TCNN: a Tensor Convolutional Neuro-Network for big data anomaly detection

Xiaolong Wu

***Abstract*—Big Data consists of multidimensional, multi-modal datasets that are so huge and complex that require new methods to process and these data. Nowadays, network data expand at a huge speed. And consequential network traffic anomaly detection, an important part of network security, appears some constrains with existing methods such as these systems ignore the relationship value between different attributes of network traffic which incur less accuracy problem; Lots of memory and computational cost, which hinders them from running in relatively low-end smart devices such as smart phones. Motivated by this, this work proposes a new method named Tensor Convolutional Neuro-Network (TCNN) to resolve network traffic anomaly detection problem which utilize the tensor decomposition technology to decompose the convolution layer and reduce the memory cost. What's more, consider the performance requirement of big data process, we improve the CP decomposition algorithm, which is one of the most critical parts for tensor decomposition performance. The experiment results show our improvement achieve better performance as well as comparable accuracy.**

***Index Terms*—Anomaly detection, Tensor Factorization, CP decomposition, Parallel computing**

I. Introduction

As the development of sciences and economics, magnanimous, dynamic, and complicated structured data become universal. Big Data consists of multidimensional, multi-modal datasets that are so huge and complex that they cannot be easily stored or processed by using standard computers. Except "5V" (High Volume, Velocity, Variety, Veracity, Value) characteristics of big data, real applications also have properties of high complexity, that is data are constructed from multi-source and multi-aspect, and the irregularity and incompleteness of data, Variety - different forms of data, Veracity - uncertainty of data, and Velocity - analysis of streaming data, comprise the challenges ahead of us (Source from [4]).

Fig. 1. 5V characteristics of big data

Xiaolong Wu is with School of Computer Science Virginia Tech, Blacksburg, VA, USA e-mail: xlwu@vt.edu.

Many problems in computational neuroscience, neuroinformatics, pattern/image recognition, signal processing and machine learning generate massive amounts of multidimensional data with multiple aspects and high dimensionality. Network traffic anomaly detection constitutes an important part of network security. However, existing network traffic anomaly detection methods all ignore the relationship between multidimensional data. For example, anomaly detection system SPADE [7], ADAM [1], or NIDES [6] models normal network traffic, usually the distribution of IP addresses and ports, and in fact use the matrix as the input with attribute in one dimension and record as the other dimension, which ignore the relationship value between different attribute.

On one hand, traditional matrices and vectors are not enough to represent this complex property, tensors as new approach has obtained broad abstraction from both academic and industrial areas. Multi-linear tensor-based data analysis becomes a research hotspot in recent years. One of the emerging technologies is tensor decomposition; CP decomposition is the most common used tensor decomposition methods.

On the other hand, this work proposes to use Convolutional neural networks (CNNs) to resolve network traffic anomaly detection problem. In deep convolutional neural networks, the output of each layer is a tensor, but the barrier here is that CNNs will require high amounts of memory and computational resources, which are unfortunately hard to be met by small-sized smart devices such as mobile phones . In order to tackle this problem, we propose to decompose covolutional layers of CNNs by using tensor decomposition. What's more, consider the performance requirement of big data process, we improve the CP decomposition algorithm, which is one of the most critical parts for tensor decomposition performance. We named our new convolutional neural network framework as TCNN (Tensor Convolutional Neural Network).

Several recent papers apply decompositions for either initialization [17] or post-training [13]. More recently, Anima's group make an attempt that apply tensor contractions as a generic layer directly on the activations or weights of a deep neural network and to train the resulting network end to-end [11]. Different from this work, our work decomposes the convolution layer instead of fully connected layer.

Considering the computation time of CP decomposition is also very important for the training procedure of tensor convolutional neural networks, we identify the bottleneck of CP composition with Alternating Least Squares method (CP-ALS) and propose an improved algorithm and parallel the new algorithm to further speed up the performance.

The main contributions are as follows:

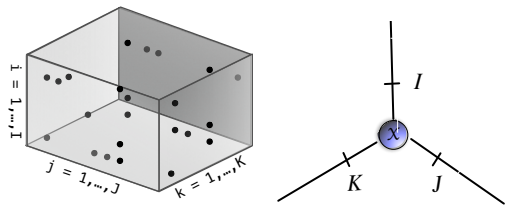


Fig. 2. (a) A third-order sparse tensor; (b) Tensor network diagram for a third-order tensor.

- we use CP decomposition algorithm to decompose the convolution layer of CNN in order to reduce of spacial overhead.
- We introduce a CP decomposition algorithm with Alternating Least Squares method (CP-ALS) and identify its bottleneck (MTTKRP operation) and further parallel it.
- We models the anomaly detection in network traffic using 3D multidimensional dataset and propose a framework named TCNN to do the network traffic anomaly detection, which consider the relationship value between different attributes of network traffic and reduce memory overhead as well as performance overhead.

The remainder of this paper is organized as follows. Section 2 describes some basic knowledge of tensors and tensor decomposition. Using CP decomposition to decompose the convolution layer of CNNs is shown in section 3. Section 4 identify the bottleneck of CP decomposition and Section 5 introduces our improved CP decomposition algorithm. We give our experimental results in section 6, and conclude this paper in section 7.

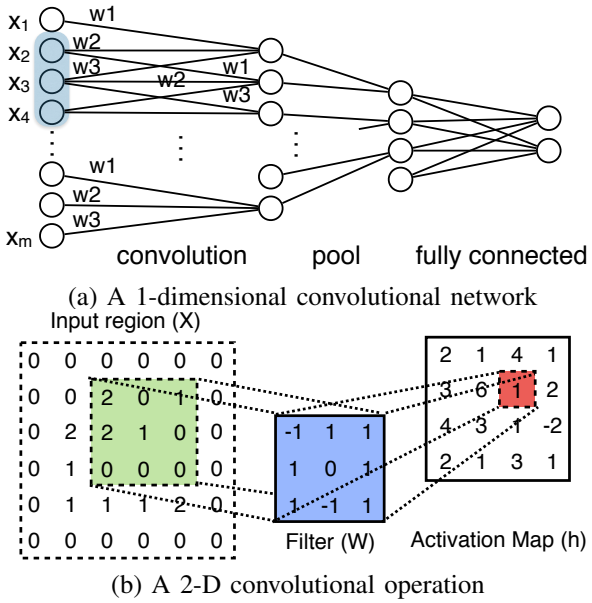
II. Background

We first introduce the essential tensor notation and operations for describing our algorithms. Several examples and definitions are drawn from the excellent overview by Kolda and Bader [10].

A. Tensors and Operations

For our purposes a sparse tensor is defined as a multi-way sparse array, illustrated graphically in Fig. 3, and denoted by a bold capital Calligraphic letter, e.g., $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. Figure 3(b) shows the tensor network diagram [5] of a third-order tensor, where each line represents a mode and their size. The *order* of a tensor is its number of dimensions or modes, which in this example is 3, whereas matrices and vectors have order 2 and 1 respectively. Matrices are denoted by boldface capital letters, e.g., $\mathbf{A} \in \mathbb{R}^{I \times J}$, and vectors by boldface lowercase letters, e.g., $\mathbf{x}$. The scalar elements of a tensor are denoted by lowercase letters, x_{ijk} is the (i, j, k) element of $\mathcal{X}$. Often, a tensor needs to be reordered into a matrix, which is called matricization or unfolding. Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ can be matricized to matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times I_1 \ldots I_{n-1} I_{n+1} \ldots I_N}$.

The ***Kronecker product*** of two tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathcal{Y} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$ is denoted by $\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y} \in \mathbb{R}^{I_1 J_1 \times I_2 J_2 \times \cdots \times I_N J_N}$. The ***Khatri-Rao product*** is the "matching columnwise" Kronecker product between two matrices.



(a) A 1-dimensional convolutional network

(b) A 2-D convolutional operation

Fig. 3. CNN illustration.

Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is denoted by $\mathbf{C} = \mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{IJ \times R}$.

The ***Hadamard product*** is an element-wise product. Given two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{I \times R}$, Hadamard product is denoted by $\mathbf{C} = \mathbf{A} * \mathbf{B}$, with each entry computed by

$$c_{i,r} = a_{i,r} b_{i,r}. \tag{1}$$

To simply our algorithm below, we extend the Hadamard product to an operation between a tensor and a matrix and introduce a reduction on one of the matrix modes. We call this operation ***Tensor-Matrix Hadamard Reduction product*** **(TMHR)**. Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N-1} \times I_N}$ and $\mathbf{M} \in \mathbb{R}^{I_{N-1} \times I_N}$, the TMHR product is denoted by

$$\mathbf{z}(i_1, \ldots, i_{N-2}, :) = \sum_{i_{N-1}=1}^{I_{N-1}} \mathbf{X}(i_1, \ldots, i_{N-2}, :, :) * \mathbf{M}. \tag{2}$$

By fixing $i_1, \ldots, i_{N-2}$, each matrix $\mathbf{X}(i_1, \ldots, i_{N-2}, :, :) \in \mathbb{R}^{I_{N-1} \times I_N}$ undergoes a Hadamard product with matrix $\mathbf{M}$. Then we sum-reduce over the first matrix mode (mode-I_{N-1}). This equation generates a $(N-1)$th-order tensor $\mathcal{Z} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N-2} \times I_N}$. Note that TMHR only operates over the nonzero entries of a sparse tensor.

The ***Matriced Tensor Times Khatri-Rao product*** **(MTTKRP)** is a core operation of CPD (introduced later). For a Nth-order tensor, its formulation on mode-n is:

$$\mathbf{M_A}^{(n)} \leftarrow \mathbf{X}_{(n)} \left(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)} \right), \tag{3}$$

where $\mathbf{X}_{(n)}$ is a sparse matricization of tensor $\mathcal{X}$ by mode-n, and $\mathbf{M_A}^{(n)} \in \mathbb{R}^{I_n \times R}, \mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$ are matrices. For efficiency, we generally perform the MTTKRP operation by operating directly on the nonzeros of sparse tensor $\mathcal{X}$ without explicitly matricizing and inherently multiply the matrices, avoiding explicit Khatri-Rao products (details in § IV).

B. CNN

CNN is a class of networks that finds non-linear models of patterns in inputs and makes data classications. The most popular type of CNNs used in image recognition are convolutional neural networks (CNNs) [15, 18]. A CNN usually consists of a stack of layers of nodes. In Figure 2 (a), the leftmost layer is the input level, with each node representing one element in the input vector (e.g., the gray value of a pixel in an input image), the rightmost is the output level, with each node representing the predicted probability for the input to belong to one of two classes. The output layer of a CNN gives the final prediction, while the other layers gradually extract out the critical features from the input. A CNN may consist of mixed types of layers, some for subsampling results (pooling layers), some for non-linear transformations. Convolution layers are of the most importance, in which, convolution shifts a small window (called a filter) across the input, and at each position, it computes the dot product between the filter and the input elements covered by the filter, as Figure 2 (b) shows in a 2-D case. In Figure 2 (a), the weights of every three edges connecting three input nodes with one layer-2 node form the filter ¡ w1, w2, w3 ¿ at that level. The result of a convolution layer is called an activation map. Multiple filters can be used in one convolution layer, which will then produce multiple activation maps. The last layer (i.e., the output layer) usually has a full connection with the previous layer.

Part of the CNN training process is to determine the proper values of the parameters in the filters (i.e., weights on the edges of the networks). In training, all the parameters in the network are initialized with some random values, which are re ned iteratively by learning from training inputs. Each training input has a label (e.g., the ground truth of its class). The forward propagation on an input through the network gives a prediction; its difference from its label gives the prediction error. The training process (via back propagation) revises the network parameters iteratively to minimize the overall error on the training inputs. In using CNN, only for- ward propagation is needed to get the prediction. Note that the size of the inputs to a CNN is typically fixed, equaling to the number of nodes in its input layer. If the raw inputs are of different sizes, they have to be normalized to the unified size.

III. TCNN

For the first layer, the input is irregular data set, different from general regular data stored in matrix form, the attributes in these irregular data set has relationship from each other. For our example, it is the network traffic data set. And the data record is a 3D topology attributes as (SourceIP, DestinationIP, Port); The output of tensor decomposition is a core tensor and three matrixes, which record the relationship value between SourceIP and DestinationIP, between SourceIP and Port and between DestinationIP and Port.

In the convolutional layer, CP decomposes a tensor as a linear combination of rank one tensors. In general, convolution layers in CNNs map a 3-way input tensor $\mathcal{X}$ of size $S \times W \times H$ into a 3-way output tensor $\mathcal{Y}$ of size $S \times W' \times H'$ using a 4-way kernel tensor $\mathcal{K}$ of size $T \times S \times D \times D$ with $\mathcal{T}$ corresponding to different output channels, $\mathcal{S}$ corresponding to different input channels, and the last two dimensions corresponding to the spatial dimension (for simplicity, we assume square shaped kernels and odd D)

$$\mathcal{Y}_{t,w',h'} = \sum_{s=1}^{S}\sum_{d'=1}^{D}\sum_{d=1}^{D}\mathcal{K}_{t,s,d',d}\mathcal{X}_{s,w_{d'},h_d} \tag{4}$$

$w_j = (w'-1)\Delta + d' - p$ and $h_i = (h'-1)\Delta + d - p$, where Δ is stride and p is zero-padding size.

CP decomposition: Now the problem is to approximate the kernel tensor $\mathcal{K}$ with rank-R CP-decomposition. This can be represented as in (3). Spatial dimensions are not decomposed as they are relatively small (e.g., 3×3 or 5×5).

$$\mathcal{K}_{t,s,d',d} = \sum_{r=1}^{R}\mathbf{U}^{(1)}_{r,s}\mathcal{U}^{(2)}_{r,d',d}\mathbf{U}^{(3)}_{t,r} \tag{5}$$

where $\mathbf{U}^{(1)}_{r,s}, \mathcal{U}^{(2)}_{r,d',d}, \mathbf{U}^{(3)}_{t,r}$ are the three components of sizes $R \times S$, $R \times D \times D$, and $T \times R$, respectively.

Substituting (5) into (4) and performing simple manipulations gives (6) for the approximate evaluation of the convolution (4) from the input tensor $\mathcal{X}$ into the output tensor $\mathcal{Y}$.

$$\mathcal{Y}_{t,w',h'} = \sum_{r=1}^{R}\mathbf{U}^{(3)}_{t,r}\Big(\sum_{d'=1}^{D}\sum_{d=1}^{D}\mathcal{U}^{(2)}_{r,d',d}\Big(\sum_{s=1}^{S}\mathbf{U}^{(1)}_{r,s}\mathcal{X}_{s,w_{d'},h_d}\Big)\Big) \tag{6}$$

Equation (6) tells us that the output tensor Y is computed by a sequence of three separate convolution operations from the input tensor X with smaller kernels:

$$\mathcal{Z}_{r,w,h} = \sum_{s=1}^{S}\mathbf{U}^{(1)}_{r,s}\mathcal{X}_{s,w,h} \tag{7}$$

$$\mathcal{Z}'_{r,w',h'} = \sum_{j=1}^{D}\sum_{i=1}^{D}\mathbf{U}^{(2)}_{r,s}\mathcal{Z}_{t,w_j,h_i} \tag{8}$$

$$\mathcal{Y}_{t,w',h'} = \sum_{r=1}^{R}\mathbf{U}^{(3)}_{t,r}\mathcal{Z}'_{t,w',h'} \tag{9}$$

where $\mathcal{Z}_{r,w,h}$ and $\mathcal{Z}'_{r,w',h'}$ are intermediate tensors of sizes $R \times W \times H$ and $R \times W' \times H'$, respectively.

IV. CP Decomposition and Its Bottleneck

We introduce a CP decomposition algorithm with Alternating Least Squares method (CP-ALS) and identify its bottleneck (MTTKRP operation) in this section. Our motivation is through optimizing MTTKRP operation to improve the overall CP-ALS's performance.

The CANDECOMP/PARAFAC Decomposition (CPD) [10] decomposes a tensor as a sum of component rank-one tensors. For example, a third-order CPD of $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ (Figure 4), is approximated as

$$\mathcal{X} \approx [\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \equiv \sum_{r=1}^{R}\lambda_r \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r, \tag{10}$$

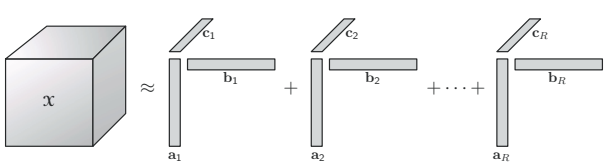
where R is the approximate rank of tensor $\mathcal{X}$, which is usually set to a small number (e.g. 10) for the purpose of low-rank approximation. $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$ are vectors which combine to dense matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ [10], and λ contains the weights when normalizing each vector to length one.

While tensor decompositions are ideal for analyzing multi-modal relationships in high-order data, this comes with the challenge of facing the 'curse of dimensionality', the exponential increase in time and storage as dimension increases. Compared to other tensor decompositions such as Tucker, CPD is more scalable as order increases for both time and storage complexity [5].

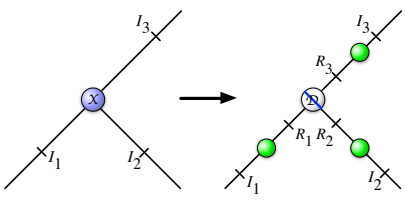
In a typical data analysis application we are interested in uncovering latent structures through a low-rank decomposition. This means fixing a rank R and computing the approximation $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$. One of the most popular algorithm is CP-ALS [10], which fixes all factor matrices except one to update that factor matrix, iterating until some convergence criterion is satisfied. This process for a Nth-order tensor is shown in Algorithm 1 and depicted in Figure 4

Algorithm 1 CP-ALS algorithm for an Nth-order sparse tensor.

Input: An Nth-order sparse tensor $\mathcal{X} \in R^{I_1 \times I_2 \times \cdots \times I_N}$ and an integer rank R;
Output: A sequence of dense factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$, $\mathbf{A}^{(i)} \in R^{I_i \times R}$;
1: Initialize $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$;
2: **do**
3: **for** $n = 1, 2, \ldots, N$ **do**
4: $\mathbf{V} \leftarrow \mathbf{A}^{(1)\dagger}\mathbf{A}^{(1)} * \cdots \mathbf{A}^{(n-1)\dagger}\mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)\dagger}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)\dagger}\mathbf{A}^{(N)}$
5: $\mathbf{A}^{(n)} \leftarrow \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})\mathbf{V}^{\dagger}$
6: Normalize columns of $A^{(n)}$ and store the norms as λ
7: **end for**
8: **while** Fit ceases to improve or maximum iterations exhausted.
9: **Return:** $[\![\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]$;



(a) Graphical representation of tensor and matrix format



(b) Tensor network diagram representation

Fig. 4. CP decomposition on a third-order tensor.

Based on our analysis below and experiments (not shown here), the bottleneck of CP-ALS is the MTTKRP sequence. For a third-order sparse tensor, MTTKRP computations make up 81% of total running time of its CPD. Assuming a Nth-order sparse cubical (equal mode size) tensor $\mathcal{X} \in \mathbb{R}^{I \times I \times \cdots \times I}$, with m nonzeroes, the time complexity of each iteration of CP-ALS is:

$$T_{CP} \approx N(T_M + NIR^2), \tag{11}$$

where T_M is the time complexity of MTTKRP, equal to $O(mR)$ [2, 3, 9, 15]. Generally, $R, N < 100$, $I << m$ for high-order tensors, thus $NIR^2 << T_M$. The runtime of CP-ALS is dominated by MTTKRP sequence,

$$T_{CP} = O(NT_M) = O(NmR). \tag{12}$$

This work focuses on parallelizing MTTKRP operation to speedup CP-ALS algorithm.

V. Sequential MTTKRP

MTTKRP operation of a third-order sparse tensor on the first mode is:

$$\mathbf{M}_{\mathbf{A}} = \mathbf{X}_{(\mathbf{1})}(\mathbf{C} \odot \mathbf{B}), \tag{13}$$

the scalar formulation of which is:

$$\mathbf{M}_{\mathbf{A}}(i, r) = \sum_{j=1}^{J} \sum_{k=1}^{K} \mathcal{X}(i, j, k)\mathbf{C}(k, r)\mathbf{B}(j, r)). \tag{14}$$

The straight-forward algorithm is first computing the Khatri-Rao product of two dense matrices $\mathbf{C}$ and $\mathbf{B}$, then doing a sparse matrix-dense matrix multiplication with the matricized sparse matrix $\mathbf{X}_{(\mathbf{1})}$. The output is stored as a dense matrix. This algorithm has memory explosion problem, since the result dense matrix from Khatri-Rao product is large even similar storage size with sparse tensor $\mathcal{X}$.

S. Smith et.al. proposed an optimized MTTKRP algorithm in SPLATT library, which only operates on nonzero entries and factors out the inner multiplication with $\mathbf{B}$ to reduce the number of Floating-Point Operations (FLOPs) [15]. Its optimized MTTKRP operation is

$$\mathbf{M}(i, r) = \sum_{j=1}^{J} \mathbf{B}(j, r) * \sum_{k=1}^{K} \mathcal{X}(i, j, k)\mathbf{C}(k, r). \tag{15}$$

However, SPLATT's MTTKRP algorithm is based on a particular sparse tensor format, which is hard to distribute on Spark platform.

We modified SPLATT's MTTKRP algorithm by using TTM operation and saving the intermediate data. Algorithm 2 shows our MTTKRP algorithm for a third-order sparse tensor $\mathcal{X}$ on mode-I. First, sparse tensor $\mathcal{X}$ times a dense matrix $\mathbf{C}$, generating a new sparse tensor $\mathcal{Y}$. Then Tensor-Matrix Hadamard Reduction product (TMHR) (mentioned in § II) is used to compute a dense matrix $\mathbf{M}_{\mathbf{A}}$ by contracting on mode-J.

Algorithm 2 is taken as our baseline to optimize MTTKRP.

VI. Parallel CP Decomposition

To introduce parallel CPD, we first introduce data structure of local and distributed sparse tensors. Based on the data structure, we designed parallel MTTKRP and CPD algorithms.

Algorithm 2 Sequential MTTKRP algorithm $\mathbf{M_A} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$.

Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense factor matrices $\mathbf{B} \in \mathbb{R}^{J \times R}, \mathbf{C} \in \mathbb{R}^{K \times R}$;
Output: Updated dense factor matrix $\mathbf{M_A} \in R^{I \times R}$;
1: Generate $\mathcal{Y}$: $\mathcal{Y} \leftarrow \mathcal{X} \times_3 \mathbf{C}$
2: $\mathbf{M_A}$ = TMHR $(\mathcal{Y}, \mathbf{B})$
3: **return** $\mathbf{M_A}$;

A. Data Structure

To implement parallel CPD algorithm, an efficient data structure is indispensable to represent sparse tensors locally and distributedly. In this section, we introduce our methods to build distributed sparse tensors.

We create a class to represent a sparse tensor, named *spTen*, which has four members: the number of dimensions (*ndims*), the number of nonzeros (*nnz*), and the size of each mode (*dims*) to describe a sparse tensor and *tuples* storing all nonzero entries. Coordinate format (COO) is used to store the nonzero entries, Figure 5 shows a third-order sparse tensor represented by it. Each row in Figure 5 stands for a nonzero value with its three indices. We use 'Array[Int]' for indices, to better represent tensors in different orders (dimensions). *tuples* is an array of such tuples, 'Array[(Array[Int],Double)]'. For a distributed sparse tensor, *SpTenDist* class is designed to distribute all nonzero entries.

[i	j	k]	val
1	1	1	1.0
1	2	1	2.0
1	2	2	3.0
2	2	1	4.0
2	2	1	5.0
2	2	2	6.0

Fig. 5. Coordinate format of a third-order sparse tensor.

We also design other data structures for the intermediate 'semi-sparse' tensor $\mathcal{Y}$ in Algorithm 3. A 'semi-sparse' tensor is a sparse tensor with one mode extremely dense (detail is below).

B. Parallel MTTKRP

The distributed sparse tensor in format *spTenDist*, each nonzero entry is distributed and computed separately. Parallel MTTKRP algorithm is illustrated in Algorith 3. First, each nonzero entry $\mathcal{X}(i,j,k)$ is multiplied with a vector $MC(k,:)$ to compute TTM operation $\mathcal{Y} \leftarrow \mathcal{X} \times_3 \mathbf{C}$, outputs $\mathcal{Y} \in \mathbb{R}^{I \times J \times R}$. For each dense vector $\mathcal{Y}(i,j,:)$, a Hadamard product is computed with $\mathbf{B}(j,:)$, to generate $\mathbf{M_A}$. We use 'reduceByKey' and 'map' RDD transformations in Spark to parallelize Algorithm 3. Finally, $\mathbf{M_A}$ is collected to local machine. Note that, Algorithm 3 computes each nonzero with a vector instead of a single element locally. This vectorization greatly improves MTTKRP performance by better spatial locality, which avoids R times memory accesses.

Algorithm 3 Parallel MTTKRP algorithm $\mathbf{M_A} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$.

Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense factor matrices $\mathbf{B} \in \mathbb{R}^{J \times R}, \mathbf{C} \in \mathbb{R}^{K \times R}$;
Output: Updated dense factor matrix $\mathbf{M_A} \in R^{I \times R}$;
▷ $\mathcal{Y} \leftarrow \mathcal{X} \times_3 \mathbf{C}$
1: **parfor** $x = 1, 2, \ldots, m$ **do**
2: $i = \mathcal{X}.tuples(x).inds(0)$
3: $j = \mathcal{X}.tuples(x).inds(1)$
4: $k = \mathcal{X}.tuples(x).inds(2)$
5: $\mathcal{Y}(i,j,:) += \sum_{k=1}^{K} \mathcal{X}(i,j,k)\mathbf{C}(k,:)$
6: **end parfor**
▷ Loop all P nonzero fibers $\mathcal{Y}(i,j,:)$. $\mathbf{M_A}$ = TMHR $(\mathcal{Y}, \mathbf{B})$.
7: **parfor** $y = 1, 2, \ldots, P$ **do**
8: $i = \mathcal{Y}.tuples(y).inds(0)$
9: $j = \mathcal{Y}.tuples(y).inds(1)$
10: $\mathbf{M_A}(i,:) += \sum_{j=1}^{J} \mathcal{Y}(i,j,:) * \mathbf{B}(j,:)$
11: **end parfor**
12: **return** $\mathbf{M_A}$;

C. Parallel CP-ALS

By far, we optimized MTTKRP, the critical operation of CP-ALS. Our parallel CP-ALS algorithm is proposed in Algorithm 4. Compared to Algorithm 1, other than parallel MTTKRP, we also optimize the Hadamard product sequence to compute $\mathbf{V}$. Instead of every iteration computes all $\mathbf{V}^{(i)}$s, we only compute $\mathbf{V}^{(n)}$ with updated matrix $\mathbf{A}^{(n)}$ then do Hadamard products.

Algorithm 4 CP-ALS algorithm for an Nth-order sparse tensor.

Input: An Nth-order sparse tensor $\mathcal{X} \in R^{I_1 \times I_2 \times \cdots \times I_N}$ and an integer rank R;
Output: A sequence of dense factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$, $\mathbf{A}^{(i)} \in R^{I_i \times R}$;
1: Initialize $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$;
2: **do**
3: **for** $i = 1, \ldots, n-1, n+1, \ldots, N-1$ **do**
4: $V^{(i)} \leftarrow \mathbf{A}^{(i)\dagger}\mathbf{A}^{(i)}$
5: **end for**
6: **for** $n = 1, 2, \ldots, N$ **do**
7: $V^{(n)} \leftarrow \mathbf{A}^{(n)\dagger}\mathbf{A}^{(n)}$
8: $V \leftarrow$ Hada-reduce $(\mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(n-1)}, \mathbf{V}^{(n+1)}, \ldots, \mathbf{V}^{N})$
9: $\mathbf{A}^{(n)} \leftarrow$ Para-MTTKRP $(\mathcal{X}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(n-1)}, \mathbf{A}^{(n+1)}, \ldots, \mathbf{A}^{N})$
10: Normalize columns of $A^{(n)}$ and store the norms as λ
11: **end for**
12: **while** Fit ceases to improve or maximum iterations exhausted.
13: **Return:** $[\![\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]$;

VII. Experiments and Analysis

A. Data Sets and Platform

In order to evaluate efficiency of our proposed method, we use the IP3D dataset [16], which is a 3D network data with dimension 500-by-500-by-100. The traffic trace consists of TCP flow records collected at the backbone router of a class-B university network. Each record in the trace corresponds to a directional TCP flow between two hosts through a server port. Here we use the model described in section 3 to detect the following three types of anomalies:

Remote to local (R2L) Attacks: For example, R2L attack occurs when an intruder who send packets to a large number of different hosts in the system.

Denial of Service (DOS) Attacks: For example, the victim of a distributed denial of service attack (DDoS), which receives high volume of traffic from a large number of source hosts. Port-scan attacks: Ports that receive abnormal volume of traffic, and/or traffic from too many hosts.

Platform We use Intel Core i7-4770K platform with 3.5GHz frequency, four cores supporting up to 8 hardware threads.

Measurement Runtime is used to measure our parallel algorithm.

B. Performance

We implemented four algorithms: CPD-sInit, CPD-sVec, CPD-pInit, and CPD-pVec.

- CPD-sInit: Algorithm 1 with the most straightforward MTTKRP algorithm, matricized sparse tensor timing the result of Khatri-Rao product, on a single machine. This algorithm is the most space-consuming.
- CPD-sVec: Algorithm 1 with MTTKRP implemented by Algorithm 2.
- CPD-pInit: Algorithm 1 with the most straightforward parallel MTTKRP algorithm. This implementation is also space-consuming.
- CPD-pVec: Parallel algorithm 4 with MTTKRP implemented by Algorithm 3.

Theoretically, CPD-pVec should get the highest performance and also space-efficient. Since CPD-sInit and CPD-pInit cannot run on large tensors, we test on both large and small tensors for better comparison. The small synthetic tensors are generated using Tensor Toolbox [2]. Figure 6 shows the comparison among the four implementations on two sparse tensors. 'Syn-500' is a third-order synthetic sparse tensor, and 'IP3D' is the sparse tensor dataset. Comparing the four implementations on 'Syn-500', our CPD-sVec and CPD-pVec are faster than CPD-sInit and CPD-pInit respectively, showing the benefit of our algorithm. While on IP3D sparse tensor, CPD-sInit and CPD-pInit cannot run because of not enough space. Comparing our two implementations CPD-sVec is faster than CPD-pVec. This is opposite from theoretical analysis, after optimizing on Spark, our CPD does not improve its performance. We profile our code and find two reasons: First, our IP3D dataset is still small, which easily fits in memory. The parallelism benefit doesn't show up from this dataset. We plan to test much larger dataset to prove this. Second, because Breeze is column-majored our implementation requires some matrix transpose, which is relatively expensive especially for low-order tensors. We plan to optimize it by changing the algorithm framework, decreasing matrix transpose from algorithm level.

Figure 7 shows our CP-ALS running time is relatively steady under variant rank values. This makes it possible to discover more phenotypes through one time execution of CPD.

Figure 8 shows the comparison of the outcome of this study with the original CNN methods. The accuracy is tested according the classification rate, which is different with the change of number of hidden layer of convolutional neuro network. But, in the end it will reach a stable value when

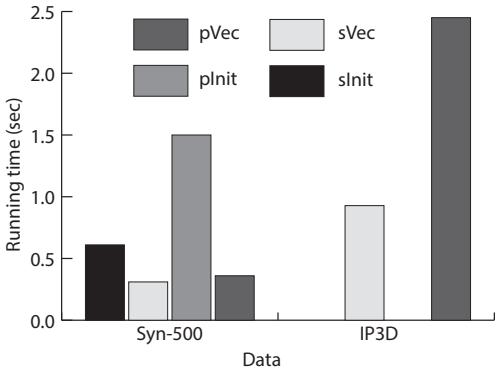


Fig. 6. CP-ALS running time on MIMIC-III and a small synthetic sparse tensor.

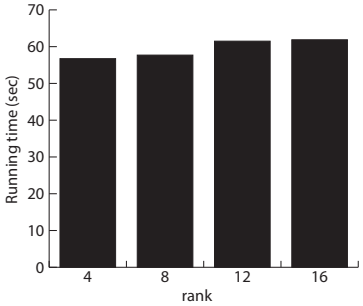


Fig. 7. CP-ALS running time under variant rank values.

choosing a fixed layer number.From this result, we can see that the accuracy of TCNN is comparable with original method.

Attack category	SVM Test Accuracy	TCNN Accuracy
R2L	70.20%	89.30%
DOS	92.50%	92.70%
Port-scan attacks	88.30%	85.40%

Fig. 8. Accuracy.

VIII. Related work

Recently, tensor methods have been used in the optimization of deep neural networks [6]. One class of broadly useful techniques within tensor methods are tensor decompositions. The properties of tensors have long been studied, recently lots of work in machine learning use tensor such as learning latent variable models [1], and developing recommender systems [8]. Several recent papers apply tensor learning and tensor decomposition to deep neural networks for the purpose of devising neural network learning algorithms with theoretical guarantees of convergence [14]. Other lines of research have investigated practical applications of tensor decomposition to deep neural networks with aims including multi-task learning [17], and speeding up convolutional neural networks [12]. Several recent papers apply decompositions for either initialization [17] or post-training [13]. Different with these work, our work decompose the convolution layer, what's more, our method improve the performance of CP decomposition

by proposing a new optimized algorithm and parallel the algorithm

IX. Conclusion and Future work

Big Data consists of multidimensional, multi-modal datasets that are so huge and complex that require new methods to process and these data. Traditional method can not match the requirement of big network traffic data anomaly detection. we proposes a new method named Tensor Convolutional Neuro-Network (TCNN) to resolve network traffic anomaly detection problem which utilize the tensor decomposition technology to decompose the convolution layer and reduce the memory cost. What's more, consider the performance requirement of big data process, we improve the CP decomposition algorithm. The experiment results show our improvement achieve better performance as well as comparable accuracy.

We intend to combine federated learning with our work to implement a distributed parallel convolutional neuro-network on amount of smart device by updating the local gradient descent to the sever.

References

[1] A. Anandkumar, R. Ge, D. J. Hsu, S. M. Kakade, and M. Telgarsky. Tensor decompositions for learning latent variable models. *CoRR*, abs/1210.7559, 2012.

[2] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox (Version 2.6). Available online, February 2015.

[3] J. H. Choi and S. Vishwanathan. Dfacto: Distributed factorization of tensors. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1296–1304. Curran Associates, Inc., 2014.

[4] A. Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *arXiv preprint arXiv:1403.2048*, 2014.

[5] A. Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014.

[6] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis. *CoRR*, abs/1509.05009, 2015.

[7] D. F. Gleich, L. Lim, and Y. Yu. Multilinear pagerank. *CoRR*, abs/1409.1465, 2014.

[8] A. Karatzoglou, X. Amatriain, L. Baltrunas, and N. Oliver. Multiverse recommendation: N-dimensional tensor factorization for context-aware collaborative filtering. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 79–86, New York, NY, USA, 2010. ACM.

[9] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 77:1–77:11, New York, NY, USA, 2015. ACM.

[10] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[11] J. Kossaifi, A. Khanna, Z. C. Lipton, T. Furlanello, and A. Anandkumar. Tensor contraction layers for parsimonious deep nets. *CoRR*, abs/1706.00439, 2017.

[12] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using finetuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[13] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. Tensorizing neural networks. *CoRR*, abs/1509.06569, 2015.

[14] H. Sedghi and A. Anandkumar. Training input-output recurrent neural networks through spectral methods. *arXiv preprint arXiv:1603.00954*, 2016.

[15] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015.

[16] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383. ACM, 2006.

[17] Y. Yang and T. Hospedales. Deep multi-task representation learning: A tensor factorisation approach. *arXiv preprint arXiv:1605.06391*, 2016.