Recursive Sketches for Modular Deep Learning

Badih Ghazi [*1] **Rina Panigrahy** [*1]
Joshua R. Wang [*1]

Abstract

We present a mechanism to compute a sketch (succinct summary) of how a complex modular deep network processes its inputs. The sketch summarizes essential information about the inputs and outputs of the network and can be used to quickly identify key components and summary statistics of the inputs. Furthermore, the sketch is recursive and can be unrolled to identify sub-components of these components and so forth, capturing a potentially complicated DAG structure. These sketches erase gracefully; even if we erase a fraction of the sketch at random, the remainder still retains the "high-weight" information present in the original sketch. The sketches can also be organized in a repository to implicitly form a "knowledge graph"; it is possible to quickly retrieve sketches in the repository that are related to a sketch of interest; arranged in this fashion, the sketches can also be used to learn emerging concepts by looking for new clusters in sketch space. Finally, in the scenario where we want to learn a ground truth deep network, we show that augmenting input/output pairs with these sketches can theoretically make it easier to do so.

1. Introduction

Machine learning has leveraged our understanding of how the brain functions to devise better algorithms. Much of classical machine learning focuses on how to *correctly* compute a function; we utilize the available data to make more accurate predictions. More recently, lines of work have considered other important objectives as well: we might like our algorithms to be small, efficient, and robust. This work aims to further explore one such sub-question: can we design a system on top of neural nets that efficiently stores information?

Our motivating example is the following everyday situation. Imagine stepping into a room and briefly viewing the objects within. Modern machine learning is excellent at answering immediate questions about this scene: "Is there a cat? How big is said cat?" Now, suppose we view this room every day over the course of a year. Humans can reminisce about the times they saw the room: "How often did the room contain a cat? Was it usually morning or night when we saw the room?"; can we design systems that are also capable of efficiently answering such memory-based questions?

Our proposed solution works by leveraging an existing (already trained) machine learning model to understand individual inputs. For the sake of clarity of presentation, this base machine learning model will be a modular deep network.[1] We then augment this model with sketches of its computation. We show how these sketches can be used to efficiently answer memory-based questions, despite the fact that they take up much less memory than storing the entire original computation.

A modular deep network consists of several independent neural networks (modules) which only communicate via one's output serving as another's input. Figure 1 presents a cartoon depiction of a modular network for our example. Modular networks have both a biological basis (Azam, 2000) and evolutionary justification (Clune et al., 2013). They have inspired several practical architectures such as neural module networks (Andreas et al., 2016; Hu et al., 2017), capsule neural networks (Hinton et al., 2000; 2011; Sabour et al., 2017), and PathNet (Fernando et al., 2017) and have connections to suggested biological models of intelligence such as hierarchical temporal memory (Hawkins & Blakeslee, 2007). We choose them as a useful abstraction to avoid discussing specific network architectures.

What do these modules represent in the context of our room task? Since we are searching for objects in the room, we think of each module as attempting to identify a particular type of object, from the low level edge to the high level cat. For the reader familiar with convolutional neural networks, it may help to think of each module as a filter or kernel. We denote the event where a module produces output as an object, the produced output vector as an attribute vector, and all objects produced by a module as a class. In fact, our sketching mechanism will detail how to sketch each object, and these sketches will be *recursive*, taking into account the sketches corresponding to the inputs of the module.

[*]Equal contribution [1]Google Research, Mountain View, CA, USA.. Correspondence to: Rina Panigrahy <rinap@google.com>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

[1]Of course, it is possible to cast many models as deep modular networks by appropriately separating them into modules.

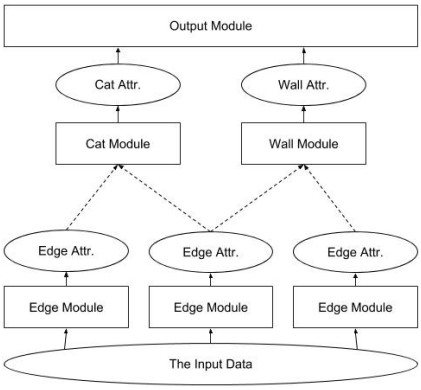

Figure 1. Cartoon depiction of a modular network processing an image of a room. Modules are drawn as rectangles and their inputs/outputs (which we refer to as object attributes) are drawn as ovals. The arrows run from input vector to module and module to output vector. There may be additional layers between low level and high level modules, indicated by the dashed arrows. The output module here is a dummy module which groups together top-level objects.

Armed with this view of modular networks for image processing, we now entertain some possible sketch properties. One basic use case is that from the output sketch, we should be able to say something about the attribute vectors of the objects that went into it. This encompasses a question like "How big was the cat?" Note that we need to assume that our base network is capable of answering such questions, and the primary difficulty stems from our desire to maintain this capability despite only keeping a small sketch. Obviously, we should not be able to answer such questions as precisely as we could with the original input. Hence a reasonable *attribute recovery* goal is to be able to approximately recover an object's attribute vector from its sketch. Concretely, we should be able to recover cat attributes from our cat sketch.

Since we plan to make our sketches recursive, we would actually like to recover the cat attributes from the final output sketch. Can we recover the cat's sketch from final sketch? Again, we have the same tug-of-war with space and should expect some loss from incorporating information from many sketches into one. Our *recursive sketch* property is that we can recover (a functional approximation of) a sketch from a later sketch which incorporated it.

Zooming out to the entire scene, one fundamental question is "Have I been in this room before?" In other words, comparing sketches should give some information about how similar the modules and inputs that generated them. In the language of sketches, our desired *sketch-to-sketch similarity* property is that two completely unrelated sketches will be dissimilar while two sketches that share common modules and similar objects will be similar.

Humans also have the impressive ability to recall approximate counting information pertaining to previous encounters (Brannon & Roitman, 2003). The *summary statistics* property states that from a sketch, we can approximately recover the number of objects produced by a particular module, as well as their mean attributes.

Finally, we would like the nuance of being able to forget information gradually, remembering more important facts for longer. The *graceful erasure* property expresses this idea: if (a known) portion of our sketch is erased, then the previous properties continue to hold, but with additional noise depending on the amount erased.

We present a sketching mechanism which simultaneously satisfies all these desiderata: *attribute recovery*, *recursive sketch*, *sketch-to-sketch similarity*, *summary statistics*, and *graceful erasure*.

1.1. Potential Applications

We justify our proposed properties by describing several potential applications for a sketching mechanism satisfying them.

Sketch Repository. The most obvious use case suggested by our motivating example is to maintain a repository of sketches produced by the (augmented) network. By the *recursive sketch* and *attribute recovery* properties, we can query our repository for sketches with objects similar to a particular object. From the *sketch-to-sketch similarity* property, we can think of the sketches as forming a knowledge graph; we could cluster them to look for patterns in the data or use locality-sensitive hashing to quickly fish out related sketches. We can combine these overall sketches once more and use the *summary statistics* property to get rough statistics of how common certain types of objects are, or what a typical object of a class looks like. Our repository may follow a generalization of a "least recently used" policy, taken advantage of the *graceful erasure* property to only partially forget old sketches.

Since our sketches are recursively defined, such a repository is not limited to only keeping overall sketches of inputs; we can also store lower-level object sketches. When we perform object attribute recovery using the *attribute recovery* property, we can treat the result as a fingerprint to search for a more accurate sketch of that object in our repository.

Learning New Modules. This is an extension of the sketch repository application above. Rather than assuming we start with a pre-trained modular network, we start with a barebones modular network and want to grow it. We feed it inputs and examine the resulting sketches for emerging clusters. When we find such a cluster, we use its points to train a corresponding (reversible) module. In fact, this clustering may be done in a hierarchical fashion to get finer classes.

For example, we may develop a module corresponding to "cat" based on a coarse cluster of sketches which involve cats. We may get sub-clusters based on cat species or even finer sub-clusters based on individual cats repeatedly appearing.

The modules produced by this procedure can capture important primitives. For example, the *summary statistics* property implies that a single-layer module could count the number of objects produced by another module. If the modules available are complex enough to capture curves, then this process could potentially allow us to identify different types of motion.

Interpretability. Since our sketches store information regarding how the network processed an input, they can help explain how that network came to its final decision. The *sketch-to-sketch similarity* property tells us when the network thinks two rooms are similar, but what exactly did the network find similar? Using the *recursive sketch* property, we can drill down into two top level sketches, looking for the pairs of objects that the network found similar.

Model Distillation. Model distillation (Buciluǎ et al., 2006; Hinton et al., 2015) is a popular technique for improving machine learning models. The use case of model distillation is as follows. Imagine that we train a model on the original data set, but it is unsatisfactory in some regard (e.g. too large or ensemble). In model distillation, we take this first model and use its features to train a second model. These stepping stones allows the second model to be smaller and less complex. The textbook example of feature choice is to use class probabilities (e.g. the object is 90% A and 10% B) versus the exact class found in the original input (e.g. the object is of class A). We can think of this as an intermediate option between the two extremes of (i) providing just the original data set again and (ii) providing all activations in the first model. Our sketches are an alternative option. To showcase the utility of our sketches in the context of learning from a teacher network, we prove that (under certain assumptions) augmenting the teacher network with our sketches allows us to learn the network. Note that in general, it is not known how the student can learn solely from many input/output pairs.

1.2. Techniques

The building block of our sketching mechanism is applying a random matrix to a vector. We apply such transformations to object attribute vectors to produce sketches, and then recursively to merge sketches. Consequently, most of our analysis revolves around proving and utilizing properties of random matrices. In the case where we know the matrices used in the sketch, then we can use an approximate version of isometry to argue about how noise propagates through our sketches. On the other hand, if we do not know the matrices involved, then we use sparse recovery/dictionary learning techniques to recover them. Our recovery procedure is at partially inspired by several works on sparse coding and dictionary learning (Spielman et al., 2012; Arora et al., 2014c), (Arora et al., 2014a) and (Arora et al., 2015) as well as their known theoretical connections to neural networks (Arora et al., 2014b; Papyan et al., 2018). We have more requirements on our procedure than previous literature, so we custom-design a distribution of block-random matrices reminiscent of the sparse Johnson-Lindenstrauss transform. Proving procedure correctness is done via probabilistic inequalities and analysis tools, including the Khintchine inequality and the Hanson-Wright inequality.

One could consider an alternate approach based on the serialization of structured data (for example, protocol buffers). Compared to this approach, ours provides more fine-grained control over the accuracy versus space trade-off and aligns more closely with machine learning models (it operates on real vectors).

1.3. Related Work

There have been several previous attempts at adding a notion of memory to neural networks. There is a line of work which considers augmenting recurrent neural networks with external memories (Graves et al., 2014; Sukhbaatar et al., 2015; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Zaremba & Sutskever, 2015; Danihelka et al., 2016; Graves et al., 2016). In this line of work, the onus is on the neural network to learn how to write information to and read information from the memory. In contrast to this line of work, our approach does not attempt to use machine learning to manage memory, but rather demonstrates that proper use of random matrices suffices. Our results can hence be taken as theoretical evidence that this learning task is relatively easy.

Vector Symbolic Architectures (VSAs) (Smolensky, 1990; Gayler, 2004; Levy & Gayler, 2008) are a class of memory models which use vectors to represent both objects and the relationships between them. There is some conceptual overlap between our sketching mechanism and VSAs. For example, in Gayler's MAP (Multiply, Add, Permute) scheme (Gayler, 2004), vector addition is used to represent superposition and permutation is used to encode quotations. This is comparable to how we recursively encode objects; we use addition to combine multiple input objects after applying random matrices to them. One key difference in our model is that the object attribute vectors we want to recover are provided by some pre-trained model and we make no assumptions on their distribution. In particular, their possible correlation makes it necessary to first apply a random transformation before combining them. In problems where data is generated by simple programs, several papers (Wu et al., 2017; Yi et al., 2018; Ellis et al., 2018; Andreas et al., 2016; Oh et al., 2017) attempt to infer programs from the

generated training data possibly in a modular fashion.

Our result on the learnability of a ground truth network is related to the recent line of work on learning small-depth neural networks (Du et al., 2017b;a; Li & Yuan, 2017; Zhong et al., 2017a; Zhang et al., 2017; Zhong et al., 2017b). Recent research on neural networks has considered evolving neural network architectures for image classification (e.g., (Real et al., 2018)), which is conceptually related to our suggested Learning New Models application.

1.4. Organization

We review some basic definitions and assumptions in Section 2. An overview of our final sketching mechanism is given in Section 3 (Theorems 1, 2, 3). In Section 4, we explain how to deduce the random parameters of our sketching mechanism from multiple sketches via dictionary learning. Further discussion and detailed proofs are provided in the supplementary material.

2. Preliminaries

In this section, we cover some preliminaries that we use throughout the remainder of the paper. Throughout the paper, we denote by $[n]$ the set $\{1, \ldots, n\}$.

2.1. Modular Deep Networks for Object Detection

For this paper, a modular deep network consists of a collection of modules $\{M\}$. Each module is responsible for detecting a particular class of object, e.g. a cat module may detect cats. These modules use the outputs of other modules, e.g. the cat module may look at outputs of the edge detection module rather than raw input pixels. When the network processes a single input, we can depict which objects feed into which other objects as a (weighted) communication graph. This graph may be data-dependent. For example, in the original Neural Module Networks of Andreas et al. (Andreas et al., 2016), the communication graph is derived by taking questions, feeding them through a parser, and then converting parse trees into structured queries. Note that this pipeline is disjoint from the modular network itself, which operates on images. In this paper, we will not make assumptions about or try to learn the process producing these communication graphs. When a module does appear in the communication graph, we say that it detected an object θ and refer to its output as x_θ, the attribute vector of object θ. We assume that these attribute vectors x_θ are nonnegative and normalized. As a consequence of this view of modular networks, a communication graph may have far more objects than modules. We let our input size parameter N denote twice the maximum between the number of modules and the number of objects in any communication graph.

We assume we have access to this communication graph, and we use $\theta_1, ..., \theta_k$ to denote the k input objects to object θ. We assume we have a notion of how important each input object was; for each input θ_i, there is a nonnegative weight w_i such that $\sum_{i=1}^{k} w_i = 1$. It is possible naively choose $w_i = 1/k$, with the understanding that weights play a role in the guarantees of our sketching mechanism (it does a better job of storing information about high-weight inputs).

We handle the network-level input and output as follows. The input data can be used by modules, but it itself is not produced by a module and has no associated sketch. We assume the network has a special output module, which appears exactly once in the communication graph as its sink. It produces the output (pseudo-)object, which has associated sketches like other objects but does not count as an object when discussing sketching properties (e.g. two sketches are not similar just because they must have output pseudo-objects). This output module and pseudo-object only matter for our overall sketch insofar as they group together high-level objects.

2.2. Additional Notation for Sketching

We will (recursively) define a sketch for every object, but we consider our sketch for the *output object* of the network to be the overall sketch. This is the sketch that we want to capture how the network processed a particular input. One consequence is that if an object does not have a path to the output module in the communication graph, it will be omitted from the overall sketch.

Our theoretical results will primarily focus on a single path from an object θ to the output object. We define the "effective weight" of an object, denoted w_θ, to be the product of weights along this path. For example, if an edge was 10% responsible for a cat and the cat was 50% of the output, then this edge has an effective weight of 5% (for this path through the cat object). We define the "depth" of an object, denoted $h(\theta)$, to be the number of objects along this path. In the preceding example, the output object has a depth of one, the cat object has a depth of two, and the edge object has a depth of three. We make the technical assumption that all objects produced by a module must be at the same depth. As a consequence, modules M also have a depth, denoted $h(M)$. To continue the example, the output *module* has depth one, the cat *module* has depth two, and the edge *module* has depth three. Furthermore, this implies that each object and module can be assigned a unique depth (i.e. all paths from any object produced by a module to the output object have the same number of objects).

Our recursive object sketches are all d_{sketch} dimensional vectors. We assume that d_{sketch} is at least the size of any object attribute vector (i.e. the output of any module). In general, all of our vectors are d_{sketch}-dimensional; we assume that shorter object attribute vectors are zero-padded (and

normalized).

3. Overview of Sketching Mechanism

In this section, we present our sketching mechanism, which attempts to summarize how a modular deep network understood an input. The mechanism is presented at a high level; see Section **??** for how we arrived at this mechanism and the supplementary material for proofs of the results stated in this section.

3.1. Desiderata

As discussed in the introduction, we want our sketching mechanism to satisfy several properties, listed below.

Attribute Recovery. Object attribute vectors can be approximately recovered from the overall sketch, with additive error that decreases with the effective weight of the object.

Sketch-to-Sketch Similarity. With high probability, two completely unrelated sketches (involving disjoint sets of modules) will have a small inner product. With high probability, two sketches that share common modules and similar objects will have a large inner product (increasing with the effective weights of the objects).

Summary Statistics. If a single module detects several objects, then we can approximately recover summary statistics about them, such as the number of such objects or their mean attribute vector, with additive error that decreases with the effective weight of the objects.

Graceful Erasure. Erasing everything but d'_{sketch}-prefix of the overall sketch yields an overall sketch with the same properties (albeit with larger error).

3.2. Random Matrices

The workhorse of our recursive sketching mechanism is random (square) matrices. We apply these to transform input sketches before incorporating them into the sketch of the current object. We will defer the exact specification of how to generate our random matrices to Section 4, but while reading this section it may be convenient for the reader to think of our random matrices as uniform random *orthonormal matrices*. Our actual distribution choice is similar with respect to the properties needed for the results in this section. One notable parameter for our family of random matrices is $\delta \geq 0$, which expresses how well they allow us to estimate particular quantities with high probability. For both our random matrices and uniform random orthonormal matrices, δ is $\tilde{O}(1/\sqrt{d_{\text{sketch}}})$. We prove various results about the properties of our random matrices in the supplementary material.

We will use R with a subscript to denote such a random matrix. Each subscript indicates a fresh draw of a matrix from our distribution, so every R_1 is always equal to every other R_1 but R_1 and $R_{\text{cat},0}$ are independent. Throughout this section, we will assume that we have access to these matrices when we are operating on a sketch to retrieve information. In Section 4, we show how to recover these matrices given enough samples.

3.3. The Sketching Mechanism

The basic building block of our sketching mechanism is the tuple sketch, which we denote s_{tuple}. As the name might suggest, its purpose is to combine k sketches $s_1, ..., s_k$ with respective weights $w_1, ..., w_k \geq 0$ into a single sketch (for proofs, we will assume that these weights sum to at most one). In the case where the k sketches are input sketches, these weights will be the importance weights from our network. Otherwise, they will all be $1/k$. Naturally, sketches with higher weight can be recovered more precisely. The tuple sketch is formally defined as follows. If their values are obvious from context, we will omit $w_1, ..., w_k$ from the arguments to the tuple sketch. The tuple sketch is computed as follows.[2]

$$s_{\text{tuple}}(s_1, ..., s_k, w_1, ..., w_k) \coloneqq \sum_{i=1}^{k} w_i \left(\tfrac{I+R_i}{2}\right) s_i$$

Note that I is the identity matrix, and we will define a tuple sketch of zero things to be the zero vector.

Each object θ is represented by a sketch, which naturally we will refer to as an object sketch. We denote such a sketch as s_{object}. We want the sketch of object θ to incorporate information about the attributes of θ itself as well as information about the inputs that produced it. Hence we also define two subsketches for object θ. The attribute subsketch, denoted s_{attr}, is a sketch representation of object θ's attributes. The input subsketch, denoted s_{input}, is a sketch representation of the inputs that produced object θ. These three sketches are computed as follows.

$$s_{\text{object}}(\theta) \coloneqq \left(\tfrac{I+R_{M(\theta),0}}{2}\right) s_{\text{tuple}}\left(s_{\text{attr}}(\theta), s_{\text{input}}(\theta), \tfrac{1}{2}, \tfrac{1}{2}\right)$$

$$s_{\text{attr}}(\theta) \coloneqq \tfrac{1}{2} R_{M(\theta),1} x_\theta + \tfrac{1}{2} R_{M(\theta),2} e_1$$

$$s_{\text{input}}(\theta) \coloneqq s_{\text{tuple}}\left(s_{\text{object}}(\theta_1), ..., s_{\text{object}}(\theta_k), w_1, ..., w_k\right)$$

Note that e_1 in the attribute subsketch is the first standard basis vector; we will use it for frequency estimation as part of our **Summary Statistics** property. Additionally, $\theta_1, ..., \theta_k$ in the input sketch are the input objects for θ and $w_1, ..., w_k$ are their importance weights from the network.

The overall sketch is just the output psuedo-object's *input*

[2] Rather than taking a convex combination of $\left(\frac{I+R_i}{2}\right) s_i$, one might instead sample a few of them. Doing so would have repercussions on the results in this section, swapping many bounds from high probability to conditioning on their objects getting through. However, it also makes it easier to do the dictionary learning in Section 4.

subsketch. It is worth noting that we do not choose to use its object sketch.

$$s_{\text{overall}} \coloneqq s_{\text{input}}(\text{output pseudo-object})$$

We want to use this to (noisily) retrieve the information that originally went into these sketches. We are now ready to present our results for this sketching mechanism. Note that we provide the technical machinery for proofs of the following results in the supplementary material and we provide the proofs themselves in the supplementary material. Our techniques involve using an approximate version of isometry and reasoning about the repeated application of matrices which satisfy our approximate isometry.

Our first result concerns **Attribute Recovery** (example use case: roughly describe the cat that was in this room).

Theorem 1. *Our sketch has the simplified **Attribute Recovery** property, which is as follows. Consider an object $\theta^\star$ at constant depth $h(\theta^\star)$ with effective weight $w_{\theta^\star}$.*

- *(i) Suppose no other objects in the overall sketch are also produced by $M(\theta^\star)$. We can produce a vector estimate of the attribute vector $x_{\theta^\star}$, which with high probability has at most $O(\delta/w_{\theta^\star})$ ℓ_∞-error.*
- *(ii) Suppose we know the sequence of input indices to get to $\theta^\star$ in the sketch. Then even if other objects in the overall sketch are produced by $M(\theta^\star)$, we can still produce a vector estimate of the attribute vector $x_{\theta^\star}$, which with high probability has at most $O(\delta/w_{\theta^\star})$ ℓ_∞-error.*

As a reminder, ℓ_∞ error is just the maximum error over all coordinates. Also, note that if we are trying to recover a quantized attribute vector, then we may be able to recover our attribute vector *exactly* when the quantization is larger than our additive error. This next result concerns **Sketch-to-Sketch Similarity** (example use case: judge how similar two rooms are). We would like to stress that the output psuedo-object does not count as an object for the purposes of (i) or (ii) in the next result.

Theorem 2. *Our sketch has the **Sketch-to-Sketch Similarity** property, which is as follows. Suppose we have two overall sketches s and $\bar{s}$.*

- *(i) If the two sketches share no modules, then with high probability they have at most $O(\delta)$ dot-product.*
- *(ii) If s has an object $\theta^\star$ of weight $w_{\theta^\star}$ and $\bar{s}$ has an object $\bar{\theta}^\star$ of weight $\bar{w}_{\theta^\star}$, both objects are produced by the same module, and both objects are at constant depth $h(\theta^\star)$ then with high probability they have at least $\Omega(w_{\theta^\star}\bar{w}_{\bar{\theta}^\star}) - O(\delta)$ dot-product.*
- *(iii) If the two sketches are identical except that the attributes of any object θ differ by at most ϵ in ℓ_2 distance, then with probability one the two sketches are at most $\epsilon/2$ from each other in ℓ_2 distance.*

Although our **Sketch-to-Sketch Similarity** result is framed in terms of two overall sketches, the recursive nature of our sketches makes it not too hard to see how it also applies to any pair of sketches computed along the way.

This next result concerns **Summary Statistics** (example use case: how many walls are in this room).

Theorem 3. *Our sketch has the simplified **Summary Statistics** property, which is as follows. Consider a module $M^\star$ which lies at constant depth $h(M^\star)$, and suppose all the objects produced by $M^\star$ has the same effective weight $w^\star$.*

- *(i) **Frequency Recovery**. We can produce an estimate of the number of objects produced by $M^\star$. With high probability, our estimate has an additive error of at most $O(\delta/w^\star)$.*
- *(ii) **Summed Attribute Recovery**. We can produce a vector estimate of the summed attribute vectors of the objects produced by $M^\star$. With high probability, our estimate has at most $O(\delta/w^\star)$ ℓ_∞-error.*

Again, note that quantization may allow for exact recovery. In this case, the number of objects is an integer and hence can be recovered exactly when the additive error is less than $1/2$. If we can recover the exact number of objects, we can also estimate the mean attribute vector.

The supplementary material explains how to generalize our sketch to incorporate signatures for the **Object Signature Recovery** property and also explains how to generalize these results to hold when part of the sketch is erased for the **Graceful Erasure** property.

4. Learning Modular Deep Networks via Dictionary Learning

In this section, we demonstrate that our sketches carry enough information to learn the network used to produce them. Specifically, we develop a method for training a new network based on (input, output, sketch) triples obtained from a teacher modular deep network. Our method is powered by a novel *dictionary learning* procedure. Loosely speaking, dictionary learning tries to solve the following problem. There is an unknown dictionary matrix R, whose columns are typically referred to as atoms. There is also a sequence of unknown sparse vectors $x^{(k)}$; we only observe how they combine the atoms, i.e., $\{y^{(k)} = Rx^{(k)}\}$. We want to use these observations $y^{(k)}$ to recover the dictionary matrix R and the original sparse vectors $x^{(k)}$.

While dictionary learning has been well-studied in the literature from both an applied and a theoretical standpoint, our setup differs from known theoretical results in several key aspects. The main complication is that since we want to apply our dictionary learning procedure recursively, our error in recovering the unknown vectors $x^{(k)}$ will become noise on the observations $y^{(k)}$ in the very next step. Note

that classical dictionary learning can only recover the unknown vectors $x^{(k)}$ up to permutation and coordinate-wise sign. To do better, we will carefully engineer our distribution of dictionary matrices R to allow us to infer the permutation (between the columns of a matrix) and signs, which is needed to recurse. Additionally, we want to allow very general unknown vectors $x^{(k)}$. Rather than assuming sparsity, we instead make the weaker assumption that they have bounded ℓ_2 norm. We also allow for distributions of $x^{(k)}$ which make it impossible to recover all columns of R; we simply recover a subset of essential columns.

With this in mind, our desired dictionary learning result is Theorem 4. We plan to apply this theorem with N as thrice the maximum between the number of modules and the number of objects in any communication graph, S as the number of samples necessary to learn a module, and H as three times the depth of our modular network. Additionally, we think of ϵ_1 as the tolerable input ℓ_∞ error while ϵ_H is the desired output ℓ_∞ error after recursing H times.

Theorem 4. *[Recursable Dictionary Learning] There exists a family of distributions $\{\mathcal{D}(b, q, d_{sketch})\}$ which produce $d_{sketch} \times d_{sketch}$ matrices satisfying the following. For any positive integers N, S, positive constant H, positive real ϵ_H, block size $b \geq poly(\log N, \log d_{sketch}, 1/\epsilon_H)$, nonzero block probability $q \geq poly(\log N, \log d_{sketch}, 1/\epsilon_H)/\sqrt{d_{sketch}}$, and dimension $d_{sketch} \geq poly(1/\epsilon_H, \log N, \log S)$, there exists:*

- *a base number of samples S where $S \leq S \leq S \cdot poly(N)$,*
- *and a sequence of ℓ_∞ errors $(0 <)\epsilon_1 \leq \epsilon_2 \leq \cdots \leq \epsilon_{H-1}(\leq \epsilon_H)$ with $\epsilon_1 \geq poly(\epsilon_H)$,*

such that the following is true. For any $h \in [H-1]$, let $S_h = SN^{h-1}$. For any unknown vectors $x^{(1)}, ..., x^{(S_h)} \in \mathbb{R}^{d_{sketch} \times N}$ with ℓ_2 at most $O(1)$, if we draw $R_1, ..., R_N \sim \mathcal{D}(d_{sketch})$ and receive S_h noisy samples $y^{(k)} := [R_1 R_2 \cdots R_N]x^{(k)} + z_1^{(k)} + z_\infty^{(k)}$ where each $z_1^{(k)} \in \mathbb{R}^{d_{sketch}}$ is noise with $\left|\left|z_1^{(k)}\right|\right|_1 \leq O(\sqrt{d_{sketch}})$ (independent of our random matrices) and each $z_\infty^{(k)} \in \mathbb{R}^{d_{sketch}}$ is noise with $\left|\left|z_\infty^{(k)}\right|\right|_\infty \leq \epsilon_h$ (also independent of our random matrices), then there is an algorithm which takes as input h, $y^{(1)}$, ..., $y^{(S_h)}$, runs in time $poly(S_h, d_{sketch})$, and with high probability outputs $\hat{R}_1, ..., \hat{R}_N, \hat{x}^{(1)}, ..., \hat{x}^{(S_h)}$ satisfying the following for some permutation π of $[N]$:

- *for every $i \in [N]$ and $j \in [d_{sketch}]$, if there exists $k^\star \in [S_h]$ such that $\left|x^{(k^\star)}_{(i-1)d_{sketch}+j}\right| \geq \epsilon_{h+1}$ then the j^{th} column of $\hat{R}_{\pi(i)}$ is $0.2d_{sketch}$ in Hamming distance from the j^{th} column of R_i.*
- *for every $k \in [S_h], i \in [N], j \in [d_{sketch}]$, $\left|x^{(k)}_{(\pi(i)-1)d_{sketch}+j} - x^{(k)}_{(i-1)d_{sketch}+j}\right| \leq \epsilon_{h+1}$.*

4.1. Recursable Dictionary Learning Implies Network Learnability

We want to use Theorem 4 to learn a teacher modular deep network, but we need to first address an important issue. So far, we have not specified how the deep modular network decides upon its input-dependent communication graph. As a result, the derivation of the communication graph cannot be learned (it's possibly an uncomputable function!). When the communication graph is a fixed tree (always the same arrangement of objects but with different attributes), we can learn it. Note that any fixed communication graph can be expanded into a tree; doing so is equivalent to not re-using computation. Regardless of the communication graph, we can learn the input/output behavior of each module regardless of whether the communication graph is fixed.

Theorem 5. *If the teacher deep modular network has constant depth, then any module $M^\star$ which satisfies the following two properties:*

- *(Robust Module Learnability) The module is learnable from $(\alpha = poly(N))$ input/output training pairs which have been corrupted by ℓ_∞ error at most a constant $\epsilon > 0$.*
- *(Sufficient Weight) In a $\left(\beta = \frac{1}{poly(N)}\right)$-fraction of the inputs, the module produces an object and all of the input objects to that object have effective weight at least w.*

can, with high probability, be learned from $poly(N)$ overall sketches of dimension $poly(1/w, 1/\epsilon)\log^2 N$.

Suppose we additionally know that the communication graph is a fixed tree. We can identify the sub-graph of objects which each have effective weight w in a $\left(\beta = \frac{1}{poly(N)}\right)$-fraction of the inputs.

Theorem 5 is proved in the supplementary material. The main idea is to just repeatedly apply our recursable dictionary learning algorithm and keep track of which vectors correspond to sketches and which vectors are garbage.

4.2. Recursable Dictionary Learning: Proof Outline

The main idea of our recursable dictionary learning algorithm is the following. The algorithm proceeds by iteratively examining each of the d_{sketch}/b blocks of each of the S received samples. For the ℓth block of the ith sample, denoted by $y^{(i)}[(\ell-1)b+1 : \ell b+1]$, it tries to determine whether it is dominated by the contribution of a single $(\sigma_s, \sigma_c, \sigma_m)$ (and is not composed of the superposition of more than one of these). To do so, it first normalizes this block by its ℓ_1-norm and checks if the result is close to a point on the Boolean hypercube in ℓ_∞ distance. If so, it rounds (the ℓ_1-normalized version of) this block to the hypercube; we here denote the resulting rounded vector by $\mathcal{R}(y^{(i)}[(\ell-1)b+1 : \ell b+1])$.

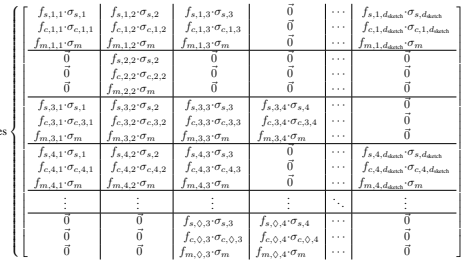

Figure 2. Example block-random matrix from distribution $\mathcal{D}(b, q, d_{\text{sketch}})$. $\Diamond := d_{\text{sketch}}/b$ denotes the number of blocks.

We then count the number of blocks $\ell' \in [d_{\text{sketch}}/b]$ whose rounding $\mathcal{R}(y^{(i)}[(\ell'-1)b+1 : \ell' b+1])$ is close in Hamming distance to $\mathcal{R}(y^{(i)}[(\ell-1)b+1 : \ell b+1])$. If there are at least $0.8qd_{\text{sketch}}$ of these, we add the rounded block $\mathcal{R}(y^{(i)}[(\ell-1)b+1 : \ell b+1])$ to our collection (if it's not close to an already added vector). Finally, we cluster the added blocks in terms of their matrix signatures σ_m in order to associate each of them to one of the matrices $R_1, \ldots, R_N$. Using the matrix signatures as well as the column signatures $\{\sigma_c\}$ allows us to recover all the essential columns of the original matrices. The absolute values of the x coordinates can also be inferred by adding the ℓ_1-norm of a block when adding its rounding to the collection. The signs of the x coordinates can also be inferred by first inferring the true sign of the block and comparing it to the sign of the vector whose rounding was added to the collection.

4.3. The Block-Sparse Distribution $\mathcal{D}$

We are now ready to define our distribution $\mathcal{D}$ on random (square) matrices R. It has the following parameters:

- The block size $b \in \mathbb{Z}^+$ which must be a multiple of 3 and at least $3\max(\lceil \log_2 N \rceil, \lceil \log_2 d_{\text{sketch}} \rceil + 3)$.
- The block sampling probability $q \in [0, 1]$; this is the probability that a block is nonzero.
- The number of rows/columns $d_{\text{sketch}} \in \mathbb{Z}^+$. It must be a multiple of b, as each column is made out of blocks.

Each column of our matrix is split into d_{sketch}/b blocks of size b. Each block contains three sub-blocks: a random string, the matrix signature, and the column signature. To specify the column signature, we define an encoding procedure Enc which maps two bits b_m, b_s and the column index into a $(b/3)$-length sequence. $\text{Enc} : \{\pm 1\}^2 \times [d_{\text{sketch}}] \to \{\pm \frac{1}{\sqrt{d_{\text{sketch}} q}}\}^{b/3}$, which is presented as Algorithm 1. These two bits encode the parity of the other two sub-blocks, which will aid our dictionary learning procedure in recovery of the correct signs. The sampling algorithm which describes our distribution is presented as Algorithm 2.

Algorithm 1 $\text{Enc}(j, b_m, b_s)$

Input: Column index $j \in [d_{\text{sketch}}]$, two bits $b_m, b_s \in \{\pm 1\}$, .
Output: A vector $\sigma_c \in \{\pm \frac{1}{\sqrt{d_{\text{sketch}} q}}\}^{b/3}$.
Set σ_c to be the (zero-one) binary representation of $j-1$ (a $\lceil \log_2 d_{\text{sketch}} \rceil$-dimensional vector).
Replace each zero in σ_c with -1 and each one in σ with $+1$.
Prepend σ_c with a $+1$, making it a $(\lceil \log_2 d_{\text{sketch}} \rceil + 1)$-dimensional vector.
Append σ_c with (b_m, b_s), making it a $(\lceil \log_2 d_{\text{sketch}} \rceil + 3)$-dimensional vector.
Append σ_c with enough $+1$ entries to make it a $(b/3)$-dimensional vector. Note that $b \geq 3(\lceil \log_2 d_{\text{sketch}} \rceil + 3)$.
Divide each entry of σ_c by $\sqrt{d_{\text{sketch}} q}$.
return σ_c.

Algorithm 2 $\mathcal{D}(b, q, d_{\text{sketch}})$

Input: Block size $b \in \mathbb{Z}^+$, block sampling probability $q \in [0, 1]$, number of rows/columns d_{sketch}.
Output: A matrix $R \in \mathbb{R}^{d_{\text{sketch}} \times d_{\text{sketch}}}$.
Initialize R to be the all-zeros $\mathbb{R}^{d_{\text{sketch}} \times d_{\text{sketch}}}$ matrix.
Sample a "matrix signature" vector σ_m from the uniform distribution over $\{\pm \frac{1}{\sqrt{d_{\text{sketch}} q}}\}^{b/3}$.
for column $j \in [d_{\text{sketch}}]$ **do**
  Sample a "random string" vector $\sigma_{s,j}$ from the uniform distribution over $\{\pm \frac{1}{\sqrt{d_{\text{sketch}} q}}\}^{b/3}$.
  for block $i \in [d_{\text{sketch}}/b]$ **do**
    Sample three coin flips $f_{m,i,j}, f_{s,i,j}, f_{c,i,j}$ as independent Rademacher random variables (i.e. uniform over $\{\pm 1\}$).
    Compute two bits $f'_{m,i,j} := f_{m,i,j} \cdot \text{sign}(\sigma_m[1])$ and $f'_{s,i,j} := f_{s,i,j} s \cdot \text{sign}(\sigma_{s,j}[1])$.
    Compute a "column signature" vector $\sigma_{c,i,j} := \text{Enc}(j, f'_{m,i,j}, f'_{s,i,j})$.
    Sample $\eta_{i,j}$ to be a (zero-one) Bernoulli random variable which is one with probability q.
    if $\eta_{i,j} = 1$ **then**
      Set $R[b(i-1)+1 : bi+1, j]$ to be $f_{s,i,j} \cdot \sigma_{s,j}$ concatenated with $f_{c,i,j} \cdot \sigma_{c,i,j}$ concatenated with $f_{m,i,j} \cdot \sigma_m$.
    end if
  end for
end for
return R.

References

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 39–48, 2016.

Arora, S., Bhaskara, A., Ge, R., and Ma, T. Provable bounds for learning some deep representations. In *International Conference on Machine Learning*, pp. 584–592, 2014a.

Arora, S., Bhaskara, A., Ge, R., and Ma, T. Provable bounds for learning some deep representations. In Xing, E. P. and Jebara, T. (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 584–592, Bejing, China, 22–24 Jun 2014b. PMLR. URL http://proceedings.mlr.press/v32/arora14.html.

Arora, S., Ge, R., and Moitra, A. New algorithms for learning incoherent and overcomplete dictionaries. In *Conference on Learning Theory*, pp. 779–806, 2014c.

Arora, S., Ge, R., Ma, T., and Moitra, A. Simple, efficient, and neural algorithms for sparse coding. In *Proceedings of The 28th Conference on Learning Theory*, pp. 113–149, 2015.

Azam, F. *Biologically inspired modular neural networks*. PhD thesis, Virginia Tech, 2000.

Brannon, E. M. and Roitman, J. D. Nonverbal representations of time and number in animals and human infants. 2003.

Buciluǎ, C., Caruana, R., and Niculescu-Mizil, A. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 535–541. ACM, 2006.

Clune, J., Mouret, J.-B., and Lipson, H. The evolutionary origins of modularity. *Proc. R. Soc. B*, 280(1755): 20122863, 2013.

Danihelka, I., Wayne, G., Uria, B., Kalchbrenner, N., and Graves, A. Associative long short-term memory. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pp. 1986–1994. JMLR.org, 2016.

Du, S. S., Lee, J. D., and Tian, Y. When is a convolutional filter easy to learn? *arXiv preprint arXiv:1709.06129*, 2017a.

Du, S. S., Lee, J. D., Tian, Y., Poczos, B., and Singh, A. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. *arXiv preprint arXiv:1712.00779*, 2017b.

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pp. 6059–6068, 2018.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

Gayler, R. W. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. *arXiv preprint cs/0412059*, 2004.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. Learning to transduce with unbounded memory. In *Advances in neural information processing systems*, pp. 1828–1836, 2015.

Hawkins, J. and Blakeslee, S. *On intelligence: How a new understanding of the brain will lead to the creation of truly intelligent machines*. Macmillan, 2007.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hinton, G. E., Ghahramani, Z., and Teh, Y. W. Learning to parse images. In *Advances in neural information processing systems*, pp. 463–469, 2000.

Hinton, G. E., Krizhevsky, A., and Wang, S. D. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pp. 44–51. Springer, 2011.

Hu, R., Andreas, J., Rohrbach, M., Darrell, T., and Saenko, K. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 804–813, 2017.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pp. 190–198, 2015.

Levy, S. D. and Gayler, R. Vector symbolic architectures: A new building material for artificial general intelligence. In *Proceedings of the 2008 Conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pp. 414–418. IOS Press, 2008.

Li, Y. and Yuan, Y. Convergence analysis of two-layer neural networks with ReLU activation. *arXiv preprint arXiv:1705.09886*, 2017.

Oh, J., Singh, S., Lee, H., and Kohli, P. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2661–2670. JMLR. org, 2017.

Papyan, V., Romano, Y., Sulam, J., and Elad, M. Theoretical foundations of deep learning via sparse representations: A multilayer sparse model and its connection to convolutional neural networks. *IEEE Signal Processing Magazine*, 35(4):72–89, 2018.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Sabour, S., Frosst, N., and Hinton, G. E. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pp. 3856–3866, 2017.

Smolensky, P. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216, 1990.

Spielman, D. A., Wang, H., and Wright, J. Exact recovery of sparsely-used dictionaries. In *Conference on Learning Theory*, pp. 37–1, 2012.

Sukhbaatar, S., Weston, J., Fergus, R., et al. End-to-end memory networks. In *Advances in neural information processing systems*, pp. 2440–2448, 2015.

Wu, J., Tenenbaum, J. B., and Kohli, P. Neural scene derendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. B. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Advances in Neural Information Processing Systems (NIPS)*, 2018.

Zaremba, W. and Sutskever, I. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

Zhang, Q., Panigrahy, R., Sachdeva, S., and Rahimi, A. Electron-proton dynamics in deep learning. *arXiv preprint arXiv:1702.00458*, 2017.

Zhong, K., Song, Z., and Dhillon, I. S. Learning non-overlapping convolutional neural networks with multiple kernels. *arXiv preprint arXiv:1711.03440*, 2017a.

Zhong, K., Song, Z., Jain, P., Bartlett, P. L., and Dhillon, I. S. Recovery guarantees for one-hidden-layer neural networks. *arXiv preprint arXiv:1706.03175*, 2017b.