Scalable, Distributed Factor Analysis in Spark with Allen Brain Observatory Data

Danielle Rager
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, Pennsylvania
drager@andrew.cmu.edu

Daoyuan Jia
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, Pennsylvania
daoyuanj@andrew.cmu.edu

Tiancheng Liu
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, Pennsylvania
tianchel@andrew.cmu.edu

ABSTRACT

We develop a scalable, distributed implementation of Factor Analysis in Spark intended as a tool to assist the neuroscience community in analyzing large Allen Observatory brain data. Factor Analysis is a dimentionality reduction technique commonly used in scientific research, especially amongst neuroscientists to analyze the covariability of neural responses in an observed network. We take an existing dsitributed implementation of a related dimensionality reduction technique, leverage some of its optimizations for solving Gaussian-Linear process EM algorithms efficiently, and develop additional optimizations specific to the model formulation of factor analysis. We also conduct experiments on our Spark FA algorithm to test its effectiveness and efficiency, using both simulated data and large neuroscience data. Our implementation is able to explain more variability in neuroscience data than existing distributed dimensionality reduction algorithms while achieving a similar time complexity.

CCS Concepts

•Mathematics of computing → Dimensionality reduction; •Theory of computation → MapReduce algorithms; Vector / streaming algorithms; •Computing methodologies → Factor analysis; Principal component analysis;

Keywords

Principal component analysis; factor analysis; stochastic principal component analysis; probabilistic principal component analysis; dimension reduction

1. INTRODUCTION

The Allen Brain Observatory data, released in July 2016, is now the largest open source neuroscience dataset.[1] It contains two-photon calcium imaging recordings of over 18,000 neurons from four areas of mouse visual cortex in response to an array of visual stimuli, which include static gratings, natural scenes, and natural movies. Researchers began collecting data for the current public release, which includes over 30 terabytes of information, in 2012, and the Allen Institute has reported plans to grow the size of the existing database at a similar rate over the next several years. The neuroscience community is hoping to use such a large (and expanding) dataset to build comprehensive, generalizable models of visual processing and cortical computation. However, most theoretical neuroscience labs currently lack the computational tools necessary to efficiently process data at this scale.

One area of active research in the neuroscience community is the analysis of "noise" in the brain – the electrical signals produced by neurons are highly variable, even when an identical stimulus is presented to a neural network across trials. Understanding trial-to-trial covariability amongst a neural population is important for understanding the purpose of the noise, e.g. its effect on the network's ability to encode stimuli. Recent research has shown that the low-rank structure of noise covariance for a network of neurons can change based on brain region, cell type, and the behavioral task that the animal is completing, but a systematic theory of noise covariance in the brain has yet to be developed. Dimensionality reduction techniques can be used to understand the low-rank structure of noise in a network.

The broad aim of our proposed project is to efficiently analyze the low-rank structure of the noise across many combinations of recordings from the Allen Brain Observatory data. Noise covariance matrices quickly become too large to be stored in memory as the dimension of the analyzed network increases, because a network of N neurons results in a covariance matrix with N^2 entries. Additionally, computing a covariance matrix explicity is computationally expensive. We therefore sought to develop a distributed implementation of a dimensionality reduction technique that did not require the explicit computation of a large covariance matrix. Furthermore, each neuron in a recorded population has a private source of variability that is roughly proportional to the neuron's mean firing rate. Neuroscientists must be careful to account for such private neural variability when estimating the co-variability of a neural population. Factor analysis is a dimensionality reduction technique in which the private variability of each feature (neuron) is estimated simulataneously with the network covariability, making it the preferred dimensionality reduction technique among neuroscientists. However, no publicly-available, distributed implementation of factor analysis currently exists.

Our project introduces sFA, a distributed version of factor analysis implemented in Spark. sFA is motivated by the desire to analyze the noise covariance of large populations of neural activity from the Allen Observatory dataset. Our sFA algorithm implementation was inspired by a distributed version of pPCA in Spark, called sPCA

[1] http://observatory.brain-map.org/visualcoding/

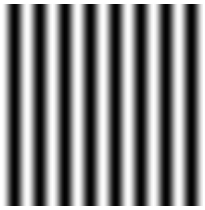

Figure 1: A Static Grating Stimulus

[6].

In the subsequent sections, we will first discuss, in greater detail, the scientific questions from the visual neuroscience community that motivate our desire to perform dimensionality reduction on neural data. We will then describe the relevant details of the Allen Observatory data collection, including the storage format of their data, and discuss our use of the Allen SDK to retrieve and pre-process the relevant time series for thousands of simultaneously recorded neurons. In Section 3, we will summarize the dimensionality reduction techniques that currently have Spark implementations and highlight the statistical differences between these algorithms and our favored dimensionality reduction technique, factor analysis. We provide tests of local, Python implementations of these various dimensionality reduction techniques to show that factor analysis is capable of finding low-rank structure in data that has non-isotropic noise. Finally, in Section 4, we will introduce sFA, our distributed implementation of factor analysis for Spark. We will discuss sFA's optimizations and test its accuracy on simulated data of known rank. We will then test sFA's efficiency on real neural data from the Allen Observatory dataset.

2. ALLEN OBSERVATORY DATA

2.1 Data of Interest

The Allen Observatory data consists of two-photon calcium imaging recordings of several visual-processing regions of mouse cortex. Two-photon calcium imaging recordings consist of time series of calcium-triggered fluorescence emissions from genetically modified mice. Neurons have large influxes of calcium as the emit action potentials, the most basic electrical communication unit of the nervous system. Therefore, a change in the fluorescence emissions of a genetically modified neuron corresponds to a change in the calcium influx of that neuron, which corresponds to the emission of an action potential. The change in the fluorescence of a neuron is quantified as DF/F, or the instantaneous fluorescence emission compared to the mean fluorescence emission windowed over recent observations.

Allen Observatory provides us with recordings of fluorescence level they have collected from experiments they conducted recently. The experiments contain multiple trials of showing different sets of stimuli to mice expressing genetically encoded calcium sensors. In each trial, the fluorescence level in each of the thousands of cells in target brain areas is recorded over a period of time as raw data. Experiment metadata describing mouse, stimuli, imaging equipment, and other experiment conditions is also included as part of the data.

Raw data from the experiments only consists of absolute fluorescence trace. Allen Observatory [3] suggests that contamination by the fluorescence of the neuropil immediately above and below the target cell requires adjustment on the original fluorescence trace before calculating the DF/F series in order to more accurately reflect electrical activities in the cell.

For example, Figure 1 is one of the static grating stimuli that will

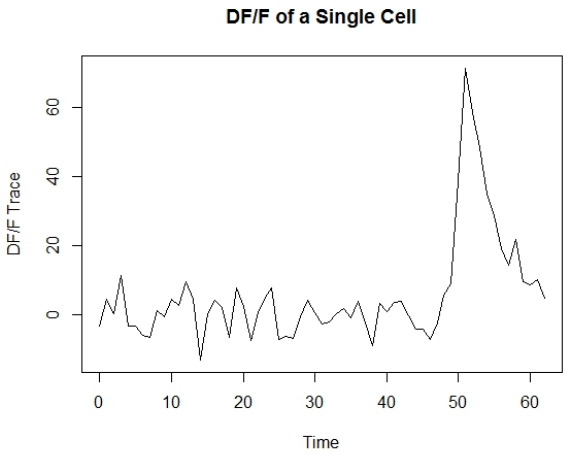

Figure 2: DF/F of a Cell

be shown to the mouse. Stimuli vary across several kinds and have parameters in each kind. Within the scope of this project, we only randomly choose one particular static grating stimulus and examine brain activities in a certain region. Figure 2 shows an example of a DF/F trace in our data set. As one can easily tell, there is a great spike around time 50, indicating that this cell has extensive electrical activities at this time when shown the stimulus. While this graph clearly shows a spike, most DF/F traces are fairly flat probably because most cells are only sensitive to a very narrow set of stimuli.

We are particularly interested in analyzing the correlation of noise of brain activities between neurons; an intuitive conjecture might be that neurons that are close to each other in a brain region should have similar functions and therefore should have similar behaviors to a particular stimulus. Hence, the core data we are looking for is the corrected DF/F trace from a number of cells in several trials.

2.2 Data Retrieval and Storage

The Allen Brain Observatory data is available online through a Python API called AllenSDK for accessing and manipulating the data. The experiment data is stored in the Neurodata Without Borders (`NWB`) format (http://www.nwb.org/), a new community standard for sharing large-scale electrophysiology data. Features in `NWB` are described in a `JSON`-like syntax that specifies the relevant data features and their storage location within an `HDF5` file (https://support.hdfgroup.org/HDF5/), a format that allows for efficient input/output of high volume data with complex hierarchical structure. The AllenSDK package not only allows us to access the experiment data using specific parameters and filters, but also provides python functions to retrieve pre-processed data such as the DF/F series for convenience.

Allen Institute released notes on how to use the Python API to access the data either by experiment and cell [1] or by stimulus [2]. In either case, the API employs a cache on disk by default to store metadata as well as the final `NWB` file. The largest structure of the database is experiment containers which store id's of individual experiments and are divided according to target brain regions. Once the target brain region is selected, a list of experiment containers is returned.

In each experiment container, all related brain cells are recorded and can be filtered according to cell characteristics such as orienta-

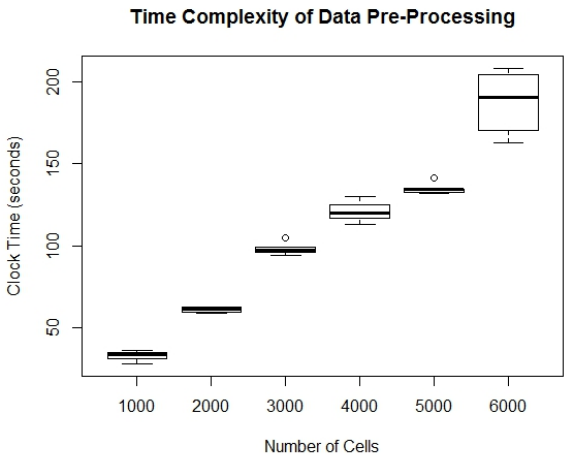

Figure 3: Time Complexity of Data Pre-Processing

tion selectivity index, direction selectivity index, peak dF/F, etc. Once the target cells are selected, the experiment data metadata can be identified and used to initiate downloading of the actual `NWB` file. If the cache option is chosen, the `NWB` file, each around 500 MB, will be stored on disk. An `NWB` file usually contains data for experiments that are closely related.

From the `NWB` file cached on disk, we are able to easily retrieve the "sweep response" recorded in a particular experiment. In the experiment that we randomly choose, the stimulus is shown to the mouse 75 times and each time the stimulus is shown, fluorescent traces for all target cells are recorded over 63 time stamps. In the experiment container that we obtained, activities of 6000 cells have been recorded and we get DF/F traces by applying the built-in DF/F calculation function. Therefore, a 3-D matrix of DF/F values is obtained for as many as 6000 cells in 75 trials of the same stimulus at 63 time stamps.

2.3 Data Pre-Processing

Once we obtain the DF/F traces, we need to subtract the mean across trials from each cell, in order to get the residual of each cell and analyze them across cells in each trial. Using Numpy functions, it is easy to get the mean at each time stamp across trials and subtract it from the original data. Now we are left with a 3-D matrix with the same dimension.

Neuroscientists try to analyze the noise between cells at the same time; therefore, we can only compare residuals in the same trial. To be less biased by one single trial, an empirical method is to concatenate time series one directly after another, assuming no inter-temporal relationship (auto-regression) in the traces. By doing so, data is no longer in a time series manner; each data point is treated independent across time. This step can be seen as flattening the dimension of trial via concatenation, and we are now left with a 2-D matrix which stores DF/F values of 6000 cells at 4662 observations.

An experiment is conducted to analyze the time complexity of data pre-processing with respect of number of cells specified. The experiment contains trials of data pre-processing of 1000 to 6000 cells in increment of 1000, 5 trials each. Shown in the box plot (Figure 3), the time complexity of pre-processing the data is linear to number of cells used.

We also notice that since each the operation for each cell is independent of others, the task can be easily distributed row-wise. Moreover, we have also come across thoughts regarding storage, retrieval, and pre-processing large, or even distributed scientific data, most beyond the scope of this project. These ideas are discussed in the last section in detail to further optimize the data pipeline for the computational neuroscience community and even the general public.

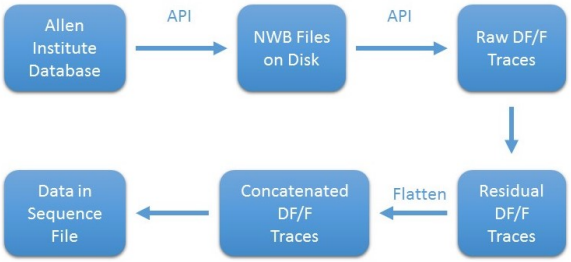

Figure 4: Summary of Data Workflow

Inspired by the sPCA implementation [5], we decide to transform our data, which is originally text files, to Hadoop SequenceFile format, using the script provided by Elgamal's sPCA implementation. Now, the sequence file is ready to be fed into our main program.

To summarize all of our data work, we provide Figure 4 as a visualization of the workflow.

3. RELATED WORK

One method of dimensionality reduction that has commonly been used to analyze low-dimensional structure in neural data is principle component analysis (PCA). In principal component analysis, all variability from an observed data matrix Y_{NxD}, where N is the total number of observations and D is the dimensionality of the data, is mapped to a lower-dimensional, orthonormal representation of the co-variability. The PCA model for a single observation vector y is therefore defined as:

$$y = C * x + \mu \tag{1}$$

where C is the Dxd transformation matrix that projects vector x with mean μ into a $d < D$ dimensional space. If we set $d = D$, it can be demonstrated that the orthonormal basis defined by C is equivalent to the eigendecomposition of the observed data. Note that the form of the PCA model is incapable of capturing any variability that is private to each of the D dimensions of the original data.

Moreover, the time complexity of the standard PCA algorithms is quite high. If PCA is performed using a standard eigendecomposition of the data covariance matrix, the complexity is $O(D^2N + D^3)$; the covariance computation has $O(D^2N)$ complexity, and the eigendecomposition has $O(D^3)$ complexity. The most commonly used algorithm for PCA analysis with Spark is currently implemented by the `MMLib` library, which computes a dsitributed implementation of an eigendecomposition of the covariance. Figure 5, replicated from Elgamal et. al, shows the complexity metrics of `MMLib` PCA. Alternatively, PCA can also be performed using a Singular Value Decomposition (SVD) on the raw data. The complexity of standard SVD is $O(min(DN^2, D^2N))$ [4]. Though this can be reduced in a distributed implementation by using Stochastic SVD, the communication complexity of stochastic SVD remains large (Figure 5).

Method to Compute PCA	Time Complexity	Communication Complexity	Example Libraries
Eigen decomp. of covariance matrix	$O(ND \times \min(N, D))$	$O(D^2)$	MLlib-PCA (Spark), RScaLAPACK
SVD-Bidiag [11]	$O(ND^2 + D^3)$	$O(\max((N+D)d, D^2))$	RScaLAPACK
Stochastic SVD (SSVD) [21]	$O(NDd)$	$O(\max(Nd, d^2))$	Mahout-PCA (MapReduce)
Probabilistic PCA (PPCA) [32]	$O(NDd)$	$O(Dd)$	sPCA

Figure 5: Comparison of PCA Methods

In contrast to the PCA model expressed in Equation 1, probabilistic formulations of dimensionality reduction algorithms that come from the same family of Gaussian Linear Models as PCA are capable of capturing both private and shared variability in observed neural data. These models take the form:

$$y = C * x + \mu + \epsilon \tag{2}$$

where ϵ expresses the noise private to each dimension of the observed data. In probabilistic PCA (pPCA), $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbb{I})$. This means that every dimension of the original data is assumed to have the same amount of private variability, a condition known as an isotropic noise constraint. In factor analysis (FA), $\epsilon \sim \mathcal{N}(0, \psi)$, where ψ is merely constrained to be a diagonal matrix; In other words, FA gives a unique estimate for the private noise of every dimension of the original data. In neuroscience data, this is equivalent to saying that each neuron has a different magintude of private variability. Indeed, neuroscientists have found that the variability in the responses of individual neurons scales with their mean firing response, and different neurons can have very different mean firing responses due to different cellular mechanics and differential tuning to the stimulus being encoded by the network.

FA (and pPCA) are latent variable models that lack closed form solutions but can be estimated by (closely-related) expectation maximization (EM) schemes. The goal of the EM algorithm is to optimize the log-likelihood of the joint probability of the data Y and the underlying low rank variable X. Given that we have observed Y, we are actually interested in estimating the marginal distribution $P(x_n|y_n)$, where y_n is the nth observation of the data Y. Note that $P(x_n|y_n)$ will take the form of a Gaussian, since X and Y are both Gaussian random variables.

The likelihood of $P(X|Y)$ (over all data observations) cannot be optimized directly because of the presence of latent variables C and Ψ. Therefore, in the E-step of the EM algorithm, C and Ψ are fixed, and we compute the marginal distribution $P(x_n|y_n)$ in terms of its first and second moments, $\mathbb{E}(x|y_n) = x_n$ and $\mathrm{Var}(x|y_n) = \Sigma_{x_n}$, respectively. In the M-step of the EM algorithm, we fix the marginal distribution and compute updated values for C and Ψ. The full FA EM scheme, computed for a set of observations Y of dimensions $D \times N$ is shown in Algorithm 1.

Notably, the EM algorithm has time complexity $O(ND)$, making it a more favorable choice for solving large-scale dimensionality reduction problems than PCA methods. Additionally, the EM algorithm more robustly deals with missing observations, which can be common in large datasets. Our work exists because there is no known, publicly available, distributed implementation of FA; however, there is an existing distributed implementation of pPCA, known as scalable PCA (sPCA), that must compute many of the same quantities as the FA EM algorithm [5]. sPCA was shown to have communication complexity $O(Dd)$, where d is the dimensionality of the projection space. Thus, the communication complexity of the EM algorithm is also superior to that of eigendecomposition.

Data: Data Matrix $\boldsymbol{Y}_{N \times D}$, int d
Result: Projection Matrix $\boldsymbol{C}_{D \times d}$
$\boldsymbol{C} = normrnd(D, d)$;
$\boldsymbol{\Psi} = diag(normrand(D, 1))$;
$\boldsymbol{Y}_m = columnMean(\boldsymbol{Y})$;
$\boldsymbol{Y}_c = \boldsymbol{Y} - \boldsymbol{Y}_m$;
while *not STOP_CONDITION* **do**
  $\boldsymbol{M} = \boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{\Psi}$;
  $\boldsymbol{X}_m = \boldsymbol{C}^T \boldsymbol{M}^{-1} \boldsymbol{Y}_c$;
  $\boldsymbol{\Sigma}_{\boldsymbol{X}_m} = \boldsymbol{I} - (\boldsymbol{C}^T \boldsymbol{M}^{-1})\boldsymbol{C} + \boldsymbol{X}_m \boldsymbol{X}_m^T$;
  $\boldsymbol{Y}_{proj} = \boldsymbol{Y}_c \boldsymbol{X}_m^T$;
  $\boldsymbol{C}_{new} = \boldsymbol{Y}_{proj} \boldsymbol{\Sigma}_{\boldsymbol{X}_m}$;
  $\boldsymbol{\Psi}_{new} = \frac{1}{N} diag(\boldsymbol{Y}_c \boldsymbol{Y}_c^T - \boldsymbol{C}_{new} \boldsymbol{X}_m \boldsymbol{Y}_c^T)$;
end

Algorithm 1: FA Expectation Maximization algorithm pseudocode

4. DESIGN OF SFA

Data: Matrix $\boldsymbol{Y}$, Vector $\boldsymbol{Y}_n$, Vector $\boldsymbol{X}_n$, Matrix CM, int D, int d
$acc_{\boldsymbol{Y}_{proj}} = spark.accumulator(newMatrix(D, d))$;
$acc_{\boldsymbol{\Sigma}_{\boldsymbol{X}_n}} = spark.accumulator(newMatrix(d, d))$;
$\boldsymbol{Y}_{proj}.map\{\boldsymbol{Y}_i =>$
  $\boldsymbol{X}_i = \boldsymbol{Y}_i * CM - \boldsymbol{Y}_n * CM$;
  $(\boldsymbol{Y}_{proj})_i = \boldsymbol{Y}_i^T(\boldsymbol{X}_i - \boldsymbol{X}_n) - \boldsymbol{Y}_n^T(\boldsymbol{X}_i - \boldsymbol{X}_n)$;
  $(\boldsymbol{\Sigma}_{\boldsymbol{X}_n})_i = \boldsymbol{X}_i^T(\boldsymbol{X}_i - \boldsymbol{X}_n) - \boldsymbol{X}_n^T(\boldsymbol{X}_i - \boldsymbol{X}_n)$;
  $acc_{\boldsymbol{Y}_{proj}}.add((\boldsymbol{Y}_{proj})_i)$;
  $acc_{\boldsymbol{\Sigma}_{\boldsymbol{X}_n}}.add((\boldsymbol{\Sigma}_{\boldsymbol{X}_n})_i)$;
$\}$;
$\boldsymbol{Y}_{proj} = acc_{\boldsymbol{Y}_{proj}}.value()$;
$\boldsymbol{\Sigma}_{\boldsymbol{X}_n} = acc_{\boldsymbol{\Sigma}_{\boldsymbol{X}_n}}.value()$;

Algorithm 2: SparkJobXY pseudocode

We present an implementation of the EM solution to the FA dimensionality reduction problem that scales to handle large datasets such as the Allen Observatory data and runs on the distributed platform Spark. By construction, the EM algorithm needs to perform all the operations shown in Algorithm 1 until the log-likelihood of the model converges to a stable maximum. This means that many variables computed in one iteration of the algorithm will need to be re-used in the next iteration of the algorithm. This form of iterative computation is naturally suited to the Spark platform, in which iterative computations are kept in memory with RDDs. The alternative solution, as implemented by Hadoop's MapReduce framework, is to save results of distributed computations to disk. While Hadoop's approach is well-suited for algorithms that only require a single pass over data, such an approach greatly increases the time spent on I/O for iterative algorithms. There are interfaces to our chosen MapReduce framework Spark from Python, Scala, and Java. We

use the Java Spark interface because of its ability to leverage the Apache Mahout environment, which includes optimized linear algebra methods for MapReduce frameworks.

4.1 Spark Job Design that Minimizes Communication

There are several ways in which one could implement the FA EM scheme in Spark with use of Mahout. One might try to exclusively use Mahout as an optimization tool for linear algebra operations and avoid launching any custom jobs in Spark. Such an implementation would scale poorly for an algebraic operation involving the very large data matrices Y and X (each of which will have one dimension greater than or equal to the number of datapoints N) as Mahout linear algebra methods require local access to each of the matrices on which they are operating. In contrast, we sought to develop an implementation that could handle an input matrix Y so large that it should be stored in RDD form and accessed a single row at a time for any required linear algebra computations. On the other extreme, one might attempt to implement each operation in Algorithm 1 as a custom Spark MapReduce job. We note, as did the developers of sPCA, that such an implementation would be naive because many of the computations only operate on matrices with dimension D (number of covariates) or d (small dimensionality of the projection space). Thus, our algorithm consists of a driver program that computes the majority of operations locally and launches Spark jobs only for the computation of X_m, Σ_{X_m}, Y_{proj}, $\texttt{Cov}(Y)$, and Y_m, all of which involve computation on the large matrices X and Y.

Furthermore, we leverage the observations by Elgamal et. al that there is a high communication cost incurred by passing intermediary matrix X_m of dimensions $N \times d$ to a Spark MapReduce job, and both the computation of Σ_{X_m} and Y_{proj} require X_m. We thus reduce the communication complexity of our algorithm in two ways. First, we compute X_m row-wise from its constiuent parts $C^T M^{-1}$ (a smaller matrix that can easily be pre-computed and passed to Spark job workers) and the rows of Y_c within each MapReduce operation that requires it. Though this increases time complexity due to the worker-side computation of X_m, it reduces the overhead of passing a monolithic matrix.

Furthermore, we leverage Elgamal et. al's observation that the operations Σ_{X_m} and Y_{proj} have no dependency on eachother and can be consolidated into the same Spark job. By combining the row-wise, distributed computation of Σ_{X_m} and Y_{proj} into a single Spark job, we ensure that X_m is only computed once. We also reduce the overhead of launching Spark jobs, broadcasting the required variables to those jobs, and managing multiple sets of Spark Accumulators needed to sum the products of the row-wise computations, producing a single output matrix.

4.2 Efficient Matrix Computations

Linear algebra operations are the bulk of the FA Algorithm and thus account for the majority of its time complexity. We optimized the matrix math of the FA algorithm in two ways. First, when we must perform operations over two large matrices, we try to use the row-outer-product formulation of matrix multiplication:

$$(A^T B) = \sum_{n=1}^{N} (A_n)^T B_n$$

where A_n is the nth row of A. We can then use a map-side join to sum over all the row-outer-product resultant matrices. This technique is used to get results Σ_{X_m} and Y_{proj}. The pseudocode for SparkJobXY in Algorithm 2 demonstrates how row-outer-product

Data: Data Matrix $\boldsymbol{Y}_{N \times D}$, int d
Result: Projection Matrix $\boldsymbol{C}_{D \times d}$
$\boldsymbol{C}$ = normrnd(D, d);
$\boldsymbol{\Psi}$ = diag(normrand(D, 1));
$\boldsymbol{Y}_m$ = MeanSparkJob($\boldsymbol{Y}$);
$\texttt{Cov}(\boldsymbol{Y})$ = CovSparkJob($\boldsymbol{Y}$,$\boldsymbol{Y}_m$);
while *not STOP_CONDITION* **do**
  $\boldsymbol{M} = \boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{\Psi}$;
  $\boldsymbol{C}_{int} = \boldsymbol{C}^T \boldsymbol{M}^{-1}$;
  $\boldsymbol{X}_m, \boldsymbol{\Sigma}_{\boldsymbol{X}_m}, \boldsymbol{Y}_{proj}$ = XYSparkJob($\boldsymbol{Y}, \boldsymbol{Y}_m, \boldsymbol{C}_{int}$);
  $\boldsymbol{C}_{new} = \boldsymbol{Y}_{proj} \boldsymbol{\Sigma}_{\boldsymbol{X}_m}$;
  $\boldsymbol{\Psi}_{new}$ =diag($\texttt{Cov}(\boldsymbol{Y})$) - $\frac{1}{N}$ diag($\boldsymbol{C}_{new} \boldsymbol{X}_m \boldsymbol{Y}_c^T$);
end

Algorithm 3: Final sFA algorithm, pseudocode

computations can be performed row-wise by workers in a Spark job and then summed using Spark accumulators.

When one of the matrices involved in the multiplication operation is relatively small, as sometimes occurs when we are doing computations on matrices with the projected dimensionality d, we can load that smaller matrix into memory. Thus, we can perform matrix multiplication of the form

$$(AB)_n = A_n B$$

where only matrix A is paritioned row-wise to worker nodes of the Spark job. Note that this formulation does not require the operation to be in outer product form; converting matrix operations to such form can result in unnecessary transpose operations. The in-memory form of matrix multiplication shown above also does not require any summation over the row-wise computations.

4.3 Leveraging Sparsity

Many large datasets are fairly sparse. Though all operations in the EM scheme from Algorithm 1 are performed on Y_c, the mean-centered version of the data Y, subtracting the mean Y_m from Y will make potentially sparse matrix Y dense. Thus, doing the mean subtraction at the beginning of the algorithm increases the number of elements that we need to compute over in all subsequent matrix operations. We can instead propogate the mean subtraction through all operations of the EM algorithm as shown in the example below.

$$Y_c C = (Y - Y_m)C = YC - Y_m C$$

Propogating the mean centering operation throughout the EM matrix computations allows us to represent Y as a Mahout SparseMatrix, in which we only store the indices of non-zeros values of the matrix. This reduces the number of elements that we must iterate over for matrix computations to only those elements that are non-zero.

This property becomes very important for the computation of $\texttt{Cov}(Y)$, an operation that we had to optimize for our FA implementation that was not part of the sPCA optimizations. We reformulate the computation of $\texttt{Cov}(Y)$ as below, leveraging the sparsity of Y before centering. We also use each row of Y as a column-vector so that we can use the row-outer-product formulation of matrix multiplication discussed in the previous section. The resulting, optimized computation of $\texttt{Cov}(Y)$ has the form below:

$$\texttt{Cov}(Y) = Y^T Y - Y m^T Y m = \sum_{n=1}^{N} ((Y_n)^T (Y_n)) - N((Y_m)^T (Y_m))$$

When a worker in a Spark MapReduce job is iterating over the row Y_n to perform part of the computation Y^TY, we need only compute the products of the non-zero elements of Y_n.

The FA EM algorithm requires the computation of the expression $(CC^T + \Psi)^{-1}$. If computed naively, solving this expression requires the inversion of a reasonably-large, dense matrix. We instead decompose this computation into the constiuent parts shown below using the Matrix Inversion Lemma:

$$(CC^T + \Psi)^{-1} = \Psi^{-1} - \Psi^{-1}C(\mathtt{I} + C^T\Psi^{-1}C)^{-1}C^T\Psi^{-1}$$

where Ψ^{-1} is a diagonal matrix that is trivially inverted and the recurring computation $\Psi^{-1}C$ can be implemented by only performing the product operations on a small subset of the elements because of Ψ's diagonality.

Algorithm 3 shows the pseudocode for our full implementation of sFA, combining all the aforementioned optimizations in this section of the paper.

5. PERFORMANCE EVALUATION

5.1 Baseline Analysis

We use standard PCA and Probabilistic PCA (pPCA) as baseline methods for the comparison. Instead of making use of real data, we chose to simulate our own low-rank data for better knowledge of the algorithm performance in a different level of noise settings.

Data Simulation: We generated 4 types of low-rank data with similar methods, denoted as $\boldsymbol{X}^{(1)} \sim \boldsymbol{X}^{(4)}$. The low-rank data matrix $\boldsymbol{D}$ is generated by:

$$\boldsymbol{D}_{n\times d} = \boldsymbol{A}_{n\times r}\boldsymbol{B}_{r\times d},$$

where $n \gg d \gg r$. That is to say, $\boldsymbol{A}_{n\times r}$ is the real low-rank data (r), and projected to a high (d) dimension space by $\boldsymbol{B}_{r\times d}$. We generated $\boldsymbol{A}$ and $\boldsymbol{B}$ by directly sampling from two normal distribution, with $\sigma_{\boldsymbol{A}} = \frac{10}{d}$ and $\sigma_{\boldsymbol{B}} = 1$, both with $\mu = 0$. Usually this guarantees $\boldsymbol{D}$ to be rank r. Ideally, PCA should recover the from $\boldsymbol{D}$ the real low-rank representation $\boldsymbol{A}$. The first simulated dataset $\boldsymbol{X}^{(1)} = \boldsymbol{D}$. Then, we define $\boldsymbol{X}^{(2)}$ as

$$\boldsymbol{X}^{(2)} = \boldsymbol{A}_{n\times r}\boldsymbol{B}_{r\times d} + \boldsymbol{\epsilon}, \text{ and } \boldsymbol{\epsilon} \sim N(\boldsymbol{0}, ss * \boldsymbol{I}).$$

The white noise $\boldsymbol{\epsilon}$ is set to be draw from the same normal distribution for every dimension with standard deviation as ss. This is what pPCA designed to handle. But in real situation, it is not necessary to be this case. The noise for each dimension, or for each neuron in our case, is draw from different normal distribution for each dimension. Therefore, we the dataset $\boldsymbol{X}^{(3)}$ and $\boldsymbol{X}^{(4)}$ both take the structure as

$$\boldsymbol{X}^{(3|4)} = \boldsymbol{A}_{n\times r}\boldsymbol{B}_{r\times d} + \boldsymbol{\epsilon}, \text{ and } \boldsymbol{\epsilon} \sim N(\boldsymbol{0}, diag(\boldsymbol{\sigma})),$$

where $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_d)$ is noise's standard deviation for each dimension. And $N(\boldsymbol{0}, diag(\boldsymbol{\sigma}))$ indicates the noise is independent for each dimension, with corresponding $\sigma_i \in \boldsymbol{\sigma}$ for each dimension. The specific σ_i is actually drew from another normal distribution by

$$\begin{aligned}\alpha &= max(std(\boldsymbol{D}_{n\times d})),\\ ss &\sim U(p\alpha, q\alpha),\\ \sigma_i &\sim N(ss, 0.25 * ss), \forall_i \in \boldsymbol{\sigma}\end{aligned}$$

Where α is an intermediate variable to depict the size of data matrix when no noise is presented. ss is draw from uniform distribution U, and it is the mean we will use to generate σ_i. Note we also used ss in determining $\boldsymbol{X}^{(2)}$ from previous equation. c and d is used to control the noise's magnitude relative to data. For $\boldsymbol{X}^{(2)}, \boldsymbol{X}^{(3)}$, we set $p = 0.01, q = 0.3$, and for $\boldsymbol{X}^{(4)}$, we set $p = 0.5, q = 1.0$. In another word, the noise added to $\boldsymbol{X}^{(2)}, \boldsymbol{X}^{(3)}$ is relatively small in magnitude comparing with each data dimension, while to $\boldsymbol{X}^{(4)}$ is much larger. The remaining parameters are set as: $n = 60$, $d = 30$ and $r = 8$. We experimented different parameters of n, d, r, p and q. The result is similar.

Simulation Experiment: We experimented PCA and pPCA on dataset $\boldsymbol{X}^{(1)} \sim \boldsymbol{X}^{(4)}$, and measure their performance by variability defined by

$$Variability = \frac{\sum_{i=1}^{r}\boldsymbol{S}_{ii}}{\sum_{j=1}^{d}\boldsymbol{S}_{jj}},$$

where $\boldsymbol{S}$ is singular matrix of $\boldsymbol{X}^{(i)}$ after SVD decomposition. We repeat the experiments for multiple times and average the variability. The result of experiment is shown in fig 6.

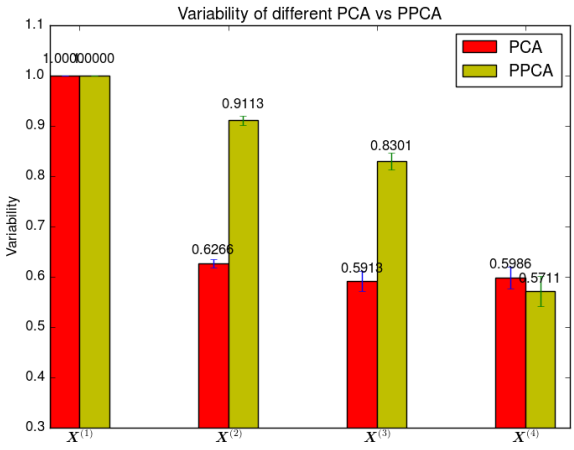

Figure 6: Experiments on simulated data with PCA and pPCA

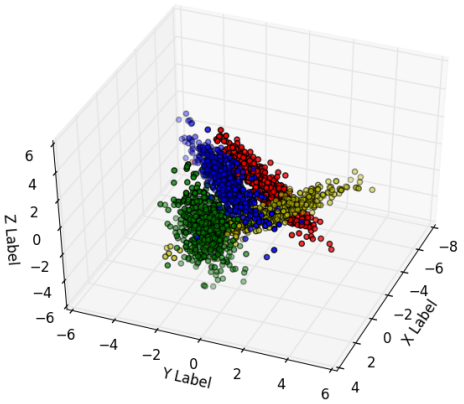

Figure 7: Visualization of 4 classes data in low-rank

Because $\boldsymbol{X}^{(1)}$ is of no noise, therefore, for PCA, the first r eigen vector can depict the whole data, which gives $variability =$

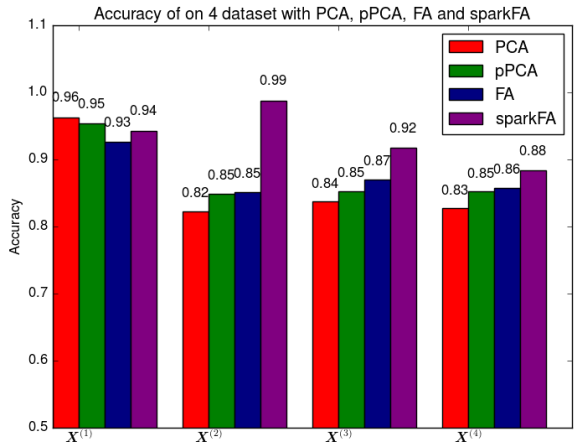

Figure 8: Classification result with different dimension reduction method

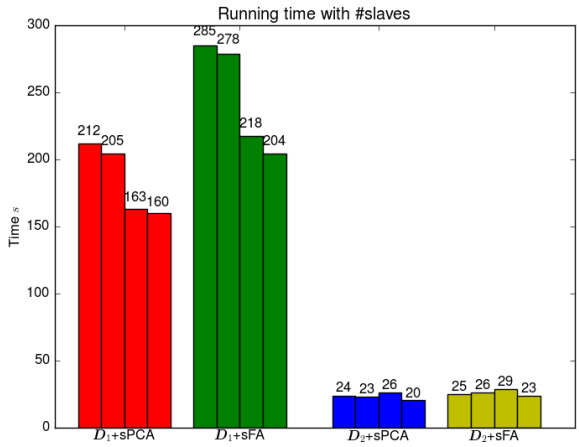

Figure 9: Running time with different number of slaves in spark cluster

1.0. Similarly, pPCA is a iterative approximation of PCA with noise. When no noise is presented, pPCA also provides good performance.

When noise presented, PCA no longer results in good performance. The variability of PCA at $\boldsymbol{X}^{(2)}, \boldsymbol{X}^{(3)}, \boldsymbol{X}^{(4)}$ are around 0.6. But for pPCA, which designed to deal with the situation of $\boldsymbol{X}^{(i)}$, still gives good results. But when the noise no longer satisfy $N(0, ss * \boldsymbol{I})$, the result deteriorates. As the noise's magnitude grow bigger, the pPCA degenerated to similar level of PCA. This meet our intuition as the noise in $\boldsymbol{X}^{(3)}, \boldsymbol{X}^{(3)}$ is $N(0, diag(\boldsymbol{\sigma}))$, which is a general case in real neuron data. The experiments on simulated data provide an intuition and suggest that we should use Factor Analysis (FA) to deal with the real neuron data.

5.2 Effectiveness Analysis

In this part we will show the effectiveness of FA when deal with $\boldsymbol{\Phi}$ noise comparing with pPCA and PCA. To approach that, we designed a classification task on simulated data to validate this idea. **Data Simulation**: We generate 2000 samples of 4 classes of data with dimension 3. Also, we control the generation process to guarantee each class of data have one cluster and linear separable. We call these data $\boldsymbol{A}_{1000\times 3}$. We can see the data distribution in the following fig. Then, we map the $\boldsymbol{A}$ to a 10 dimension space with 4 types of noise, which we mentioned in sect 6.1. Those high dimension data are denoted as $\boldsymbol{X}^i, \forall i = 1, .., 4$. **Classification Experiment**: We later use PCA, pPCA, FA (sklearn) and our sparkFA on these 4 datasets, and reduce the dimension back to 3. Because of noise presents, the high dimension matrix cannot easily be converted back with same, original low rank structure like $\boldsymbol{A}$ did. It's likely they are no longer linear separable. Therefore, a linear classifier like svm will not produce same accuracy on the 4 datasets. The classifier we used is a linear kernel svm, with $C = 1$. The result is listed below. We can see from fig **??** that the our implemented sparkFA obtained best classification on dataset $\boldsymbol{X}^3$ and $\boldsymbol{X}^4$, where light and heavy $\boldsymbol{\Phi}$ noise presented respectively. This proves the correctives of our sparkFA implementation. Further, the accuracy of classification task, represents the quality of the dimension reduction method with different noise type. We can see the either pure local FA (implemented with sklearn) or sparkFA produce better result for $\boldsymbol{X}^3$ and $\boldsymbol{X}^4$, which is common in neural science.

5.3 Efficiency Analysis

In this part, we evaluate our algorithm with Allen Institute Data, which is real neural data collected from mice brain. We collected two group of data, one with 10000 samples over 300 neurons, and the other one is composed with 4662 samples over 100 neuron. We denote them as $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ individually. The second dataset is about 12M in text format.

For hardware, we build a spark cluster on Amazon, and experimented with local, and 1, 2, 4 slaves. Each master and slave node is m1.large, with 2 cpu and 7.5 GiB memory. Also, we experimented with the same setting with sPCA. We set the principal components / factors to for $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ to be 30 and 10 respectively. The time is listed below. The bar chart in grouped into 4 cluster, in each cluster there are 4 bars denoting running spark locally (single node, or 0 slaves), 1 slave, 2 slaves and 4 slaves. For the dataset $\boldsymbol{D}_1$, we can easily see the boosting effect of more nodes. However, on $\boldsymbol{D}_2$ things are little bit different. By using distributed mode instead on local mode spark, the running time increased. This can be explained by communication and scheduling overhead by distributed spark. When the dataset is small, the overhead will take a greater share in total running time. But for big dataset like $\boldsymbol{D}_1$, increase slaves do result better efficiency.

The other thing we node is when running time on the same dataset, the sPCA uses less time than sFA. We think it is because our sFA is based on the sPCA instead of writing from scratch. Therefore, some detailed optimization that originally works for sPCA might not work well for sFA. For example, the key variables that are expensive to calculate in sFA might not be fully optimized, distributed or re-used.

Therefore, if we want to scale it up to process several thousands of neuron at the same time, we can definitely benefit from using sparkFA.

6. FUTURE WORK

6.1 Bottleneck in Data Retrieval

Before introducing the core algorithm, we have mentioned our workflow to retrieve data from Allen Institute. Though we have not worked on improving data retrieval since it seems that network is the main bottleneck, which is beyond our control, we have done a

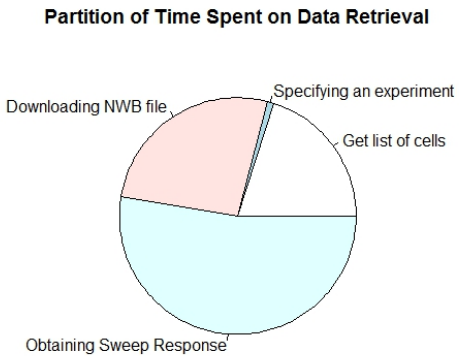

Figure 10: Partition of Time Spent on Data Retrieval

simple analysis on the time complexity of our data retrieval. Figure 10 shows the breakdown of time spent on data retrieval. It takes the server a decent amount of time to get the list of all cells available in a particular experiment and manage to return it to us. Downloading the NBW takes a while, but finishes with a reasonable time, considering each NBW file is more than half gigabytes large. However, obtaining sweep response from the local NBW files takes the most time, and since we only use the functions in the API as directed in the instructions, we have little knowledge about how their functions operate to return results and about how NWB files are organized.

Nevertheless, we do believe that it is possible to make the entire API compatible with distributed algorithms. Since end-users of Allen Institute's API will utilize each function as a black box, it would be nice for Allen Institute to re-design some of the functions to be able to queries of more granularity so that workers from a distributed program can effectively retrieve a small range of data each of them needs.

6.2 HDF5 File Format

On the other hand, having a scalable data format is as important, if not more, to reduce overall time complexity. The NWB file format itself is a specific format of HDF5 format, i.e. Hierarchical Data Format V. Aiming to help deal with large scientific data, `HDF5` file format supports unlimited size and format, flexible and efficient I/O, and complex data subsetting.

Furthermore, the `H5Spark` package (https://www.nersc.gov/ users/data-analytics/data-management/i-o-libraries/hdf5-2/h5spark/), developed by Lawrence Berkeley National Lab for use on their `NERSC` HPC resources [6], uses `PySpark` to enable MapReduce operations across data contained in multiple `HDF5` files. Together, these tools will provide a framework for streaming Brain Observatory data with specified features from many `HDF5` files, bringing scalability to a new higher level.

7. CONCLUSION

Scientific research often involves working with extra large data sets and complex matrix calculations; therefore, techniques commonly used by different scientific communities are in great need of parallelization. Particularly we look into one of the dimensionality reduction techniques used in neuroscience community, Factor Analysis. Compared to PCA and stochastic PCA, which have been implemented in parallel, Factor Analysis loosens the assumptions even more to accommodate research needs. We employ neuroscience data from Allen Brain Institute as our primary data source.

Our project focuses on designing and implementing Factor Analysis in Spark. Using Spark can greatly reduce total time complexity since it distributes workload to different workers concurrently. Moreover, with RDD structures in Spark, I/O cost is also greatly reduced for repetitive reads and writes, which happens quite often in large matrix operations.. We start from the conventional EM algorithm for Factor Analysis and have employed different techniques to improve our algorithm in a distributed settings. These techniques include distributing large matrix operations, leveraging matrix sparsity, etc.

After implementing the FA algorithm in Spark, we also conduct experiments both locally and on AWS to test its effectiveness as well as efficiency. For effectiveness, our test is based on simulated data of different variance assumptions. Our test results show that Factor Analysis does a better job explaining the variability in the data set when data violates assumptions used in PCA and sPCA. Baseline analysis between PCA and PPCA is also included. To analyze the algorithm efficiency, our experiments are on the Allen Institute Data and the results show that the Spark FA algorithm achieves similar clock time as sPCA, despite that Factor Analysis requires more complex calculations compared to sPCA. We are very excited to see that by applying more techniques for large matrix operations, our Spark Factor Analysis implementation outperforms the sPCA implementation both in effectiveness and in efficiency.

8. REFERENCES

[1] Brain observatory. *Allen Brain Observatory*.

[2] Brain observatory trace analysis. *Allen Brain Observatory*.

[3] Technical white paper: Overview. *Allen Brain Observatory*, Jun 2016.

[4] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM Journal on Scientific and Statistical Computing*, 11(5):873–912, 1990.

[5] T. Elgamal, M. Yabandeh, A. Aboulnaga, W. Mustafa, and M. Hefeeda. spca: Scalable principal component analysis for big data on distributed platforms. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 79–91, New York, NY, USA, 2015. ACM.

[6] J. Liu, E. Racah, Q. Koziol, and R. S. Canon. H5spark: Bridging the i/o gap between spark and scientific data formats on hpc systems.