An All–at–Once CP Decomposition Method for Count Tensors

Teresa M. Ranadive
Laboratory for Physical Sciences
College Park, MD 20740
tranadive@lps.umd.edu

Muthu M. Baskaran
Reservoir Labs, Inc.
New York, NY 10012
baskaran@reservoir.com

***Abstract*—CANDECOMP/PARAFAC (CP) tensor decomposition is a popular method for detecting latent behaviors in real–world data sets. As data sets grow larger and more elaborate, more sophisticated CP decomposition algorithms are required to enable these discoveries. Data sets from many applications can be represented as count tensors. To decompose count tensors, one should minimize the sum of the generalized Kullback–Leibler divergences from each tensor entry to each corresponding decomposition entry. Most often, this is done using the algorithm CP–APR (CP–Alternating Poisson Regression). In view of the fact that all–at–once optimization algorithms for related CP decomposition problems often achieve better decomposition accuracy than alternating algorithms like CP–APR, we develop CP–POPT–GDGN, an all–at–once algorithm for count tensor decomposition that utilizes a generalized damped Gauss-Newton method. We then implement a highly efficient version of CP–POPT–GDGN into the tensor package ENSIGN. After decomposing several tensors formed from real–world data sets, many of which are related to network traffic, we see that CP–POPT–GDGN typically outperforms CP–APR, both in terms of decomposition accuracy and latent behavior detection.**

***Index Terms*—Big Data Analytics, Count Tensor Decomposition, Generalized Gauss–Newton, High Performance Computing**

I. Introduction

CANDECOMP/PARAFAC (CP) tensor decomposition is an unsupervised machine learning method often utilized to discover latent patterns in data sets with multiple attributes. While first employed for applications in psychology [6], CP decomposition has since been used for various other applications including geospatial analysis [12], [14], text analysis [1], and cyber security [11]. As data sets grow larger and increasingly intricate, more advanced algorithms are required to obtain CP decompositions that adequately describe the data. Therefore, in this paper, we propose, develop, and describe the implementation of a novel CP decomposition algorithm for count tensors that achieves significant improvements in accuracy and latent pattern detection over existing algorithms.

To compute a CP tensor decomposition of a tensor $\mathcal{X}$, one typically seeks to minimize the sum, over all tensor entries, of some loss function that measures the "distance" between each entry of $\mathcal{X}$ and its corresponding CP decomposition model entry. Most often, this loss function is given by the square of the difference between each of the two corresponding entries. However, in minimizing this sum–of–squares function, one is computing the CP decomposition that is the maximum likelihood estimator (MLE) of $\mathcal{X}$, under the assumption that the entries of $\mathcal{X}$ are independent and identically distributed (i.i.d.) with a Gaussian distribution [16]. The assumption that this distribution is Gaussian is unreasonable if $\mathcal{X}$ is a *count tensor*, i.e., a tensor with entries that are all whole numbers, as is the case for many applications. It is more appropriate to assume that the i.i.d. entries of a count tensor $\mathcal{X}$ follow a Poisson distribution. Under this second assumption, one must instead minimize the sum, over all tensor entries, of the generalized Kullback–Leibler (gKL) divergence from each entry of $\mathcal{X}$ to its corresponding model entry, to obtain the CP decomposition that is the MLE of $\mathcal{X}$ [8]. We refer to the objective given by this sum of gKL divergences as *gKL_obj*.

There are several algorithms already available to minimize gKL_obj. One such algorithm is CP–Alternating Poisson Regression (CP–APR), a deterministic algorithm that minimizes gKL_obj by optimizing over alternating sets of variables (each set corresponding to a particular tensor mode), while fixing the remaining sets of variables [8]. CP–APR may solve the subproblem of optimizing over a given set of variables using either a multiplicative update [8, Algorithm 3] or Newton based [13] approach. Other deterministic alternating approaches have also been considered [7], [18], [36]. However, CP–APR is the most effective algorithm for minimizing gKL_obj that is both alternating and deterministic [16]. Recently, SmartCPD, a stochastic alternating algorithm for minimizing gKL_obj (as well as other CP decomposition objectives) via stochastic mirror descent and a strategic sampling scheme was proposed in [28]. SmartCPD appears promising, although the authors do not give a comparison between SmartCPD and CP–APR in their work. There are also several all–at–once optimization algorithms for generalized CP decomposition that can be utilized to minimize many different CP decomposition objectives, including gKL_obj. These are (i) GCP–OPT, which uses stochastic gradient descent [16], [21], and (ii) an algorithm that uses a generalized Gauss–Newton method [34]. A comparison of several algorithms, including three novel ones, that are used to minimize gKL_obj for nonnegative *matrix* factorization is given in [15]. However, two of these algorithms have yet to be generalized to higher order tensors, while SmartCPD from [28] is a stochastic version of the third for tensors. Finally, [10], [22], [31], [37] consider performing tensor decompositions by

minimizing objectives, characterized by the gKL divergence, related to alternative tensor decomposition models via other tensor algorithms. However, the focus of this work is on the critical problem of CP tensor decomposition (rather than other tensor decomposition models and problems) as the CP decomposition model is an effective tool for pattern discovery [11], [14] that requires only a relatively small number of parameters.

Previous work has illustrated that certain all–at–once optimization CP decomposition algorithms that minimize the sum–of–squares objective achieve higher decomposition accuracy than the alternating algorithm CP–Alternating Least Squares [6] (used to minimize the same objective), without requiring much additional time to compute decompositions [25], [26], [29], [33]. CP–ALS converges to relatively inaccurate decompositions quickly and then offers no further accuracy improvements, even if run for additional time, while certain all–at–once algorithms require additional time to converge, but compute more accurate decompositions. However, to the best of our knowledge, there are currently no tensor algorithms that have been shown to achieve more accurate decompositions (in terms of final gKL_obj value) than the alternating algorithm CP–APR. No comparison is given between the CP–APR and the decomposition algorithms in [28] and [34]. In [34] the authors demonstrate that their all–at–once algorithm achieves better decomposition accuracy than the alternating CP decomposition algorithm from [7], but the algorithm from [7] is not as effective as CP–APR. Additionally, GCP–OPT achieves nearly, but not quite, the same accuracy as CP–APR [16], [21].

Therefore, inspired by techniques from [8, Algorithm 3] and [34], we develop CP–POPT–GDGN, an all–at–once OPTimiztion algorithm for CP count tensor decomposition that uses a generalized damped Gauss–Newton method to optimize gKL_obj. The first 'P' in "POPT" represents the assumption that the tensor entries are i.i.d. with a Poisson distribution.

We then implement CP–POPT–GDGN into ENSIGN [9], a commercially available tensor software package written in C with (i) highly efficient implementations of CP tensor decomposition algorithms, including CP–APR, and (ii) sparse tensor data structures, namely, mode–specific sparse (MSS) and mode–generic sparse (MGS) data structures that enable the efficient implementations of ENSIGN's algorithms [3]. ENSIGN's tensor decomposition algorithms are parallelized and optimized for computation and memory efficiencies, with both shared and distributed memory versions of several tensor algorithms available [4], [5]. We focus on the shared memory versions of ENSIGN's algorithms, as we have not yet implemented a distributed memory version of CP–POPT–GDGN.

Finally, we compare the performance, in terms of decomposition accuracy and latent behavior detection, of CP–POPT–GDGN to that of the above CP decomposition algorithms. This is done by decomposing several count tensors (some large–scale) formed from real–world data sets. We observe that for all of these tensors, CP–POPT–GDGN obtains at least as (and often much more) accurate decompositions as (than) all of the other decomposition algorithms, including CP–APR, although CP–POPT–GDGN requires more time to converge than CP–APR (we note that running CP–APR for additional time after it converges does not lead to improvements in accuracy). Of particular interest, is that CP–POPT–GDGN obtains more accurate decompositions than those computed using CP–APR for all of the tensors formed from network traffic data sets. We also see how, for two large–scale tensors formed from NetFlow data, CP–POPT–GDGN detects behaviors that are not detected by CP–APR, but CP–APR does not detect any behaviors that are not detected by CP–POPT–GDGN.

This paper is organized as follows. In Section II we introduce some notation, present background information on count tensors and their CP decompositions via the minimization of gKL_obj, and list previous algorithms that have been used to perform this minimization, along with their available implementations. In Section III we develop CP–POPT–GDGN and discuss its implementation into ENSIGN. Next, in Section IV, we decompose several tensors formed from real–world data sets using CP–POPT–GDGN, as well as many of the algorithms discussed in Section II, comparing the accuracy of the various decompositions. Finally, in Section V, we compare the performance, in terms of latent behavior detection, of CP–POPT–GDGN to that of CP–APR, for two large–scale sparse tensors formed from network traffic data. We conclude with a summary in Section VI. More information on previously developed algorithms for the CP decomposition of count tensors, further details on CP–POPT–GDGN, and descriptions of the data sets that are used for evaluation are given in the Appendices.

II. Notation and Background Information

In this section, we introduce some notation (cf. Section II-A), provide background information on count tensors and their CP decompositions (cf. Section II-B), and list the related state–of–the–art and most widely used CP count tensor decomposition algorithms (cf. Section II-C). For more information on tensors and their decompostions, see [20].

A. Notation

Let $\mathbb{R}$, $\mathbb{R}_+$, $\mathbb{N}$, and $\mathbb{N}_0$ denote the sets of real, nonnegative real, natural, and whole numbers, respectively. For $N \in \mathbb{N}$, we define $[N] := \{1, \ldots, N\}$. Scalars, vectors, matrices, and tensors are written with lowercase (e.g., x), bold lowercase (e.g., $\mathbf{x}$), bold uppercase (e.g., $\mathbf{X}$), and calligraphic (e.g., $\mathcal{X}$) letters, respectively. For $\mathbf{x} \in \mathbb{R}^M$, $(\mathbf{x})_+$ denotes the entry-wise maximum of $\mathbf{x}$ and zero. If $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is a tensor and $\mathbf{i} := (i_1, \ldots, i_N) \in [I_1] \times \cdots \times [I_N]$, then $x_{\mathbf{i}}$ denotes the entry of $\mathcal{X}$ indexed by $\mathbf{i}$. For a set $\mathcal{F} \subseteq [M]$ let the matrix $\mathbf{A}_{\mathcal{F}\mathcal{F}}$ (resp. vector $\mathbf{b}_{\mathcal{F}}$) denote the submatrix (resp. subvector) of $\mathbf{A} \in \mathbb{R}^{M \times M}$ (resp. $\mathbf{b} \in \mathbb{R}^M$) whose rows and columns (resp. entries) are indexed by $\mathcal{F}$.

B. Count Tensors and CP Decomposition

For $N \in \mathbb{N}$, an N**-mode tensor** $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is a multimodal array. Tensors generalize the concepts of vectors and matrices: a vector is a 1-mode tensor, while a matrix is a 2-mode tensor. Tensors may be formed from data sets with

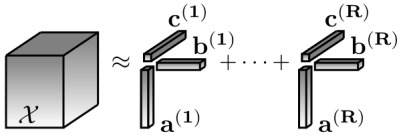

Fig. 1. A rank-R CP decomposition of a three-mode tensor [20].

multiple attributes. For example, after collecting the metadata, with fields sender, recipient, and day sent, of all emails sent to and from 50 employees in a company during a given week, one may form a 3-mode tensor $\mathcal{X} \in \mathbb{R}^{50\times 50\times 7}$ from this data set, with the three modes of $\mathcal{X}$ corresponding to the three respective metadata fields. This is done by entering the number n into the (i,j,k)-th entry of $\mathcal{X}$ to indicate that sender i sent n emails to recipient j on day k, for all $(i,j,k) \in [50]\times[50]\times[7]$. We note that the tensor $\mathcal{X}$ formed in such a way is a **count tensor**, i.e., all entries of $\mathcal{X} \in \mathbb{N}_0^{I_1\times\cdots\times I_N}$ are whole numbers. Count tensors are an important class of tensors, as tensors are formed in the above way for many applications.

The multi-modal nature of a tensor $\mathcal{X}$ makes it difficult to determine any interpretable patterns that characterize that tensor. Consequently, there have been many efforts to decompose tensors into lower dimensional structures to extract such patterns. One such type of decomposition is the **CP decomposition**. A CP decomposition $\mathcal{M}$ of an N-mode tensor $\mathcal{X}$ is an approximation of $\mathcal{X}$ given by the sum of R outer products of appropriately sized vectors, for some *a priori* chosen $R \in \mathbb{N}$ (cf. Figure 1). We call R the **rank** of $\mathcal{M}$, say the sum of the outer products is a **rank-R decomposition** of $\mathcal{X}$, and refer to the individual outer products forming $\mathcal{M}$ as the **rank-1 components** of $\mathcal{M}$ [20]. We note that a CP decomposition of $\mathcal{X}$ is itself an N-mode tensor.

In Figure 1, we see how a CP decomposition of a 3-mode tensor $\mathcal{X}$ is given by R outer products of three vectors. The vectors corresponding to the first (resp. second and third) mode of $\mathcal{X}$ are given by $\{\mathbf{a^{(r)}}\}_{r\in[R]}$ (resp. $\{\mathbf{b^{(r)}}\}_{r\in[R]}$ and $\{\mathbf{c^{(r)}}\}_{r\in[R]}$). For each mode, we collect the vectors corresponding to that mode and form a matrix whose columns are given by those vectors, e.g., we form $\mathbf{A} := \left[\mathbf{a^{(1)}}|\ldots|\mathbf{a^{(R)}}\right]$, In the same way, we form form $\mathbf{B}$ from $\{\mathbf{b^{(r)}}\}_{r\in[R]}$ and $\mathbf{C}$ from $\{\mathbf{c^{(r)}}\}_{r\in[R]}$. We call $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ the **factor matrices** of $\mathcal{M}$ and we may write the rank-R CP decomposition $\mathcal{M}$ of $\mathcal{X}$ in terms of its factors matrices [20]: $\mathcal{M} := [[\mathbf{A},\mathbf{B},\mathbf{C}]]$.

In order to determine the "best" rank-R CP decomposition of a tensor $\mathcal{X}$, one must solve some sort of minimization problem whose objective measures the "distance" between $\mathcal{X}$ and each admissible rank-R CP decomposition $\mathcal{M}$. Because the entries of a count tensor $\mathcal{X}$ take on discrete, whole number values, it is most reasonable to assume that if the entries of $\mathcal{X}$ are i.i.d. random variables, then the underlying distribution of those entries follow a Poisson distribution, rather than a Gaussian or some other well–known type of distribution. Under this assumption, in order to compute the rank-R CP decomposition $\mathcal{M}$ that is the maximum likelihood estimator of $\mathcal{X}$, we must minimize the sum, over all entries of $\mathcal{X}$, of the generalized Kullback–Leibler divergence from each entry of $\mathcal{X}$ to its corresponding entry in $\mathcal{M}$, i.e., we solve

$$\underset{\substack{\text{rank}(\mathcal{M})=R,\\ \text{all } m_{\mathbf{i}}\geq 0.}}{\text{minimize}} \sum_{\mathbf{i}\in[I_1]\times\cdots\times[I_N]} (m_{\mathbf{i}} - x_{\mathbf{i}}\log(m_{\mathbf{i}})) + c_{\mathcal{X}}, \qquad (1)$$

where $c_{\mathcal{X}} := \sum_{\mathbf{i}\in[I_1]\times\cdots\times[I_N]} x_{\mathbf{i}}(\log(x_{\mathbf{i}})-1)$ is a constant determined by the data and the nonnegativity constraints on the entries of $\mathcal{M}$ are needed so the objective in (1) is defined.

Using the above notation, we may rewrite the rank-R decomposition $\mathcal{M}$ of the N-mode tensor $\mathcal{X}$ in terms of factors matrices, $\mathbf{A^{(1)}} \in \mathbb{R}^{I_1\times R},\ldots,\mathbf{A^{(N)}} \in \mathbb{R}^{I_N\times R}$, so that $\mathcal{M} = [[\mathbf{A^{(1)}},\ldots,\mathbf{A^{(N)}}]]$, and for each $\mathbf{i} = (i_1,\ldots,i_N)$,

$$m_{\mathbf{i}} = [[\mathbf{A^{(1)}},\ldots,\mathbf{A^{(N)}}]]_{\mathbf{i}} = \sum_{r=1}^{R} a^{(1)}_{i_1 r}\ldots a^{(N)}_{i_N r}. \qquad (2)$$

Let $\epsilon \geq 0$ and $M := (I_1+\cdots+I_N)\times R$. Then in view of (2), define $f_\epsilon : \mathbb{R}_+^M \to \mathbb{R}$ such that

$$f_\epsilon(\mathbf{A^{(1)}},\ldots,\mathbf{A^{(N)}}) := \sum_{\mathbf{i}\in[I_1]\times\cdots\times[I_N]} (m_{\mathbf{i}} - x_{\mathbf{i}}\log(m_{\mathbf{i}}+\epsilon)) + c_{\mathcal{X}}. \qquad (3)$$

Therefore, when $\epsilon = 0$, we may rewrite (1) as

$$\underset{\substack{\mathbf{A^{(n)}}\geq 0,\\ \text{for all } n\in[N]}}{\text{minimize}}\ f_\epsilon(\mathbf{A^{(1)}},\ldots,\mathbf{A^{(N)}}). \qquad (4)$$

The nonnegativity constraints on the $\mathbf{A^{(n)}}$'s guarantee the nonnegativity of the entries of $\mathcal{M}$. Our goal is to solve (4) to obtain the CP decompositions of count tensors. Note that we do not actually set $\epsilon = 0$ in practice, as f_ϵ must be defined for all feasible sets of factor matrices. Instead we let $\epsilon > 0$ be some small value, e.g., 10^{-10}.

C. Previous Decomposition Algorithms for Count Tensors

The most widely used and state–of–the–art algorithms used to solve (4) are listed below, along with abbreviations and their available (to us) implementations. We use the abbreviations throughout the rest of the paper. Thorough descriptions of these algorithms are given in Appendix A.

- **MU**: CP–APR–MU [8, Algorithm 3] - ENSIGN and MATLAB [2] implementations are available.
- **PQNR**: CP-APR-PQNR [13] - ENSIGN and MATLAB [2] implementations are available.
- **PDNR**: CP-APR-PDNR [13] - ENSIGN and MATLAB [2] implementations are available.
- **SmartCPD**: [28] - no implementations available.
- **GCP**: GCP–OPT [16] - MATLAB [2] and C++ (in the package GenTen [27]) implementations are available.
- **GGN** [34] - a MATLAB [35] implementation is available.

III. The CP–POPT–GDGN Algorithm

CP–POPT–GDGN (POPT) is developed in Section III-A; we discuss its implementation into ENSIGN in Section III-B.

A. Overview of CP–POPT–GDGN

Our algorithm, POPT, is outlined in Algorithm 1, where the various vectors in the outline represent the collections of factor matrices such as those in (4), reshaped into a single vector. We abuse notation, by writing f_ϵ as a function of such vectors instead of as a function of the factor matrices. Roughly, POPT is given by applying an active–set trust–region method to (4), i.e., for each iteration of POPT, we

(i) compute a Cauchy point $\mathbf{z}^{(\mathbf{k})}$ (cf. line 3),
(ii) use $\mathbf{z}^{(\mathbf{k})}$ to determine which entries of the next iterate will be active (fixed at zero) and which will be free,
(iii) compute a Newton based direction to update the free variables and use that update to (ideally) obtain a point $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ such that $f_\epsilon(\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}) < f_\epsilon(\mathbf{z}^{(\mathbf{k})})$ (cf. line 5).
(iv) decide whether the next iterate should be given by $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ or the Cauchy point $\mathbf{z}^{(\mathbf{k})}$ (cf. lines 6–18), and
(v) adjust the damping parameter in order to control the trust–region radius for the next iteration (cf. line 19).

However, due to the rapidly changing nature of the $\log(\cdot)$ function in (3) when its argument is near zero, we sometimes perturb certain entries of the Cauchy point $\mathbf{z}^{(\mathbf{k})}$ and $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ away from zero (cf. lines 4 and 9) to improve the performance of POPT in practice. We discuss the main steps of POPT in more detail below, dropping the superscript (k) to simplify notation. Further details, including parameter choices and the procedure for perturbing points, are given in Appendix B.

Algorithm 1 CP–POPT–GDGN

1: Choose initial guess $\mathbf{x}^{(\mathbf{0})}$, initial damping parameter λ, and initial perturbation parameter κ.
2: **for** $k = 0, 1, \ldots$ maxit **do**
3: Compute Cauchy point $\mathbf{z}^{(\mathbf{k})}$, starting search from $\mathbf{x}^{(\mathbf{k})}$.
4: Perturb certain entries of $\mathbf{z}^{(\mathbf{k})}$ near zero to get $\tilde{\mathbf{z}}^{(\mathbf{k})}$.
5: Compute the GDGN direction $\mathbf{d}^{(\mathbf{k})}$ at $\tilde{\mathbf{z}}^{(\mathbf{k})}$.
Set $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})} = (\tilde{\mathbf{z}}^{(\mathbf{k})} + \mathbf{d}^{(\mathbf{k})})_+$.
6: **if** $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ meets conditions (C1)–(C4) **then**
7: Set $\mathbf{x}^{(\mathbf{k+1})} = \mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ and go to line 19.
8: **else**
9: Perturb certain entries of $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ to get $\tilde{\mathbf{x}}_{\mathbf{GN}}^{(\mathbf{k})}$.
10: **if** $\tilde{\mathbf{x}}_{\mathbf{GN}}^{(\mathbf{k})}$ meets conditions (C1)–(C4) **then**
11: Set $\mathbf{x}^{(\mathbf{k+1})} = \tilde{\mathbf{x}}_{\mathbf{GN}}^{(\mathbf{k})}$ and go to line 19.
12: **end if**
13: **end if**
14: **if** $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$ meets conditions (C1) and (C5) **then**
15: Set $\mathbf{x}^{(\mathbf{k+1})}$ to $\mathbf{x}_{\mathbf{GN}}^{(\mathbf{k})}$.
16: **else**
17: Set $\mathbf{x}^{(\mathbf{k+1})}$ to Cauchy point $\mathbf{z}^{(\mathbf{k})}$.
18: **end if**
19: Adjust damping and perturbation parameters, λ and κ.
20: **end for**

Line 3 - computing the Cauchy point: We wish to solve the constrained optimization problem given by (4) using an active–set, trust–region method. Therefore, since (4) has only bound constraints, it is appropriate to compute a Cauchy point in order to determine which variables will be free and which will be active when computing the next iterate [23]. Let $\mathbf{g} : \mathbb{R}_+^M \to \mathbb{R}^M$ and $\hat{\mathbf{H}} : \mathbb{R}_+^M \to \mathbb{R}^{M\times M}$ denote the gradient and (generalized) Gauss-Newton Hessian approximation (cf. [34]) of $f := f_\epsilon$ from (3), respectively, for some fixed $\epsilon > 0$. To obtain the Cauchy point, we first model f by a quadratic function $m_{\mathbf{x}} : \mathbb{R}_+^M \to \mathbb{R}$ near $\mathbf{x}$, such that the model value, gradient, and Hessian at $\mathbf{x}$ are given by $f(\mathbf{x})$, $\mathbf{g}(\mathbf{x})$, and $\hat{\mathbf{H}}(\mathbf{x})$, respectively. We then consider the piecewise linear path formed by projecting the ray starting at $\mathbf{x}$ and moving in the direction $-\mathbf{g}(\mathbf{x})$ onto $\mathbb{R}_+^M$. The Cauchy point $\mathbf{z}$ is the first minimizer of $m_{\mathbf{x}}$ on that piecewise linear path, and the set of free variables for computing the next iterate is then given by

$$\mathcal{F} := \{i \in [M] \,|\, \mathbf{z}_i > 0 \text{ or } \mathbf{g}(\mathbf{z})_i > 0\}. \tag{5}$$

Line 4 - perturbing the Cauchy point: In line 5, we would like to solve a Newton–like linear system whose matrix of coefficients is given by a damped version of $\hat{\mathbf{H}}(\mathbf{z})$. However, if certain entries of $\mathbf{z}$ are near zero, related entries of $\hat{\mathbf{H}}(\mathbf{z})$ could become extremely large, making the linear system ill-conditioned and impossible to accurately solve. Therefore, we 'scooch' or perturb certain entries of $\mathbf{z}$ near zero, by setting those entries equal to some small $\kappa > 0$, similar to [8]. Note that even though we are perturbing $\mathbf{z}$ to obtain $\tilde{\mathbf{z}}$, and use $\tilde{\mathbf{z}}$ in place of $\mathbf{z}$ in subsequent steps, we need not be concerned that this step is somewhat unconventional: we may always choose our next iterate to be the Cauchy point (cf. line 17) if $\mathbf{x}_{\mathbf{GN}}$, given by the Newton based direction is unsatisfactory, and we may decrease κ later on in line 19 if this perturbation parameter is currently too large. In practice, this step greatly improves the decomposition accuracy achieved by POPT.

Line 5 - computing the generalized damped Gauss–Newton (GDGN) direction: In this step, we compute the GDGN direction $\mathbf{d}$, using (i) $\hat{\mathbf{H}}(\tilde{\mathbf{z}})$ to approximate the Hessian and (ii) the diagonal matrix $\mathbf{D}$ given (mostly) by the diagonal entries of $\hat{\mathbf{H}}(\tilde{\mathbf{z}})$ as the damping matrix (if any diagonal entries of $\hat{\mathbf{H}}(\tilde{\mathbf{z}})$ fall below a particular threshold, the corresponding entries of $\mathbf{D}$ are instead set equal to that threshold). We also set any entries of $\mathbf{d}$ indexed by $i \notin \mathcal{F}$ (cf. (5)) equal to zero. The linear system we solve to obtain $\mathbf{d}_{\mathcal{F}}$ is then

$$-(\hat{\mathbf{H}}(\tilde{\mathbf{z}}) + \lambda\mathbf{D})_{\mathcal{F}\mathcal{F}}\, \mathbf{d}_{\mathcal{F}} = \mathbf{g}(\tilde{\mathbf{z}})_{\mathcal{F}}. \tag{6}$$

To solve this linear system, we use a preconditioned conjugate gradient (PCG) method, with a Jacobi preconditioner. Once we compute $\mathbf{d}$, we then obtain $\mathbf{x}_{\mathbf{GN}}$ by moving from $\tilde{\mathbf{z}}$ along $\mathbf{d}$ and projecting the result back onto $\mathbb{R}_+^M$.

Lines 6–18 - choosing the next iterate: Let $m_{\tilde{\mathbf{z}}} : \mathbb{R}_+^M \to \mathbb{R}$ be the quadratic model of f such that the the value, gradient, and Hessian of $m_{\tilde{\mathbf{z}}}$ at $\tilde{\mathbf{z}}$ are given by $f(\tilde{\mathbf{z}})$, $\mathbf{g}(\tilde{\mathbf{z}})$, and $\hat{\mathbf{H}}(\tilde{\mathbf{z}})$, respectively. Notice that we may consider how well $m_{\tilde{\mathbf{z}}}$ models f at a point $\mathbf{x}$ near $\tilde{\mathbf{z}}$ by considering the quantity

$$\eta := \frac{f(\tilde{\mathbf{z}}) - f(\mathbf{x})}{m_{\tilde{\mathbf{z}}}(\tilde{\mathbf{z}}) - m_{\tilde{\mathbf{z}}}(\mathbf{x})}. \tag{7}$$

If $\eta \approx 1$, $m_{\tilde{\mathbf{z}}}$ is an excellent model of f at $\mathbf{x}$, while if $\eta < 0$, $m_{\tilde{\mathbf{z}}}$ is a very poor model.

Having computed $\mathbf{x_{GN}}$, we must decide if $\mathbf{x_{GN}}$ is a suitable next iterate. We require the following conditions ((C1)–(C4)) to be met in order to immediately set the next iterate to $\mathbf{x} = \mathbf{x_{GN}}$. These conditions are similar to those in related trust–region methods [23], with (C3)–(C4) together roughly indicating that $\mathbf{x}$ is a minimizer of f inside the trust region:

(C1) $f(\mathbf{x}) < f(\mathbf{z})$ - f obtains a lower value at $\mathbf{x}$ than at $\mathbf{z}$.
(C2) $\eta > 0.75$ - $m_{\tilde{\mathbf{z}}}$ is a good model of f at $\mathbf{x}$ (cf. (7)).
(C3) $\mathbf{d}^T\mathbf{g}(\tilde{\mathbf{z}}) < 0$ - $\mathbf{d}$ is a descent direction.
(C4) $\mathbf{d}^T\mathbf{g}(\mathbf{x}) < 0.1 \times |\mathbf{d}^T\mathbf{g}(\tilde{\mathbf{z}})|$ or $|\mathbf{d}^T\mathbf{g}(\mathbf{x})| < 0.9 \times |\mathbf{d}^T\mathbf{g}(\tilde{\mathbf{z}})|$ - moving along $\mathbf{d}$ reduces the positive curvature of f.

If any of these conditions are not met, we perturb $\mathbf{x_{GN}}$ in a way similar to the way $\mathbf{z}$ is perturbed in line 4, hoping that the failure of $\mathbf{x} = \mathbf{x_{GN}}$ to meet conditions (C1)–(C4) is only due to the rapidly changing behavior of f observed when certain entries of $\mathbf{x_{GN}}$ are near zero and by perturbing $\mathbf{x_{GN}}$ only slightly to obtain $\tilde{\mathbf{x}}_{\mathbf{GN}}$, (C1)–(C4) will be satisfied with $\mathbf{x} = \tilde{\mathbf{x}}_{\mathbf{GN}}$. We note that when we perturb $\mathbf{x_{GN}}$, the direction $\mathbf{d} = \tilde{\mathbf{x}}_{\mathbf{GN}} - \tilde{\mathbf{z}}$ is also changed. If $\mathbf{x} = \tilde{\mathbf{x}}_{\mathbf{GN}}$ satisfies (C1)–(C4), we take $\tilde{\mathbf{x}}_{GN}$ to be our next iterate. Otherwise, we check to see if $\mathbf{x_{GN}}$ at least satisfies (C1) and

(C5) $\eta > 10^{-4}$ - $m_{\tilde{\mathbf{z}}}$ is not a bad model of f near $\tilde{\mathbf{z}}$ (cf. (7)).

If this is not the case, we take the next iterate to be $\mathbf{z}$.

Line 19 - updating the damping and perturbation parameters: In many trust–region algorithms, the trust–region radius ρ is increased if the candidate for the next iterate satisfies conditions similar to (C1)–(C4). Therefore, if either (resp. both) $\mathbf{x} = \mathbf{x_{GN}}$ or (resp. and) $\mathbf{x} = \tilde{\mathbf{x}}_{\mathbf{GN}}$ satisfy (resp. do not satisfy) (C1)–(C4) and (resp. or) the PCG method converges (resp. does not converge) before exceeding a certain number of iterations, we decrease (resp. may increase) the damping parameter λ, which roughly corresponds to increasing (resp. decreasing) ρ. We note that the smaller λ is, the more iterations the PCG method requires to converge. Therefore, we do not decrease (and may even increase) λ if the PCG method requires too large of a number of iterations to converge during a given iteration of POPT, even if (C1)–(C4) are satisfied. Finally, we consider adjusting the perturbation parameter κ. If (C5) is satisfied, but (C1) is not satisfied, this suggests that moving in the generalized damped Gauss-Newton $\mathbf{d}$ did not decrease f by a sufficient amount to compensate for the amount that f was (likely) increased by moving from the Cauchy point $\mathbf{z}$ to the perturbed Cauchy point $\tilde{\mathbf{z}}$. Therefore, κ is understood to be too large and therefore decreased.

B. Implementation of CP–POPT–GDGN into ENSIGN

We implement POPT (cf. Algorithm 1) for shared memory systems into ENSIGN, using OpenMP to parallelize many of POPT's subroutines. All steps of POPT have ample opportunities for parallelization, with the exception of the step that involves computing a Cauchy point (cf. line 3). Fortunately, it does not take long to compute a Cauchy point in comparison to certain other steps of POPT; the main performance bottleneck is computing the generalized damped Gauss-Newton direction (cf. line 5), by solving the linear system in (6).

Recall that the system in (6) is solved via the PCG method using a Jacobi preconditioner. Since $\mathbf{D}$ is a diagonal matrix, the main challenge of computing the product of $(\hat{\mathbf{H}}+\lambda\mathbf{D})$ and a vector $\mathbf{p}$ is computing the product $\hat{\mathbf{H}}\mathbf{p}$, where we suppress the dependence of $\hat{\mathbf{H}}$ on $\tilde{\mathbf{z}}$. Let nnz denote the number of nonzeros in the sparse N-mode tensor $\mathcal{X}$ (cf. (3)). In view of [34, pp. 4456], $\hat{\mathbf{H}}$ can be written as the sum of nnz symmetric rank one matrices, where each rank one matrix can be written in terms of only $(NR + 1)$ parameters (recall R is the rank of the CP decomposition $\mathcal{M}$ (cf. (1) and (3))). Therefore, using a strategy similar to that in [4], we compute and store these $(NR+1)$ parameters for each nonzero before solving the linear system in (6) and avoid having to recompute these parameters each iteration of the PCG method. This technique enables a very efficient implementation of POPT.

Data set	Dimensions	No. of Non-zeros
Chicago	6,186 x 24 x 77 x 32	5,330,673
LANL1	1,433 x 22,077 x 534,687 x 58,389 x 11	40,266,345
LANL2	3,761 x 11,154 x 8,711 x 75,147 x 9	69,082,467
LBNL	1,605 x 4,198 x 1,631 x 4,209 x 868,131	1,698,825
Plant	562 x 124 x 123	8,370
RL1	279 x 248 x 1,757 x 10	46,591
RL2	1,512 x 433 x 4,568 x 4,743	314,276

TABLE I
TENSORS GENERATED FROM SEVEN DIFFERENT DATA SETS.

IV. COMPARISON OF DECOMPOSITION ACCURACY

We now consider the decomposition of seven different count tensors, formed from real–world data sets, using POPT and many of the algorithms listed in Section II-C. Table I gives a summary of these tensors; detailed data set descriptions are given in Appendix C. We note that the LANL1, LANL2, LBNL, RL1, and RL2 tensors are formed from network traffic data sets. Additionally, the LAN1 and LANL2 tensors are large–scale tensors, with tens of millions of nonzeros, created from NetFlow data. We compare the decomposition accuracy achieved by the various algorithms to demonstrate POPT's ability to obtain accurate CP decompositions.

We attempted to decompose each of the above tensors using POPT and each algorithm listed in Section II-C, with the exception of SmartCPD (no code was available) and GGN (required too much memory to decompose even the smallest network traffic tensor (RL1)), running ten decomposition trials for each tensor and algorithm pair, and starting each trial with a different random initial guess. In order to compare our results with those from [28], we performed rank 10 CP decompositions of the Chicago and Plant tensors. We also performed a rank 50 CP decompositions of the tensor, and rank 100 CP decompositions of the remaining tensors. All decompositions obtained with MU, PQNR, PDNR, and POPT, were computed using ENSIGN, with each trial run on a single node with 20 cores and 512 GB of memory; all decompositions computed with GCP were done in MATLAB, running on a desktop computer, with the GCP default settings. Note that

	MU	PQNR	PDNR	GCP	POPT
Chicago	4.2E-1	4.2E-1	4.2E-1	4.2E-1	4.2E-1
LANL1	**2.2E-11**	1.2E-10	6.3E-11	N/A	**2.2E-11**
LANL2	3.5E-9	4.6E-9	4.8E-9	N/A	**3.4E-9**
LBNL	3.1E-11	1.5E-10	1.6E-10	N/A	**2.6E-11**
Plant	7.4E-3	7.4E-3	7.4E-3	8.1E-3	7.4E-3
RL1	**2.5E-4**	2.8E-4	2.6E-4	7.7E-4	**2.5E-4**
RL2	5.9E-8	9.9E-8	6.9E-8	1.0E-6	**5.7E-8**

TABLE II
BEST VALUE OBTAINED FOR (3) WITH $\epsilon = 10^{-10}$ OUT OF TEN TRIALS, DIVIDED BY THE NUMBER OF TENSOR ENTRIES (LOWER IS BETTER).

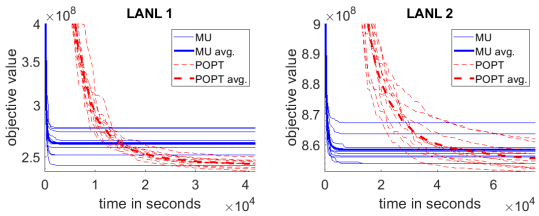

Fig. 2. Objective value (cf. (3)) with $\epsilon = 10^{-10}$ vs. time in seconds for each of the two large–scale NetFlow tensors from Table I obtained by MU and POPT. For each data set and algorithm, we plot the results for 10 different initial guesses (thin lines), as well as the average of those results (thick lines). Convergence to a lower objective value indicates better decomposition accuracy. Hence, POPT obtains better decomposition accuracy than MU.

we did not decompose some of the larger network traffic tensors (LANL1, LANL2, and LBNL) with GCP, as (i) computing such decompositions with GCP in MATLAB would be costly and (ii) based off of the relatively poor decomposition accuracy achieved by GCP for the smaller network traffic tensors (RL1 and RL2), we did not expect GCP to achieve accurate decompositions of the larger network traffic tensors. To ensure a fair comparison between the algorithms, we ran all algorithms (except GCP) for the same amount of time. To determine how much time to run the algorithms for each data set, we first ran ten trials of POPT for 200 iterations, and then allowed the other algorithms to run for the maximum amount of time taken by any of those trials. The length of time for running decompositions of the various tensors for ranged from slightly more than a second (for the Plant tensor) to roughly 20 hours (for the LANL2 tensor). Because we were running the less efficient MATLAB implementation of GCP on different hardware than the other algorithms, GCP was allowed to run until convergence, regardless of how much time that required.

The best objective value obtained for (3) out of the ten trials, divided by the number entries (including zeros) of the tensor $\mathcal{X}$, is recorded in Table II. For all seven tensors, POPT either obtained or tied for the best decomposition accuracy, out of all five algorithms in that table. In view of [28, Section V-B], POPT obtains better decomposition accuracy than GGN and SmartCPD for the Chicago and Plant tensors as well. To further illustrate the effectiveness of POPT in decomposing large tensors for network traffic applications, we also plot the objective values (cf. (4)) obtained by MU and POPT over time for the LANL1 and LANL2 tensors, in Figure 2. Since, PQNR and PDNR obtained poor decomposition accuracy relative to MU and POPT for the LANL tensors, we chose not to include the results for those algorithms in our plots.

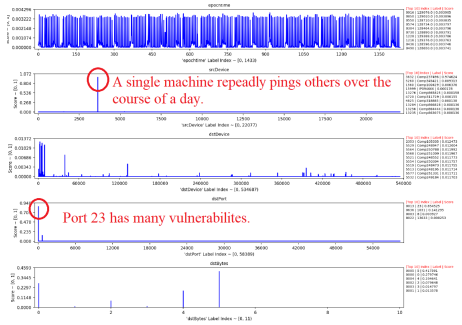

Fig. 3. Potentially suspicious behavior: one source device repeatedly pings several destination devices over port 23 (Telnet) throughout the day.

V. LATENT BEHAVIOR DETECTION

In this section, we discuss how, for the two large–scale NetFlow data sets (LANL1 and LANL2), POPT detected behaviors that went undetected by MU, but MU did not detect any behaviors undetected by POPT. We do not consider the latent behaviors detected by PQNR, PDNR, and GCP, as these algorithms achieved relatively poor decomposition accuracy (cf. Table II). We also do not discuss the smaller tensors (cf. Table I), but note that (i) for some of these tensors neither MU, nor POPT detected behaviors that went undetected by other, and (ii) for the remaining tensors, both MU and POPT detected behaviors that the other did not detect.

For two CP decompositions $\mathcal{M}_1$ and $\mathcal{M}_2$ of the same tensor, we say that a behavior given by one of the components of $\mathcal{M}_1$ is also detected by $\mathcal{M}_2$, if the entry–wise cosine similarity between that component and $\mathcal{M}_2$ exceeds a certain threshold. For each tensor (LANL1 and LANL2) and algorithm (MU and POPT), we select the most accurate decomposition out of each of those computed in the ten trials (cf. Section IV), in order to see which behaviors were captured by the two algorithms. Setting a threshold of 10^{-6}, we see that POPT detected one (resp. two) behavior(s) that MU did not detect in the LANL1 (resp. LANL2) data set. However, for both data sets, MU did not detect any behaviors other than those detected by POPT.

The component representing the behavior detected by POPT, but not MU, for the LANL1 data set, is displayed in Figure 3. The behavior illustrated by this component is particularly interesting: we see how a single machine repeatedly pings several other machines many times over the course of a day on port 23. This pattern is potentially suspicious due to its repetitive nature and the fact that port 23 (used for Telnet) has many vulnerabilities. The fact that POPT, but not MU, detected this potentially suspicious behavior highlights the utility of using POPT to compute count tensor decompositions.

VI. SUMMARY

In this paper, we developed and implemented an all–at–once optimization CP count tensor decomposition algorithm that uses a generalized damped Gauss-Newton method, CP–POPT–GDGN. This algorithm often outperforms other widely used and state–of–the–art CP count tensor decomposition algorithms in terms of accuracy and latent behavior detection.

REFERENCES

[1] E. Acar, S. A. Camtepe, M. S. Krishnamoorthy, and B. Yener, "Modeling and multiway analysis of chatroom tensors," *Proc. IEEE Int. Conf. Intell. and Secur. Inform.*, Atlanta, GA, USA, May 19–20, 2005, pp. 256–268.

[2] B.W. Bader, T. G. Kolda, et. al., "Tensor Toolbox for MATLAB, Version 3.2.1," Accessed: Jul. 7, 2021. [Online]. Available: https://www.tensortoolbox.org.

[3] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sept. 10–12, 2012.

[4] M. Baskaran et al., "Memory-efficient parallel tensor decompositions," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sept. 12–14, 2017.

[5] M. Baskaran, T. Henretty, and J. Ezick, "Fast and scalable distributed tensor decompositions," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sept. 24–26, 2019.

[6] J. D. Carroll and J.–J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[7] A. Cichocki and A.-H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Trans. on Fundam. of Electron., Commun., and Comput. Sci.*, vol. 92, no. 3, pp. 708–721, 2009.

[8] E. C. Chi and T. G. Kolda, "On tensors, sparsity, and nonnegative factorizations," *SIAM J. Matrix Anal. and Appl.*, vol. 33, pp. 1272–1299, 2013.

[9] *ENSIGN Tensor Toolbox*, Reservoir Labs. Accessed: Jul. 6, 2021. [Online]. Available: https://www.reservoir.com/ensign/

[10] B. Ermis, E. Acar, and A.T. Cemgil, "Link prediction in heterogeneous data via generalized couped tensor factorizations," *Data Min. and Knowl. Disc.*, vol. 29, pp. 203–236, 2015.

[11] J. Ezick et. al., "Combining tensor decompositions and graph analytics to provide cyber situational awareness at HPC scale," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sept. 24–26, 2019.

[12] S. S. Ghosal, "Travel Time Prediction of Taxis using Tensor Factorization," *Proc. ACM India Joint Int. Conf. on Data Sci. and Manage. of Data*, Kolkata, India, Jan. 3–5, 2019.

[13] S. Hansen, T. Plantenga, and T. G. Kolda, "Newton-based optimization for Kullback-Leibler nonnegative tensor factorizations," *Optim. Methods and Softw.*, vol. 30, no. 5, pp. 1002–1029, 2015.

[14] T. Henretty, M. Baskaran, J. Ezick, D. Bruns–Smith; T.A. Simon, "A quantitative and qualitative analysis of tensor decompositions on spatiotemporal data," in *IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sept. 12–14, 2017.

[15] L.T.K. Hien and N. Gillis, "Algorithms for nonnegative matrix factorization with the Kullback-Leibler divergence," *J. Sci. Comput.*, vol. 87, 2021.

[16] D. Hong, T. G. Kolda, and J. A. Duersch, "Generalized canonical polyadic tensor decomposition," *SIAM Rev.*, vol. 62, no. 1, pp. 133–163, 2020.

[17] J.A. Jones, R.A. Hutchinson, and V.W. Pfeiffer, "Plant pollinator data at HJ Andrews Experimental Forest, 2011 to 2015," *Environmental Data Initiative*, ver. 5. Accessed: Jul. 15, 2021. [Online]. Available: https://doi.org/10.6073/pasta/e482b510e3ce8ee6a62b39a6a9c308e6/

[18] N. Kargas and N.D. Sidiropoulos, "Learning mixtures of smooth product distribution: identifiability and algorithm," *Proc. Int. Conf. Artif. Intell. and Statist.*, vol. 89, pp. 388–396, 2019.

[19] A.D. Kent, "Cybersecurity data sources for dynamic network research," in *Dynamic Networks and Cyber–Security*, World Scientific, 2015, ch. 2, pp. 37-65.

[20] T. G. Kolda and B. W. Bader, "Tensor decomposition and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455-500, 2009.

[21] T.G. Kolda and D. Hong, "Stochastic gradients for large-scale tensor decomposition" *SIAM J. Math. Data Sci.*, vol. 2, no. 4, pp. 1066-1095, 2020.

[22] D. Krompaβ, M. Nickel, X. Jiang, and V. Tresp, "Non-negative tensor factoriztion with RESCAL," *Tensor Methods for Mach. Learn.*, ECML workshop, 2013.

[23] J. Nocedal and S.J. Wright, "Numerical Optimization," Springer, 2006.

[24] R. Pang, M. Allman, V. Paxson, and J. Lee, "The devil and packet trace anonymization," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 29–38, 2006.

[25] A.-H. Phan, P. Tichavsky, and A. Cichocki, "Fast damped Gauss-Newton method for sparse and nonnegative tensor factorization," in *Proc. IEEE Int. Conf. on Acoust., Speech and Signal Process. (ICASSP)*, Prague, Czech Republic, 2011, pp. 1988–1991.

[26] A.-H. Phan, P. Tichavsky, and A. Cichocki, "Low complexity damped Gauss–Newton algorithms for CANDECOMP/PARAFAC," *SIAM J. Matrix Anal. and Appl.*, vol. 34, no. 1, pp. 126–147, 2013.

[27] E.T. Phipps, T.G. Kolda, D. Dunlavy, G. Ballard, and T. Plantenga, "Genten: software for generalized tensor decompositions v. 1.0.0," Accessed: Jul. 7, 2021. [Online]. Available: https://www.osti.gov//servlets/purl/1373350

[28] W. Pu, S. Ibrahim, X. Fu, and M. Hong, "Stochastic mirror descent for low-rank-tensor decomposition under non-Euclidean losses," *arXiv preprint arXiv:2104.14562*, 2021.

[29] T.M. Ranadive and M.M. Baskaran, "Large-scale sparse tensor decomposition using a damped Gauss-Newton Method," *Proc. IEEE High Perform. Extreme Comput. Virtual Conf.*, Sept. 21–25, 2020.

[30] S. Smith et. al., *FROSTT: The formidable repository of open sparse tensors and tools*, 2017. [Online]. Available: http://frostt.io/

[31] M. Sugiyama, H. Nakahara, and K. Tsuda, "Legendre decomposition for tensors," in *Proc. Adv. in Neural Inform. Process. Syst.*, vol. 31, 2018.

[32] M. Turcotte, A. Kent and C. Hash, "Unified host and network data set," in *Data Science for Cyber-Security*, World Scientific, Nov. 2018, ch. 1, pp. 1-22.

[33] P. Tichavsky, A.-H. Phan and A. Cichocki, "A further improvement of a fast damped Gauss-Newton algorithm for CANDECOMP-PARAFAC tensor decomposition," in *Proc. IEEE Int. Conf. on Acoust., Speech and Signal Process. (ICASSP)*, Vancouver, Canada, 2013, pp. 5964–5968.

[34] M. Vandecappelle, N. Vervliet, and L. D. Lathauwer, "A second-order method for fitting the canonical polyadic decomposition with non-least-squares cost," *IEEE Trans. Signal Process.*, vol. 68., pp. 4454–4465, 2020.

[35] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, "Tensorlab 3.0," Accessed: Jul. 7, 2021. [Online]. Available: https://www.tensorlab.net/.

[36] S. Zafeiriou and M. Petrou, "Nonnegative tensor factorization as an alternative Csiszar–Tusnady procedure: algorithms, convergence, probabilistic interpretations and novel probabilistic tensor latent variable analysis algorithms," *Data Min. Knowl. Disc.*, vol. 22, pp.419–466, 2011.

[37] X. Zhang and M.K. Ng, "Sparse nonnegative tensor factorization and completion with noisy observations," *arXiv preprint arXiv:2007.10626*, 2021.

APPENDIX A
PREVIOUS DECOMPOSITION ALGORITHMS

In this section, we provide further information on the algorithms MU, PQNR, PDNR, SmartCPD, GCP, and GGN (cf. Section II-C).

MU (Multiplicative Update) [8, Algorithm 3] is an alternating algorithm that optimizes (3) over a single factor matrix, while fixing the remaining factor matrices. It alternates optimizing over all factor matrices, optimizing over each factor matrix once per iteration. To optimize (3) over a given $\mathbf{A}^{(\mathbf{n})}$, MU uses a multiplicative update approach, while 'scooching' certain entries of $\mathbf{A}^{(\mathbf{n})}$ away from zero as needed. The extremely efficient ENSIGN implementation of MU uses techniques from [4] to quickly compute certain columns of the Khatri–Rao product

$$\bigodot_{m \neq n} \mathbf{A}^{(\mathbf{m})} := \mathbf{A}^{(\mathbf{N})} \odot \cdots \odot \mathbf{A}^{(\mathbf{n+1})} \odot \mathbf{A}^{(\mathbf{n-1})} \odot \cdots \odot \mathbf{A}^{(\mathbf{1})},$$

which are repeatedly used while optimizing (3) over $\mathbf{A}^{(\mathbf{n})}$.

PQNR (Poisson, Quasi-Newton, Row update) [13] is an alternating algorithm that uses a quasi-Newton algorithm to optimize (3) over a given factor matrix. While optimizing (3) over $\mathbf{A}^{(\mathbf{n})}$, PQNR takes advantage of the separable nature of the objective that is obtained once the other factor matrices are fixed and updates each row of $\mathbf{A}^{(\mathbf{n})}$ independently from the others. Unfortunately, PQNR is unable to take advantage of the techniques from [4] as MU is. Consequently, the ENSIGN version of PQNR requires more time to converge than the ENSIGN version of MU, even though the opposite holds true for other versions of these algorithms [13].

PDNR (Poisson, Damped-Newton, Row update) [13] is another alternating algorithm, which uses a damped-Newton method to optimize (3) over a given $\mathbf{A}^{(\mathbf{n})}$. It also takes advantage of the separable nature of the objective and updates each row of $\mathbf{A}^{(\mathbf{n})}$ independently of the others. In ENSIGN, PDNR, like PQNR, takes longer to converge than MU.

SmartCPD [28] is a stochastic alternating algorithm that utilizes stochastic mirror descent and clever sampling schemes to solve CP decomposition optimization problems such as (4). We have been unable to find any published code for SmartCPD.

GCP [16] is an all–at–once optimization algorithm that uses stochastic gradient descent to compute CP decompositions and can be used to solve (4).

GGN [34] is an all–at–once optimization algorithm that uses a generalized damped Gauss-Newton method to solve optimization problems related to CP decomposition including (4).

Appendix B
Additional Details on CP–POPT–GDGN

In this Section, we give further details on CP–POPT–GDGN (cf. Algorithm 1 and Section III), specifically on (i) parameter choices and (ii) how perturbations are performed.

In line 1 of Algorithm 1, we set the initial damping parameter $\lambda = 10^3$ and the initial perturbation parameter $\kappa = 10^{-4}$ (resp. $\kappa = 10^{-5}$) for network traffic (resp. non-network traffic) data sets.

In line 4, we perturb entries of the Cauchy point $\mathbf{z}$ to obtain $\tilde{\mathbf{z}}$. The point $\tilde{\mathbf{z}}$ is given precisely by

$$\tilde{\mathbf{z}}_i = \begin{cases} \kappa \text{ if } \mathbf{z}_i < \kappa \text{ and } (\mathbf{g}(\mathbf{z}))_i < 0, \text{ and} \\ \mathbf{z}_i \text{ otherwise.} \end{cases} \tag{8}$$

Suppose $\{\mathbf{A}^{(\mathbf{n})}\}_{n\in[N]}$ denotes the collection of factor matrices that, when reshaped, form $\tilde{\mathbf{z}}$. The goal of setting certain entries of $\tilde{\mathbf{z}}$ equal to κ is to avoid having any $m_{\mathbf{i}}$ (cf. (2)) near zero, for which $x_{\mathbf{i}} \neq 0$, so that f_ϵ does not change too rapidly near $\tilde{\mathbf{z}}$, in view of the $\log$ function in (3). In practice, the scheme from (8) tends to achieve this goal. Note that (8) is preferable to setting all $\tilde{\mathbf{z}}_i$, for which $\mathbf{z}_i < \kappa$, equal to κ, as the latter strategy prevents us from obtaining sparse factor matrices. In line 9 $\mathbf{x_{GN}}$ is perturbed in a similar manner.

In line 19, we update λ and κ. If, during a given iteration of CP–POPT–GDGN, conditions (C1)–(C4) (cf. Section III) are satisfied with either $\mathbf{x} = \mathbf{x_{GN}}$ or $\mathbf{x} = \tilde{\mathbf{x}}_{\mathbf{GN}}$, and the PCG method converges in 50 or fewer iterations, we decrease λ by a factor of 1.25. If instead $\eta < 0.25$ (cf. (7)), or the PCG method does not converge in 50 iterations, we increase λ by a factor of 2.5. If κ is to be decreased, we reduce κ by a factor of $10^{1/2}$, but do not allow κ to fall below 10^{-6}.

Appendix C
Data Set Descriptions

In this section, we provide additional details on the data sets used to form the tensors in Section IV (cf. Table I).

The Chicago Crime data set is a collection of crime reports made in Chicago, for crimes occurring between January 1st, 2001 and December 11th, 2017. It is available at `www.cityofchicago.org`. This data set was used to form a tensor downloaded from [30], with modes corresponding to the attributes day, hour, community, and crime type.

The two LANL tensors were formed from publicly available anonymized NetFlow data sets published by Los Alamos National Laboratories [19], [32]. The first data set [32] contains NetFlow data collected over 90 days; we use the data from just one of those days to form the first LANL tensor. This tensor has modes corresponding to timestamp (binned by minute), source device, destination device, destination port, and number of bytes transferred (binned in logarithmic scale). The second data set [19] contains NetFlow data collected over 58 days. We use all of this data to form the second LANL tensor, with the same mode attributes as first LANL tensor, although the timestamps for this tensor are binned by ten minute intervals. Both of these tensors were formed using ENSIGN's tensor construction utility (`csv2tensor`).

The LBNL network data set [24] was used to form a tensor downloaded directly from [30]. This tensor has modes corresponding to the attributes source IP, source port, destination IP, destination port, and timestamp (binned by second).

The Plant-Pollinator data set [17] documents observed interactions between various species of plants and pollinators during the summers of 2011-2015, at the HJ Andrews Experimental Forest. (Note that this is not the most up-to-date version of this data set. We use this version so we can compare our results to those in [28]). We form a tensor whose modes correspond to the fields plant name, pollinator name, and day, where we throw out any records in the data set that do not include either the plant name or pollinator name. This tensor was also formed with ENSIGN's `csv2tensor` utility.

The two Reservoir Labs (RL) tensors are formed from network traffic data from Reservoir Labs' internal network. The first RL tensor has modes corresponding to source IP, destination IP, destination port, and number of bytes transferred (binned in logarithmic scale), while the second has modes corresponding to timestamp (binned by minute), source IP, destination IP, and destination port. These two tensors were formed with ENSIGN's `csv2tensor` utility as well.