here.

We also do not attempt here to address the problem of determining the feature sets - we simply assume that they are given to us, and we look at the problem of mapping them to an efficient neural network computation. However, it is worth pointing out that significant research is currently ongoing that is relevant to this problem [25, 8, 12].

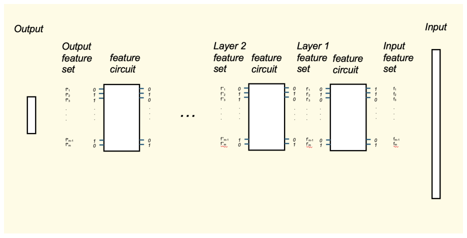

Figure 2: The feature circuit representation of multiple layers of a neural network.

2.3 Complexity and Superposition

Our primary measure of complexity is n, the number of neurons used by the neural network $N(P)$ for computation, but we will also examine the number of bits required to represent parameters. We describe the computation to be performed using a feature circuit. We are agnostic as to how the feature circuit F, specifically the features f_i, are described. However, we do point out that if extracting the logic behind the feature circuit is resource intensive, this will cause the algorithm that translates F into $N(F)$ to be resource intensive as well. In general though, given our focus on n, we do not provide an analysis here of the computational requirements of translating F into $N(F)$. However, all algorithms we provide in our upper bounds have a running time that is polynomial in m. We also are confident that they are more efficient than running a traditional training algorithm to determine $N(F)$.

The inputs to a feature circuit are said to have *feature sparsity* if only a small fraction of the m input features entries are activated (are non-zero) for every input. This corresponds to observed behaviors of real neural networks [12]. There is one more property of a feature circuit that we wish to examine: feature influence. The *feature influence* of an input x_i in a feature circuit is the number of output features f_j, for which there exists a setting s of the other inputs, such that keeping those other inputs fixed at s while changing x_i results in a change to f_j. The maximum (average, minimum, respectively) feature influence of a feature circuit is the maximum (average, minimum, respectively) influence of all its input features. As mentioned above, the algorithms of our upper bounds for 2-AND work for all variations of feature influence, but it does have a significant impact on what techniques are deployed in those algorithms. We also point out in Section 5.4 that if the maximum feature influence of a 2-OR problem is too high, then superposition is not helpful.

We will say that a layer of a neural network *computes in superposition* if $n < m'$, 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.

We focus our discussion on the problem of converting a feature circuit on feature sparse inputs into a superposed neural network. We assume that both the input and the output to that superposed neural network are provided in superposition, but if this is not the case (for example, at the first layer of a neural network), then it is straightforward to transform it into that form. The main question we are asking is how much superposition is possible: can we use significantly less neurons in computation than we have outputs that we are producing.

2.4 The k-AND problem

Our upper bound work will focus on a specific algorithm that converts a 2-AND feature circuit into a superposed neural network computation. In a 2-AND feature circuit, the input is m Boolean variables, and the output is m' different conjunctions of any two of these m inputs $\left(m' \leq \binom{m}{2}\right)$ returning a 1 in any output entry corresponding to two non-zero inputs, and zero otherwise. Our interest will be in computing this function using a superposed network that has significantly fewer than m' neurons. Later in the paper, we will explain how to compute k-AND from a collection of 2-AND computations, where a k-AND feature circuit produces m' different conjunctions of k inputs.

3 Lower Bounds

We here present our lower bounds for parameterized algorithm. We start by assuming that the parameterized algorithm does not make any errors but will add errors to the mix later below. For the error-free case, we are perhaps slightly more formal than might be necessary; we do so in order to set up a framework that makes it much easier to demonstrate the lower bound for when the parameterized algorithm can make mistakes.

Let U and V be finite sets, and let $\mathcal{F}$ be a set of distinct functions that map $U \to V$. Let T be a parameterized algorithm that can compute any function $F \in \mathcal{F}$ where the decision which function is computed by T only depends on the setting of the parameters of T. Let $P(F)$ be any description function describing the parameters used by T when T is computing F, and let $|P(F)|$ be the length of the description $P(F)$ measured in bits.

Theorem 3.1. *For almost all $F \in \mathcal{F}$, $|P(F)| \geq \log_2 |\mathcal{F}|$.*

It is important to stress that this is not a bound on an instantiation of an algorithm that has been designed to compute a specific function $F \in \mathcal{F}$, for example, a neural network with an architecture for computing a single 2-AND function by keeping only the weights that connect the inputs to neurons fitting

F's desired 2-AND outputs. Such a network would have very low expressibility, and in fact, in our parameterized algorithm model, it would not require any parameters at all. Rather, this theorem is a bound on a network that can compute arbitrary $F \in \mathcal{F}$s, and the choice of which F is dependent on the setting of the parameters. This is similar to Komogorov Complexity, which provides lower bounds on the description length of almost all strings in a set, but does not provide lower bounds on the description length of individual strings.

Proof. We prove this by assuming that there is a T that violates this theorem and show that such a T would allow us to construct a protocol to transmit more information between two parties than is possible. Specifically, we show that T would allow us to transmit $|\mathcal{F}|$ distinct messages using less than $\log |\mathcal{F}|$ bits. This is a contradiction since we know from Kolmogorov Complexity that representing each of $|\mathcal{F}|$ elements uniquely as a binary string requires using at least $\log_2 |\mathcal{F}|$ bits for almost all elements. For simplicity, we assume that $|\mathcal{F}|$ is a power of 2, but this technique generalizes to arbitrary $|\mathcal{F}|$.

We assume that we have two parties, Alice and Bob, where Alice will send Bob the identity of an arbitrary $\log |\mathcal{F}|$-bit string. Prior to the start of the transmission protocol, Alice and Bob agree on what the protocol will be, and as a precursor to that protocol, they can exchange any arbitrary information. Once the protocol is set up, Alice is then given an arbitrary $\log |\mathcal{F}|$-bit string, and her task is to inform Bob of what string she has been given using the protocol. We are concerned with how many bits Alice must send to perform this task. We use the following protocol:

Setup:

- Alice and Bob are both given T, $\mathcal{F}$, and $P(F)$ for all $F \in \mathcal{F}$.
- Alice and Bob agree on a bijection B from the elements of $\mathcal{F}$ to $\log |\mathcal{F}|$-bit strings.

Protocol:

- Alice is given S, a $\log |\mathcal{F}|$-bit string.
- Alice determines $F = B(S)$, the $F \in \mathcal{F}$ that S corresponds with in B.
- Alice sends Bob $P(F)$. By assumption, this requires less than $\log |\mathcal{F}|$ bits for most $F \in \mathcal{F}$.
- Bob now uses T and $P(F)$ to determine what F is. To do so, Bob uses T with parameter settings defined by $P(F)$ to compute $F(x)$ for every possible input to F. This may be computationally very expensive, but we are only concerned with communication.
- Once Bob knows F, he can use B to recover S.

We have shown that you can send any message you want with Alice's protocol, and since that message can be sent with fewer bits than Kolmogorov complexity tells us is possible, we have reached a contradiction.

□

Note that to convey $P(F)$ in the above protocol, Alice does not need to include any ordering information of the parameters since Alice and Bob can use some agreed-upon ordering for encoding the parameters, so the lower bound really does represent how many bits are required to represent the parameters themselves as opposed to any ordering information on the parameters.

3.1 Parameterized algorithms with errors

The lower bound we just showed does not really capture what is going on in neural networks since neural networks are typically allowed to make mistakes on some inputs. There are two different ways that an algorithm can make mistakes: either there are random choices during the execution of the algorithm and a mistake is dependent on those choices, or there is some subset of the inputs where the algorithm will always make a mistake. The former type of mistake is actually relatively easy to incorporate into the above protocol. Bob would simply try every input enough times to have very high confidence in the resulting mapping (assuming that the parameterized algorithm returns the correct answer with probability greater than 0.5). By doing so, he is very likely to find the correct answer, and we again reach a contradiction (although some care needs to be taken to ensure it is in fact a contradiction when the correct message is not transmitted 100% of the time).

The harder case for errors is the one that represents what happens with neural networks in practice: the algorithm T parameterized by $P(F)$ correctly computes $F(x)$ for some inputs x but is allowed to make mistakes on other inputs. The lower bound we show does not depend on how the mistaken inputs are chosen – they can be randomly or arbitrarily chosen, but we here argue the case where they are arbitrarily chosen. More specifically, given a set of functions $\mathcal{F}$, a set X of allowed inputs to $\mathcal{F}$, and any $\epsilon < 0.5$, the goal is to define a parameterized algorithm T such that for any $F \in \mathcal{F}$, T parameterized by $P(F)$ must determine $F(x)$ correctly for at least a fraction $(1 - \epsilon)$ of the inputs $x \in X$. We say that any T with this property on a set of functions $\mathcal{F}$ and inputs X computes $\mathcal{F}$ ϵ-correctly.

In this case, we are not able to demonstrate as general a result as the error-free case since the ability to make errors could allow the algorithm to use a smaller parameter description. Consider, for example, the case where all of the differentiation between the functions in $\mathcal{F}$ is on the same input (i.e., all other inputs always map to the same result). In this case, T can have a single error on that one input for all $F \in \mathcal{F}$, which means that there is effectively only one function that T needs to compute, which in turn means that no parameters at all are needed. Instead, we will prove a lower bound based on a subset of functions in $\mathcal{F}$ that can always be distinguished from each other. Specifically, we say that $\mathcal{F}' \subseteq \mathcal{F}$ is β-robust if for all $F_1, F_2 \in \mathcal{F}'$ with $F_1 \neq F_2$, F_1 and F_2 map strictly more than a fraction of β of the inputs to different outputs.

Let $\mathcal{F}$ be any set of functions $U \to V$ such that there exists a non-empty subset of $\mathcal{F}$ that is 2ϵ-robust. Let $\mathcal{F}'$ be a largest such subset, i.e. having the maximum cardinality. Let T be a parameterized algorithm that computes $\mathcal{F}$ ϵ-correctly for any $\epsilon < 0.5$. Let $P(F)$ be any description function describing the parameters used by T when T is computing F, and let $|P(F)|$ be the length of the description $P(F)$ measured in bits. Let $T(Y, x)$ be the result of running algorithm T with parameter settings defined by Y on input x.

Theorem 3.2. *For almost all $F \in \mathcal{F}'$, $|P(F)| \geq \log |\mathcal{F}'|$.*

Proof. We will again demonstrate a protocol for Alice and Bob to transmit a binary string more efficiently than should be possible and thus reach a contradiction. In this case, however, instead of using all the functions in $\mathcal{F}$ as codewords, we will use the functions in $\mathcal{F}'$ as the codewords, and the robustness of those functions means they can be used as the codewords to an error-correcting code.

Setup:

- Alice and Bob are both given T, ϵ, $\mathcal{F}$, $\mathcal{F}'$, and $P(F)$ for all $F \in \mathcal{F}$.
- Alice and Bob agree on a bijection B from the elements of $\mathcal{F}'$ to $\log |\mathcal{F}'|$-bit strings.

Protocol:

- Alice is given S, a $\log |\mathcal{F}'|$-bit string.
- Alice determines $F = B(S)$, the $F \in \mathcal{F}'$ that S corresponds with in B.
- Alice sends Bob $P(F)$. By assumption, this does not require at least $\log |\mathcal{F}'|$ bits for almost all $F \in \mathcal{F}'$.
- Bob computes $T(P(F), x)$ for every possible $x \in U$. This defines a function F^*.
- Bob compares F^* to every function in $\mathcal{F}'$ and finds the function $\hat{F} \in \mathcal{F}'$ that matches F^* on the most inputs x. Bob determines that $F = \hat{F}$.
- Based on this F, Bob uses B to recover S.

Bob is guaranteed to determine the correct F since the 2ϵ-robustness of $\mathcal{F}'$ ensures that F^* will have more outputs that are the same as F than any other $F' \in \mathcal{F}'$, provided that at least a fraction of $(1 - \epsilon)$ of the inputs are computed correctly. □

We next demonstrate how to apply this to the 2-AND function. As an intermediate step towards that, we first prove a lower bound on parameterized algorithms for the problem of computing a permutation. While we do not formally provide a reduction from permutation to 2-AND, permutation does provide intuition for how to tackle the 2-AND problem, and it really is the inherent permutation required by the 2-AND problem that allows us to prove that lower bound. For the permutation problem, let $\mathcal{F}$ be the set of all permutation functions on elements of a set U. Let T be a parameterized algorithm that computes $\mathcal{F}$ ϵ-correctly for any $\epsilon < 0.5$.

Corollary 3.2.1. *Any such T requires a parameter description length of at least* $\log[((1-2\epsilon)|U|)!]$.

Thus, for any $\epsilon < \frac{1}{2}$, any such T for the permutation function requires a parameter description length that is $\Omega(|U|\log|U|)$.

Proof. By Theorem 3.2, we only need to demonstrate a subset $\mathcal{F}' \subseteq \mathcal{F}$ that is 2ϵ-robust and has size $(|U|(1-2\epsilon))!$. To build such a subset, we will essentially construct a permutation code [22]. However, since we are only concerned with the asymptotics of the logarithm of the subset $\mathcal{F}'$ (instead of achieving full channel capacity), we can get away with a fairly simple construction. To construct such a subset, we start with any element $F \in \mathcal{F}$ and place it in $\mathcal{F}'$. We then remove F from consideration as well as any other permutations that do not differ from F in strictly more than a fraction of 2ϵ of the inputs. We iterate this process until everything from $\mathcal{F}$ has either been placed in $\mathcal{F}'$ or removed from consideration.

Each item that is placed in $\mathcal{F}'$ can remove from consideration at most $\binom{|U|}{2\epsilon|U|}(2\epsilon|U|)!$ permutations, and so the total number of permutations placed in $\mathcal{F}'$ is at least

$$\frac{|U|!}{\binom{|U|}{2\epsilon|U|}(2\epsilon|U|)!} = ((1-2\epsilon)|U|)!.$$

□

Corollary 3.2.2. *Any parameterized algorithm T that computes the 2-AND function at least ϵ-correctly where the input has m entries and the output has m' entries, for $m' \leq \binom{m}{2}$, requires a parameter description length of at least $\Omega(m' \log m')$.*

Note that we have made no assumptions here about whether the inputs and/or outputs are stored in superposition, and so this bound applies in all four combinations of superposition or not. Also, note that we can assume that $m' \geq \frac{m}{2}$ since if m' is smaller than that, then we can remove any unused input entries from the problem, thereby reducing m.

Proof. For any m' and m we can construct a set X of inputs and a set $\mathcal{F}$ of 2-AND functions that demonstrate this lower bound. A function $F \in \mathcal{F}$ will consist of any m' pairwise ANDs of the m inputs. All other functions in $\mathcal{F}$ will compute the same m' pairwise ANDs, but in a different order. All possible orderings (i.e., permutations) of those pairwise ANDs will appear in $\mathcal{F}$, and there are no other functions in $\mathcal{F}$. This gives $\mathcal{F}$ such that $|\mathcal{F}| = m'!$. For the set X of inputs, we will only allow two hot (exactly two entries are set to 1) inputs, and only those two hot inputs where one of the ANDs in F is the AND of those two inputs. (Note that if $m' \ll \binom{m}{2}$, then most two-hot inputs will not result in entry of the output being a 1; hence the restriction.) All inputs $x \in X$ have m entries, but the number of inputs in X is m', so $|U| = |V| = m'$. We are essentially computing all the permutations of the output entries. The Corollary now follows from the exact same argument as was used for the permutation function. $\square$

We point out that for any neural network performing matrix multiplication using square matrices, this lower bound implies that the number of neurons required will be at least $\Omega\left(\sqrt{m' \log m'}\right)$. Note that a superposition representation of an m variable Boolean input with a constant number of active variables can be as small as $O(\log m)$ bits, but when we want to compute some function, such as a permutation or 2-AND, the description of the parameters of the neural network performing that mapping must be exponentially larger than that!

3.2 Other potential applications

The last topic we address here with respect to lower bounds is showing lower bounds on the parameterization of the problems we use neural networks for in practice, such as LLMs and image generation. In these cases, it might seem like there is only one (or a small number) of functions being computed, such as a specific LLM, and so this lower bound does not apply. In theory, this is probably true: if there exists an algorithm that computes that LLM, there exists a parameter-free algorithm for it. In practice, however, that is not how LLMs are constructed. Instead, all existing neural networks use a general network structure that could be trained to compute very different functions through their parameterization. They start with a structure that has a high degree of expressibility that is then made more specific through a training process that sets their parameters. Our lower bound applies to the expressability of the neural network prior to training.

As a result, our lower bound technique could potentially be used to prove lower bounds on the parameterization of LLMs. While we do not prove any such lower bounds, we demonstrate here that it has the potential to do so. The idea is that if we take a fixed neural network structure, and train it on two very different data sets, this should lead to it computing very different things. (The details of the training process itself can cause differences as well, but we ignore that here). For example, for a neural network structure being used as an LLM, we will get very different results if we train it with a data set consisting entirely of English language text versus only using Mandarin Chinese. We formalize this concept here using our lower bound framework:

Corollary 3.2.3. *Let Υ be any neural network that is capable of computing every function $F \in \mathcal{F}'$ ϵ-correctly with some setting of its parameters for any $\epsilon < 0.5$ and any set of functions $\mathcal{F}'$ that is 2ϵ-robust. Almost all functions $F \in \mathcal{F}'$ require at least* $\log |\mathcal{F}'|$ *bits to describe their parameter settings in* Υ.

Given this, the question comes down to how large a 2ϵ-robust set $\mathcal{F}'$ can be constructed for a given neural network structure. Proving bounds on this is beyond the scope of this paper and the abilities of its authors, but we can offer a potential technique for doing so. Consider training a neural network Υ designed to be an LLM on a data set D consisting of r total tokens. Now consider training the same Υ using a dataset D' that is D with a random permutation applied to the individual tokens of D. If D is a real-world dataset, then with very high probability, D' will differ from D in most positions, and D' will not be a permutation of the documents making up the dataset. As a result, we conjecture (with a fair bit of confidence) that training Υ on D will result in very different results than training Υ on D'. It seems unlikely that Υ will be able to "understand" language based on the very jumbled D'.

A harder question is what happens if we construct D'', a second and independent random permutation of the tokens of D. While it seems likely that Υ trained on D'' will be very different from Υ trained on D, it is less obvious that Υ trained on D'' will be very different from Υ trained on D'. A jumbled training set probably produces very different results from a clean training set, but do two training sets that are jumbled very differently from each other produce different results from each other? We here conjecture that this is also true. This conjecture is important since if we could show that for a neural network structure Υ there is a training set D with r tokens, such that applying any set of permutations $\mathcal{P}$ to the individual tokens of D, where every pair of permutations $p_1, p_2 \in \mathcal{P}$ are significantly different from each other, leads to Υ computing a significantly different function for each permutation of D, this would provide us with a lower bound of $\Omega(r \log r)$ bits on the description of the parameters of Υ, where r is now as large as the training set, that is, in the trillions of tokens. This follows from a similar counting argument as was used above for the proof of the lower bound on permutation functions.

4 Upper Bounds

In this section, we provide an algorithm that converts any description of a 2-AND feature circuit into a neural network that computes that feature circuit in superposition, with both the input and output in superposition as well. Given any 2-AND circuit with m inputs and m' outputs, this neural network requires $n = O(\sqrt{m'}\log m')$ neurons. Even though our model allows the network to make errors on some of the inputs, our resulting network does not take advantage of this, and will always be correct. We here assume that the input consists of at most 2 inputs being active in any input; in Section 5 we describe how to extend this to a larger number of inputs, albeit with an exponential dependency on the number of active inputs. The bound on neurons achieved by this algorithm nearly matches the lower bound of the previous section: since $\Omega(m' \log m')$ parameters are needed for 2-AND by any algorithm that uses a constant number of bits per parameter, any $n \times n$ matrix multiplication based algorithm requires $n = \Omega(\sqrt{m' \log m'})$. Our algorithm uses a constant number of $n \times n$ matrices, and requires $O(m' \log^2 m')$ parameters.

4.1 Superposed Inputs

In the following, the computation proceeds in superposition throughout. However, it is helpful to think about the design of the algorithm in terms of going back and forth between representing the input and subsequent computations using the n-vector superposed representation, and the m-vector monosemantic representation (we describe below how to do this in a way such that the final computation is entirely in superposition). As a warm-up, we start by describing how to move back and forth between these representations. Let y be an m-vector representation and x be the n-vector representation of the same state, where $n = O(\sqrt{m} \log m)$. To go from y to x we can multiply y on the left by C, an $n \times m$ binary matrix where all entries are chosen i.i.d. with a probability $p = O(\log m/n)$ of each entry being a 1. To compress y into x, we use $x = Cy$. We will refer to a matrix like C as a compression matrix but will use different forms of compression matrices below.

We see that if y is sparse then Cy is fairly sparse as well and we can recover y from Cy. In order to do so, we use a decompression matrix D where D depends on C as follows: let C^T be the transpose of C. D is C^T with every entry equal to 1 in C^T (which is binary) replaced by 1 divided by the number of 1s in that entry's respective row of C^T. Consider the matrix $R = DC$ and consider the entry R_{ij} of R. If $i = j$ then $R_{ij} = 1$ since the non-zero entries will line up exactly and we are normalizing. And if $i \neq j$ then R_{ij} will be very close to 0 since the non-zero entries in row i of D will (with high probability) have very little overlap with the non-zero entries in column j of C. Thus D is an approximate left inverse of C and can be used to recover the original y through $y \approx DCy$. The approximation here is due to some of the values intended to be zero not being exactly zero.

We can also do the compression/decompression pairing in the reverse order: $x \approx CDx$. And we can insert other matrices in between C and D. For example, we could insert a permutation matrix P to get $x' \approx CPDx$, which would allow us to approximately compute the permutation P of the compressed representation of x to the compressed representation x' of the permutation of x. Note that while P is an $m \times m$ matrix, CPD is an $n \times n$ matrix, and thus maintains superposition. The challenge here is that Dx contains some noise in it due to the small but non-zero values that should have been zero: in expectation, Dx will have $O(m \log^2 m/n)$ non-zero values. As a result, x' also has some noise in it, and in fact the multiplication by C causes the noise to add up from the different entries of Dx, and become distributed to many entries of CDx. The expected noise in each entry of CDx is $O(m \log^2 m/n^2)$. If that noise is small (say less than 1/4), then we can remove it through a ReLU operation, as described below. This leads us to a value of $n = O(\sqrt{m} \log m)$, which is sufficient to compute the permutation function in superposition.

We know from our lower bounds that we cannot compute the permutation with an $n \times n$ matrix for $n = \omega(\sqrt{m \log m})$; it is exactly this noise that keeps us from making n any smaller, and in fact, when n is too small, the noise becomes so significant that the resulting permutation cannot be extracted from $CPDx$ even with unlimited computational power.

With the above compression and decompression techniques in hand, we are ready to tackle the 2-AND problem. We will assume that the input comes in the compressed form, i.e., the input is x_0, an n-vector representing the input in superposition, but if it starts in its uncompressed form of size m, then we can compress it down by multiplying on the left by a compression matrix C. We also assume that we know the representation being used for x_0 in the form of D_0, the decompression matrix for x_0.

4.2 Algorithm for maximum feature influence 1

We start off by providing an algorithm for a special case of the problem. Let the maximum feature influence of any input be denoted by t. We here consider the case where $t = 1$, which we call the *single-use 2-AND* problem, since each input feature can be used in at most one output feature. This provides some intuition for our main algorithm, and in fact, we will use the basic structure of the single-use algorithm in our main algorithm. Note that here, the number of output features m' for 2-AND when $t = 1$ is $m' \leq m/2$.

To compute single-use 2-AND in superposition, we start by multiplying x_0 on the left by D_0 to uncompress x_0. This allows us to more easily manipulate the individual inputs of the problem. We then multiply D_0x_0 by C_0, an $n \times m$ matrix that is a compression matrix but different from the one described above. C_0 is a binary matrix whose m columns are populated with values chosen from m' different types of column specifications $s_1, \dots, s_{m'}$, one for each output of the 2-AND. Each column specification s_i has binary entries chosen i.i.d. with probability $p = O(\log m/n)$ of being a 1 and 0 otherwise. As before $n = O(\sqrt{m} \log m)$. The column specification s_i will appear in column j of the matrix C_0 if the jth input appears in the ith output that is being computed. In other words, the ith column specification will appear exactly twice in C_0: once for each input that appears in the ith AND that is computed.

After multiplying by the compression matrix, we add b to the result where b is a column vector consisting of n entries, all of which are -1. And then we take the pointwise ReLU of the result giving us $x_1' = \text{ReLU}[C_0D_0x_0 + b]$. The n-vector x_1' is a compressed form of the output to the 2-AND problem. The intuition for this is that because each column specification s_i is fairly sparse, two distinct column specifications will not overlap much. As a result we can use each column specification as a kind of computational channel to compute each of the outputs. Since the input vector is also sparse, these channels will not interfere much with each other.

To extract the outputs of the 2-AND problem from x_1', we multiply on the left by an $m' \times n$ decompression matrix, D_1. Note that this decompression matrix is slightly different from the version above since the compression matrix is also different. D_1 depends on the column specification vectors $s_1, \dots, s_{m'}$ used to construct C_0. Specifically, row i of D_0 will have non-zero entries in the

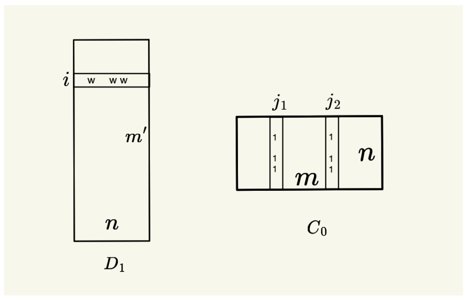

Figure 3: Matrices D_1 and C_0. Output i computes $j_1 \wedge j_2$. Columns j_1 and j_2 of C_0 are identical and defined by s_i, which also defines the non-zero entries of row i in D_1. Value w is the reciprocal of the number of these non-zero entries.

same positions as s_i has 1s except that the row and column are transposed. And the value of those non-zero entries will be $1/|s_i|$ where $|s_i|$ is the number of 1s in $|s_i|$. In other words, D_1 is constructed as if the corresponding compression matrix had consisted of the matrix formed by joining the vectors $s_1, \ldots, s_{m^*}$. We depict C_0 and D_1 in Figure 3.

As a last step of the computation, we will multiply on the left by another $n \times m$ compression matrix, which we call C_1'. This matrix is a standard compression matrix: an $n \times m$ binary matrix where all entries are chosen i.i.d. with a probability $p = O(\log m/n)$ of each entry being a 1. Finally we perform a ReLU operation on the result of that multiplication, providing us with the final result $x_1 = \text{ReLU}[C_1' D_1 \text{ReLU}[C_0 D_0 x_0 + b]]$. The matrix C_1' actually serves several purposes:

- It compresses $D_1 x_1'$ down to the right form to serve as an input to the next layer. This includes being in the compressed form we assume, but we will also use it below to set up all the inputs correctly for the more involved computation we require.
- It removes additional noise introduce by the matrix D_1 (this will require a slightly more involved use of ReLU, which we describe below).
- It ensures that the encoding used at the start of the algorithm (in this case for the next layer) uses a random compression matrix.

Again, x_1 is used as the input to the next layer of the neural network, and the resulting computation looks as depicted in Figure 4 for a single layer of the computation.

To see that the correct entries of x_1 are set to 1 in the result, consider the case where inputs indexed j_1 j_2 are both active, and so x_1 should represent j_1 AND j_2 as being active. If the ith output computes j_1 AND j_2 then both the j_1th and j_2nd columns of the matrix C_0 will be s_i. If no other inputs are 1, then $C_0 D_0 x_0$ will be an n-vector with 2s in the entries where s_i has 1s and almost zeros elsewhere (the "almost" comes from the noise in the decompression

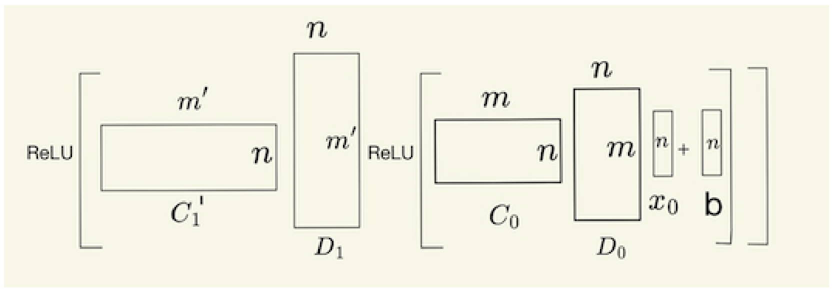

Figure 4: One layer of computation

process). As a result, ReLU$[C_0 D_0 x_0 + b]$ will be very close to s_i. This noise will be subsequently removed by the compression matrix C_1'. We will see below that we have now successfully computed x_1 which can then be used as the input for the next layer of the neural network.

Before proceeding, it is worth noting that our neural network model of computation requires us to use matrices of size $n \times n$, and yet in the construction above, we are using matrices of size $m' \times n$ and $n \times m'$ which are much larger. This is fine since we are differentiating between the construction of the neural network and actually using the neural network (analogous to training and inference in traditional neural networks). During the construction phase of the neural network, we do indeed work with the larger $m' \times n$ matrices (we expect that the resulting computational cost is still significantly lower than traditional training). However, before proceeding to inference, we can multiply out $C_0 D_0$ explicitly and then simply use the resulting $n \times n$ matrix for inference. The result looks as depicted in Figure 5.

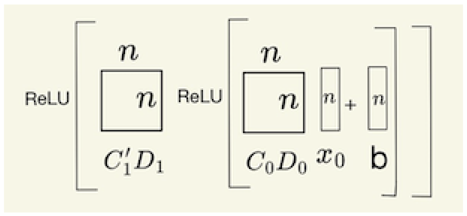

Figure 5: Resulting $n \times n$ matrices used for inference

This is one of the really powerful aspects of this model: as long as we are able to multiply together any matrices that we are using for the construction and get matrices and vectors of size $n \times n$ and n, we can use (and/or think about) larger matrices in the process of constructing the algorithm.

Furthermore, in the analysis that follows, we will describe the computation through the lens of the larger matrices and more specifically the random choices made for constructing the larger matrices. In this analysis, we will group together the matrices being multiplied out in different ways; this is valid due to the associativity of matrix multiplication.

In the algorithm described above, we have the column specifications tied to the outputs. In a sense, there is a "computational channel" for each output, and we route each input to its correct channel (essentially, by computing a permutation), where it can then be combined with the appropriate other input (in this case, via the AND function). This technique seems foundational, as it can be used to solve any other Boolean function as well, and we are curious if anything similar is happening in neural networks after they are trained. We also note that this is a different approach from [24]. In [24], the column specification is an encoding specific to the input represented by that column with the goal of the random binary string having enough non-zero entries to randomly line up with any other inputs that are active together.

We will refer to the technique above as using *output channels* and the technique of [24] as using *input channels.* To generalize our single-use 2-AND algorithm to an arbitrary 2-AND problem, we use both types of channels, although the input channel approach requires additional components beyond that of [24] to work when the computation starts and ends in superposition. Which type of channel is utilized depends on the feature influences of the specific 2-AND problem. Input channels are useful for inputs that have high feature influence, since that single encoding can be used multiple times. However, they create a lot of noise relative to the number of inputs, and so if we want to keep n small, they can only be used when the number of input channels is significantly smaller than m'. Output channels, on the other hand, are useful when the number of inputs is closer to m', which happens when the average feature influence is small, such as in the single use case above. However, if output channels are used for high feature influence inputs, the different output channels start to interfere with each other. Using the right type of channel for each input is one of the main challenges our algorithm overcomes. To do so, we will divide the 2-AND into three different subproblems, based on the feature influence specifics of the 2-AND problem to be solved.

The rest of this Section is organized as follows. In Section 4.3 we provide a high level overview of our algorithm, describe how to divide it up into the different cases, and cover some preliminaries that we will use in the analysis of those cases. In Section 4.4 we describe our algorithm for the case of low maximum feature influence ($t \leq {m'}^{1/4}$), which we believe to be the most interesting of the cases, as it seems likely to represent the actual level of feature influence seen in real neural networks. Then, in Section 4.5 we provide the algorithm for the case of high average influence and in Section 4.6 we describe how to cover the case of high maximum feature influence but low average feature influence.

4.3 High level outline of algorithm

Our goal is to demonstrate that $n = O(\sqrt{m'} \log m')$ is sufficient. We divide the problem up into three subproblems, dependent on feature influence, and these subproblems will be solved using the three algorithms described in the following three subsections. In all cases, we use the same structure of matrices as described above, and depicted in Figure 4. We call this structure the *common structure*. Except as otherwise described, each of the algorithms only changes the specific way that matrices C_0 and D_1 are defined.

Here is a high level description of our algorithm:

- Label each input *light* or *heavy* depending on how many outputs it appears in, where light inputs appear in at most ${m'}^{1/4}$ outputs and heavy inputs appear in more than ${m'}^{1/4}$ outputs.
- Label each output as *double light*, *double heavy* or *mixed*, dependent on how many light and heavy inputs that output combines.
- Partition the outputs of the 2-AND problem into three subproblems, based on their output labels. Each input is routed to the subproblems it is used in, and thus may appear in one or two subproblems. Otherwise, the subproblems are solved independently.
- Solve the double light outputs subproblem using algorithm **Low-Influence-AND**, described in Section 4.4.
- Solve the double heavy outputs subproblem using algorithm **High-Influence-AND**, described in Section 4.5.
- Solve the mixed outputs subproblem using algorithm **Mixed-Influence-AND**. We describe this algorithm in Section 4.6.

To route the inputs to the correct subproblems, we use the matrix C'_1 of the previous layer, or if this is the first layer, we can either assume that we have control over the initial encoding, or if not, then we can insert a preliminary decompress-compress pair to the left of x_0, followed by a ReLU operation to remove any resulting noise before starting the algorithm above. The partition of the outputs and the computation allocates unique rows and columns to each of the subproblems in every matrix of the computation except C'_1 (since that is used to set up the partition for the input to the next layer). As a result, the subproblems do not interfere with each other, and in fact the description of the algorithms below treats each subproblem as if they are standalone. This is depicted in Figure 6 for the case of two subproblems.

We will prove that $n = O(\sqrt{m'} \log m')$ neurons are sufficient for each of the subproblems, and thus that bound also applies to the overall problem as long as there are a constant number of subproblems. We note that when some of the outputs are placed in a subproblem, the inputs that remain may go from being heavy to light (since they have lost some of their outputs). We use the convention that we continue to classify such inputs with their original designation. Also, one or two of the subproblems may become much smaller than the original problem. However, when we partition the problem into these subproblems, we will treat each subproblem as being of the same size as the original input: we will use a value of $n = O(\sqrt{m'} \log m')$ for each of the subproblems, regardless of how small it has become.

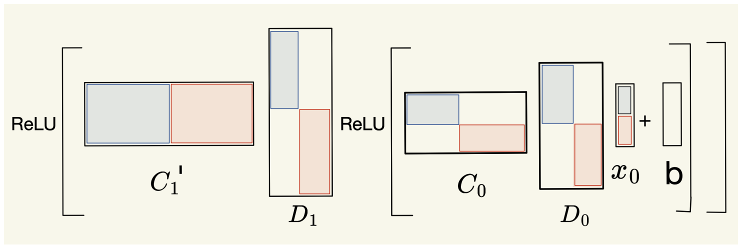

Figure 6: A partition of 2-AND into two subproblems. The red regions compute one subproblem, and the blue regions the other. All entries in other regions will be 0. Note that in C_1', the rows do overlap. This is to set up the outputs of this layer as the inputs to next layer, specifically to allow the same resulting input to appear in up to two subproblems.

We also note that in this section, when we refer to ReLU, when in fact we mean something slightly more involved built on top of the typical definition of ReLU. Specifically, we mean that we take each of the n entries of a vector in its compressed form, and round them to either 0 or 1, where anything less than 1/4 becomes a 0 and anything greater than 3/4 becomes 1 (we don't allow values to go to the intermediate range). This can be done with two rounds of an actual ReLU operation as follows: First, subtract 1/4 from all entries and then perform a ReLU operation. Then, do $1-\text{ReLU}(-2x+1)$ on all entries, which guarantees the objective. We can do this any time we have an intermediate result that is in superposed representation, and so we only need to be concerned with getting our superposed results to be close to correct. Note that we cannot use ReLU when an intermediate result is in its uncompressed form, since that would require at least $m \gg n$ ReLU operations. We will continue to use the convention of using ReLU to refer to the above operation.

In the analysis that follows, we frequently make use of Chernoff bounds [18] to prove high probability results. In all cases, we use the following form of the bound:

$$\Pr(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2\mu}{2+\delta}}, \quad 0 \leq \delta,$$

We always use *with high probability* to mean with probability at most $O(1/m^c)$, where c can be made arbitrarily large by adjusting the constants in the algorithm hidden by the Big-O notation. As mentioned above, our algorithms are always correct for all inputs. In our method of constructing the algorithm, there is a small probability that the construction will not work correctly (with high probability it will work). However, we can detect whether this happened by trying all pairs of inputs being active, and verifying that the algorithm works correctly. If it does not, then we restart the construction process from scratch, repeating until the algorithm works correctly (but these restarts do not add appreciably to the expected running time of the process of constructing the algorithm).

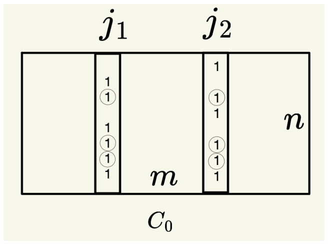

Figure 7: The matrix C_0 for **Low-Influence-AND**. The circled 1s are those that correspond to s_i, where output i computes $j_1 \wedge j_2$, and thus the 1s in those rows will line up between j_1 and j_2. Other rows with 1s come from different column specifications, and thus only line up by chance, but when that happens it causes spurious 1s to appear after the second ReLU. When there are at most $O({m'}^{1/4} \log m')$ total 1s in each column, it is likely there will be $O(\log m')$ such spurious 1s. However, since $n = O(\sqrt{m'} \log m')$, if there were more 1s in both columns, the number of spurious 1s would become too large to handle. This is why ${m'}^{1/4}$ represents such an important phase change for what techniques are effective for this problem.

4.4 Algorithm for double light outputs

We now handle the case where all inputs are light. This means that the maximum feature influence is at most ${m'}^{1/4}$. We show that in this case $n = O(\sqrt{m'} \log m')$ is sufficient. The algorithm uses the common structure, defined above, with the following changes:

Algorithm **Low-Influence-AND**

- Define m' different types of column specifications $s_1, \dots, s_{m'}$, one for each output, where a column specification is a binary n-vector. Each column specification s_i has binary entries chosen i.i.d. with probability $p = O(\log m/n)$ of being a 1 where $n = O(\sqrt{m} \log m)$ and 0 otherwise.

- In the matrix C_0, there is one column for each input j, and that column lines up with the entry for j in $D_0 x_0$. Entry e of column j is a 1 if any column specification s_i that corresponds to an output that input j participates in has a 1 in entry e. Otherwise entry e is a 0. C_0 is still an $n \times m$ matrix.

- The matrix D_1 is still an $m' \times n$ matrix with one row for each output i, and that row is the transpose of s_i, with each 1 replaced by $1/|s_i|$, where $|s_i|$ is the sum of the entries in s_i.

- The other matrices are defined exactly as they were before, except that now C_1' is an $n \times m'$ matrix (with the same likelihood of a 1).

The matrix C_0 is depicted in Figure 7. We point out that we are still using output channels here, since we are actively routing inputs that need to be paired up to the channels specified by the column specifications. We then combine all the channel specifications for a given input into a single column for that input. We say that a neural network algorithm *correctly computes in superposition* x_1 from x_0, if x_0 and x_1 are represented in superposition, and for any input x_0, x_1 represents the output of the 2-AND problem for that x_0, with all intended 0s being numerically 0 and all intended 1s numerically 1.

Theorem 4.1. *When the maximum feature influence is at most $m'^{1/4}$, at most 2 inputs are active, and $n = O(\sqrt{m'} \log m')$, Algorithm* **Low-Influence-AND** *with high probability correctly computes in superposition x_1 from x_0.*

Note that this subsumes the single-use case above.

Proof. We already demonstrated in our discussion of the single use case that we will get values that are 1 in the entries of x_1 that were supposed to be 1s; a very similar argument holds here, and so we only need to demonstrate that the inherent noise of the system does not result in too large values in the entries of x_1 that are supposed to be 0s. There are two sources of noise in the system:

(a) Multiplying by the decoding matrix D_0 is not perfect: there is the potential to have entries of $D_0 x_0$ that are supposed to be zero but are actually nonzero since each row of D_0 can have a 1 in the same column as a row that corresponds to an input that's a 1. Or equivalently, each row i of D_0 can have a 1 at a location that lines up with a 1 in the row representing the active x_0. We need to show that the resulting noise in $D_0 C_0 x_0$ is small enough to be removed by the first ReLU operation.

(b) Multiplying by the decoding matrix D_1 is also not perfect for the same reason. There can be overlap between the different output channels. Furthermore, since an input can be used multiple times (but in this case at most $m'^{1/4}$ times) we can also get 1s in the matrix $\mathrm{ReLU}[C_0 D_0 x_0]$ in places outside the correct output channel. Both of these effects lead to noise after multiplying by D_1, and potentially after subsequently multiplying by C_1' as well. We also need to show that this noise is small enough to be removed by the second ReLU operation.

We point out that as long as it is small, the noise of type (a) is removed by the first ReLU operation (right after the multiplication by C_0), and thus will not contribute to the noise of type (b). Thus, we can analyze the two types of noise independently. We handle noise of type (b) first. Let $y_1 = D_1 \mathrm{ReLU}[C_0 D_0 x_0]$. Our goal is to show that $\mathrm{ReLU}[C_1' y_1]$ only has non-zero entries in the correct places. Let $x_1' = \mathrm{ReLU}[C_0 D_0 x_0]$.

Claim 4.2. *Any entry of $C_1' y_1$ that does not correspond to a correct 1 of the 2-AND problem has value at most ϵ due to noise of type (b) with high probability.*

Proof. For any matrix M, we will refer to entry (i, j) in that matrix as $M(i, j)$, and similarly we will refer to entry k in vector V as $V(k)$. Let e be the index of any entry of $C_1' y_1$ that should not be a 1. We will show that with high probability the value of the entry $C_1'y_1(e)$ is at most ϵ. Due to our ReLU specific operation, we can assume that all non-zero entries of x_1' are at most 1. We refer to the two active inputs as i and j, where $i \neq j$ (there is no type (b) noise if there is only one active input). We first consider the expectation of $C_1'y_1(e)$. In the following expression for $E[C_1'y_1(e)]$, we let a range over the entries of C_1' in row e and b range over the columns of D_1. For entry $x_1'(b)$ to be a 1, we need $C_0(b,i) = 1$ and $C_0(b,j) = 1$. For this to translate to a non-zero value in its term of the sum for entry $y_1(a)$, we also need $D_1(a,b) = 1$. Since the entries in D_0 will be $O(\frac{1}{\log m'})$ with high probability, this gives the following:

$$E[C_1'y_1(e)] = O(\frac{1}{\log m'} \sum_{a\in 1\ldots m'} \sum_{b \in 1\ldots n} \Pr[C_0(b,i)=1] \Pr[C_0(b,j)=1|C_0(b,i)] \cdot \\ \Pr[D_1(a,b)=1|C_0(b,i), C_0(b,j)] \Pr[C_1'(e,a)=1]) \quad (1)$$

Note that $C_1'(c,e)$ is independent of all the other events we are conditioning on, and so we do not need to condition for the probability associated with that event. The other three events are independent for most terms in the sum, but not so for a small fraction of them. $D_1(a,b)$ is independent of $C_0(b,i)$ whenever i is not used in the output for row a of D_1, and similarly for $C_0(b,j)$. $C_0(b,i)$ is independent of $C_0(b,j)$, as long as $D_0(d,b) = 0$, where row d of D_0 corresponds to the output that is an AND of i and j (since then we know that for all such entries b of row d, $s_d(c) = 0$). The entries $y_1(d)$ where $D_0(d,b) > 0$ are supposed to be non-zero, but they can still contribute noise when multiplied by the compression matrix C_1'.

Thus, we will evaluate this sum using two cases: where there is some dependence between any pair of the four probabilities, and where there is not. We deal with the latter case first, and in this case $\Pr[C_1'(e,a)=1] = \Pr[D_1(a,b) = 1] = O(\log m'/n)$. Using a union bound and the fact that no input is used more than ${m'}^{1/4}$ times, we see that $\Pr[C_0(b,i) = 1] = O({m'}^{1/4}\log m'/n)$ and also $\Pr[C_0(b,j)=1] = O({m'}^{1/4}\log m'/n)$. This tells us that the contribution of the independent terms is at most

$$O\left(\frac{1}{\log m'} nm' \left(\frac{\log m'}{n}\right)^2 \left(\frac{{m'}^{1/4}\log m'}{n}\right)^2\right) = O(1).$$

For the case where there is dependency between the different events, we first consider what happens when i is used in the output for row a of D_1. In this case, we can simply assume that $D_1(d_c) = 1$ always, which means we lose a factor of $\log m'/n = 1/\sqrt{m'}$ in the above equation. However, since i can be used in at most ${m'}^{1/4}$ outputs, there are now only ${m'}^{1/4}$ values of a to consider instead of m', so we also lose a factor ${m'}^{3/4}$. From this we see that the terms of this case do not contribute meaningfully to the value of the sum, and similarly for when j is used in the output for row a. For the case where $D_0(d,b) > 0$, where i and j

are used in the output for row d, we see that all three of the dependent variables will be 1. However, there is only 1 such row d, and we also know that with high probability that row will contain $O(\log m')$ 1s. Thus, the number of terms in the sum is reduced by $m'^{3/2}$ and we still have the $\frac{\log m'}{n}$ from $\Pr[C'_1(e, a) = 1]$, and so this case does not contribute significantly to the sum either.

To convert this expectation to a high probability result, we can rearrange the terms of the sum to consider only those rows d of D_1 that correspond to columns of C'_1 where $C'_1(d, e) = 1$ and those columns of D_1 that correspond to entries of x'_1 where $x'_1(c) = 1$ incorrectly (i.e. c such that there exist a and b such that both $C_0(c, a) = 1$ and $C_0(c, b) = 1$). The number of non-zero entries in a row of C'_1 is $O(m' \log m'/n)$ with high probability (from how C'_1 is built).

The number of non-zero entries in x'_1 is $O(\log m')$ with high probability. Thus, with high probability, we are summing a total of $O(m' \log^2 m'/n) = O(\sqrt{m'} \log m')$ entries of D_1. Each of those entries is either $\Theta(1/\log m')$ or 0 and takes on the non-zero value with probability $\log m'/n$. Thus the expectation of that sum is $O(m' \log^2 m'/n^2) = O(1)$. We can define indicator variables on whether or not each such entry of D_1 is non-zero. We can assume these random variables are chosen independently, and their expected sum is $O(\log m')$, and so a standard Chernoff bound then demonstrates that the number of non-zero entries in D_1 will be within a constant of its expectation with high probability. Thus, $C'_1 y_1(e)$ will be $O(1)$ with high probability. We can make that constant smaller than any ϵ be increasing the size of n by a constant factor dependent on ϵ. $\square$

Thus, all noise of type (b) will be removed by the second ReLU operation. We now turn to noise of type (a): there can be incorrect non-zeros in the vector $D_0 x_0$ and we want to make sure that any resulting incorrect non-zero entry in the vector $C_0 D_0 x_0$ has size at most ϵ and thus will be removed by the first ReLU function.

Claim 4.3. *The amount of type (a) noise introduced to any entry of $C_0 D_0 x_0$ is at most ϵ with high probability.*

Proof. For any row e of C_0, let C_0^e be the set of columns a of C_0 such that $C_0(e, a) = 1]$. We first show that for any e, with high probability, $|C_0^e| = O(\sqrt{m'})$. This follows from how the columns of C_0 are chosen: they are defined by the column specifications $s_1, \ldots, s_{m'}$. Every column specification s_i where $s_i(e) = 1$ contributes at most 2 new columns to C_0^e - one for each input used for output i. There are m' column specifications, and the entries are all chosen i.i.d., with probability of a 1 being $1/\sqrt{m'}$, and so a straightforward Chernoff bound shows that with high probability there are at most $O(\sqrt{m'})$ specifications s_i where $s_i(e) = 1$. Thus $|C_0^e| = O(\sqrt{m'})$ with high probability.

When we multiply C_0 by $D_0 x_0$, we will simply sum together the non-zero entries of $D_0 x_0$ that line up with the columns in C_0^e. For any a that does not correspond to an active input, using the fact that x_0 has $O(\log m')$ non-zero entries and a union bound, we see that

$$\Pr[D_0 x_0(a) > 0] \leq O\left(\frac{\log^2 m'}{n}\right) = O\left(\frac{\log m'}{\sqrt{m'}}\right).$$

Thus, the expected number of non-zero terms in the sum $C_0 D_0 x_0(e)$ is $O(\log m')$. Furthermore, since the entries of D_0 are chosen independently of each other, we can use a Chernoff bound to show that the the number of non-zero terms in the sum for $C_0 D_0 x_0(e)$ is $O(\log m')$ with high probability. Finally, we point out that the incorrect non-zeros in $D_0 x_0$ have size at most $c/\log m'$ for a constant c with high probability which follows directly from the facts that each entry of $D_0 x_0$ is the sum of $\log m'$ pairwise products of two entries, divided by $\log m'$ and the probability of each of those products being a 1 is at most $\log m'/n$. Putting all of this together shows that for any e, $C_0 D_0 x_0(e)$ is at most $O(1)$ with high probability. This can be made smaller than any ϵ by increasing n by a constant factor dependent on ϵ. □

□

4.5 Algorithm for double heavy outputs

We here provide the algorithm called **High-Influence-AND**, which is used by our high level algorithm for outputs that have two heavy inputs. Let $\bar{t}$ be the average influence of the feature circuit. The algorithm **High-Influence-AND** requires only $n = O(\sqrt{m'} \log m')$, provided that $\bar{t} > m'^{1/4}$. Note that the high level algorithm uses **High-Influence-AND** on a subproblem that has a *minimum* feature influence of $m^{1/4}$. This implies that $\bar{t} > m'^{1/4}$. However, **High-Influence-AND** applies more broadly than just when the minimum feature influence is high - it is sufficient for the *average* feature influence to be high. We here describe the algorithm in terms of the more general condition to point out that if the overall input to the problem meets the average condition, we can just use **High-Influence-AND** for the entire problem, instead of breaking it down into various subproblems.

This algorithm uses input channels, in the sense that the column specifications do not depend on which outputs the inputs appear in. We can do so here for all inputs, because the number of inputs m is significantly smaller than m', and we define n relative to m', not m. Specifically, if $\bar{t} > m'^{1/4}$, then $m' > m \cdot m'^{1/4}/2$, which implies that $m < 2m'^{3/4}$. This algorithm follows the same common structure as above, with the following modifications to C_0 and D_0:

Algorithm **High-Influence-AND**

- In the matrix C_0, there is one column for each input, and that column lines up with the entry for that input in $D_0 x_0$. Each entry in this column is binary, and chosen independently, with a probability of 1 being $\frac{1}{m'^{1/4}}$. No further columns are allocated to C_0.

- Every row of D_1 corresponds to an output, and the entries in that row that are non-zero are those entries where both of the inputs for that output have a 1 in the corresponding entry of their column in C_0.

Theorem 4.4. *With high probability Algorithm* **High-Influence-AND** *correctly computes x_1 from x_0, provided that at most 2 inputs are active, $\bar{t} > m'^{1/4}$, and $n = O(\sqrt{m'}\log m')$.*

Proof. We first point out that for any output that should be active as a result of the AND, the entries of x_1 that should be 1 for that output, will in fact be a 1. This follows from the fact that for any pair of inputs that appear in an output, the expected number of entries of overlap in their respective columns of C_0 is $\Theta(\log m')$, and thus we can use a Chernoff bound to show that it will be within a constant factor of that value. From there, we see that the correct value of $D_1\text{ReLU}(C_0D_0x_0 + b)$ will be a 1. Thus, we only need to demonstrate that there is not too much noise of either type (a) or type (b) (as defined in Section 4.4). We demonstrate this with the following two claims:

Claim 4.5. *Any entry of C'_1y_1 that does not correspond to a correct 1 of the 2-AND problem has value at most ϵ due to noise of type (b) with high probability.*

Proof. For any column of C_0, the expected number of 1 entries is $O(n/m'^{1/4}) = O(m'^{1/4}\log m')$, and will be no larger with high probability. With this in hand, we can use an argument analogous to that in the proof of Claim 4.2. Specifically, for any entry e, Equation 1 still represents $E[C'_1y_1(e)]$, and so it follows that $E[C'_1y_1(e)] = \left(\frac{m'^{3/2}\log^3 m'}{n^3}\right)$. A similar Chernoff bound as in Claim 4.2 shows that $C'_1y_1(e)$ will be within a constant of its expectation with high probability. Thus, $n = O(\sqrt{m'}\log m')$ is sufficient to make $C'_1y_1(e) \leq \epsilon$ with high probability. □

Claim 4.6. *The amount of type (a) noise introduced to any entry of $C_0D_0x_0$ is at most ϵ with high probability.*

Proof. Let $N(e)$ be the contribution to entry e in $C_0D_0x_0$ due to this kind of noise. We first provide an expression for $\text{E}[N(e)]$. Let $H(C_0)$ be the columns of C_0, except those that correspond to the active inputs. In this expression, we let a range over the columns of $H(C_0)$ and b range over all the columns of D_0. We see that

$$E[N(e)] = \frac{1}{\log m'}\sum_{a\in H(C_0)}\sum_{b\in 1\ldots n}\Pr[x_0(b)=1]\Pr[D_0(a,b)>0]\Pr[C_0(e,a)=1], \quad (2)$$

where the active input not being in $H(C_0)$ implies that the three probabilities listed are independent. Since $|H(C_0)| \leq m$, there are at most nm terms in this sum, and the first two probabilities are $\frac{\log m'}{n}$, and the third is $\frac{1}{m'^{1/4}}$. This gives us that

$$E[N(e)] = O\left(\frac{1}{\log m'} nm \left(\frac{\log m'}{n}\right)^2 \frac{1}{m'^{1/4}}\right) = O\left(\frac{m \log m'}{nm'^{1/4}}\right) = O(1),$$

where the last equality uses the fact that $m \leq 2m'^{3/4}$, which follows from the fact that $\bar{t} \geq m'^{1/4}$. This gap between m and m' is why we are able to use this algorithm in the case of high average feature influence, but not when that average is smaller. Since the summation of probabilities is divided by a $\log m'$ factor, a fairly straightforward Chernoff bound over the choices of $C_0(e, a)$, for $a \in H(C_0)$, shows that this is no higher than its expectation by a constant factor with high probability. The resulting constant can be made smaller than any ϵ by increasing n by a constant factor dependent only on ϵ. $\square$

This concludes the proof of Theorem 4.4. $\square$

4.6 Algorithm for mixed outputs

We now turn to the most challenging of our three subproblems, the case where the outputs are mixed: one heavy and one light input. As stated above, **High-Influence-AND** from Section 4.5 is actually effective when some outputs are mixed, provided that the average feature influence of the feature circuit is sufficiently high. However, what **High-Influence-AND** is not able to handle (with $n = O(\sqrt{m'} \log m')$), is the case where the feature circuit has low average influence, but high maximum influence. Our algorithm here is used by the high level algorithm for all the mixed outputs, but most importantly it addresses that case of feature circuits with low average influence and high maximum influence. This involves a combination of input channels for heavy inputs and output channels for light inputs. Furthermore, we see below that just how high the feature influence of a heavy input is impacts how the problem is divided into input and output channels. As a result, we will further partition the outputs into two subcases based on a further refinement of the heavy features. Since we overall performed four partitions, this does not affect the overall complexity of the solution.

Algorithm **Mixed-Influence-AND**

- Label any input that appears in more than $m'^{1/2}$ outputs as *super heavy.* We further partition this subproblem into two based on this label: we treat the regular heavy mixed outputs separately from the super heavy mixed outputs.
- For the regular heavy mixed outputs:
  - In the encoding for x_0 and the matrix D_0, partition the encoding of the light inputs and the super heavy inputs, so that they do not share any rows or columns.

  - In the matrix C_0, there is one column for each heavy input. Each entry in this column is binary, and chosen independently, with a probability of 1 being $\frac{1}{m'^{1/4}}$.
  - In C_0, there is also one column for each light input j. Each entry in this column is binary. For entry k for j, if there is a heavy input i such that i and j appear in the same output and entry k for column i is a 1, then entry k for column j is chosen independently with the probability of a 1 being $\frac{1}{m'^{1/4}}$. Otherwise, entry k in column j is a 0.
  - No further columns are allocated to C_0, and the remainder of the algorithm is constructed analogously to the algorithms **Low-Influence-AND** and **High-Influence-AND**, where the entries of D_1 that are non-zero for a given output are those entries where both of its inputs have a 1 in the corresponding entry of their column in C_0.

- For the super heavy mixed outputs:
  - In the encoding for x_0 and the matrix D_0, partition the encoding of the light inputs and the super heavy inputs, so that they do not share any rows or columns. Furthermore, none of the super heavy inputs will share any rows or columns with each other.
  - In the matrix C_0, there is one column for each heavy input. Each entry in this column is binary, and chosen i.i.d., with a probability of 1 being $\frac{1}{\gamma}$, for a constant γ to be determined below.
  - In C_0, there is one column for each light input j. Each entry in this column is binary, and chosen i.i.d., with a probability of 1 being $\frac{2\gamma}{\sqrt{m'}}$.
  - There is an additional mechanism, called **detect-two-active-heavies**, which will be described below.
  - The remainder of the algorithm is constructed analogously to the algorithms **Low-Influence-AND** and **High-Influence-AND**, where the entries of D_1 that are non-zero for a given output are those entries where both of its inputs have a 1 in the corresponding entry of their column C_0.

In the case of regular heavy inputs, we can view the heavy inputs as using input channels (since they are not dependent on how those inputs are used), and the light inputs as using output channels (since they are routed to the channel of the input they share an output with). We see below that this is effective for regular heavy inputs. However, for super heavy inputs, this would not work: a super heavy input would have too many light inputs routed to it. If we do not increase the size of the input channel for the super heavy input, there will be too many light inputs routed to too little space, and as a result, those light inputs would create too much type (a) noise. And if we do increase the size of the input channel for super heavy inputs, then the super heavy inputs will create too much type (b) noise with each other. Thus, we need to deal with the super heavy inputs separately, as we did above. Key to this is **detect-two-active-heavies** which is a way of shutting down this entire portion of the algorithm when two super heavy inputs are active. This allows us to remove what would otherwise be too much noise in the system. The output for that pair of inputs will instead be produced by Algorithm **High-Influence-AND**.

Theorem 4.7. *With high probability, Algorithm* **Mixed-Influence-AND** *correctly computes x_1 from x_0, provided that at most 2 inputs are active, each output contains both a heavy and a light input, and $n = O(\sqrt{m'} \log m')$.*

Proof. This follows from the two lemmas below.

Lemma 4.8. *The subproblem of Algorithm* **Mixed-Influence-AND** *on the regular heavy mixed outputs produces the correct result provided that at most 2 inputs are active and $n = O(\sqrt{m'} \log m')$.*

Proof. We first point out that for any output that should be active as a result of the AND, the entries of x_1 that should be 1 for that output, will in fact be a 1. This follows from the fact that for any pair of inputs that appear in a regular heavy mixed output, the expected number of rows of overlap in their respective columns of C_0 is $\Theta(\log m')$, and thus we can use a Chernoff bound to show that it will be within a constant factor of that value. The Lemma now follows from the following two claims:

Claim 4.9. *Any entry of $C_1' y_1$ that does not correspond to a correct 1 of the 2-AND problem has value at most ϵ due to noise of type (b) with high probability.*

Proof. For any column of C_0 (corresponding to either a light or a heavy input), the expected number of 1 entries is $O(n/{m'}^{1/4}) = O({m'}^{1/4} \log m')$, and will be no larger with high probability. With this in hand, we can use an argument analogous to that in the proof of Claim 4.2. □

Claim 4.10. *The amount of type (a) noise introduced to any entry of $C_0 D_0 x_0$ is at most ϵ with high probability.*

Proof. We need to argue this for both the light inputs and the heavy inputs. However, since we partitioned those inputs in D_0, they will not interfere with each other, and we can handle each of those separately. We first examine the heavy inputs, and note that there can be at most ${m'}^{3/4}$ of them, since each will contribute at least ${m'}^{1/4}$ distinct outputs. Let $N_h(e)$ be the contribution to $C_0 D_0 x_0(e)$ of this kind of noise from heavy inputs. We first provide an expression for $\mathrm{E}[N_h(e)]$. Let $H(C_0)$ be the columns of C_0 in row e that correspond to heavy inputs, not counting the active input. Let $H(D_0)$ be the columns of D_0 that are used by the heavy inputs. In this expression, we let a range over the columns in $H(C_0)$ and b range over the columns of $H(D_0)$. We see that

$$E[N_h(e)] = \frac{1}{\log m'} \sum_{a \in H(C_0)} \sum_{b \in H(D_0)} \Pr[x_0(b) = 1] \Pr[D_0(a,b) > 0] \Pr[C_0(e,a) = 1], \quad (3)$$

where the active input not being in $H(C_0)$ implies that the three probabilities listed are independent. Since $|H(C_0)| \le {m'}^{3/4}$ and $|H(D_0)| \le n$, there are at most $n{m'}^{3/4}$ terms in this sum, and the first two probabilities are $\frac{\log m'}{n}$, and the third is $\frac{1}{{m'}^{1/4}}$. This gives us that

$$E[N_h(e)] = O\left(\frac{1}{\log m'} n{m'}^{3/4} \left(\frac{\log m'}{n}\right)^2 \frac{1}{{m'}^{1/4}}\right) = O(1).$$

Since the summation of probabilities is divided by a $\log m'$ factor, a fairly straightforward Chernoff bound over the choices of $C_0(e,a)$, for $a \in H(C_0)$, shows that this is no higher than its expectation by a constant factor with high probability. The resulting constant can be made smaller than any ϵ by increasing n by a constant factor dependent only on ϵ.

We next turn to light inputs. This is more challenging than the heavy inputs for two reasons. First, if we define $L(C_0)$ analogously to $H(C_0)$, then $|L(C_0)|$ can be larger than ${m'}^{3/4}$ because each light input appears in at most ${m'}^{1/4}$ outputs. It can be $\Theta(m)$, which means we would need to evaluate the sum in the expectation a different way. Second, the choices of $C_0(e,a)$, for $a \in L(C_0)$, are no longer independent, since those choices for two light inputs that share the same heavy input will both be influenced by the choice in row e for that heavy input (see Figure 8). Thus the Chernoff bound to demonstrate the high probability result needs to be done differently. In fact, this lack of independence is why we need to handle the super heavy inputs differently in the algorithm. If, for example, there were a single heavy input h that appeared in the same output as all of the light inputs, consider any row e_h such that $C_0(e_h,h) = 1$. The expectation of $C_0D_0x_0(e_h)$ is $\Theta(m^{3/4}/n)$, which is too large. In other words, with such a super heavy input, even though the expectation $E[N_l(e)] = O(1)$ for every e, the distribution is such that with high probability there will be some e_h such that $N_l(e_h) = \Theta(m^{3/4}/n)$.

Instead, we take a different approach here. For any row e of C_0, and any set of columns S, let $C_0^S(e)$ be the set of entries $C_0(e,a)$ that are 1 for $a \in S$. We first show that with high probability, $|C_0^{L(C_0)}(e)| = O(\sqrt{m'})$. To do so, we demonstrate that the entries in C_0 in row e for the heavy inputs leave at most $O({m'}^{3/4})$ light inputs that make a choice for their entry in row e; the remainder are only in rows where all heavy inputs that appear with them have a 0 in row e, and thus they are set to 0 without making a choice. More precisely, for any heavy input a, let $\delta(a)$ be the set of light inputs that appear in an output with a. We wish to show that

$$\left|\bigcup_{a \in C_0^{H(C_0)}(e)} \delta(a)\right| = O({m'}^{3/4}).$$

To do so, first note that $\sum_{a \in H(C_0)} \delta(a) \le m'$, since there are at most m' outputs, and each output has at most one light entry. We can now define random variables z_a for each $a \in H(C_0)$, where $z_a = 0$ when $a \notin C_0^{H(C_0)}(e)$, and $z_a =$

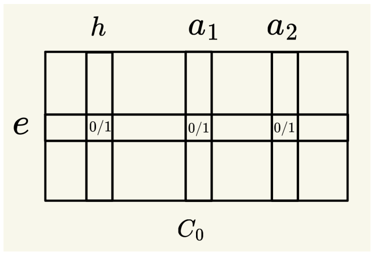

Figure 8: The dependence in light inputs between the different choices for $C_0(e,a)$, when $a \in L(C_0)$. Here a_1 and a_2 are light inputs, and so $a_1, a_2 \in L(C_0)$, and h is a heavy input such that both $h \wedge a_1$ and $h \wedge a_2$ are computed. If we ignore the impact of other heavy inputs then if $C_0(e,h) = 0$, then both $C_0(e,a_1) = 0$ and $C_0(e,a_2) = 0$. Thus, $\Pr[C_0(e,a_2) = 1 | C_0(e,a_1) = 1] \gg \Pr[C_0(e,a_2) = 1 | C_0(e,a_1) = 0]$, and so $C_0(e,a_2)$ and $C_0(e,a_1)$ are not independent.

$\delta(a)/\sqrt{m'}$ when $a \in C_0^{H(C_0)}(e)$, which happens with probability $\frac{1}{m'^{1/4}}$. Since there are no super heavy inputs in $H(C_0)$, $\forall a \in H(C_0), |\delta(a)| \leq \sqrt{m'}$, and so $0 \leq z_a \leq 1$. Also, the z_a are mutually independent. Thus, $E\left[\sum_{a \in H(C_0)} z_a\right] \leq m'^{1/4}$, and a standard Chernoff bound shows that $\sum_{a \in H(C_0)} z_i = O(m'^{1/4})$ with high probability. From this it follows that

$$\left| \bigcup_{a \in C_0^{H(C_0)}(e)} \delta(a) \right| \leq \sum_{a \in C_0^{H(C_0)}(e)} |\delta(a)| = \sqrt{m'} \sum_{a \in H(C_0)} z_a = O(m'^{3/4}),$$

with high probability. Given this, at most $O(m'^{3/4})$ light inputs make a choice for their entry in row e, and each of those is a 1 independently with probability $\frac{1}{m'^{1/4}}$. A standard Chernoff bound now shows that $|C_0^{L(C_0)}(e)| = O(\sqrt{m'})$ with high probability.

To finish the proof of this claim, let $N_l(e)$ and $L(D_0)$ be defined analogously to $N_h(e)$ and $H(D_0)$ respectively, for light inputs. We see that

$$E[N_l(e)] = \frac{1}{\log m'} \sum_{a \in C_0^{L(C_0)}(e)} \sum_{b \in L(D_0)} \Pr[x_0(b) = 1] \Pr[D_0(a,b) > 0]$$

$$= O\left(\frac{n\sqrt{m'}}{\log m'} \left(\frac{\log m'}{n} \right)^2 \right) = O(1).$$

We can now use a Chernoff bound over the choices of the relevant entries in D_0 to show that $N_l(e) = O(1)$ with high probability as well. The resulting constant can be made smaller than any ϵ by increasing n by a constant factor dependent only on ϵ. □

This concludes the proof of Lemma 4.8. □

Lemma 4.11. *The subproblem of Algorithm* **Mixed-Influence-AND** *on the super heavy mixed outputs produces the correct result provided that at most 2 inputs are active and $n = O(\sqrt{m'}\log m')$.*

Proof. We first point out that for any output that should be active as a result of the AND, the entries of x_1 that should be 1 for that output, will in fact be a 1. This follows from the fact that for any pair of inputs that appear in a super heavy mixed output, the expected number of rows of overlap they share in C_0 is $\Theta(\log m')$, and thus we can use a Chernoff bound to show that it will be within a constant factor of that value. The Lemma now follows from the following two claims:

Claim 4.12. *The amount of type (a) noise introduced to any entry of $C_0 D_0' y_1$ is at most ϵ with high probability.*

Proof. Since the super heavy inputs do not have any overlapping columns in D_0 with each other or with light inputs, none of the super heavy inputs will produce type (a) noise. Note that there can be at most $\sqrt{m'}$ super heavy inputs (or we would have more than m' outputs), and so $n = O(\sqrt{m'}\log m')$ is sufficient space to provide a non-overlapping input encoding in x_0 for each super heavy input. (This is why we cannot treat regular heavy inputs the same as super heavy inputs - there might be too many of them.) Thus, we only need to concern ourselves with type (a) noise produced by pairs of inputs that are both light. Demonstrating that this noise is at most ϵ is analogous to the proof that there is not too much type (a) noise for heavy inputs in Claim 4.10. In fact, the expected such noise is given by an expression almost identical to Equation 3. In evaluating that expression, we only need to change the number of choices of $H(C_0)$ from ${m'}^{3/4}$ to m, and the $\Pr[C_0(e,a) = 1]$ from $\frac{1}{{m'}^{1/4}}$ to $\frac{2\gamma}{{m'}^{1/2}}$. The facts that the expectation of this noise is $O(1)$, that it is not much higher with high probability, and that it can be made smaller than ϵ by increasing n by a constant all follow the same way as in the proof of Claim 4.10. $\square$

Claim 4.13. *When the two active inputs to the 2-AND problem consist of at most one super heavy input, then any entry of $C_1' y_1$ that does not correspond to a correct 1 of the 2-AND problem has value at most ϵ due to noise of type (b) with high probability.*

Proof. Type (b) noise occurs when the 1s that appear in $ReLU(C_0 D_0 x_0)$ are picked up by non-zero entries in unintended rows during the multiplication by D_1, and then remain after being subsequently multiplied by C_1'. Since we assume there is at most 1 super heavy input, we only need to handle two cases: one active light input and one active super heavy input, as well as two active light inputs. For two light inputs, the number of 1s in $ReLU(C_0 D_0 x_0)$ is $O(1)$ with high probability. For the mixed case, the number of 1s in $ReLU(C_0 D_0 x_0)$ is $O(\log m')$ with high probability, and thus is more challenging, and in fact the two active light inputs case can be handled similarly, so we here only present the argument for the mixed case.

Let s be the active super heavy input, and let l be the active light input. The 1s in $ReLU(C_0D_0x_0)$ from those two active inputs can be picked up by an unintended row of D_1 that corresponds to an incorrect output that combines a light input l' and a heavy input s', where either $s' \neq s$ or $l' \neq l$, or both. We will combine these three possibilities into two cases: in the first, $s' \neq s$, but $l' = l$, and in the second, $l' \neq l$, but s' may or may not be the same as s.

In the first case, the number of entries of D_1 in the row for any given output that overlap with 1s, for each s', can be at most $O(\log m')$, and since l is light, there can be at most ${m'}^{1/4}$ such s'. Thus, this way only contributes at most $O({m'}^{1/4} \log m')$ nonzero entries to $y_1 = D_1 ReLU(C_0D_0x_0)$. Furthermore, each of these entries has size at most $2/\gamma$ with high probability. The type (b) noise of any entry e of C'_1y_1 will consist of the sum of each of those entries multiplied by either 0 or a 1, with the probability of a 1 being $\log m'/n$. Thus, from a union bound the probability that this sum is non-zero is at most $O({m'}^{1/4} \log^2 m'/n) = O(\log m'/{m'}^{1/4})$. Furthermore, with high probability that sum will have at most $O(1)$ non-zero entries, and thus the type (b) noise when we hold l fixed is at most $O(1)$, and that constant can be made smaller than any ϵ by increasing the constant γ.

We next turn to the second case: noise of type (b) that combines s' with l', where $l' \neq l$. In this case, the number of entries of D_1 in the row for any given output that overlap with 1s will be $O(1)$ with high probability, and thus any non-zero entry of D_1 has value $O(\frac{1}{\log m'})$ with high probability. There are at most m' rows of D_1 that could have such overlap, and the probability of overlap for each of them is $O\left(n(\frac{1}{\sqrt{m'}})^2\right) = O\left(\frac{\log m'}{\sqrt{m'}}\right)$. Thus the resulting expected number of entries of y_1 that are non-zero is $O(\sqrt{m'}\log m')$. Using the fact that for any pair of rows of D_1 that involve two different light inputs, the entries in those rows will be independent, and the fact that every light input can appear in at most $m^{1/4}$ rows, we can use a Chernoff bound to show that it will not be higher by more than a constant factor.

Again, any entry e of C'_1y_1 will consist of the sum of each of those entries multiplied by either 0 or a 1, with the probability of a 1 being $\log m'/n$. The expected number of non-zero terms in that sum will be $O\left(\frac{\log^2 m'}{n\sqrt{m'}}\right) = O(\log m')$, and can be shown with a Chernoff bound to be within a constant factor of its expectation with high probability. Finally, since each of these terms is $O(\frac{1}{\log m'})$ with high probability, we see that this contribution to any entry of C'_1y_1 is at most $O(1)$. This can be made smaller than any ϵ by increasing n by a constant factor. $\square$

Claim 4.13 assumes that no two super heavy inputs are active. However, as described thus far, if two super heavy inputs were to be active, than a constant fraction of the entries in $ReLU(C_0D_0x_0)$ would be 1s, and this would wreak havoc with the entries in C'_1y_1. Fortunately, we do not need to handle the case of two active super heavy inputs here: if an output has two super heavy inputs, it will be handled by algorithm **High-Influence-AND**. However, we still have to ensure that when there are two active super heavy inputs, all of the mixed outputs return a 0. Specifically, when there are two active super heavy inputs, there is so much noise of type (b) that if we do not remove that noise, many mixed outputs would actually return a 1. The mechanism **detect-two-active-heavies** is how we remove that noise.

To construct this mechanism, we add a single row to the matrix C_0, called the cutoff row. Every column of C_0 corresponding to a super heavy input will have a 1 in the cutoff row, and all other columns will have a 0 there. D_1 will have a cutoff column which lines up with the cutoff row in C_0, and that column will have a value of $-Z$ in every row, where Z is large enough to guarantee that all entries of y_1 will be negative. Thus, all entries of $C'_1 y_1$ will be non-positive, and will be set to 0 by the subsequent $ReLU$ operation. Note that since we are using non-overlapping entries of x_0 to represent the super heavy inputs, there will not be any noise added to the cutoff row, and so this mechanism will not be triggered even partially when less than two super heavy inputs are active.

We point out that this operation is very reliant on there being at most two active inputs, and so the algorithm as described thus far does not work if three or more inputs are active (for example, two super heavy inputs and one light input would only return zeros for the mixed outputs). However, we describe below how to convert any algorithm for two active inputs into an algorithm that can handle more than two active inputs. □

This concludes the proof of Theorem 4.7. □

5 Generalizing the constructions

We here demonstrate how the above algorithm can be extended to more general settings, adding the ability to structurally handle more than two active inputs, handle multiple layers, and handle the k-AND function.

5.1 More than two active inputs

We have assumed throughout that at most 2 inputs are active at any time. It turns out that most of the pieces of our main algorithm work for any constant number of inputs being active, but one significant exception to that is the **detect-two-active-heavies** mechanism of Algorithm **Mixed-Influence-AND**, which requires at most 2 active inputs in order to work. Thus, we here describe a way to handle any number of active inputs, albeit at the cost of an increase in n. Let v be an upper bound on the number of active inputs.

We start with the case where $v = 3$, where there are three possible pairs of active inputs to a 2-AND. The idea will be to create enough copies of the problem so that for each of the three possible pairs of active inputs, there is a copy in which the pair appears without the third input active. To handle that, we make $O(\log m)$ copies of the problem (and thus increase n by that factor). These copies are partitioned into pairs, and each input goes into exactly one of the copies in each pairing. The choice of copy for each input is i.i.d. with probability

1/2. Each of the copies are now computed, using our main algorithm, except that only the outputs that have both of their inputs in a copy are computed, and we have an additional mechanism, similar to **detect-two-active-heavies**, that detects if a copy has 3 active inputs, and if so, it zeroes out all active outputs in that copy. Finally, we combine all the copies of each output that are computed, summing them up and then cutting off the result at 1. The number of copies is chosen so that for every set of three inputs, with high probability there will be a copy where each of the three possible pairs of inputs in that set of three inputs appears without the third input. Thus, for any set of three active inputs, each pair will be computed correctly in some copy, and so with high probability this provides us with the correct answer.

We can extend this to any bound v on the number of active inputs. We still partition the copies into pairs, and we need any set of v inputs to have one copy where each set of two inputs appears separately from the other $v-2$ inputs. The probability for this to happen for a given set of v inputs and a pair within that set is $1/2^{v-1}$. The number of choices of such sets is $\binom{m}{v}\binom{v}{2}$. Thus, to get all of the pairings we need to occur, the number of copies we need to make is $O\left(2^{v-1}\log\left[\binom{m}{v}\binom{v}{2}\right]\right) \leq O(v2^v \log m)$. As a result, we can still compute in superposition up to when $v = O(\log m')$. We note that some care needs to be taken with the initial distribution of copies of each input to ensure that process does not create too much (type (a)) noise, but given how quickly n grows with v due to the number of copies required, this is not difficult.

5.2 Multiple layers

These algorithms can be used to compute an unlimited number of layers because, as discussed above, as long as the output of a layer is close to the actual result (intended 1s are at least 3/4 and intended 0s are at most 1/4), we can use ReLU to make them exact Boolean outputs. Therefore, the noise introduced during the processing of a layer is removed between layers, and so does not add up to become a constraint on depth. Also, as discussed, the high probability results all are with respect to whether or not the algorithm for a given layer works correctly. Therefore, each layer can be checked for correctness, and redone if there is an error, which ensures that all outputs are computed correctly for every layer. Thus, there is no error that builds up from layer to layer.

5.3 Computing k-AND

We next turn our attention to the question of k-AND. To do so, we simply utilize our ability to handle multiple layers of computation to convert a k-way AND function to a series of pairwise AND functions. Specifically, to compute each individual k-AND, we build a binary tree with k leaves where each node of the tree is a pairwise AND of two variables. These then get mapped to a binary tree of vector 2-AND functions where each individual pairwise AND is computed in exactly one vector 2-AND function, to compute the entire vector

k-AND function. We thus end up with $2k$ 2-AND functions to compute, which we do with an additional factor of $\log k$ in the depth of the network.

As described so far, this will increase the number of neurons required by the network by a factor of $O(k)$. However, for k-AND to be interesting, there would need to be the possibility of at least k inputs being active ($v \geq k$), and so for any interesting case of k-AND, we would be using our technique for more than 2 active inputs described above, and so if we want to compute in superposition, we have the limitation that $k = O(\log m')$. However, since that technique already ensures that every pair of the k-AND appears by itself in one of the copies of the network, we can embed the leaves of our binary tree into those copies. We would then do the same thing with the next level of the tree, and so on until we get to the root of the tree. Since each level of the tree has half the number of outputs as the previous level, we get a telescoping sum, and thus k-AND can be added to our implementation of at least k active inputs without changing the asymptotics of n. It does however add an additional factor of $\log k$ to the depth of the network.

5.4 Arbitrary Boolean circuits

The results we have demonstrated in this paper can be extended to arbitrary Boolean functions. However, presenting these extensions is beyond the scope of this paper, and will be described in a followup manuscript. Instead, we here only point out that with more general Boolean functions, the notion of feature influence becomes even more important. In fact, without an upper bound on the maximum feature influence, even computing pairwise ORs wholly in superposition is not possible: if a single input appears in (for example) half of the pairwise ORs, then when that input is a 1, half the outputs will be 1. Thus, the outputs cannot be even represented in superposition.

6 Conclusion

To the best of our knowledge this is the first paper to address the complexity of neural network computation in superposition, an important new topic in the field of mechanistic interpretability. Our work delivers the first upper and lower bounds on such computation and offers insights into what types of techniques can be effective in neural networks.

There are many questions that emerge from our work; here are some examples. Do real world, trained neural networks exhibit any of the techniques we have described in our algorithms? It may be easier to uncover what these networks are actually doing in practice when armed with techniques that we know can be effective. Can our algorithms be helpful in designing real networks by using them to map a known feature circuit to a superposed neural network? This may reduce the computational effort of building a model, and also improve its effectiveness and/or interpretability. Can our lower bound techniques be used to prove useful lower bounds on the parameter description of neural networks for real world problems, such as LLMs and image generation? Can we close the remaining gap of the various complexity measures? This includes the gap of $\sqrt{\log m'}$ for n, and determining the precise dependence on the number of active features and the size of the formula being computed. What is the impact of using non-Boolean variables and other activation functions besides ReLU?

Looking forward, we hope our work can be viewed by the theory community as setting up the elements of this new computation model so as to enable further research. For the deep learning mechanistic interpretability community, we aim to build a bridge to complexity theory. The important research analyzing the way neural network expressibility is captured by features is at its infancy, and it would be good to find ways to co-develop the safety aspects of this research together with an understanding of its complexity implications, in particular given the enormous costs involved in running neural computation.

References

[1] Parameterized approximation algorithm. *https://en.wikipedia.org/wiki/Parameterized_approximation_algorithm.* Accessed: 2023-08-10.

[2] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *https://arxiv.org/abs/1611.01491*, 2018.

[3] Sanjeev Arora, Yingyu Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. Linear algebraic structure of word senses, with applications to polysemy. *Transactions of the Association for Computational Linguistics*, 6:483–495, 2018.

[4] Burton Howard Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nicholas Turner, Cem Anil, Charles Denison, Amanda Askell, Robert Lasenby, Yuhuai Wu, Shane Kravec, Nicholas Schiefer, Tristan Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Kathy Nguyen, Ben McLean, James E. Burke, Tristan Hume, Shan Carter, Tom Henighan, and Chris Olah. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, 2023.

[6] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2002.

[7] Lawrence Chan. Superposition is not "just" neuron polysemanticity. *AI Alignment Forum*, 2024. Accessed from *https://www.alignmentforum.org/posts/8EyCQKuWo6swZpagS/superposition-is-not-just-neuron-polysemanticity.*

[8] 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.

[9] Marek Cygan et al. Parameterized algorithms. *https://www.mimuw.edu.pl/˜malcin/book/parameterized-algorithms.pdf*. Accessed: 2023-08-10.

[10] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[11] Philipp Grohs, Shokhrukh Ibragimov, Arnulf Jentzen, and Sarah Koppensteiner. Lower bounds for artificial neural network approximations: A proof that shallow neural networks fail to overcome the curse of dimensionality. *https://arxiv.org/abs/2103.04488*, 2024.

[12] 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.

[13] Christoph Hertrich, Amitabh Basu, Marco Di Summa, and Martin Skutella. Towards lower bounds on the depth of relu neural networks. In *Advances in Neural Information Processing Systems*, volume 34. Curran Associates, Inc., 2021.

[14] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, abs/1503.02531, 2015.

[15] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 23:1–124, September 2021. Submitted 4/21; Revised 6/21; Published 9/21.

[16] William B. Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[18] Michael Mitzenmacher and Eli Upfal. Probability and computing: Randomized algorithms and probabilistic analysis. *Cambridge University Press*, pages 63–68.

[19] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *https://distill.pub/2020/circuits/zoom-in/*, March 2020.

[20] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. *https://arxiv.org/abs/1901.03429*, 2019.

[21] Adam Scherlis, Kshitij Sachan, Adam Jermyn, Joe Benton, and Buck Shlegeris. Polysemanticity and capacity in neural networks. *https://arxiv.org/abs/2210.01892*, 2023.

[22] David Slepian. Permutation modulation. *Proceedings of the IEEE*, 53(3):228–236, Mar 1965.

[23] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, Alex Tamkin, Esin Durmus, Tristan Hume, Francesco Mosconi, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits, https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html*, 2024.

[24] 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.

[25] Raphael Yuster and Uri Zwick. Sparse representations. *https://www.cs.tau.ac.il/ zwick/papers/sparse.pdf*. Accessed: 2023-08-10.