Can simple assembly algorithms compute robust, high-dimensional means?

Kiran Vodrahalli (knv2109)

COMS 6998-06: Computation and the Brain, Fall '18

Contents

1 Introduction

The recently-introduced assembly framework of Papadimitriou and Vempala [2018] is an attempt at describing low-level operations of groups of neurons which are (a) biologically plausible, in the sense that there are actual experiments have validated these operations happen in real brains and which (b) are computationally feasible, both in the sense that they are robust operations and in the sense that they can be strung together to form simple programs which compute non-trivial calculations a real brain may be likely to calculate.

In this report, we initiate the study of whether it is possible to implement algorithms for robust statistics in the assembly framework, and in the process,

develop a rudimentary framework for handling linear algebraic operations purely with assembly machinery. One potential limitation of the assembly framework is that we assume that neurons are binary — either active or non-active — and thus assemblies themselves are represented as binary, sparse vectors. In this work, we make some progress in building a set of assembly primitives for handling real-valued input. As a bonus, we can interpret this framework in the context of how humans (actively reasoning, not subconsciously reasoning in their minds) may use assemblies to carry out mathematical operations and reason about linear algebra (though there are some pretty unrealistic things in our model). We also note that there is perhaps something a little strange about this problem: We already essentially have vector representations (though they are binary and sparse) in terms of assemblies; why can we not just use the sparse vector structure already existing to perform basic operations, rather than treating assemblies as blackboxes for concepts which adhere to a predefined assembly programming language? We try to resolve some of this tension by examining both approaches.

To determine whether the assembly framework can compute robust statistics, we examine the best existing algorithms for calculating robust statistics (biasing towards simple algorithms) and attempt to implement them in our linear algebraic assembly framework. To reduce the scope, we focus on robust mean estimation for a restricted class of distributions. The best algorithms for solving this problem can be expressed as semidefinite programs (SDPs) [Kothari and Steurer, 2017], and it is known [Kale, 2007] that we can efficiently approximately solve SDPs with the multiplicative weights (MW) algorithm. The MW algorithm has the additional advantage of being relatively simple to implement. Thus, we focus our goal as developing an assembly implementation of the linear algebra operations used by the MW algorithm.

Some caveats: It is perhaps a little crazy to think that the brain is actively solving SDPs for SOS algorithms in order to do robust mean estimation — for now, we defer the question of how crazy this is and see how far we can take the idea. The second question that comes up is that while the brain is robust to some forms of perception, it may not be robust in the adversarial sense that robust statistics takes as its definition. Therefore, we should investigate the ability of actual humans to do "robust estimation", and craft a definition of robustness compatible with experimental results. Then we can re-examine algorithms for solving this (perhaps simpler) problem.

2 Background

2.1 Assemblies and their Primary Operations

Here we will summarize the main ideas of Papadimitriou and Vempala [2018].

Definition 1. Brain areas.
We assume the brain is divided into a finite number of separate areas. We

assume that the synaptic connections between areas are part of the objects of interest.

Definition 2. An *assembly* is a group of neurons of size $k << n$, where n is the total number of neurons. In particular, the group of neurons at a given time step t is selected by a recurrent random projection and cap operation: A stimulus of k neurons fires in one area, and a random projection to another area occurs, increasing the dimension of the input. We can think of the activity of the neurons in the new space as a sum of the activity weights flowing in from the first area. Then a k-winner-take-all operation occurs, selecting only the top k active neurons to be active. This new set of neurons is fed back into the input with the stimulus in a recurrent manner, and this process continues for T timesteps, resulting in sequence of length T $(A_1, \cdots, A_T)$. This resulting process is an assembly, and we assume that the activity patterns over the A_t form a firing pattern. Here, $k = |\cup_t A_t|$ is the size of the support for the assembly A.

The first main set of technical results of Papadimitriou and Vempala [2018] are bounds on the size of the support of assemblies under certain parameter settings (mainly depending on k and the plasticity parameter of the synaptic connections (how easily they update). Essentially, what is shown is that assemblies created using the projection procedure will never overtake the whole brain — it is possible to distinctly separate out assemblies in the brain. They also give a bound on the size of the intersection between the resulting projections after k-cap given assemblies which initially overlap. Given an initial overlap, assemblies will continue to overlap, thus *preserving the overlap under random projections.* These technical results give a foundation for describing how assemblies can be formed as distinct entities in the brain.

The next group of results concerns the use of assemblies as a programming language. Papadimitriou and Vempala [2018] show that a certain group of operations over n neurons are able to simulate $\mathcal{O}(\sqrt{n})$-space computations as a Turing machine, which is an indication of the potential computational power of this language. They do this under the assumption that (a) any newly created assembly is a random set of $k \sim \sqrt{n}$ neurons in its area, (b) assemblies can interfere destructively only if they overlap in at least $\epsilon\sqrt{n}$ cells, and (c) synaptic weights fade with time. They show that assumption (b) is mild enough that new assemblies interfere is exponentially small in $\epsilon\sqrt{n}$. In this paper, we will be interested in the ability of this language to simulate efficient computations of robust statistics. The subset of the language we will use is as follows:

1. $\texttt{project}(x, A, y) : \boxed{x}_{\text{area}(x)} \rightarrow \boxed{y}_A$. This creates an assembly y in area A whose $\texttt{parent}(y) = x$. We make a note that we can make this operation permanent in memory with `permanent_project`.
2. $\texttt{merge}(x, y, A, z) : \boxed{x}_{\text{area}(x)} \rightarrow \boxed{z}_A \leftarrow \boxed{y}_{\text{area}(y)}$. This creates a new assembly in area A who has two parents, x and y.

3. `fade`(x) : This removes an assembly.

4. `activate`(x) : This activates assembly x for a few steps.

5. `enable`(A, B) : This allows synaptic connections to be formed between areas A and B.

2.2 Computationally Efficient Robust Statistics

Now we introduce the notion of robust statistics.

Definition 3. Outlier Robust Mean Estimation with parameter ϵ.
Let $x_1, \ldots, x_n \in \mathbb{R}^d$. Our goal is to estimate mean μ under the assumption that an adversary can perturb an ϵ-fraction of the data points.

This definition is extendable to estimating higher-order data moments. In particular, people care about this sort of problem because many algorithms in machine learning and statistics involve method-of-moment (MOM) approaches, which estimate high-dimensional moments as a part of the algorithm [Kothari and Steurer, 2017]. Kothari and Steurer [2017] give computationally inefficient methods for robustly estimating such moments for a certain (large) class of distributions, and via a modification of this distribution class, are able to convert the previously inefficient methods into efficient ones. The main technique by which they do this is the sum-of-squares (SOS) framework. The resulting quality of mean estimation is of order $\mathcal{O}(\epsilon^{1-1/d})$ for a large class of distributions, where d is the parameter controlling the number of moments we end up using to estimate robustly. The higher the d, the more computation we pay as well. As d gets large, we see that we get close to the optimal rate of $\mathcal{O}(\epsilon)$ for Gaussian data distributions, and get smaller for the information theoretically optimal rate of $\mathcal{O}(\sqrt{\epsilon})$ if we have a general data distribution. The algorithms in this paper attain an essentially optimal robust estimatino rate while still being "tractable", though in a somewhat weak sense given that their algorithms run in time $\mathcal{O}(n^d)$, where n is the dimension of the mean being estimated.

2.2.1 Sum of Squares

The aforementioned "efficient" algorithms are SDPs derived from the sum-of-squares framework. We explain the SOS framework here.

Definition 4. SOS framework.
Let $\mathcal{A} = \{f_1 \geq 0, \ldots, f_m \geq 0\}$ be a collection of polynomial constraints. We say that a degree$-\ell$ *sum-of-squares proof* that $p \geq 0$ given the constraints $\mathcal{A}$ consists of polynomials $\{q_S\}_{S \subseteq [m]}, \{r_t\}_{t=1}^T$ such that:

$$p = \sum_{S \subseteq [m]} q_S \prod_{i \in S} f_i + \sum_{t=1}^{T} r_t^2,$$

and for every subset S, $\deg(q_S \prod_{i \in S} f_i) \leq \ell$ and for all t $\deg(r_t) \leq \ell/2$. We write $\mathcal{A} \vdash_\ell \{p \geq 0\}$ (read this as: $\mathcal{A}$ implies a degree$-\ell$ SOS proof of the following inequality).

SOS satisfies an interesting duality property: It turns out that one can identify the existence of a valid, constraint-satisfying degree-ℓ pseudo-distribution (a linear operator $\tilde{\mathbb{E}}$ called the pseudo-expectation puts this in context: We must have $\tilde{\mathbb{E}}_\mu[1] = 1$ and ℓ^{th} pseudo-moment $\tilde{\mathbb{E}}_{x \sim \mu}[(1,x)^{\otimes \ell/2}(1,x)^{\otimes \ell/2^T}]$ must be PSD, where we view $(1,x)^{\otimes \ell/2}$ as a flattened vector, representing the requirement that the pseudo-expectation with respect to squared functions must be non-negative) with a degree-ℓ SOS proof. In particular, the pseudo-distribution certifies the proof.

The consequence of this duality property is that we can optimize over pseudo-distributions to find one that satisfies the SOS constraints (e.g., a feasible solution). If we took $\ell = \infty$, pseudo-distributions would always satisfy the PSD constraint, and thus would always be actual distributions. It turns out that there is no efficient separation oracle for the (convex set) of degree-ℓ moment tensors of an actual distribution, but there is an efficient separation oracle if we relax to the (convex set) of degree-ℓ pseudo-moments. This fact gives us an algorithmic advantage: our algorithms can now run in time $n^{\mathcal{O}(\ell)}$, where n is the dimension of the space the polynomials are defined over.

In particular, it is possible to solve this feasibility problem with an SDP: Recall the pseudo-moment matrix which we require to be PSD. The pseudo-moment matrix's PSDness forms our constraint over matrices. Then, the function which we care about is indeed linear in the entries of the matrix $M = (1,x)^{\otimes \ell/2}(1,x)^{\otimes \ell/2^T}$: We want to find weights W for each of these elements (e.g., $W \cdot M$) such that this is always $W \cdot M \geq 0$. Here we interpret $\cdot$ as an inner product in the vector sense.

2.2.2 Robust Mean Estimation

Recall that we have a dataset of points $\mathcal{D} = \{x_1, \cdots, x_m\}$. Our goal is to come up with a robust estimate of the mean μ of the distribution (robust if an adversary can perturb an ϵ-fraction of the data points). Now, what is the exact SOS algorithm we want to write down to solve robust mean estimation? We describe it here. The basic idea is to come up with two sets of constraints: one group implements a robust identifiability proof and the other implements a constraint on the moments of the distribution known as certifiable subgaussianity. Together, these constraints form a polynomial system of inequalities in terms of variables corresponding to a an alternative set of data points $\{x'_1, \cdots, x'_m\}$, binary selection variables $w_1, \cdots, w_m$ (for the robust identifiability proof), and variables associated with an SOS proof of certifiable subgaussianity inequalities which we require to hold. The robust mean estimation algorithm will find a feasible assignment to all of these variables subject to the constraints with an SDP, as described above. Then, we use the assignment of the x'_i to compute our robust estimate: $\hat{\mu}_{\text{robust}} := \frac{1}{m} \sum_{i=1}^m x'_i$.

We will fill in some of the details now.

Definition 5. Robust identifiability proof.
Let X be a typical sample drawn from the original data distribution D. Let Y be the data after corrupting an ϵ-fraction. Output a set of vectors $X' = \{x'_1, \ldots, x'_n\}$ such that:

1. $x'_i = x_i$ for $1 - \epsilon$ fraction of data
2. uniform distribution over over X' is contained in a specific class of distributions $\mathcal{C}$,

where in particular, we assume that $D \in \mathcal{C}$.

A robust identifiability proof corresponds to the notion that it is information theoretically possible to recover the correct parameter of interest from the corrupted data. A proof of this sort of identifiability can be turned into an efficient algorithm if we can get an SOS proof of identifiability. To ensure identifiability in our setting, we will require some assumptions on the moments (that we will shortly define). In particular, we will require these assumptions to be certifiable by SOS proof as well, so that we can proceed with the strategy outlined previously.

The key condition upon the moments that we require is certifiable subgaussianity:

Definition 6. (k, ℓ)-certifiable subgaussianity (CSG).
Let D be a distribution over $\mathbb{R}^d$ with mean μ. D is (k, ℓ)-CSG with parameter $c > 0$ if for every positive integer $k' \leq \frac{k}{2}$, there exists a degree-ℓ SoS proof of:

$$\forall u \in S^{d-1} : \mathop{\mathbb{E}}_{x \sim D} \left[\langle x - \mu, u \rangle^{2k'} \right] \leq \left(c \cdot k' \mathop{\mathbb{E}}_{x \sim D} \left[\langle x - \mu, u \rangle^2 \right] \right)^{k'}.$$

Note that many distributions satisfy this property: Poincaré and log concave distributions are notable examples, in addition to distributions built from simple constructions of other (k, ℓ)-CSG distributions.

The strategy to show this definition makes sense is as follows: under distribution-closeness requirement $\|X - Y\|_{TV} \leq \epsilon$ and certifiable subgaussianity, Kothari and Steurer [2017] shows we can estimate moments well, and each step of the proof is SOS certifiable (e.g., only uses inequalities and relationships that themselves have SOS proofs — this is what is meant by the fact that SOS is a proof system: You can compose inequalities under the same set of original assumptions with certain rules). We will not discuss these details in this report.

The key thing to understand from this setup is exactly what the constraints are (e.g., the SOS inequality versions of the robust identifiability proof and certifiable subgaussianity). For convenience, we write the constraint sets here (these are what go on the left-hand-side of all the SOS proofs):

Definition 7. Robust identifiability system.
Let $A_{Y,\epsilon}$ be the polynomial equations in $\mathbb{R}[w_1, \ldots, w_n]$. Then,

1. $w_i^2 = w_i$, for all $i \in [n]$
2. $\sum_{i=1}^n w_i = (1-\epsilon)$
3. $w_i(y_i - x_i') = 0$.

The first condition ensures that $w_i = 0, 1$, the second allows for an ϵ-fraction to be corrupted, and the third gives the corruption condition.

Definition 8. Certifiable subgaussianity system.
Let $B_{c,k,\ell}$ be the set of polynomial equations: for all $k' \in [k/2]$:

$$\frac{1}{n}\sum_{i=1}^n \langle x_i', u\rangle^{2k'} - \left(ck' \cdot \frac{1}{n}\sum_{i=1}^n \langle x_i', u\rangle^2\right)^{k'} = -\langle p_{k'}, (1,u)^{\otimes \ell}\rangle \left(\|u\|^2 - 1\right) - \left\|Q_{k'} \cdot (1,u)^{\otimes \ell/2}\right\|^2,$$

where:

1. $x_1', \ldots, x_n' \in \mathbb{R}^d$,
2. $p_1, \ldots, p_{k/2}$ are vectors,
3. $Q_1, \ldots, Q_{k/2}$ are matrices.

Note that here, u is a dummy variable which we are not actually solving for. This is just an efficient way of writing out all the inequalities we require: To get the list of all inequalities, we would expand out these equations and match terms of u with each other. Note that this system of constraints is exactly the SOS polynomial condition with respect to the certifiable subgaussianity constraint.

2.3 Simple Algorithms for Solving SDPs

Recall that our motivation is to implement robust mean estimation algorithms in the assemblies framework. So far, we have seen that robust mean estimation reduces to solving a certain SDP. We write the SDP as follows:

$$\begin{aligned} &\min 1 \\ \forall j \in [p] : A_j \cdot X &\geq b_j \\ X &\succeq 0 \end{aligned} \tag{1}$$

where X corresponds to the degree-ℓ moment variable in the SOS equation, and the A_j, b_j correspond to the various constraints imposed by our systems. We don't care about the objective since we just need a feasible solution.

It is possible to approximately solve SDPs with the multiplicative weights update algorithm [Kale, 2007, Gupta, 2011]. The generic algorithm is given below.

We will also need an oracle to find a feasible solution to a single constraint of the form of the constraints in the SDP (e.g., we need to be able to implement

Algorithm 1 Multiplicative Weights

Input: Finite set of p decisions to choose from.
Fix $\eta < 1/2$.
For each decision $i \in [p]$, associate $w_i^{(0)} = 1$.
for $t = 0, 1, \cdots, T$ **do**
    Choose decision according to the discrete weight distribution $w^{(t)}$.
    Observe cost of all decisions $m^{(t)}$.
    Update: $w_i^{(t+1)} = w_i^{(t)}(1 - \eta m_i^{(t)})$
end for
return $\frac{1}{T}\sum_{t=1}^{T} w^{(t)}$

an algorithm which can find a PSD solution Y to a constraint $M \cdot X \geq c$, for some constraint matrix M and some bound c (again, $\cdot$ is dot product treating M, X as vectors). How do we pick the oracle? For SDPs, the answer is simple: We merely need an oracle to check whether $\lambda_{\max}(M) \geq c$, and if it is, to return vv^T, where v is the top eigenvector of M. Together with this oracle, we can use multiplicative weights to solve the feasible SDP problem is as follows:

1. Identify an expert (decision) with each constraint of our SDP.
2. Each round, we produce a vector $w^{(t)} \in \mathbb{R}^p$ which gives us a convex combination over the constraints.
3. Use the oracle to find feasible solution $X^{(t)}$ for $M = \sum_{i=1}^{p} w_i^{(t)} A_i$ and $c = {w^{(t)}}^T b$. If not feasible, exit with certificate $w^{(t)}$ of non-feasibility.
4. Return cost $m_i^{(t)} := \lambda_{\max}(A_i) - c_i$ for all i and use the multiplicative weights update on w. Iterate previous three steps for T rounds.
5. Return $\frac{1}{T}\sum_{t=1}^{T} X^{(t)}$ as solution.

We set the cost this way since we think of positive values of $\lambda_{\max}(A_i) - c_i$ as having over-satisfied that particular constraint. We are working with a distribution over the constraints, and are essentially trying to eventually zero in on the hardest constraints, so we can focus on satisfying those constraints well [Gupta, 2011]. If the cost is negative, we failed that constraint and need to allocate more weight to it in the next round.

To implement this algorithm for finding a feasible solution to an SDP, we need the following operations:

1. Representation of inputs: real-valued scalars, vectors, matrices
2. Addition and subtraction (scalars, vectors)
3. Multiplication (scalars)
4. Top eigenvector/value computation of a matrix (for the oracle)

5. Control flow of program (variable definitions, for loops, etc.)

6. Sampling from discrete distribution

Notably, these are relatively simple operations that we can hope to implement with assemblies! In this report, we will leave out sampling (for future work) and control flow (this already seems to be relatively established in Papadimitriou and Vempala [2018]) and focus on the linear algebraic operations.

For the rest of this report, we will simply try to implement some of these operations with assemblies, and understand that if we can implement all the operations with assemblies, we can run multiplicative weights on the SDP defined by the SOS system defined by the robust identifiability and certifiable subgaussianity constraints.

3 Linear Algebra for Assemblies

We now present our framework for doing linear algebra with assemblies. Throughout, we will think of assemblies as k-sparse binary vectors with the ability to project and merge, as defined previously. First, we will figure out how to represent real-valued scalars and the operations of addition, subtraction, and multiplication using assemblies and the project operation.

3.1 Scalar Addition and Subtraction via Project

First we will do something slightly insane, and decide to identify a discretized set of bounded real values (along with ∞, ± 1, and 0) each with its own assembly (k-sparse binary vector). Then, we will connect these assemblies in an appropriate way via project operations to simulate the discretized real number line as a list. This number line will be exclusively non-negative, we will use the -1 assembly to deal with negative numbers. Concretely, suppose we will maintain real numbers between $[0, B]$ up to accuracy δ. Then we will have $B/\delta + 3$ assemblies: for all $i \in [B]$: $\boxed{i \cdot \delta}_{R_i}$, $\boxed{-1}_{R_{-1}}$, $\boxed{0}_{R_0}$, and $\boxed{\infty}_{R_\infty}$. We will create these assemblies each in their own brain area (which will be quite small), and we will create them permanently using `project_permanent` from a (large) assembly we will call r for "real numbers" which lives in area R ($\boxed{r}_R$). So to clarify, the structure so far is we have an assembly r which will permanent-project to all our real numbers. Specifically, we run `permanent_project`$(r, R_i, i \cdot \delta)$ and similarly `permanent_project`$(r, R_{-1}, -1)$, `permanent_project`$(r, R_0, 0)$,

and `permanent_project`(r, R_∞, ∞). Let us call these assemblies the "platonic concept of the real numbers". To actually work with the real numbers, we will need to make copies via a projection from the platonic concepts, and we will work with them when actually manipulating specific numbers (given by input, for instance).

We will now specify how to do addition. Subtraction will be essentially the same algorithm, except we will have to do a little casework and use the $\boxed{-1}_R$ assembly.

Basically, we will set up two projection chains for the $\{\boxed{i \cdot \delta}_i\}_{i \in [B]}$ assemblies, one going forwards and one going backwards (think of these like linked lists, where the links will occur due to synaptic projections. We will run for L_F `permanent_project`$(i \cdot \delta, R_{i+1}, (i+1) \cdot \delta$ for all $i \in [0, \cdots, B-1]$ (forward list).

Similarly, we will run for L_B: `permanent_project`$(i \cdot \delta, R_{i-1}, (i-1) \cdot \delta$ for all $i \in [1, \cdots, B]$ (backward list). Now we are ready to add.

Suppose we want to add $a + b$. First we figure out which assemblies contain the real values a and b and identify them with those assemblies. For now assume $a, b \geq 0$. First, we need to move to a copy brain areas C_1, C_2. Create two pointer assemblies p_a and p_b in C_1 who project to a and b initially: Do `project`$(p_a, R_i, a = i \cdot \delta)$ and `project`$(p_b, R_j, b = j \cdot \delta)$. We will also store copies a', b': Do

1. `project`(a, C_2, a')
2. `project`(b, C_2, b'), and
3. `project`(a', C_1, p_a),
4. `project`(b', C_1, p_b).

This way, we'll be able to remember which pointer is which! Next, calculate $\min(a, b)$ and $\max(a, b)$. We do this as follows: Using both pointers, we will trigger the synaptic connections along L_B one time step at a time, first activating the synaptic connection at p_a, then activating it at p_b. After you `activate`(p_a), check if $\boxed{0}_{R_0}$ is active or not. If it is first active when you trigger the a-chain, stop — we know a is min and b is max. Otherwise, apply `fade`(p_a). Then trigger the assembly in the b-chain and check if $\boxed{0}_{R_0}$ is active or not. If it is we're done again; otherwise, apply `fade`(p_b). Now we need to step with both p_a and p_b. Before applying `fade`(p_a), check the assembly it activated (call it x), and now `activate`(x) – check the resulting assembly (y), and while it's still active, do a `fade`(p_a) to remove the old connections. Finally, do `project`$(p_a, R_k, y = k \cdot \delta)$ and we've updated the pointer. We do an exactly analogous thing for p_B. Now, assume one of the pointers has activated $\boxed{0}_{R_0}$. WLOG let it be p_a. Recover $a' =$ `parent`(p_a), and $a =$ `parent`(a'), and re-initialize p_a by doing `activate`(a) (as described before). Do likewise for p_b, except this time, use the L_F list for b instead of the L_B list. We carry out the same process for p_a (getting to 0), but this time, each time we step towards 0 with p_a, we step away from 0 with p_B. When we activate 0 with p_a, we stop at p_B, and after creating a new assembly in C_2 called $a+b$, do `project`$(a+b, R_\ell, a+b = \ell \cdot \delta)$. This allows us to calculate sums of positive numbers. One note: If p_b activates the synapse and nothing else gets activated, then it means we need to build on our platonic concept some more (e.g., extend the size of B). This is easy: We just permanent project as before when we originally constructed the platonic concept of the reals.

For subtraction, we do something almost identical. The first key difference is that a' and b' are defined a little bit differently depending on whether we do $a - b$ or $b - a$, assuming $a, b \geq 0$. For whichever term is negative (WLOG say

a), we also do $\texttt{project}(-1, C_2, a')$. Another difference is we instead calculate $\min(|a|, |b|)$ and $\max(|a|, |b|)$ in the same way as for addition. After we figure out which is larger, we check whether $\boxed{-1}_{R_{-1}}$ is a parent of the larger absolute value. If it is not a parent of the larger value, we do the same algorithm as before except we start at the larger value (which is positive), and instead of using L_F as in positive addition, we just use L_B. If $\boxed{-1}_{R_{-1}}$ is a parent of the larger absolute value, then we do the same thing, except at the end, we do a `merge` of the output with $\boxed{-1}_{R_{-1}}$ (incidentally, this is how we will represent negative numbers).

We can now relax the condition that $a, b \geq 0$, and assume we can have negative values too (just check if they're merged with $\boxed{-1}_{R_{-1}}$ to see how to treat them). The last remaining case, $-a - b$, is analogous to $a + b$.

There are also some uninteresting edge cases with 0 and ∞. In these special cases, you do the obvious thing ($0 + x = 0, \infty + x = \infty$).

3.2 Scalar Multiplication

Let us calculate $a * b$. Again we identify them with the appropriate assemblies. First we will quickly describe how to handle negative signs. Basically, just check each of the inputs to see if it has a negative sign assembly downstream. If both do, copy the inputs to an assembly which is not merged to a negative sign (or just de-merge it). If both do not, do not do anything. If one does, de-merge it but create a new temporary assembly which points to -1. After the rest of the computation is finished, we will merge it at the end. From now on, we suppose $a, b > 0$ (0 and ∞ are again trivial cases, as are 1 and -1).

The key idea for scalar multiplication is as an area estimate using the δ-net on the non-negative reals. What we can do is estimate the number of tiles in the cross product of L_F and a copy of L_F. For simplicity, we will call these L_F^x and L_F^y, to denote and x-dimension and a y-dimension. If we can count the number of tiles with assemblies, our estimate of the area will be $\#\text{tiles} \cdot \delta^2$. Therefore, we will in particular identify the value of $a * b$ with $\#\text{tiles} \cdot \delta$ on L_F, since each term is already a factor of δ.

Thus there are two questions: (1) How do we count the number of tiles with assemblies and (2) How do we find the right assembly to identify $\#\text{tiles} \cdot \delta$ with? Our method must answer both simultaneously: We have to count the tiles *with respect to* L_F from the very beginning, so that we can correctly identify the assembly it maps to. The method will be analogous to the previous section. We will again have two pointers tracing along different paths in different areas, but now, one of the pointers will be on a "winding-square" path while the other pointer will start at 0 on L_F and keep moving forward. We will have to construct the winding-square path as distinct from L_F and L_B. After it is constructed, we will use almost the same strategy as for addition — it will be constructed so that the winding square path will tabulate all tiles just as it reaches $\boxed{0}_{R_0}$, and thus, we can check if 0 is active to stop the other pointer at the #tiles-assembly on L_F. Thus we have reduced the problem to simplifying multiplying

by one, system-wide number: δ. First we fill in the rest of the details of the winding-square path.

3.2.1 Construction of the Winding-Square Path

Think of a square of dimensions $a \times b$ (subset of $L_F^x \times L_F^y$), with the bottom left corner containing $\boxed{0}_{R_0}$. The winding square path will start at $(0, b)$, move to (a, b), down to $(a, b - \delta)$, right to $(0, b - \delta)$, and so on, tracing out every single assembly in the intersection until it reaches $(0, 0)$, which corresponds to $\boxed{0}_{R_0}$. We can get away with building exactly one of these squares, once at the beginning along with L_F and L_B (projected to as usual from r). In building this square, each vertical real-value that we store will have an orientation: We basically will alternate horizontal copies of L_F^x and L_B^x from the boundary B until we reach the 0 level. We only have copies of L_B vertically from 0 until we reach the boundary B, since on the y-axis, we are always heading down from B to 0. In this way we build the whole grid.

3.2.2 Control Flow on Winding-Square Path

There is another detail before we are finished calculating the number of tiles (in exactly the same way as for addition). Given that we only build one winding square, when we calculate $a * b$, we have to make sure to store special assemblies for a and b (like a' and b', which do not change). This is because each time we take a horizontal step (either forward or backward) on any of the levels of the winding square, we need to check whether the resulting assembly which gets activated is (WLOG) a, and in the vertical case, (WLOG) b. This is exactly analogous to the terminal condition of checking whether we have hit 0 yet or not. Basically, we need to be able to select the next assembly on the L^y axis after we complete a row of the L^x axis. Therefore, we have a control flow statement like "if `activate`(x) activates `parent`(a'), activate next turn on L^y synapse instead of on the current L^x synapse".

3.2.3 The $1/\delta$ Factor

Now that we have number of tiles, we need to multiply it by δ to get to the right assembly on L_F. How can we do this? We think of δ as a system-wide constant for now, and set aside a special procedure for multiplying by δ. How do we do this? Since δ is known, we mark out the assembly which corresponds to it — this is precisely the first non-zero assembly. Then, we also look at the assembly corresponding to 1. We can easily count the number of δ-steps it takes to get the the 1-assembly, this number is $1/\delta$. As we counted it out, we (by using parallel pointers) were also able to identify the assembly corresponding to $1/\delta$, which we have a dummy variable ($1/\delta$-pointer) project to. We basically want to divide $\#tiles$ by $1/\delta$.

How do we do this? Again, we use the parallel pointer trick we have been using throughout. What we will do is again have two pointers, this time on L_B

and a copy of L_B that we will call L'_B. The idea is that the L_B pointer will start at 1 and go down to δ, and the other pointer (L'_B) will start at $\#tiles$, and move much more quickly per step (L'_B moves $1/\delta$ tiles per step of the L_B pointer). When the L_B pointer reaches δ-assembly (in the same manner as usual), the L'_B pointer also stops, and we have successfully divided by $1/\delta$. The resulting assembly is then stored as usual in a new assembly pointing to the platonic assembly.

The last remaining issue is to spell out how we move by $1/\delta$ steps at a time. The answer to this is yet another copy of L_B, and our stored $(1/\delta)$-pointer. Just like in the calculation of the $\min(a, b)$ we saw in the addition section, we will step in L'_B as long as the $(1/\delta)$-pointer does not trigger the 0-assembly. Once it triggers the 0-assembly, we stop moving in L'_B for that step, and re-set the $(1/\delta)$-pointer (by having previously stored a $(1/\delta)'$-assembly).

3.3 Tensor Products via Merge

Tensor products are remarkably simple compared to the scalar operations in assemblies. This is because we can use merge to create vectors easily (and higher order tensor products). Recall that a tensor product of $a, b \in \mathbb{R}^n$ is just an association with $a \otimes b$, which is a collection of all possible $a_i b_j$. Therefore, just calculate all $a_i b_j$ in their own separate scalar assemblies, and then merge them all in a new assembly (e.g., $\texttt{merge}(\{a_i b_j\}_{i,j}, C_3$ where C_3 is some new area of the brain). In this manner, we can do any kind of tensor product now that we have scalar multiplication. Note that to represent a vector in $\mathbb{R}^n$, you do n merge operations of scalar assemblies. To do a d-order tensor (let's say it's symmetric for simplicity) based on a vector in $\mathbb{R}^n$, you need d^n merges.

3.4 Top Eigenvector Computation

A simple, well-known algorithm to compute the top eigenvector of a matrix A is the power method: Take a random vector x on the unit sphere and keep multiplying the matrix by the vector over and over again: $A^k x$. For k large enough, $A^k x$ will converge to the top eigenvector. One can do a simple matrix-vector multiplication to get the top eigenvalue. Thus, we only need to be able to implement matrix multiplication with our setup. Fortunately, this is now easy since we have all the necessary tools. Inner products are just sums of products of two vectors (so just do the elementwise products of two vectors a, b as a new vector by doing scalar multiplication in each dimension, and then adding them all up gradually (like a `foldl` operation from functional programming), accumulating the sum in a given assembly and then continually adding to it). Matrix-vector multiplication is just the tensor product of a bunch of inner products (which map to scalars), so we use merge to create the new tensor. Throughout all these procedures, we are sensible and discard any temporary assemblies which we don't need using operations like `fade`.

4 Conclusion and Future Work

In this work, we identified an interesting question of whether or not it is possible to compute robust statistics like the mean under an ϵ-adversarial corruption of the data using the primitives of the assembly model of the brain due to Papadimitriou and Vempala [2018]. We gave the background on assemblies and robust statistics, and identified multiplicative weights as a simple, underlying core algorithm for calculating a robust mean. We then came up with a rudimentary framework for doing real-valued linear algebra with assemblies, which we hope will be useful in successive works which try to figure out how to implement various other algorithms in the assembly framework.

The key aspect we left out in this work is an assemblies implementation of sampling. We leave this to future work to complete the story on robust SDP implementations in the assemblies framework. It would also be interesting to calculate exactly how many operations end up happening in just building the SDP — it is not super easy to tell from the equation constraints, but should be possible to check – roughly it is of the order $n^{2\ell}$ different constraints if the data comes from $\mathbb{R}^n$. Then we need to store all the matrices, which is another factor of n^2 for each of these terms. This ends up being a huge number of terms. It would be nice to figure out other ways to use the intrinsic vector-space properties of k-sparse binary vectors to represent vector and other tensor data. The difficulty is both in the fact that these vectors are sparse and that they are binary. Additionally, we want to be able to use other operations not defined in Papadimitriou and Vempala [2018] more freely, perhaps for instance taking advantage of the fact that we have real-valued weights in synapses for more than just proving things about the dynamics of the projection process (e.g., actually use in the framework as well — but, it seems very unrealistic to think we can program individual weights to specific values). Finally it makes a lot of sense to try to better characterize what the dynamic patterns in the assemblies look like (rather than just using a core support as in Papadimitriou and Vempala [2018], we could potentially encode more information in the dynamics of a single assembly).

Another missing point is the fact that the properties of `merge` have not yet been studied, and yet we used it to define tensor products in this paper. More theoretical understanding of how `merge` is carried out is necessary going forward.

Another thing to check is how realistic it is to be able to solve SOS programs with the MW approach, given that it is not as precise as interior point methods. How do these errors trade off with the $\epsilon^{1-1/d}$ rate achieved by the SOS algorithm, assuming it is solved to precision? The discretization of the brain is another worrisome issue.

It would also be interesting to see if anything remotely like this setup is what happens in practice when someone is trying to learn linear algebra in real life. It seems a little plausible, but there are lots of incredibly unrealistic aspects of the computation model defined in this report. In a similar vein, it would be interesting to figure out if humans are even capable of calculating

high-dimensional means of any kind (for instance, perhaps humans can calculate robust means of images, if shown a bunch of images in a row from a distribution centered around a certain image).

Finally, we would like to finish by noting the broad research program that opens up when we consider trying to implement various existing algorithms in the assembly framework. In the past, "neurally plausible" algorithms have traditionally referred to algorithms implementable by neural networks [Arora et al., 2015]. While assemblies are certainly somewhat related to deep networks (the random projection and cap are similar in flavor to one layer of a deep net), they also provide a more formal framework for reasoning about what is going on at the neuronal level, and are which perhaps have more experimental evidence for at least some of their operations. In particular, it will be interesting to find algorithms which are implementable *naturally* by assemblies, which do not rely on kludgy formulations and huge numbers of assemblies to work (in particular, it will be helpful to keep in mind the expected number of assemblies existing in various parts of the human brain to see if they are reasonable).

References

Sanjeev Arora, Rong Ge, Tengyu Ma, and Ankur Moitra. Simple, efficient, and neural algorithms for sparse coding. *Proceedings of the 28th Conference on Learning Theory, Proceedings of Machine Learning Research*, 40, 2015.

Anupam Gupta. Solving lps/sdps using multiplicative weights, 2011. URL `https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f11/www/notes/lecture17.pdf`.

Satyen Kale. *Efficient Algorithms using the Multiplicative Weights Update Method*. PhD thesis, Princeton, 2007. URL `http://www.satyenkale.com/papers/thesis.pdf`.

Pravesh Kothari and David Steurer. Outlier-robust moment-estimation via sum-of-squares. 2017.

Christos H. Papadimitriou and Santosh S. Vempala. Random projection in the brain and computation with assemblies of neurons. *10th Innovations in Theoretical Computer Science Conference*, pages 55:1–55:19, 2018. URL `https://www.cc.gatech.edu/ vempala/papers/assemblies.pdf`.