Deep Model Compression and Inference Speedup of Sum-Product Networks on Tensor Trains

Ko, Ching Yun; Chen, Cong; He, Zhuolun; Zhang, Yuke; Batselier, Kim; Wong, Ngai

DOI
10.1109/TNNLS.2019.2928379

Publication date
2020

Document Version
Final published version

Published in
IEEE Transactions on Neural Networks and Learning Systems

Brief Papers

Deep Model Compression and Inference Speedup of Sum–Product Networks on Tensor Trains

Ching-Yun Ko, Cong Chen, Zhuolun He, Yuke Zhang, Kim Batselier, and Ngai Wong

***Abstract*— Sum–product networks (SPNs) constitute an emerging class of neural networks with clear probabilistic semantics and superior inference speed over other graphical models. This brief reveals an important connection between SPNs and tensor trains (TTs), leading to a new canonical form which we call tensor SPNs (tSPNs). Specifically, we demonstrate the intimate relationship between a valid SPN and a TT. For the first time, through mapping an SPN onto a tSPN and employing specially customized optimization techniques, we demonstrate improvements up to a factor of 100 on both model compression and inference speedup for various data sets with negligible loss in accuracy.**

***Index Terms*— Model compression, sum–product network (SP), tensor train (TT).**

I. Introduction

Density estimation is one of the most general tasks in machine learning, where the aim is to learn an estimator for a joint probability distribution over a set of random variables (RVs) from a set of samples. Such an estimator can be used to do inference, namely, computing the probability of queries over those RVs. There are many classical density estimators such as probabilistic graphical models (PGMs) [1], like Markov networks and Bayesian networks, whose exact inference is #P or NP-hard and, therefore, computationally *infeasible*. To develop traceable graphical models, a new deep network structure called the sum–product network (SPN) [2] has been proposed which can compute marginal and conditional probabilities in *linear* time with respect to the size of the network. Moreover, an SPN exhibits a clear semantics of mixtures (sum nodes) and features (product nodes): given a high-dimensional data set $\boldsymbol{x}_k \in \mathbb{R}^d$ $(k = 1 \ldots N)$, an SPN learns and encodes a probability distribution over the data and implicit latent (hidden) variables. Many works have emerged utilizing SPNs in computer vision [3]–[6], speech modeling [7], [8], and robotics [9], [10]. An SPN uses only sum and product nodes, which largely simplifies hardware deployment [11] and forms a strong candidate for lightweight probabilistic neural networks on terminal or edge devices. However, despite the above-mentioned advantages, the SPNs learned by the existing structure and weight learning approaches (see [12]–[15]) are often oversized and contain much redundancy, thus preventing the full exploitation of SPNs as compact graphical networks.

On the other hand, the recent surge of tensor arithmetic [16], [17] in various neural networks has also blossomed in the machine learning community [18]–[22]. The existence of a low-rank tensor approximation in various practical problems, analogous to low-rank matrix factorizations, can often lift the curse of dimensionality and reduce computation and storage from exponential complexities to a linear cost. In line with these works, this brief reveals the intimate connection between SPNs and tensor trains (TTs) [23]. Most importantly, a natural TT representation of an SPN (abbreviated hereafter as a tensor SPN/ tSPN) will be proposed which allows the use of a compact TT to represent this SPN when the sample probabilities are reasonable.

In particular, we leverage the wealth of the existing SPN learning algorithms and attempt to turn their inherently wide SPN tree outputs (due to the intrinsic way of learning through partitioning the data matrix) into a "deep" tree by means of a tensor decomposition subject to a unique nonnegativity constraint. To the best of our knowledge, such a mapping of an SPN onto a tSPN is proposed for the first time, which automatically enforces the sharing of weights through the TT cores. The tensor representation has an inference computational complexity of $O(NR^2d)$, compared with that of $O(NN_wd)$ in an original SPN, where N is the number of samples, R is the maximal TT-rank, N_w is the number of SPN subtrees, and d is the number of variables. Experiments show that a typical N_w is at least ten times larger than R^2, which explains why the faster inference is possible with the proposed tensor representation. Compared to SPNs, tSPNs are able to both compress the number of parameters and speedup the inference up to a factor of 100, with negligible loss in the probabilistic modeling accuracy.

II. Related Works

Although deep networks show great potential in many scenarios, their large model size quickly becomes a bottleneck for real-world deployment. A trending topic in recent years has been in lowering the computational costs of deep networks by model compression and parameter quantization [24]–[26]. By doing so, the model storage, memory bandwidth, and computation can then be reduced to facilitate terminal or edge computing.

This work concerns the compression of SPN model parameters through a TT. Before going into the details, we review a common compression approach in neural networks: *pruning*. Pruning takes a large network and deletes features or parameters under specific guidelines. For example, the optimal brain damage [27] and optimal brain surgeon [28] techniques prune networks to reduce the number of connections based on the Hessian of the loss function. Alternatively, in [29], connections are eliminated based on the parameter magnitudes. The HashedNets [30] technique reduces model sizes using hash functions to randomly group connections into hash buckets, where all connections within the same bucket share a single parameter. Hu *et al.* [31] delete connections based on the output statistics of activated neurons.

Manuscript received November 29, 2018; revised May 10, 2019; accepted July 3, 2019. Date of publication August 9, 2019; date of current version July 7, 2020. This work was supported in part by the Hong Kong Research Grants Council under Project 17246416 and in part by the University Research Committee of The University of Hong Kong. *(Ching-Yun Ko and Cong Chen contributed equally to this work.) (Corresponding author: Ngai Wong.)*

C.-Y. Ko, C. Chen, Z. He, and N. Wong are with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong (e-mail: cyko@eee.hku.hk; chencong@eee.hku.hk; zleonhe@hku.hk; nwong@eee.hku.hk).

Y. Zhang is with the Ming Hsieh Department of Electrical and Computer Engineering, University of South Carolina, Columbia, SC 29208 USA (e-mail: yukezhan@usc.edu).

K. Batselier is with the Delft Center for Systems and Control, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: k.batselier@tudelft.nl).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2019.2928379

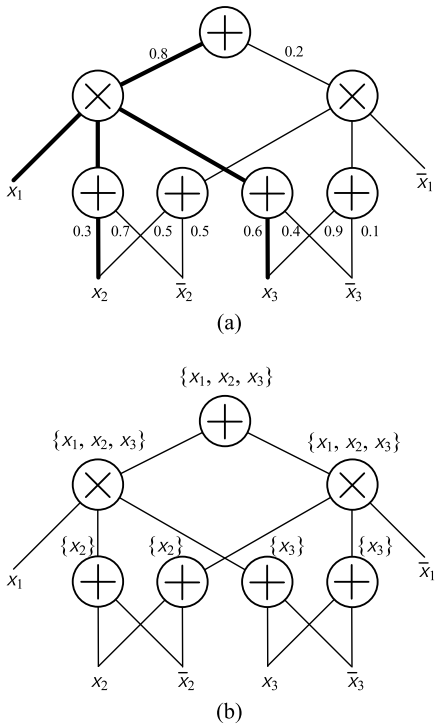

Fig. 1. (a) Example SPN with Boolean variables. Bold edges: induced tree example (see Definition 3). (b) Scopes, denoted by $\{\circ\}$, for every node in the example SPN.

These schemes, however, do not generalize to SPNs because the validity (see Definition 2 in the following) of an SPN may be violated after pruning the trivial weights. There are relatively few works on SPN compression, most existing schemes [4], [13], [32], [33] prune learned SPNs by simply discarding edges with zero weights, and recursively removing nonroot parentless nodes. In this way, the resultant SPNs still preserve *completeness* and *consistency* and are, therefore, still *valid* SPNs. An alternative to pruning is the conversion of SPNs into graph SPNs [34]. Specifically, similarities of subtrees stemming from one identical variable (node) in an SPN are evaluated in a bottom-up fashion. Subtrees are then merged if their similarities exceed a predefined threshold[1]. To this end, we aim at finding compact yet valid tSPNs from a trained reference SPN and will compare with approaches that preserve validity.

III. Preliminaries

A. SPN Basics

We use a modified SPN example from [2], [35], shown in Fig. 1(a), to motivate some concepts and operations of SPNs. Boolean variables are chosen for the ease of illustration, while their generalization to multi-nominal or continuous variables is straightforward [2]. To begin with, an SPN is a directed acyclic graph with alternating layers of sum and product (internal) nodes and a root node on top. The edges emanating from sum nodes have nonnegative weights, while the edges emanating from product nodes are all of unit weight. The leaves contain the set of RVs $\boldsymbol{X} = \{X_1, \ldots, X_d\}$. For boolean variables, the indicator functions x_i and $\bar{x}_i$ are 1 when X_i and $\bar{X}_i$ are 1, respectively, and 0 otherwise.

Definition 1: The *scope* of an SPN is the set of variables appearing in its leaves. The scope of an internal sum or product node is the scope of the corresponding *sub-SPN* rooted at that node, as illustrated in Fig. 1(b).

[1]Whether this procedure preserves SPNs' validity, however, is uncertain.

Definition 2: An SPN is *complete* when all children of a sum node have identical scope. It is *consistent* when no variable appears negated in one child of a product node and nonnegated in another. An SPN is *valid* when all its sum nodes are complete and all product nodes are consistent.

Most existing algorithms learn valid SPNs [12]–[14], which act as the starting point for the contributions in this brief. The sum nodes have the semantics of a *mixture* of components, while the product nodes represent *features*. An SPN is called a *normalized* SPN when the edges emanating from a sum node have a total weight of one. Consequently, the SPN in Fig. 1(a) is a valid and normalized SPN. We use $\mathcal{S}_{\boldsymbol{w}}(\boldsymbol{x}) \in \mathbb{R}$ to denote the SPN output where $\boldsymbol{w}$ is the vector containing all (nonnegative) weights in the network, and $\boldsymbol{x} \in \mathbb{R}^d$ contains all RVs. A distribution is *tractable* if any marginal probability can be computed in linear time proportional to the number of graph edges.

Definition 3: An induced tree [14] is a subtree of an SPN originating from the root following two rules: 1) only one edge out of a sum node is selected at a time and 2) all edges out of a product node are selected. It can be readily checked that the total number of induced trees arising from an SPN is $\tau = \mathcal{S}_{\mathbf{1}}(\mathbf{1})$, i.e., by setting $\boldsymbol{w} = \mathbf{1}$ and $\boldsymbol{x} = \mathbf{1}$ where $\mathbf{1}$ is the all-ones vector of the appropriate size.

For instance, the bold edges in Fig. 1(a) denote an induced tree by selecting the left route out of each sum node. An important concept that serves as a stepping stone to TTs is that of the network polynomial [36]:

Definition 4: Let $f(\boldsymbol{x})$ be the probability mass function of a set of discrete RVs $\boldsymbol{X} = \{X_1, \ldots, X_d\}$. The network polynomial of $f(\boldsymbol{x})$ is the multilinear polynomial $\sum_{\boldsymbol{x}} f(\boldsymbol{x}) \prod_{\boldsymbol{x}} \lambda(\boldsymbol{x})$, where $\prod_{\boldsymbol{x}} \lambda(\boldsymbol{x})$ is the product of evidence indicators that has a value of 1 in the state $\boldsymbol{x}$. Any joint probability function of d I-valued discrete RVs is represented by I^d probabilities. The corresponding network polynomial has, therefore, I^d terms. For example, the joint probability function $f(\boldsymbol{x})$ of the SPN in Fig. 1 has a network polynomial that consists of $2^3 = 8$ terms

$$\begin{aligned} f(\boldsymbol{x}) = {} & (0.8)(0.3)(0.6)x_1x_2x_3 + (0.8)(0.3)(0.4)x_1x_2\bar{x}_3 \\ & + (0.8)(0.7)(0.6)x_1\bar{x}_2x_3 + (0.8)(0.7)(0.4)x_1\bar{x}_2\bar{x}_3 \\ & + (0.2)(0.5)(0.9)\bar{x}_1x_2x_3 + (0.2)(0.5)(0.1)\bar{x}_1x_2\bar{x}_3 \\ & + (0.2)(0.5)(0.9)\bar{x}_1\bar{x}_2x_3 + (0.2)(0.5)(0.1)\bar{x}_1\bar{x}_2\bar{x}_3. \quad (1) \end{aligned}$$

An SPN can thereby be viewed as a network polynomial $\mathcal{S}_{\boldsymbol{w}}(\boldsymbol{x}) := f(\boldsymbol{x})$ that encodes a probability function. The beauty of an SPN lies in its exact and tractable inference. Equation (1) is an instance of a normalized SPN. For an unnormalized SPN, there are two ways to normalize it. One is to scale the edge weights out of each sum node such that they add up to one, i.e., turning it back into a normalized SPN. Alternatively, we can compute the *partition function* in one bottom-up pass by setting $\boldsymbol{x} = \mathbf{1}$, namely, $Z = \mathcal{S}_{\boldsymbol{w}}(\mathbf{1})$, such that $\mathcal{S}_{\boldsymbol{w}}(\boldsymbol{x})/Z$ is a probability function.

Example 1: Assuming a normalized SPN, the probability of a fully specified state (also called a complete evidence) $\boldsymbol{x}$, e.g., $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ in Fig. 1(a), is easily computed through a bottom-up pass by setting $x_i = 1$ and $\bar{x}_i = 0$ for $i = 1, 3$ and $x_2 = 0$ and $\bar{x}_2 = 1$.

Example 2: Assuming a normalized SPN, the probability of some evidence, e.g., $x_1 = 1$ in Fig. 1(a), can be computed by marginalizing over x_2 and x_3. This is computed through a bottom-up pass by setting $x_1 = 1$ and $\bar{x}_1 = 0$, and $x_i = \bar{x}_i = 1$ for $i = 2, 3$.

These two examples can be easily verified by comparing with (1). Similar tractable operations allow us to compute the conditional probability, as well as the most probable explanation (MPE) by

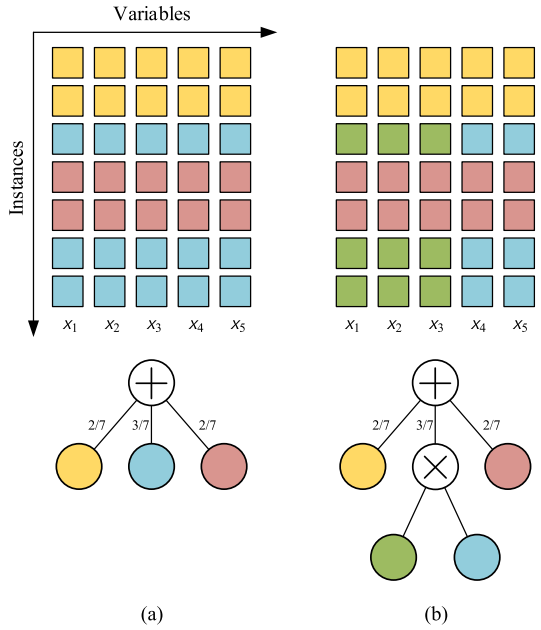

Fig. 2. LearnSPN operations. (a) Slicing. (b) Chopping.

augmenting an SPN to incorporate the sum nodes' latent variables (namely, a selective SPN) and using maximum nodes in place of sum nodes as described in [37].

Now, to transform an SPN into a tSPN, we need to slightly modify the induced tree (Definition 3) by terminating at the leaf nodes. This implies that the bottom-layer sum nodes of the SPN have a univariate scope. We remark that the leftmost x_i in Fig. 1(a) can be regarded as a leaf node with one edge having zero weight, namely, $(x_1 \ \bar{x}_1)(1 \ 0)^T$, while the one adjacent to it is $(x_2 \ \bar{x}_2)(0.3 \ 0.7)^T$. In fact, prevailing SPN learning algorithms (e.g., LearnSPN, SPN-B and SPN-BT[2] [12], [13]) all produce SPN trees terminating at leaf nodes. Although SPN illustrations often utilize networks with shared weights (e.g., the two top branches in Fig. 1(a) are shared among many induced trees), conventional learning algorithms are all based on the "slice" and "chop" operations on the data set matrix [15], or variants with additional regularization constraints. A toy example illustrates the basic learnSPN flow. Referring to Fig. 2(a), the slicing operation constructs children of a sum node by clustering similar sample instances. This is often done via k-means clustering or expectation–maximization (EM) for Gaussian mixture models (GMMs). In Fig. 2(b), the chopping operation constructs children of a product node by grouping-dependent variables. This is often done by the G-test or mutual information methods wherein a scoring formula is used to determine whether variables belong to the same group.

These hierarchical divisive clustering steps are surprisingly simple and effective but they proceed in a top-down fashion and never look back, which often leads to inherently wide SPN trees. For example, in the standard NLTCS benchmark, learnSPN (with default hyperparameters) generates an SPN with 19 layers and 1420 leaf nodes even though there are only 16 variables. This example shows that the existing learning algorithms do not readily produce shared edges (and weights) across different induced trees and do not generate SPNs that can otherwise be represented compactly.

B. Tensor Basics

Tensors are high-dimensional arrays that generalize vectors and matrices to higher orders. A d-way or d-order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ is an array where each entry is indexed by d indices $i_1, i_2, \ldots, i_d$. We use the convention $1 \leq i_k \leq I_k$ for $k = 1, \ldots, d$. When $I_1 = \ldots = I_d = I$, the tensor is called *cubical*. MATLAB notation is used to denote entries of tensors. Boldface capital calligraphic letters $\mathcal{A}, \mathcal{B}, \ldots$ denote tensors, boldface capital letters $\boldsymbol{A}, \boldsymbol{B}, \ldots$ denote matrices, boldface letters $\boldsymbol{a}, \boldsymbol{b}, \ldots$ denote vectors, and Roman letters $a, b, \ldots$ denote scalars. A set of d tensors, like the cores of a TT [23], is denoted $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)}$. The notion of a rank-1 matrix is generalized to tensors as follows:

Definition 5 [16, p. 460]: For a given set of vectors $\boldsymbol{a}_1 \in \mathbb{R}^{I_1}, \ldots, \boldsymbol{a}_d \in \mathbb{R}^{I_d}$, the entries of the corresponding rank-1 tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ are defined as

$$\mathcal{A}(i_1, i_2, \ldots, i_d) := \boldsymbol{a}_1(i_1)\boldsymbol{a}_2(i_2) \cdots \boldsymbol{a}_d(i_d).$$

A rank-r tensor is the sum of r rank-1 tensors. The matrix-vector product is extended to the multiplication of a vector to a tensor along one of its modes.

Definition 6 [16, p. 458]): The k-mode product of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ with a vector $\boldsymbol{u} \in \mathbb{R}^{I_k}$ is denoted $\mathcal{B} = \mathcal{A} \times_k \boldsymbol{u}^T \in \mathbb{R}^{I_1 \times \cdots \times I_{k-1} \times I_{k+1} \times \cdots \times I_d}$ for which the corresponding entries $\mathcal{B}(i_1, \cdots, i_{k-1}, i_{k+1}, \cdots, i_d)$ are defined as

$$\sum_{i_k=1}^{I_k} \boldsymbol{u}(i_k)\mathcal{A}(i_1, \cdots, i_{k-1}, i_k, i_{k+1}, \cdots, i_d).$$

We will also require the notions of the Khatri–Rao product and tensor vectorization:

Definition 7: If $\boldsymbol{A} \in \mathbb{R}^{N_1 \times M}$ and $\boldsymbol{C} \in \mathbb{R}^{N_2 \times M}$, then their Khatri–Rao product $\boldsymbol{A} \odot \boldsymbol{C}$ is the $N_1 N_2 \times M$ matrix $[\boldsymbol{A}(:, 1) \otimes \boldsymbol{C}(:, 1), \ \cdots, \ \boldsymbol{A}(:, M) \otimes \boldsymbol{C}(:, M)]$, where $\otimes$ denotes the standard Kronecker product.

Definition 8: The vectorization of a tensor $\mathcal{A}$, denoted $\text{vec}(\mathcal{A})$, reshapes $\mathcal{A}$ indexwise into a column vector with the same number of entries.

The storage of a d-way tensor with dimensions I requires I^d elements. Tensor decompositions are crucial in reducing the exponential storage requirement of a given tensor. In this work, we utilize the TT decomposition [23].

Definition 9 [23, p. 2296]: A TT representation of a tensor $\mathcal{A}$ is a set of d three-way tensors $\mathcal{A}^{(1)} \in \mathbb{R}^{1 \times I_1 \times R_2}, \mathcal{A}^{(2)} \in \mathbb{R}^{R_2 \times I_2 \times R_3}, \ldots, \mathcal{A}^{(d)} \in \mathbb{R}^{R_d \times I_d \times 1}$ such that $\mathcal{A}(i_1, i_2, \ldots, i_d)$ can be computed from

$$\sum_{r_2, \ldots, r_d=1}^{R_2, \ldots, R_d} \mathcal{A}^{(1)}(1, i_1, r_2)\mathcal{A}^{(2)}(r_2, i_2, r_3) \cdots \mathcal{A}^{(d)}(r_d, i_d, 1).$$

Here, $R_2, \ldots, R_d$ are called the TT-ranks, and the three-way tensors $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(d)}$ are called the TT-cores.

Our key idea is to represent a network polynomial of any joint probability function by a low-rank TT. In this way, all I^d (if $I_1 = \ldots = I_d = I$) probabilities can be computed from $O(dIR^2)$ numbers where R is the maximal TT-rank. Without loss of generality, we consider only boolean variables ($I = 2$).

Definition 10: For a given network polynomial $f(\boldsymbol{x})$ of d binary RVs, we define the corresponding TT consisting of d three-way tensors $\mathcal{F}^{(1)} \in \mathbb{R}^{1 \times 2 \times R_2}, \mathcal{F}^{(2)} \in \mathbb{R}^{R_2 \times 2 \times R_3}, \ldots, \mathcal{F}^{(d)} \in \mathbb{R}^{R_d \times 2 \times 1}$ such that the evaluation of $f(\boldsymbol{x})$ for a given state $\boldsymbol{x}$ can be computed from

$$\left(\mathcal{F}^{(1)} \times_2 \begin{pmatrix} x_1 \\ \bar{x}_1 \end{pmatrix}^T\right)\left(\mathcal{F}^{(2)} \times_2 \begin{pmatrix} x_2 \\ \bar{x}_2 \end{pmatrix}^T\right) \cdots \left(\mathcal{F}^{(d)} \times_2 \begin{pmatrix} x_d \\ \bar{x}_d \end{pmatrix}^T\right). \tag{2}$$

Note that the $\mathcal{F}^{(1)} \times_2 (x_1 \ \bar{x}_1)$ and $\mathcal{F}^{(d)} \times_2 (x_d \ \bar{x}_d)$ factors are row and column vectors, respectively. The other factors are matrices such

[2]Binary row clustering (B); Tree distributions as leaf nodes (T).

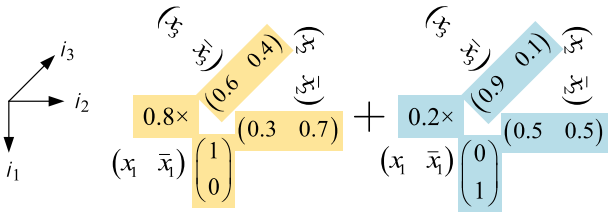

Fig. 3. tSPN equivalence of the SPN shown in Fig. 1(a).

that the whole product results in the scalar $f(\boldsymbol{x})$. Small TT-ranks imply small matrix factors, which, in turn, gives rise to a massive reduction in both the number of network parameters and the inference time.

IV. SPN to tSPN Conversion

The major innovation of this brief stems from the important observation that an SPN induced tree terminated at leaf nodes is, in fact, a rank-1 tensor. Using Fig. 1(a) again as an example, there are two such induced trees that can be regarded as the addition of two rank-1 terms with mode products $(x_k \ \bar{x}_k)$ onto their kth mode, as shown in Fig. 3. Consequently, summing all the rank-1 terms produces a d-way cubical tensor of dimension $I = 2$. This tensor can then hopefully be sufficiently approximated by a low-rank TT as a particular kind of tSPN. We aim at building a tSPN based on the TT structure, depicted in Fig. 4, while satisfying the following constraints.

1) The TT-cores contain only nonnegative entries.
2) There is a *mixed core*, whose position is arbitrary, with all its entries summing up to 1.
3) Every core to the left of the mixed core is a *left normalized core*, which means that each of its vertical slices $\boldsymbol{\mathcal{F}}^{(k)}(:,:,\alpha_{k+1})$ sums up to 1.
4) Every core to the right of the mixed core is a *right normalized core*, which means that each slice $\boldsymbol{\mathcal{F}}^{(k)}(\alpha_k,:,:)$ sums up to 1.
5) When we encounter a slice that contains all zeros, it means the two slices (one vertical and one horizontal) in two adjacent cores corresponding to the same α_k can be removed and the dimension R_k is shrunk by one.

The first constraint ensures that a tSPN has nonnegative weights. The remaining constraints ensure that the partition function $Z = 1$ when $x_k = \bar{x}_k = 1$ for all k's. A tSPN obeying the above constraints is called a normalized tSPN in analogy to a normalized SPN. The left/right normalized cores and the mixed core are strongly analogous to the mixed-canonical form of a TT which consists of left/right orthogonalized cores and a mixed core. We remark that a tSPN having a TT structure automatically enforces the desired weight parameter sharing as well as a deep network. This is because each scalar (namely, probability) evaluation of a TT-based tSPN, when contracted with $(x_k \ \bar{x}_k)$ at its kth mode, $k = 1, \ldots, d$, results in a matrix product across all TT-cores (see Definition 10).

Recalling from Fig. 3, a tSPN is fully captured by a d-way cubical tensor $\boldsymbol{\mathcal{F}}$ through summing all rank-1 terms (induced trees) extracted from the learned SPN. The conversion of such a full-tensor tSPN into a TT-based tSPN then boils down to converting $\boldsymbol{\mathcal{F}}$ into its TT format $\boldsymbol{\mathcal{F}}^{(1)}, \ldots, \boldsymbol{\mathcal{F}}^{(d)}$. A direct way to obtain the TT form of $\boldsymbol{\mathcal{F}}$ is by regarding each rank-1 tensor term, corresponding to an induced tree, as a rank-1 TT and sum them all up into a new TT [23]. However, this makes the TT-ranks $R_2, \ldots, R_d$ equal to the number of rank-1 terms and, therefore, impractically high. Although TT-rounding [23, p. 2305] by using the singular value decomposition (SVD) between successive cores may reduce the TT-ranks, it will destroy the nonnegativity of the weights and result in cores with negative values. Similar issues arise when we use nonnegative tensor factorization (NTF) algorithms [38] on $\boldsymbol{\mathcal{F}}$ which also produce a large number of rank-1 tensor terms. In fact, constructing the full tensor $\boldsymbol{\mathcal{F}}$ explicitly is computationally prohibitive when the number of variables d go beyond 17 on our computers. This motivates us to develop an SPN-to-tSPN construction algorithm, called `spn2tspn`, through a recently proposed tensor-network nonlinear system identification method [39] as explained in the following.

A. Algorithm: spn2tspn

Starting with a valid SPN learned from a given data set, we compute by exact inference the probabilities of a set of training input samples and randomly generated samples. This step has a complexity linear to the number of SPN edges and generates a set of multi-input single-output (MISO) data suitable for the identification of the TT underlying the tSPN. More specifically, training samples are meaningful data and constitute *positive samples* used in the SPN learning and, therefore, correspond to higher probabilities. Whereas, the uniformly generated samples are *negative samples* outside the data set[3]. They are fed into the SPN for their probabilities that are mostly close to zero. We then utilize these MISO data to identify a TT-based tSPN by adapting the approach in [39]. In particular, with a set of N (positive and negative) samples together with their probabilities, the goal is to obtain a tensor $\boldsymbol{\mathcal{F}} \in \mathbb{R}^{2\times\cdots\times 2}$ in a TT form such that the probability distribution it represents is aligned with that of the SPN. We first collect the N column vectors $(x_k \ \bar{x}_k)^T$ into the matrix $\boldsymbol{S}^{(k)} \in \mathbb{R}^{2\times N}$ for $k = 1, \ldots, d$. Next, we formulate the optimization problem

$$\min_{\boldsymbol{\mathcal{F}}} \ ||\boldsymbol{S}^T \ \mathrm{vec}(\boldsymbol{\mathcal{F}}) - \boldsymbol{y}||_2^2 \tag{3}$$

where $\mathrm{vec}(\boldsymbol{\mathcal{F}})$ is represented by a TT with nonnegative cores $\boldsymbol{\mathcal{F}}^{(k)}$, and $\boldsymbol{S}^T \in \mathbb{R}^{N\times 2^d}$ is computed from

$$\boldsymbol{S} = \boldsymbol{S}^{(d)} \odot \boldsymbol{S}^{(d-1)} \odot \ldots \odot \boldsymbol{S}^{(1)} \tag{4}$$

and $\boldsymbol{y} \in \mathbb{R}^{N\times 1}$ is the vector of probabilities of the N samples.

Following from [39], (3) is broken into least-squares subproblems of smaller sizes solved by the alternating linear scheme (ALS). However, different from [39], we aim at obtaining a nonnegative $\boldsymbol{\mathcal{F}}^{(k)}$ to ensure clear probabilistic semantics. Therefore, a nonnegativity constraint is further imposed on each subproblem, which is then solved by the nonnegative least-squares (NNLS) method [40] within each ALS iteration. This formulation also resembles the tensor completion work [41] that employs a TT format but without the nonnegativity constraint. In short, one solves the following least-squares subproblem for $\boldsymbol{\mathcal{F}}^{(k)}$ by NNLS

$$\boldsymbol{y} = \begin{pmatrix} \boldsymbol{a}_{>k,1}^T \otimes \boldsymbol{s}_1^{(k)T} \otimes \boldsymbol{a}_{<k,1} \\ \boldsymbol{a}_{>k,2}^T \otimes \boldsymbol{s}_2^{(k)T} \otimes \boldsymbol{a}_{<k,2} \\ \vdots \\ \boldsymbol{a}_{>k,N}^T \otimes \boldsymbol{s}_N^{(k)T} \otimes \boldsymbol{a}_{<k,N} \end{pmatrix} \mathrm{vec}(\boldsymbol{\mathcal{F}}^{(k)}) \tag{5}$$

where $\boldsymbol{s}_l^{(k)} \in \mathbb{R}^{I_k\times 1} (1 \le l \le N)$ denotes the lth column of $\boldsymbol{S}^{(k)}$, and $\boldsymbol{a}_{<k,l}^T$ and $\boldsymbol{a}_{>k,l}$ are the auxiliary notations defined as

$$\boldsymbol{a}_{<k,l}^T := (\boldsymbol{\mathcal{F}}^{(1)} \times_2 \boldsymbol{s}_l^{(1)T}) \ldots (\boldsymbol{\mathcal{F}}^{(k-1)} \times_2 \boldsymbol{s}_l^{(k-1)T}) \in \mathbb{R}^{R_k}$$
$$\boldsymbol{a}_{>k,l} := (\boldsymbol{\mathcal{F}}^{(k+1)} \times_2 \boldsymbol{s}_l^{(k+1)T}) \ldots (\boldsymbol{\mathcal{F}}^{(d)} \times_2 \boldsymbol{s}_l^{(d)T}) \in \mathbb{R}^{R_{k+1}}.$$

[3]Empirically, negative samples generated from uniform sampling yield consistently good performance in all our numerical experiments.

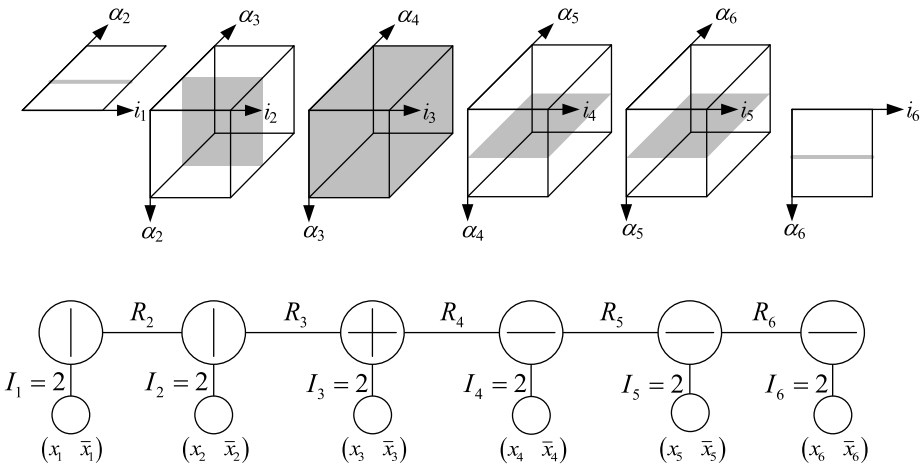

Fig. 4. Normalized tSPN analogous to a normalized SPN, wherein the shaded parts within a core have entries summing up to unity. The vertical, cross, and horizontal lines in the lower tensor diagram denote left normalized, mixed, and right normalized cores, respectively.

TABLE I
DATA SET ATTRIBUTES

Dataset	# variables	avg. log prob. of samples
KDDCup2K	64	−2.32
NLTCS	16	−5.98
MSNBC	17	−6.01
MSWeb	294	−9.46
Retail	135	−10.60
Plants	69	−12.42

This update is followed by a normalization step to ensure $\mathcal{F}^{(k)}$ is left (right) normalized. Each of the TT-cores is updated sequentially until the maximum number of iterations is reached or when the residual in (3) falls below a given tolerance. The pseudo-codes are summarized in Algorithm 1.

Algorithm 1 spn2tspn (SPN-to-tSPN Mapping)

Input: Initial TT-ranks (defaulted at 20), a valid SPN and N positive and negative samples.

Output: A compressed TT-based tSPN.

1: Construct d input matrices $\boldsymbol{S}^{(1)}, \boldsymbol{S}^{(2)}, \ldots, \boldsymbol{S}^{(d)}$ as described above
2: Infer the probabilities of the N samples via the valid SPN and stack them into $\boldsymbol{y}$
3: Randomly initialize nonnegative TT-cores with prescribed initial TT-ranks
4: **while** stopping criteria not met **do**
5: **for** $k = 1, \ldots, d-1$ **do**
6: $\text{vec}(\mathcal{F}^{(k)}) \leftarrow$ solve (5) using NNLS
7: $\boldsymbol{b} \leftarrow$ sum over the first and second indices of $\mathcal{F}^{(k)}$
8: Identify nonzero slices $\Omega \leftarrow \text{find}(\boldsymbol{b} \neq 0)$
9: $\mathcal{F}^{(k)} \leftarrow \mathcal{F}^{(k)}(:, :, \Omega) \times_3 \text{diag}(1./\boldsymbol{b}(\Omega))$
10: $\mathcal{F}^{(k+1)} \leftarrow \mathcal{F}^{(k+1)}(\Omega, :, :) \times_1 \text{diag}(\boldsymbol{b}(\Omega))$
11: **end for**
12: **for** $k = d, \ldots, 2$ **do**
13: $\text{vec}(\mathcal{F}^{(k)}) \leftarrow$ solve (5) using NNLS
14: $\boldsymbol{b} \leftarrow$ sum over the second and third indices of $\mathcal{F}^{(k)}$
15: Identify nonzero slices $\Omega \leftarrow \text{find}(\boldsymbol{b} \neq 0)$
16: $\mathcal{F}^{(k)} \leftarrow \mathcal{F}^{(k)}(\Omega, :, :) \times_1 \text{diag}(1./\boldsymbol{b}(\Omega))$
17: $\mathcal{F}^{(k-1)} \leftarrow \mathcal{F}^{(k-1)}(:, :, \Omega) \times_3 \text{diag}(\boldsymbol{b}(\Omega))$
18: **end for**
19: **end while**

The core steps in Algorithm 1 are Lines 6–10 and Lines 13–17, where Lines 6 and 13 include the use of NNLS for enforcing the nonnegativity of cores and Lines 7–10 and 14–17 are the left/right normalization of cores. Notably, zero slices are automatically removed during the normalization process. Once a tSPN is built, a reserved portion of the data set input samples (not used in the learning) is used as test inputs and their probabilities are used as test outputs to check the quality of the tSPN. We remark that the differences in Algorithm 1 from that in [39] are threefold: 1) additional nonnegativity constraints in subproblems; 2) supplementary autotrimming and redundancy removal; and 3) disparate normalization procedures due to a newly defined normalized canonical form.

The most computationally expensive steps in Algorithm 1 are the NNLS solves, with a complexity of $O(R^6)$ flops in each iteration, where R is the maximal TT-rank. The inference in a tSPN inherits the efficiency of traditional SPNs and can further exploit the TT structure. This is done as described by (2) by computing the two-mode product of each TT-core $\mathcal{F}^{(k)} \in \mathbb{R}^{R_k \times 2 \times R_{k+1}}$ with the vector $(x_k \ \bar{x}_k)$ and then multiplying the obtained matrices and vectors. This implies that batch inference can be performed by a sequence of *two-mode product-Khatri–Rao product* between TT-cores and inputs. This requires $O(NR^2d)$ flops compared with $O(NN_wd)$ flops in an SPN, where N is the number of samples, R is the maximal TT-rank, N_w is the number of SPN subtrees, and d is the number of variables.

V. EXPERIMENTS

A. Data sets and Implementations

We evaluate the proposed spn2tspn algorithm on publicly available benchmark data sets[4]. Relevant details of number of variables and average probability (in logarithm) of training samples are listed in Table I. We sort the data sets by their sample probabilities, which we will refer back when we discuss the applicability of the proposed algorithm. Reference SPNs are trained on the above data sets by Spyn [13] (Python implementation.[5]) Algorithm 1 is implemented in MATLAB[6] and all experiments were run on a desktop computer with an Intel i5 quad-core processor at 3.2-GHz and 16-GB RAM.

B. Metrics and Baselines

Our aim is to find an alternative valid SPN representation that exhibits negligible probabilistic modeling loss. Subsequently, we want to ensure samples' probabilities are significantly larger than

[4]https://github.com/arranger1044/spyn/tree/master/data
[5]https://github.com/arranger1044/spyn
[6]https://github.com/IRENEKO/tSPN

TABLE II
SPN AND tSPN INFORMATION FOR VARIOUS DATA SETS

Dataset	max TT-rank	# SPN sub-trees	# parameters in SPN/ tSPN (×)	infer. time(s) by SPN/ tSPN (×)	FPR (%) when FNR=2% SPN	FPR (%) when FNR=2% baseline SPN	FPR (%) when FNR=2% tSPN	FPR (%) when FNR=5% SPN	FPR (%) when FNR=5% baseline SPN	FPR (%) when FNR=5% tSPN	tSPN runtime (s)
KDDCup2K	13	511	15922/2015 (8)	4.521/0.260 (17.4)	0.00	0.00	0.00	0.00	0.00	0.00	273
NLTCS	3	260	3245/103 (32)	0.058/0.009 (6.8)	9.00	100.00	23.00	3.00	100.00	11.00	31
MSNBC	23	1310	14105/807 (17)	4.946/0.146 (33.9)	0.27	9.62	2.41	0.02	4.21	0.16	566
MSWeb	8	872	37725/1736 (22)	4.678/0.032 (146.7)	0.00	0.00	0.00	0.00	0.00	0.00	186
Retail	2	554	34446/372 (93)	1.284/0.015 (83.4)	0.00	0.00	0.00	0.00	0.00	0.00	190
Plants	3	432	20917/252 (83)	0.418/0.008 (50.9)	4.00	100.00	100.00	0.00	100.00	100.00	119

those of negative samples. To validate this condition[7], we adopt the metrics of *false positive rate* (FPR, Type I error) and *false negative rate* (FNR, Type II error) [42] with a threshold probability θ, where

$$\text{FNR}_\theta = \frac{\#\{\text{sample} : P(\text{sample}) < \theta\}}{\#\text{samples}}$$
$$\text{FPR}_\theta = \frac{\#\{\text{neg.sample} : P(\text{neg.sample}) \geq \theta\}}{\#\text{neg.samples}}.$$

With FNR_θ and FPR_θ, we are able to measure how many negative samples can, indeed, have larger inference probabilities than some of the samples. Specifically, when we fix the FNR_θ to, say, 2% and evaluate the corresponding FPR_θ, it means we want to quantify the portion of negative samples that have larger probabilities than 2% of the samples. This yields a fair and effective quantification of how good a probabilistic model can discriminate samples from negative samples in both SPNs and tSPNs. In addition, to fairly compare our proposed method in the context of validity-preserving approaches, a baseline pruning approach that discards weights with the few smallest values is also applied to the reference SPNs with the same parameter reduction ratio as in the spn2tspn conversions.

C. Results and Applicability

We summarize the experimental results in Table II, where data sets are ordered as in Table I. As the inference complexities of tSPNs and SPNs are $O(NR^2d)$ flops and $O(NN_wd)$ flops, respectively, the difference in complexities boils down to the comparison of the squared maximum TT-rank R^2 and the number of subtrees N_w. On account of the above, the maximum TT-rank R in tSPNs is listed together with the number of SPN subtrees N_w in Table II, alongside the inference times.

After obtaining the reference SPNs, we apply Algorithm 1 to map the SPNs to their tSPNs. As given in Table II, the total number of parameters in the tSPNs are up to a 100 times smaller compared to the original SPNs, which implies that the original SPNs, indeed, contain nonneglectable redundancy. Furthermore, one can readily check that the tSPNs inference is up to 146.7× faster than the conventional SPN inference, which is explained by different values of R^2 and N_w.

Besides, the dramatic reductions in the number of parameters and speedups in the inference time, it is also remarked that there are small differences between the statistical outputs of the SPNs and tSPNs of the *KDDCup2K*, *MSNBC*, *MSWeb*, and *Retail* data sets, relatively small differences for the *NLTCS* data set, and larger differences for the *Plants* data set. Specifically, no negative samples will be interpreted as samples when the FNR of the samples is 2% and 5% on the *KDDCup2K*, *MSWeb*, and *Retail* data sets. On the *NLTCS* and *MSNBC* data sets, the tSPN exhibits a much better capability in distinguishing samples and negative samples than the baseline pruned SPN when FNR= 2%, 5%. As the average probability of data set samples decreases, both the baseline pruned SPN and the tSPN of the *Plants* data set fail to distinguish at least 5% of samples with the smallest probabilities from negative samples. We summarize the runtime of our proposed spn2tspn algorithm in the last column in Table II. A comparison of the total runtime between our proposed method and that of SPN training (Spyn) is difficult as different languages were used for the implementation (MATLAB vs Python). Some additional remarks are in order as follows.

1) The depth of a tSPN (corresponding to the number of TT-cores) is inherently high, while its width (corresponding to TT-ranks) is usually low. This means a higher expressive efficiency is obtained.
2) We observed poorer fitting onto tSPNs in data sets with small average sample probability, where the inferred tSPN's ability in distinguishing samples from negative samples degrades. We propose two possible causes: 1) numerical (ill-conditioned) issues in linear equations and 2) negative samples used in tSPN training are bound to be insufficient when the number of variables is large.
3) Regarding the above-mentioned problems, a natural follow-up question is whether tSPN learning can be directly performed on the TT or other tensor structures rather than starting from the SPN followed by tSPN conversion. Research along this line is underway and results will be reported in our upcoming work.

[7]A typical metric, Kullback–Leibler divergence, requires the two distributions to satisfy absolute continuity, which is typically not satisfied in SPNs and their tSPNs.

VI. CONCLUSION

This brief has mapped an SPN with d variables onto a d-way tensor named tensor SPN or tSPN. The transformation of the latter into a TT then allows inherent sharing of originally distributed weights in an SPN tree, thereby leading to an often dramatic reduction in the number of network parameters as shown in various numerical experiments, with little or negligible loss of modeling accuracy. The TT-based tSPN also automatically guarantees a deep and narrow neural-network architecture. These promising new results have demonstrated tSPN to be a more natural canonical form for realizing an SPN compared to the existing tree structure.

REFERENCES

[1] D. Koller, N. Friedman, and F. Bach, *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA, USA: MIT Press, 2009.
[2] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proc. 27th Conf. Uncertainty Artif. Intell.*, Nov. 2011, pp. 337–346.
[3] M. R. Amer and S. Todorovic, "Sum product networks for activity recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 4, pp. 800–813, Apr. 2016.
[4] Z. Yuan, H. Wang, L. Wang, T. Lu, S. Palaiahnakote, and C. L. Tan, "Modeling spatial layout for scene image understanding via a novel multiscale sum-product network," *Expert Syst. Appl.*, vol. 63, pp. 231–240, Nov. 2016.

[5] F. Rathke, M. Desana, and C. Schnörr, "Locally adaptive probabilistic models for global segmentation of pathological OCT scans," in *MICCAI*, M. Descoteaux, L. Maier-Hein, A. Franz, P. Jannin, D. Collins, and S. Duchesne, Eds. Cham, Switzerland: Springer, 2017, pp. 177–184.

[6] J. Wang and G. Wang, "Hierarchical spatial sum–product networks for action recognition in still images," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 1, pp. 90–100, Jan. 2018.

[7] R. Peharz, G. Kapeller, P. Mowlaee, and F. Pernkopf, "Modeling speech with sum-product networks: Application to bandwidth extension," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 3699–3703.

[8] M. Ratajczak, S. Tschiatschek, and F. Pernkopf, "Sum-product networks for sequence labeling," 2018, *arXiv:1807.02324*. [Online]. Available: https://arxiv.org/abs/1807.02324

[9] A. Pronobis, F. Riccio, and R. Rao, "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments," in *Proc. ICAPS Workshop Planning Robot.*, Jun. 2017, pp. 1–9.

[10] K. Zheng, A. Pronobis, and R. P. N. Rao, "Learning graph-structured sum-product networks for probabilistic semantic maps," in *Proc. 32nd AAAI Conf. Artif. Intell.*, Apr. 2018, pp. 4547–4555.

[11] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to FPGA-based accelerators," in *Proc. Intl. Conf. Comput. Des. (ICCD)*, Oct. 2018, pp. 1–8.

[12] R. Gens and P. Domingos, "Learning the structure of sum-product networks," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Apr. 2013, pp. 873–880.

[13] A. Vergari, N. D. Mauro, and F. Esposito, "Simplifying, regularizing and strengthening sum-product network structure learning," in *Proc. Eur. Conf. Mach. Learn. Princ. Pract. Knowl. Discovery Databases (ECML-PKDD)*, Aug. 2015, pp. 343–358.

[14] H. Zhao, P. Poupart, and G. J. Gordon, "A unified approach for learning the parameters of sum-product networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2016, pp. 433–441.

[15] C. J. Butz, J. S. Oliveira, and A. E. dos Santos, "On learning the structure of sum-product networks," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Nov./Dec. 2017, pp. 1–8.

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

[17] A. Cichocki *et al.*, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE Signal Process. Mag.*, vol. 32, no. 2, pp. 145–163, Mar. 2015.

[18] N. Cohen, O. Sharir, Y. Levine, R. Tamari, D. Yakira, and A. Shashua, "Analysis and design of convolutional networks via hierarchical tensor decompositions," 2017, *arXiv:1705.02302*. [Online]. Available: https://arxiv.org/abs/1705.02302

[19] V. Khrulkov, A. Novikov, and I. Oseledets, "Expressive power of recurrent neural networks," in *Proc. Intl. Conf. Learn. Represent. (ICLR)*, Feb. 2018, pp. 1–12.

[20] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," Dec. 2014, *arXiv:1412.6553*. [Online]. Available: https://arxiv.org/abs/1412.6553

[21] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.* Cambridge, MA, USA: MIT Press, 2015, pp. 442–450.

[22] Z. Chen, K. Batselier, J. A. K. Suykens, and N. Wong, "Parallelized tensor train learning of polynomial classifiers," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 10, pp. 4621–4632, Oct. 2018.

[23] I. V. Oseledets, "Tensor-train decomposition," *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2295–2317, Sep. 2011.

[24] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 3123–3131.

[25] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 525–542.

[26] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent.*, Oct. 2016, pp. 1–14.

[27] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, 1990, pp. 598–605.

[28] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Proc. Adv. Neural Inf. Process. Syst.*, 1993, pp. 164–171.

[29] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[30] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2015, pp. 2285–2294.

[31] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," Jul. 2016, *arXiv:1607.03250*. [Online]. Available: https://arxiv.org/abs/1607.03250

[32] M. R. Amer and S. Todorovic, "Sum-product networks for modeling activities with stochastic structure," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 1314–1321.

[33] P. Luo, X. Wang, and X. Tang, "A deep sum-product architecture for robust facial attributes analysis," in *Proc. IEEE Int. Conf. Comput. Vis.*, Dec. 2013, pp. 2864–2871.

[34] T. Rahman and V. Gogate, "Merging strategies for sum-product networks: From trees to graphs," in *Proc. UAI*, Jun. 2016, pp. 1–10.

[35] R. Gens and P. Domingos, "Discriminative learning of sum-product networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 3239–3247.

[36] A. Darwiche, "A differential approach to inference in Bayesian networks," *J. ACM*, vol. 50, no. 3, pp. 280–305, May 2003.

[37] R. Peharz, R. Gens, F. Pernkopf, and P. Domingos, "On the latent variable interpretation in sum-product networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 10, pp. 2030–2044, Oct. 2017.

[38] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework," *J. Global Optim.*, vol. 58, no. 2, pp. 285–319, Feb. 2014.

[39] K. Batselier, Z. Chen, and N. Wong, "Tensor Network alternating linear scheme for MIMO Volterra system identification," *Automatica*, vol. 84, pp. 26–35, Oct. 2017.

[40] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*, vol. 15. Philadelphia, PA, USA: SIAM, 1995.

[41] C.-Y. Ko, K. Batselier, W. Yu, and N. Wong, "Fast and accurate tensor completion with total variation regularized tensor trains," 2018, *arXiv:1804.06128*. [Online]. Available: https://arxiv.org/abs/1804.06128

[42] S. M. Kay, *Fundamentals of Statistical Signal Processing, Volume II: Detection Theory: 002*. Upper Saddle River, NJ, USA: Prentice-Hall, 1993.