Reverse-mode differentiation in arbitrary tensor network format: with application to supervised learning

Gorodetsky, Alex, A.[1] goroda@umich.edu
Safta, Cosmin[2] csafta@sandia.gov
Jakeman, John, D.[3] jdjakem@sandia.gov

[1]Aerospace Engineering, University of Michigan, 3053 FXB, 1320 Beal Avenue, Ann Arbor, MI, 48109 USA
[2]Quantitative Modeling and Analysis, Sandia National Laboratories, Livermore, CA, 94550, USA
[3]Optimization and Uncertainty Quantification, Sandia National Laboratories, Albuquerque, NM, 87123, USA

Abstract

This paper describes an efficient reverse-mode differentiation algorithm for contraction operations of tensor networks that may have arbitrary and unconventional network topologies. The approach leverages the tensor contraction tree of Evenbly and Pfeifer (2014), which provides an instruction set for the contraction sequence of a network. We show that this tree can be efficiently leveraged for differentiation of a full tensor network contraction using a recursive scheme that exploits (1) the bilinear property of contraction and (2) the property that trees have single path from root to leaves. While differentiation of tensor-tensor contraction is already possible in most automatic differentiation packages, we show that exploiting these two additional properties in the specific context of contraction sequences can improve efficiency. Following a description of the algorithm and computational complexity analysis, we investigate its utility for gradient-based supervised learning for low-rank function recovery and for fitting real-world unstructured datasets. We demonstrate improved performance over alternating least-squares optimization approaches and the capability to handle heterogeneous and arbitrary tensor network formats. When compared to alternating minimization algorithms, we find that the gradient-based approach requires a smaller oversampling ratio (number of samples compared to number model parameters) for recovery. This increased efficiency extends to fitting unstructured data of varying dimensionality and when employing a variety of tensor network formats. Here, we show improved learning using the hierarchical Tucker method over the tensor-train in high-dimensional settings on a number of benchmark problems.

Keywords— Tensor networks, supervised learning, gradient descent, alternating least squares

1 Introduction

Tensor networks [5, 6] provide a compact means to represent multidimensional functions and arrays. They have found tremendous success in the physical and chemical sciences for representing the solutions of various governing equations and for representing high dimensional functions in general. This capacity for representing high-dimensional objects has also recently been leveraged in the applied mathematics, computer science, and data science communities for both representing the solutions of a wider variety of partial differential equations and for providing the functional form in the context of supervised and unsupervised learning. In these applications, a small set of network topologies has found tremendous utility and led to substantial development. Examples include the matrix product states/tensor-train (MPS/TT) [7, 25, 18], projected entangled pair states (PEPS) [24](a higher dimensional MPS), Hierarchical Tucker [13, 9] (Tree-tensor), and the multi-scale entanglement renormalization ansatz (MERA) [26]. Our focus in this paper is to provide an algorithm to enable the development of new gradient-based algorithms for fitting these tensor networks, and also tensor networks with more arbitrary topologies.

The majority of algorithms to compress multidimensional objects (arrays or functions) into tensor network formats rely on using a multi-linear structure to reformulate an optimization problem into a sequence of quadratic subproblems within an alternating optimization scheme that optimizes for a single node of the network at a time. In these schemes, repeated sweeps across the nodes are then made to obtain a final solution. The most famous of such algorithms is alternating least squares (ALS). These algorithms have been leveraged for supervised learning applications for a number of commonly-used tensor formats such as the tensor-train decomposition [21] and the hierarchical (tree) Tucker formats [11]. These approaches can also be extended to other formats more generally, as discussed in [22]. There are several key advantages to adapting the alternating optimization approaches. The first, and primary, advantage is that they yield a sequence of tractable *linear least-squares* problems by fixing all but one or two cores at a time [3, 23, 10]. As a result, these sub-problems can utilize existing computational approaches to scalably solve such problems, and all regularization-based variations. The second advantage of these alternating optimization approaches is the ability to leverage density matrix renormalization group (DMRG) to dynamically adapt the ranks of a tensor network structure [27].

Inspired by our recent results in [8], we explore the usage of generalized gradient-based optimization algorithms for learning parameters of arbitrary tensor network formats. In our previous work we demonstrated that supervised learning in tensor-train format can be more data efficient using gradient descent than alternating least squares. More concretely, as the number of data decreased, we found that gradient-descent was able to recover the underlying function with greater accuracy. In this paper we extend these ideas to arbitrary tensor network formats, and we find similar results. The primary barrier to these algorithms is an efficient computation of the gradient. In this paper, we describe how these gradients can be computed.

Automated gradient computations have become the cornerstone of modern machine learning frameworks. They are the primary enablers of state-of-the-art optimization approaches and are the core of unstructured model learning. See for instance [19] for the description of an automatic differentiation [12] approach that PyTorch uses. Indeed automatic differentiation is itself becoming more general purpose and undergoes names such as "differentiable programming" and can leverage advances in dynamic computational graphs. For example, back-propagation in neural networks can be viewed as a single type of automatic differentiation that requires a forward and backward pass, and the major machine learning frameworks encode these on a computational graph. For a recent review of general automatic differentiation in machine learning we refer the reader to [1].

In the language of automatic differentiation, this paper describes how to perform generalized reverse-mode differentiation for the class of tensor network models. More specifically, contraction operations for tensor network must be carefully planned and executed to avoid excessive computational costs. These operations can be formally described via so-called "contraction trees" [4]. The main two contributions of this paper are then:

1. A reverse-mode differentiation algorithm for a sequence of contractions in tensor networks that builds on the idea of contraction trees of Evenbly and Pfeifer [4], and
2. A demonstration of improved tensor-recovery resulting from gradient-based optimization as compared to alternating minimization.

Differentiation in generalized tensor network format was first referenced in [3] but an algorithm for its efficient computation was not described. Recently [16] developed differentiable programming for computing the gradients of linear algebra operations on tensors (but not tensor networks) including singular value decomposition, eigenvalue decomposition, and QR factorization. Similar to our previous work [8], the authors were then able to show that optimization-based algorithms were indeed able to outperform their ALS counterparts.

In this paper we present specialized algorithms for the types of tensor contractions necessary to compute gradients through contraction sequences in tensor networks. In particular, Section 4.1 highlights how to perform efficient back-propagation for the particular computational graph topologies that are found in tensor contraction applications. We show that the efficiency of standard chain-rule capabilities, which an out-of-the-box solution like PyTorch provide, can be increased by further exploiting bilinearity and the structure of a tensor contraction tree that encodes a contraction ordering.

The remainder of this paper is structured as follows: Section 2 provides the background on our notation and tensor contraction, Section 3 describes tensor networks and the idea of tensor contraction trees as an instruction set, Section 4 describes reverse-mode differentiation on this contraction set, and Section 5 describes how we apply our technology to supervised learning and provides numerous examples on a wide-ranging set of tensor network structures.

2 Notation and contractions

In this section we provide a definition for tensor contraction and introduce the necessary notation. We follow the notation of [15] for representing tensors, which for the purposes of this paper are multidimensional arrays. Tensors are denoted by uppercase bold Euler script, e.g. $\boldsymbol{\mathcal{A}}$. Elements of any dimension are denoted by lower-case letters with subscripts denoting indices, e.g., a_i, a_{ij}, and a_{ijk}. The indices range from 1 to their capital version, e.g., $i = 1, \ldots, I$. The dimensions of a tensor are called its "modes", where I refers to the mode sizes in the previous sentence. It is useful to permute the modes of a tensor to clarify certain operations like contractions. Let $\mathbb{Z}_+$ denote the positive integers and $\sigma : \mathbb{Z}_+ \to \mathbb{Z}_+$ denote a permutation function (one-to-one). Two tensors $\tilde{\boldsymbol{\mathcal{A}}}$ and $\boldsymbol{\mathcal{A}}$ are considered equal up to a permutation if we have

$$\tilde{a}_{i_1,\ldots,i_n} = a_{\sigma(i_1,\ldots,i_n)} \qquad \forall i_1, \ldots, i_n. \tag{1}$$

We will also ease notation by sending subsets of the full index sets, e.g. $(j_1, j_2) = \sigma(i_1, i_2)$, through the permutation function. In these cases, it will always be clear that the inputs and outputs are subsets of a larger space. Permutations can be interpreted as a reshaping of the tensor that results in a tensor of the same order.

We denote tensor-tensor multiplication (contraction) using the shorthand $\boldsymbol{\mathcal{A}}\ {}_{i}\!\times_{j}\ \boldsymbol{\mathcal{B}}$ where the indices on the left correspond to the contraction indices of the left tensor and the indices are on the right correspond to those for the right array. When it is clear from context, we will ignore i and j, and just write $\boldsymbol{\mathcal{A}} \times \boldsymbol{\mathcal{B}}$. As an example, let $\boldsymbol{\mathcal{A}}$ have size $I_1 \times I_2 \times I_3 \times I_4$ and $\boldsymbol{\mathcal{B}}$ have size $J_1 \times J_2 \times J_3$, then for the contraction $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\ {}_{3}\!\times_{2}\ \boldsymbol{\mathcal{B}}$ we require $I_3 = J_2$ and

$$\boldsymbol{\mathcal{C}} \in \mathbb{R}^{I_1 \times I_2 \times I_4 \times J_1 \times J_3}, \quad \text{and} \quad c_{i_1 i_2 i_4 j_1 j_3} = \sum_{k=1}^{I_3} a_{i_1 i_2\, k\, i_4} b_{j_1\, k\, j_3},$$

where the common dimension between the two tensors disappears and the dimensions not shared by the tensors are concatenated. Note that the requirement $I_3 = J_2$ refers to the fact that the third dimension of tensor $\boldsymbol{\mathcal{A}}$ must be the same as the second dimension of tensor $\boldsymbol{\mathcal{B}}$. The following statement generalizes this simple example to define contractions over more than one mode.

Definition 1 (Tensor contraction). *A tensor contraction is a binary operation on two tensors $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_{d_A}}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{J_1 \times \cdots \times J_{d_B}}$ yielding a tensor $\boldsymbol{\mathcal{C}}$. The operation is parameterized by two index sets, $\Lambda = \{\lambda_1, \ldots, \lambda_\ell\}$ and $\Upsilon = \{\eta_1, \ldots, \eta_\ell\}$, satisfying three conditions:*

1. *$1 \le \lambda_k \le d_A$ for each $\lambda_k \in \Lambda$*
2. *$1 \le \eta_k \le d_B$ for each $\eta_k \in \Upsilon$*
3. *$I_{\lambda_k} = J_{\eta_k}$ for $k = 1, \ldots, \ell$*

The operation, denoted by $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\ {}_{\Lambda}\!\times_{\Upsilon}\ \boldsymbol{\mathcal{B}}$, is defined as follows. Let $\sigma_A(i_1, \ldots, i_{d_A}) = (\Lambda, j_1, \ldots, j_{d_A - \ell})$ define a permutation of the modes of $\boldsymbol{\mathcal{A}}$ so that the contraction dimensions Λ are first, and the rest of the modes are in increasing order $j_1 < j_2 < \cdots < j_{d_A-\ell}$. Let $\tilde{\boldsymbol{\mathcal{A}}}$ denote the tensor permuted according to this ordering. Let $\sigma_B(i_1, \ldots, i_{d_B}) = (\Upsilon, k_1, \ldots, k_{d_B-\ell})$ define a permutation of the modes of $\boldsymbol{\mathcal{B}}$ so that the contraction dimensions Υ are first, and the rest of the modes are in increasing order $k_1 < k_2 < \cdots < k_{d_B-\ell}$. Let $\tilde{\boldsymbol{\mathcal{B}}}$ denote the equivalent tensor permuted according to this ordering. Now we have $\Lambda = \Upsilon$ corresponding to the first ℓ modes of $\tilde{\boldsymbol{\mathcal{A}}}$ and $\tilde{\boldsymbol{\mathcal{B}}}$, and the tensor $\boldsymbol{\mathcal{C}}$ is obtained by summing over the first ℓ modes of these tensors

$$c_{j_1,\ldots,j_{d_A-\ell},k_1,\ldots,k_{d_B-\ell}} = \sum_{\lambda_1=1}^{I_{\lambda_1}} \cdots \sum_{\lambda_\ell=1}^{I_{\lambda_\ell}} \tilde{a}_{\lambda_1,\ldots,\lambda_\ell,j_1,\ldots,j_{d_A-\ell}} \tilde{b}_{\lambda_1,\ldots,\lambda_\ell,k_1,\ldots,k_{d_B-\ell}}, \tag{2}$$

with $\boldsymbol{\mathcal{C}}$ having order $d_A + d_B - 2\ell$.

Using this definition we can contract tensors by: (1) reshaping the tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ into appropriately sized matrices, (2) performing matrix-matrix multiplication, and (3) reshaping the result into an appropriately shaped and ordered tensor. Using Definition 1 tensor-vector contraction can be written

$$\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\ {}_{n}\!\times_{1}\ \mathbf{b}, \tag{3}$$

$$c_{i_1 \ldots i_{n-1} i_{n+1} \ldots i_d} = \sum_{k=1}^{I_n} a_{i_1 \ldots i_{n-1}\, k\, i_{n+1} \ldots i_d} b_k, \tag{4}$$

where $\mathbf{b} \in \mathbb{R}^{I_n}$, and the tensor-matrix contraction can be expressed as

$$\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\ {}_{n}\!\times_{1}\ \mathbf{B}, \tag{5}$$

$$c_{i_1 \ldots i_{n-1} i_{n+1} \ldots i_d j} = \sum_{k=1}^{I_n} a_{i_1 \ldots i_{n-1}\, k\, i_{n+1} \ldots i_d} b_{kj}, \tag{6}$$

where $\mathbf{B} \in \mathbb{R}^{I_n \times J}$ has the same number of rows, I_n, as the n-th mode of $\boldsymbol{\mathcal{A}}$. The tensor order of $\boldsymbol{\mathcal{C}}$ is the same as that of $\boldsymbol{\mathcal{A}}$ — $\ell = 1$ and $d_B = 2$ in Definition 1 so that $d_A + 2 - 2\ell = d_A$. Tensor-matrix contractions are often used to represent a change of basis, and it is common to additionally permute the indices of c after contraction so that the corresponding indices are in the same dimensions as in the original tensor.

Finally, our notation uses superscripts in parentheses to denote elements in a sequence or set, for instance $\boldsymbol{\mathcal{A}}^{(k)}$ is the k-th tensor in a sequence. In this notation, the tensor-train representation [18] of a tensor $\boldsymbol{\mathcal{A}}$ of order d is

$$\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{A}}^{(1)}\ {}_{3}\!\times_{1}\ \boldsymbol{\mathcal{A}}^{(2)}\ {}_{3}\!\times_{1}\ \cdots\ {}_{3}\!\times_{1}\ \boldsymbol{\mathcal{A}}^{(d)}, \tag{7}$$

with cores $\boldsymbol{\mathcal{A}}^{(k)} \in \mathbb{R}^{r_{k-1} \times I_k \times r_k}$. Since $r_0 = r_d = 1$ the final tensor is scalar-valued and has order d with mode sizes I_k. If $r_0 \neq 1$ or $r_d \neq 1$ then one can either interpret the new tensor as having order $d+1$ with mode sizes $I_1, \ldots, I_d, r_d$ (for the case where r_d is not 1) or one can interpret it as a vector-valued tensor $\boldsymbol{\mathcal{A}} : \mathbb{R}^{I_1} \times \cdots \times \mathbb{R}^{I_d} \to \mathbb{R}^{r_d}$.

3 Tensor Networks

In this section, we describe tensor networks and their contraction paths according to contraction trees. We work with the following definition of a tensor network.

Definition 2 (Tensor Network)**.** *A tensor network is a connected directed graph $\mathcal{TN} = (V, E)$ where each vertex $\boldsymbol{\mathcal{V}}^{(i)} \in V$ is a tensor of order $d^{(i)}$ and the edges denote contractions. An edge $E^{(ij)}$ from vertex $\boldsymbol{\mathcal{V}}^{(i)}$ to vertex $\boldsymbol{\mathcal{V}}^{(j)}$ is a pair of multi-indices $E^{(ij)} = \{\mathbf{i}, \mathbf{j}\}$ and denotes the contraction $\boldsymbol{\mathcal{V}}^{(i)}\ {}_{\mathbf{i}}\times_{\mathbf{j}}\ \boldsymbol{\mathcal{V}}^{(j)}$. The first element is itself an ordered set of indices and is referenced as $E_1^{(ij)} \triangleq \mathbf{i}$. Similarly, the second element will be referenced as $E_2^{(ij)} \triangleq \mathbf{j}$. Moreover, a tensor network has the property that if a vertex has multiple incoming or outgoing edges, then the contraction indices are not allowed to intersect (a node cannot contract with more than one other node along the same index).*

Definition 2 is sometimes called *tensor network states*, and is similar to many others found in the literature [3, 28, 29]. The following minor difference arise because we have explicitly defined how mode ordering works in a tensor contraction. Firstly, our network is lexigraphically ordered with directed to respect the order of contractions. This is a superficial difference because any two directed graphs with the same underlying undirected graph give isomorphic tensor networks. However, we believe that this construction more clearly enables stable contraction that preserves the tensor network node orderings. Second, we force this graph to be connected to avoid pathological classes that are not of interest here.

A tensor that is a vertex of the network may have modes that are uncontracted. These can be viewed as "free" edges in the graph. The result of contracting all the tensors in a network will be an *equivalent* tensor whose modes will be the concatenation of all free (uncontracted modes) of the vertices. The following definition of the *free* order of each vertex in a tensor network will be particularly useful for discussing the order of a tensor represented by a tensor network.

Definition 3 (Free order $d_f^{(i)}$ of a vertex)**.** *The free order $d_f^{(i)}$ of a vertex $\boldsymbol{\mathcal{V}}^{(i)}$ of a tensor network is the number of modes which are not connected to any other vertex*

$$d_f^{(i)} = d^{(i)} - \sum_{E^{(ij)} \in E} |E_1^{(ij)}| - \sum_{E^{(ki)} \in E} |E_2^{(ki)}|. \tag{8}$$

In other words, it is the number modes that are not involved in contractions with other vertices. Note that the intersection $E_1^{(ij)} \cap E_1^{(i\ell)} = E_1^{(ij)} \cap E_2^{(ki)} = E_2^{(mi)} \cap E_2^{(ki)} = \emptyset$ — for all j, k, ℓ, m where $E^{(ij)}, E^{(i\ell)}, E^{(ki)}$, and $E^{(mi)}$ are all in E — is empty due to the restriction on shared contraction indices of Definition 2.

The contraction operation represented by an edge of a tensor network can be executed to form a new tensor network that combines the relevant nodes and creates several edge modifications. Contracting an edge $E^{(ij)}$ refers to the operation

$$\boldsymbol{\mathcal{W}} = \boldsymbol{\mathcal{V}}^{(i)}\ {}_{E_1^{(ij)}}\times_{E_2^{(ij)}}\ \boldsymbol{\mathcal{V}}^{(j)}. \tag{9}$$

Recall from Definition 1 that the modes of $\boldsymbol{\mathcal{W}}$ are obtained via the permutation functions $\sigma_{\boldsymbol{\mathcal{V}}^{(i)}}$ and $\sigma_{\boldsymbol{\mathcal{V}}^{(j)}}$. Let $|E_1^{(i)}| = \ell$, then the first $d^{(i)} - \ell$ indices correspond to the modes of $\boldsymbol{\mathcal{V}}^{(i)}$ that are not involved in the contraction, and the last $d^{(j)} - \ell$ correspond to the modes of $\boldsymbol{\mathcal{V}}^{(j)}$ that are not involved in the contraction. These two sets of nodes must be "reattached" to the neighbors of $\boldsymbol{\mathcal{V}}^{(i)}$ and $\boldsymbol{\mathcal{V}}^{(j)}$ according to the edges that are going into and out of them. This rearrangement requires mapping the original modes of each of the neighbor tensors into the appropriate modes of $\boldsymbol{\mathcal{W}}$. This mapping is provided by the permutations $\sigma_{\boldsymbol{\mathcal{V}}^{(i)}}$ and $\sigma_{\boldsymbol{\mathcal{V}}^{(j)}}$. For instance if mode n of $\boldsymbol{\mathcal{V}}^{(i)}$ is not involved in the contraction leading to $\boldsymbol{\mathcal{W}}$, then a new edge must be established between $\sigma_{\boldsymbol{\mathcal{V}}^{(i)}}(n)$ and the tensor to which $\boldsymbol{\mathcal{V}}^{(i)}$ was previously attached. This argument is explicitly provided by the following result.

Proposition 1 (Edge contraction)**.** *Given a tensor network (V, E), performing the contraction operation Eq. (9), defined by $E^{(ij)}$, forms a new tensor network (W, F) with vertices*

$$W = \{\tilde{\boldsymbol{\mathcal{W}}}\} \cup \bigcup_{\substack{\boldsymbol{\mathcal{V}}^{(k)} \in V \\ k \neq i,j}} \boldsymbol{\mathcal{V}}^{(k)} \tag{10}$$

where $\tilde{\boldsymbol{\mathcal{W}}} = \boldsymbol{\mathcal{V}}^{(i)}\ {}_{E_1^{(ij)}}\times_{E_2^{(ij)}}\ \boldsymbol{\mathcal{V}}^{(j)}$ is the result of contracting the two nodes of the vertex. Furthermore, the edges are the union of unmodified and modified edges $F = E_u \cup E_m$. The unmodified edges

$$E_u = \bigcup_{\substack{E^{(k\ell)} \in E \\ k,\ell \neq i,j}} E^{(k\ell)} \tag{11}$$

Algorithm 1 `contract-sub-tree`: Tensor network contraction according to a contraction tree.

```
Inputs Contraction tree $\mathcal{C}$, Edge tensor $\boldsymbol{\mathcal{W}}$
1: if $\boldsymbol{\mathcal{W}}$ is a leaf or has been visited then
2:     return $\boldsymbol{\mathcal{W}}$
3: else
4:     Mark $\boldsymbol{\mathcal{W}}$ visited
5:     $\boldsymbol{\mathcal{W}}^{(l)} =$ contract-sub-tree($\mathcal{C}$, left-parent($\mathcal{C}, \boldsymbol{\mathcal{W}}$))
6:     $\boldsymbol{\mathcal{W}}^{(r)} =$ contract-sub-tree($\mathcal{C}$, right-parent($\mathcal{C}, \boldsymbol{\mathcal{W}}$))
7:     $E =$ get-contraction-indices($\mathcal{C}, \boldsymbol{\mathcal{W}}^{(l)}, \boldsymbol{\mathcal{W}}^{(r)}$)
8:     return $\boldsymbol{\mathcal{W}}^{(l)} \;{}_{E_1}\!\times_{E_2}\; \boldsymbol{\mathcal{W}}^{(r)}$
9: end if
```

are those edges in E that do not go into or out of either $\boldsymbol{\mathcal{V}}^{(i)}$ or $\boldsymbol{\mathcal{V}}^{(j)}$.

The modified edges reattach the neighbors of $\boldsymbol{\mathcal{V}}^{(i)}$ and $\boldsymbol{\mathcal{V}}^{(j)}$ to the required modes of $\tilde{\boldsymbol{\mathcal{W}}}$

$$E_m = \left[\bigcup_{\substack{E^{(ik)} \in E \\ k \neq j}} \left(\sigma_{\boldsymbol{\mathcal{V}}^{(i)}}(E_1^{(ik)}), E_2^{(ik)}\right)\right] \bigcup \left[\bigcup_{\substack{E^{(ki)} \in E \\ k \neq j}} \left(E_1^{(ki)}, \sigma_{\boldsymbol{\mathcal{V}}^{(i)}}(E_2^{(ki)})\right)\right] \bigcup \tag{12}$$

$$\left[\bigcup_{\substack{E^{(jk)} \in E \\ k \neq i}} \left(\sigma_{\boldsymbol{\mathcal{V}}^{(j)}}(E_1^{(jk)}), E_2^{(jk)}\right)\right] \bigcup \left[\bigcup_{\substack{E^{(kj)} \in E \\ k \neq i}} \left(E_1^{(kj)}, \sigma_{\boldsymbol{\mathcal{V}}^{(j)}}(E_2^{(kj)})\right)\right] \tag{13}$$

where the first union modifies edges going from $\boldsymbol{\mathcal{V}}^{(i)}$ to its neighbors other than $\boldsymbol{\mathcal{V}}^{(j)}$, the second refers to edges coming into $\boldsymbol{\mathcal{V}}^{(i)}$ from its neighbors, and the last two are the same modifications to the edges associated with $\boldsymbol{\mathcal{V}}^{(j)}$.

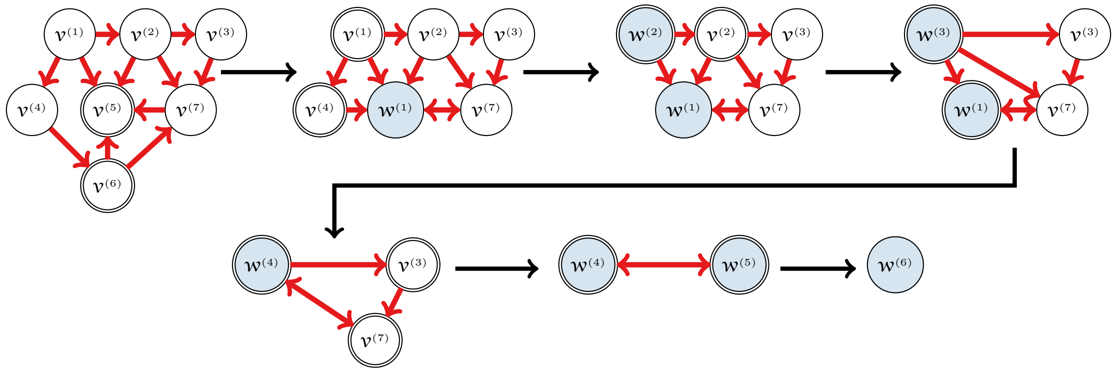
Following the contraction of the edge, the tensor network has one fewer nodes. Proceeding in this fashion, one can continue contracting each set of neighbors until only one node remains. This remaining node will have order $d = \sum_{i=1}^{|V|} d_f^{(i)}$, where V is the original set of nodes. As such, we see that every tensor network represents a tensor of order equal to the number of free dimensions. We will call this the *equivalent* tensor.

When performing these contractions, for instance to evaluate an element of the equivalent tensor, it is prudent to order the operations to achieve significant reduction in computational cost. While a full path-finding scheme utilizing breadth-first, depth-first, and/or other dynamic programming approaches can be used to find the optimal contraction order, these approaches are NP-hard [2] and therefore expensive for tensor networks with a large numbers of nodes. More recently, efficient modified search approaches that depend on several heuristics have been shown to provide significant benefit [20].

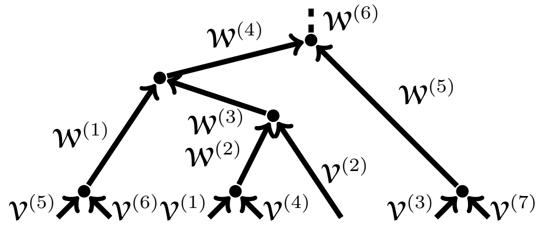
Following [4] we can represent any contraction order via a so-called *contraction tree.* Such a contraction tree can then be used to evaluate any element of the equivalent tensor of a tensor network. An example, adapted from [4], is provided in Figure 1.

Definition 4 (Contraction tree [4])**.** *A contraction tree $\mathcal{C}$ is a rooted, unbalanced binary tree with edges directed from the child to the parent nodes. The edges of the tree are representative of tensors, with edges at maximal depth representing the initial tensors (vertices) of the tensor network and internal edges representing tensors obtained in intermediate stages of contraction. Each vertex of the tree represents the contraction of a pair of tensors, and the root vertex of the tree represents the contraction of two tensors into the final equivalent tensor*

Given a contraction tree $\mathcal{C}$, contracting a tensor network to its equivalent tensor, or element of the equivalent tensor by fixing the free tensor modes, can be performed via a breadth or depth-first pass through the tensor elements. An example algorithm is provided by Algorithm 1. This algorithm is written as a recursive depth-first search over edges (tensors). For the example in Figure 1, the initial call would then be `contract-sub-tree`$(\mathcal{C}, \boldsymbol{\mathcal{W}}^{(6)})$. The algorithm then makes a depth-first traversal contracting the parent edges until completion. This algorithm makes two standard calls available for tree-type structures: the call to `left-parent` and `right-parent` get the two tensors whose contraction forms the current tensor, and `get-contraction-indices` gets the value of the tree-node for which the specified tensors are the incoming edges.



(a) Sample contraction sequence converting a tensor network into a scalar. Double circles indicate the pair of nodes contracted at the corresponding operation. Blue circles indicate intermediate tensors formed by the sequence of contractions.



(b) Contraction tree representation for the contraction sequence in sub-figure 1a. The maximal depth edges correspond to the initial nodes of the tensor network, and the nodes correspond to contraction. The internal edges represent intermediate tensors, and the top dashed edge corresponds to the final contraction result.

Figure 1: Example tensor network contraction and corresponding contraction tree. Adapted from [4].

Algorithm 2 `contract-sub-tree-deriv`: Differentials of the equivalent tensor with respect to all nodes in a tensor network, These are stored within the contraction tree

```
Inputs Contraction tree $\mathcal{C}$, Intermediate tensor $\mathcal{W}$, propagated differential $d\mathcal{W}$
1: if $\mathcal{W}$ is a leaf or has been visited then
2:     return Finished
3: else
4:     Mark $\mathcal{W}$ visited
5:     $\mathcal{W}^{(l)}$ = left-parent($\mathcal{C}, \mathcal{W}$)
6:     $\mathcal{W}^{(r)}$ = right-parent($\mathcal{C}, \mathcal{W}$)
7:     $d\mathcal{W}^{(l)}$ = chain-rule-update($d\mathcal{W}, \mathcal{W}^{(r)}$) {Compute differentials of the output with respect to the left branch of the sub tree}
8:     contract-sub-tree-deriv($\mathcal{C}, \mathcal{W}^{(l)}, d\mathcal{W}^{(l)}$) {Depth first recursion}
9:     $d\mathcal{W}^{(r)}$ = chain-rule-update($d\mathcal{W}, \mathcal{W}^{(l)}$) {Compute differentials of the output with respect to the right branch of the sub tree}
10:    contract-sub-tree-deriv($\mathcal{C}, \mathcal{W}^{(r)}, d\mathcal{W}^{(r)}$) {Depth first recursion}
11: end if
```

4 Reverse-mode differentiation

We now discuss how to leverage the tensor network machinery to obtain derivative information about a full tensor network contraction. Specifically, our goal is to obtain the derivatives of the equivalent tensor elements with respect to each element of every tensor $\mathcal{V}^{(i)}$ in the tensor network. We will show that this derivative computation can utilize the same contraction tree as the element evaluation.

Algorithm 2 recursively traverses an *evaluated* contraction tree to compute the derivative of an element of the output with respect to *all elements* of all tensor network nodes. We emphasize that prior to using this algorithm, the tensor must be contracted according to a given tensor contraction tree, and all the intermediate tensors $\mathcal{W}^{(k)}$ need to be available. Then this algorithm is initialized via the differential of the root of the contraction tree.

In this section we describe this algorithm in detail, and provide the three properties that Algorithm 2 exploits for efficiently computing the derivative through chain rule on the tensor contraction tree. We then provide an example to concretely illustrate the inefficiencies caused by using a derivative computation that does not leverage these properties. Finally, we analyze the algorithm's computational complexity.

4.1 Algorithm overview

Algorithm 2 computes the derivative of a single element of a tensor to all elements of the tensor network nodes. This algorithm is best described by first considering the general process of differentiation through a nested set of bilinear mappings (contractions), and then specializing this procedure by leveraging the binary nature of tensor contraction tree.

4.1.1 Exploiting bilinearity

Algorithm 2 exploits bilinearity to efficiently compute derivatives. Let $d\mathcal{W}$ be a differential representing the derivative of some element of the tensor $\mathcal{W}$ with respect to some parameter θ. The tensor $\mathcal{W}$ is formed by the contraction of its parents $\mathcal{W}^{(l)}$ and $\mathcal{W}^{(r)}$ according to

$$\mathcal{W}(\theta) = \mathcal{W}^{(l)}(\theta) \times \mathcal{W}^{(r)}(\theta), \tag{14}$$

where we have written that all the tensors implicitly depend on θ. The bilinear nature of the contraction suggests that the chain rule is now given as

$$d\mathcal{W}(\theta) = d\mathcal{W}^{(l)}(\theta) \times \mathcal{W}^{(r)}(\theta) + \mathcal{W}^{(l)} \times d\mathcal{W}^{(r)}(\theta) \tag{15}$$

Algorithm 2 then proceeds recursively in a depth-first search to evaluate $d\mathcal{W}^{(l)}$ and $d\mathcal{W}^{(r)}$. The recursion terminates after returning $d\mathcal{V}^{(i)}(\theta)$ for all nodes in the network.

4.1.2 Single paths from leaves to root

The second property that we exploit is the fact that the specific parameter of interest is exactly the list of nodes in the tensor network (the leaves of the tensor contraction tree) $\theta = [\mathcal{V}^{(1)}, \mathcal{V}^{(2)}, \ldots]$. As a result the differential of each leaf is only non-zero when considering the derivative with respect to the elements of that leaf $d\mathcal{V}^{(i)}(\mathcal{V}^{(j)}) = \delta_{ij}$. Then by noting that each leaf has only one path to the root of a rooted binary-tree, we see that Equation (15) only has one term, that is

$$d\mathcal{W}(\mathcal{V}^{(i)}) = \begin{cases} d\mathcal{W}^{(l)}(\mathcal{V}^{(i)}) \times \mathcal{W}^{(r)}(\theta) & \text{if } \mathcal{V}^{(i)} \in \textit{ancestors}\left(\mathcal{W}^{(l)}\right) \\ \mathcal{W}^{(l)}(\theta) \times d\mathcal{W}^{(r)}(\mathcal{V}^{(i)}) & \text{otherwise } (\mathcal{V}^{(i)} \in \textit{ancestors}\left(\mathcal{W}^{(r)}\right)) \end{cases} \tag{16}$$

Here we have specified that the non-differential terms depend on the full θ, whereas the differential terms depend only on the leaf of interest. In practice, the non-differential terms only depend on all leaves in their ancestry, which is a subset of θ. These two recursions are represented on Lines 8 and 10 in the algorithm.

4.1.3 Chain rule update (`chain-rule-update`)

We now describe the final component of Algorithm 2, the chain rule update in Lines 7 and 9. Let $\mathbf{i}$ denote the index of the root of the contraction tree (the tensor that the network represents), and let $\gamma = \mathcal{W}_{\mathbf{i}}^{root}$ represent this value. This component is standard in any computational graph automatic differentiation scheme, as such it is not specific to the tensor contraction case.

At each call of Algorithm 2, the propagated differential $d\mathcal{W}$ represents the derivative of γ with respect to the current intermediate tensor

$$d\mathcal{W} \equiv \frac{\partial \gamma}{\partial \mathcal{W}}. \tag{17}$$

When $\mathcal{W}$ is actually a leaf tensor, then the differential represents the derivative of the output with respect to the leaf of interest. Otherwise the recursive calls must then be passed derivatives $\frac{\partial \gamma}{\partial \mathcal{W}^{(l)}}$ and $\frac{\partial \gamma}{\partial \mathcal{W}^{(r)}}$ in Lines 8 and 10, respectively. The chain rule update then computes $d\mathcal{W}^{(l)}$ from $d\mathcal{W}$ by a simple rule

$$d\mathcal{W}_{\mathbf{j}}^{(l)} \equiv \frac{\partial \gamma}{\partial \mathcal{W}_{\mathbf{j}}^{(l)}} = \sum_{\mathbf{k}} \frac{\partial \gamma}{\partial \mathcal{W}_{\mathbf{k}}} \frac{\partial \mathcal{W}_{\mathbf{k}}}{\partial \mathcal{W}_{\mathbf{j}}^{(l)}} = \sum_{\mathbf{k}} d\mathcal{W}_{\mathbf{k}} \frac{\partial \mathcal{W}_{\mathbf{k}}}{\partial \mathcal{W}_{\mathbf{j}}^{(l)}}, \tag{18}$$

where the second equality is the definition of chain rule, where the summation over the multi-indices $\mathbf{k}$ represents the summation over all elements in $\mathcal{W}$, and the third equality is again the definition of $d\mathcal{W}$. This update is performed for all elements of $\mathcal{W}^{(l)}$, represented by multi-index $\mathbf{j}$. The process is the same for $\mathcal{W}^{(r)}$.

The only quantity that we have not yet described is the final partial derivative $\frac{\partial \mathcal{W}_k}{\partial \mathcal{W}_{\mathbf{j}}}$. This quantity is the derivative of a standard tensor contraction with respect to its argument. To make it concrete, the computation is given in Proposition 2

Proposition 2 (Derivative of the tensor contraction)**.** *Consider the contraction $\mathcal{C} = \mathcal{A} \;{}_{\Lambda}\!\times_{\Gamma}\; \mathcal{B}$ of tensors of appropriate sizes and shapes. Then the derivative of an element of $\mathcal{C}$ with respect to an element of $\mathcal{A}$ is given by*

$$\frac{\partial c_{j_1,\dots,j_{d_A-\ell},k_1,\dots,k_{d_B-\ell}}}{\partial a_{\sigma_A^{-1}(\nu_1,\dots,\nu_\ell,j_1,\dots,j_{d_A-\ell})}} = \tilde{b}_{\nu_1,\dots,\nu_\ell,k_1,\dots,k_{d_B-\ell}}, \tag{19}$$

and similarly for the derivatives with respect to $\boldsymbol{B}$. Here σ_A^{-1} is the inverse of the permutation mapping σ_A.

We then use this formula to iterate through all the terms in the summation in Equation (18).

4.2 Example of inefficiencies arising from a direct approach

In this section we apply Algorithm 2 to the tensor network in Figure 1. We use this example to highlight how the recursion avoids significant recomputation and that the single path from leaf-to-root property does indeed result in the cancellation of numerous terms. Our goal here is to determine the derivative of some element of the root node $\mathcal{W}^{(root)} \equiv \mathcal{W}^{(6)}$ with respect to the original tensor network nodes $\mathcal{V}^{(1)}$ and $\mathcal{V}^{(4)}$. Figure 2 depicts the contraction and highlights, in red, the path from the two nodes $\mathcal{V}^{(1)}$ and $\mathcal{V}^{(4)}$ to $\mathcal{W}^{(6)}$ needed to compute the desired derivative.

Beginning with the derivative with respect to $\mathcal{V}^{(1)}$, we proceed recursively. For clarity we ignore the indices of contraction and strike though terms that cancel. Specifically, we have

$$\begin{aligned} d\mathcal{W}^{(6)} &= d\mathcal{W}^{(4)} \times \mathcal{W}^{(5)} + \mathcal{W}^{(4)} \times \cancel{d\mathcal{W}^{(5)}} \\ &= \left(\cancel{d\mathcal{W}^{(1)}} \times \mathcal{W}^{(3)} + \mathcal{W}^{(1)} \times d\mathcal{W}^{(3)}\right) \times \mathcal{W}^{(5)} \\ &= \left(\mathcal{W}^{(1)} \times \left(d\mathcal{W}^{(2)} \times \mathcal{V}^{(2)} + \mathcal{W}^{(2)} \times \cancel{d\mathcal{V}^{(2)}}\right)\right) \times \mathcal{W}^{(5)} \\ &= \left(\mathcal{W}^{(1)} \times \left(\left(d\mathcal{V}^{(1)} \times \mathcal{V}^{(4)} + \mathcal{V}^{(1)} \times \cancel{d\mathcal{V}^{(4)}}\right) \times \mathcal{V}^{(2)}\right)\right) \times \mathcal{W}^{(5)} \\ &= \left(\mathcal{W}^{(1)} \times \left(\left(d\mathcal{V}^{(1)} \times \mathcal{V}^{(4)}\right) \times \mathcal{V}^{(2)}\right)\right) \times \mathcal{W}^{(5)}, \end{aligned}$$

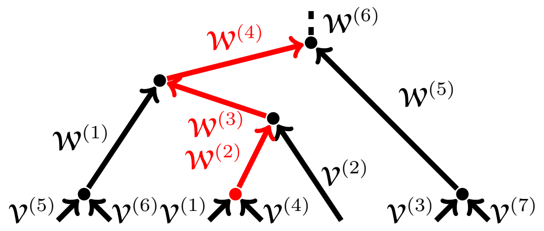


Figure 2: Reproduction of Figure 1 with the single path from leaves of interest to the root highlighted in red.

where the canceled terms are not functions of $\mathcal{V}^{(1)}$ and so are zero. We have now obtained an expression that is a function of some intermediate tensors and the network tensors. Naively, we can proceed in this manner for every differential of interest. However, this procedure would result in significant recomputation. With this in mind, consider the differential with respect to $\mathcal{V}^{(4)}$. Recursing through the contraction tree yields

$$\begin{aligned}
d\mathcal{W}^{(6)} &= d\mathcal{W}^{(4)} \times \mathcal{W}^{(5)} + \mathcal{W}^{(4)} \times \cancel{d\mathcal{W}^{(5)}} \\
&= \left(\cancel{d\mathcal{W}^{(1)}} \times \mathcal{W}^{(3)} + \mathcal{W}^{(1)} \times d\mathcal{W}^{(3)}\right) \times \mathcal{W}^{(5)} \\
&= \left(\mathcal{W}^{(1)} \times \left(d\mathcal{W}^{(2)} \times \mathcal{V}^{(2)} + \mathcal{W}^{(2)} \times \cancel{d\mathcal{V}^{(2)}}\right)\right) \times \mathcal{W}^{(5)} \\
&= \left(\mathcal{W}^{(1)} \times \left(\left(\cancel{d\mathcal{V}^{(1)}} \times \mathcal{V}^{(4)} + \mathcal{V}^{(1)} \times d\mathcal{V}^{(4)}\right) \times \mathcal{V}^{(2)}\right)\right) \times \mathcal{W}^{(5)} \\
&= \left(\mathcal{W}^{(1)} \times \left(\left(\mathcal{V}^{(1)} \times d\mathcal{V}^{(4)}\right) \times \mathcal{V}^{(2)}\right)\right) \times \mathcal{W}^{(5)}.
\end{aligned}$$

The derivative calculations, with respect to $\mathcal{V}^{(1)}$ and $\mathcal{V}^{(4)}$, can both be written in the following form

$$d\mathcal{W}^{(6)} = \left(\mathcal{W}^{(1)} \times \left(\mathcal{A} \times \mathcal{V}^{(2)}\right)\right) \times \mathcal{W}^{(5)},$$

where $\mathcal{A} = d\mathcal{V}^{(1)} \times \mathcal{V}^{(4)}$ and $\mathcal{A} = \mathcal{V}^{(1)} \times d\mathcal{V}^{(4)}$, respectively. Assuming a suitable rearrangement of contraction indices, We can rewrite this general expression as

$$d\mathcal{W}^{(6)} = \mathcal{A} \times \left(\mathcal{W}^{(1)} \times \mathcal{V}^{(2)} \times \mathcal{W}^{(5)}\right).$$

which clearly highlights the redundant computations, that is evaluating the term in brackets.

To increase the efficiency of this computation we store, along each edge, the information that is needed to compute the derivatives of its ancestors. These variables are precisely the $d\mathcal{W}^{(l)}$ and $d\mathcal{W}^{(r)}$ that are computed in each call to Algorithm 2. In the example considered we store, along the edge represented by $\mathcal{W}^{(2)}$, the variable $\mathcal{W}_d^{(2)} = \mathcal{W}^{(1)} \times \mathcal{V}^{(2)} \times \mathcal{W}^{(5)}$ which represents the information needed to compute the derivatives of the ancestors $\mathcal{V}^{(1)}$ and $\mathcal{V}^{(4)}$.

4.3 Analysis

In this section we provide a simple asymptotic upper bound on the computational cost of computing the derivative, assuming that we have already performed a forward contracting pass. The computational complexity of Algorithm 2 is dominated by the two calls to `chain-rule-update` whose formula is given in Equation (18). Suppose that $\mathcal{W}$ has dimension smaller than d and mode sizes smaller than I ($I_n < I$). Then there are $\mathcal{O}(I^d)$ terms in the summation. Furthermore, Proposition 2 shows that the gradient computation $\frac{\partial \mathbf{w}_k}{\partial \mathbf{w}_{\mathbf{j}}^{(l)}}$ simply requires picking out the appropriate element of an argument to the contraction, and so does not require any mathematical operations. If we assume that $\mathcal{W}_j^{(l)}$ also has dimension less than d and mode sizes less than I then this summation has to be computed for $\mathcal{O}(I^d)$ elements. Therefore, each of the two calls to the chain rule update requires $\mathcal{O}(I^{2d})$ operations.

This cost is incurred for every interior edge of the contraction tree. The number of edges in the contraction tree is guaranteed to be less than the number of contraction edges in the original network. Thus we incur a total cost of $\mathcal{O}(|E|I^{2d})$ operations. This result is summarized by the following proposition.

Proposition 3 (Computational complexity of the derivative)**.** *Let $\mathcal{TN} = (V, E)$ be a tensor network and $\mathcal{C}$ be an associated contraction tree. Let the dimension of each interior edge $\mathcal{W}$ in the contraction tree be lower than d and each mode size be smaller than I. Then the number of operations required for computing the derivative is $\mathcal{O}(|E|I^{2d})$.*

Note that this proposition suggests a substantional difference in the computational complexity associated with the computation of the derivative with respect to a single element of the equivalent tensor and with respect to all elements of the equivalent tensor. Namely, if we are considering a single element of the equivalent tensor, then the free edges of the tensor network are fixed to a single element and disappear. This means that the order d of the result of each contraction in the contraction tree is smaller than that of the combined orders of the parents. Once the root is reached we are left with a scalar $d = 1$. This is computationally cheaper than carrying the free edges and having a potentially growing order during the contraction sequence. Thus, it is much less expensive to obtain derivatives of a few elements of the equivalent tensor, rather than all of them.

5 Supervised learning

In this section, we highlight several applications of the derived gradients to supervised learning. We focus on the problem of real-valued regression, with the primary goal of comparing the commonly used alternating (linear) least squares (ALS) algorithm with gradient based nonlinear least squares (NLS) estimation.

To set up the supervised learning problem we assume that the tensor network represents a tensor of coefficients of a tensor-product basis. To this end, we consider the standard setting where we seek to learn a function $f_\theta : \mathcal{X} \to \mathbb{R}$ parameterized by unknowns θ. We are given N input-output pairs $(x^{(i)}, y^{(i)})_{i=1}^N$ with $x^{(i)} \in \mathcal{X}$ for all i. We denote the optimal θ as that which minimize the empirical risk with squared loss

$$\theta^* = \arg\min_\theta \frac{1}{2} \sum_{i=1}^{N} \left(f_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{20}$$

We assume the input space is a tensor-product space $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ and we choose a basis $\phi_i : \mathcal{X}_i \to \mathbb{R}^{p_i}$ for each of these inputs. The tensor network described by the parameters θ is then a $p_1 \times \ldots \times p_d$ array and the functional representation can be written as

$$f_\theta(x) = \mathcal{W}_\theta \;{}_1\!\times_1\, \phi_1(x_1) \;{}_2\!\times_1\, \phi_2(x_2) \times \cdots \;{}_d\!\times_1\, \phi_d(x_d), \tag{21}$$

where $x = (x_1, \ldots, x_d)$, and $\mathcal{W}_\theta$ is the equivalent tensor for the network represented by θ. The left index of each of the contractions above is to be understood as relating to the relevant mode of $\mathcal{W}_\theta$.

In the subsequent numerical examples we compare the minimization of Equation (20) with ALS and NLS and demonstrate that the NLS approach can be significantly more data-efficient in some cases. The gradient-based optimization using the setup above is performed via the *least_squares* function from the *scipy.optimize* python package.

5.1 Tensor network recovery statistics

In this section we compare the performance of ALS and NLS when recovering a known tensor network. In each of the examples we provide a schematic of the network under consideration and a set of probabilistic recovery results. The probabilistic recovery results perform sweeps over training set sizes. For each size of the training set we generate a set of random tensors for the nodes of the network (50 unless otherwise specified) of the given topology. For each of these randomized tensor networks, data is obtained through Equation (21) using Chebyshev polynomials as the basis functions. We then seek to recover the network using ALS and NLS. We present the relative mean squared error and probability of recovery using a validation test set of 1000 samples (a separate one for each of the 50 random networks). Accurate recovery is defined to occur when the root mean squared error (RMSE) between the predicted and exact values, at the validation samples, is smaller than 10^{-4}. We found this threshold to be sufficiently distinguishing between different qualitative performances (fitting and not fitting).

5.1.1 Two-dimensional functions

Consider the task of learning two dimensional tensor product functions with coefficient matrices $\mathbf{W} \in \mathbb{R}^{p_1 \times p_2}$. Using a tensor network we represent this coefficient matrix as $\mathbf{W} = \mathbf{V}^{(1)}\mathbf{V}^{(2)}\mathbf{V}^{(3)}$, where $\mathbf{V}^{(1)} \in \mathbb{R}^{p_1 \times r}$, $\mathbf{V}^{(2)} \in \mathbb{R}^{r \times r}$ and $\mathbf{V}^{(3)} \in \mathbb{R}^{r \times p_2}$. If the middle matrix was diagonal, and the left and right matrices were orthonormal, this would represent a Singular Value Decomposition. A schematic of this decomposition as a tensor network is shown in Figure 3. Figures 4 ($p_1 = p_2 = 3$, $r = 3$); 5 ($p_1 = p_2 = 5$, $r = 2$); and 6 ($p_1 = p_2 = 5$, $r = 4$) show the recovery behavior under the indicated settings.

Figure 4 indicates identical performance for ALS and NLS. We attribute the level of convergence to be an artifact of the convergence tolerance — not recovery performance. In this context the true coefficient matrix has nine elements, while the parameterization has $3(3) + 3(3) + 3(3) = 27$ elements. As such, this is an interesting pathological case where the network over parameterizes the underlying matrix. In this setting there are many possible solutions, and

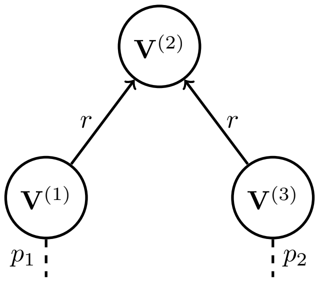


Figure 3: Tensor network for the two dimensional functions of Section 5.1.1, Free dimensions are drawn with dashed lines, indicating the network represents a matrix.

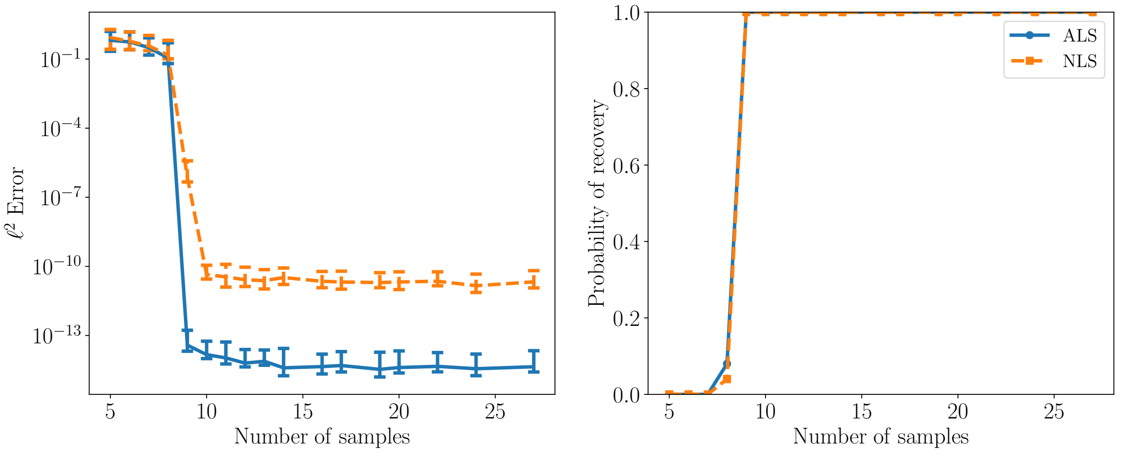


Figure 4: Recovery of two dimensional functions of Section 5.1.1 using $p_1 = p_2 = 3$, and $r = 3$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

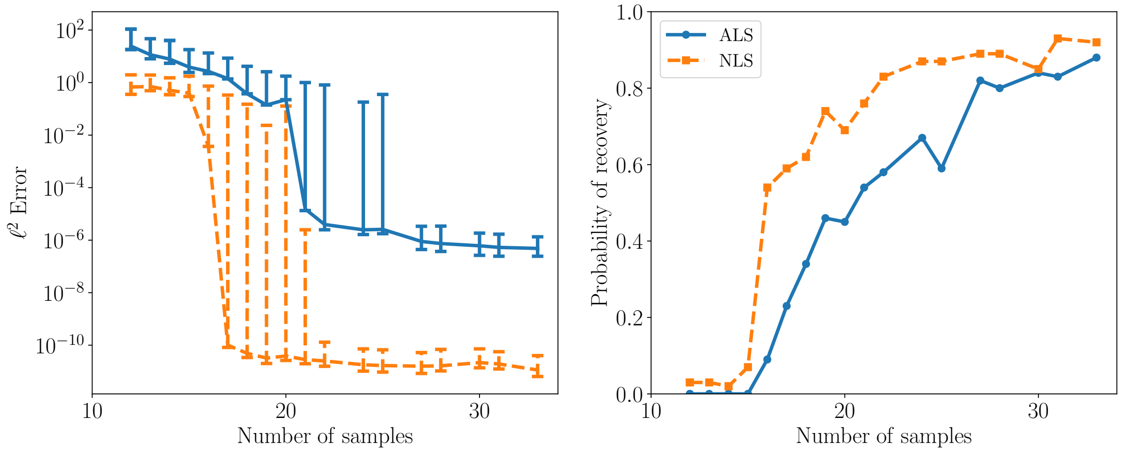


Figure 5: Recovery of two dimensional functions of Section 5.1.1 using $p_1 = p_2 = 5$, and $r = 2$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

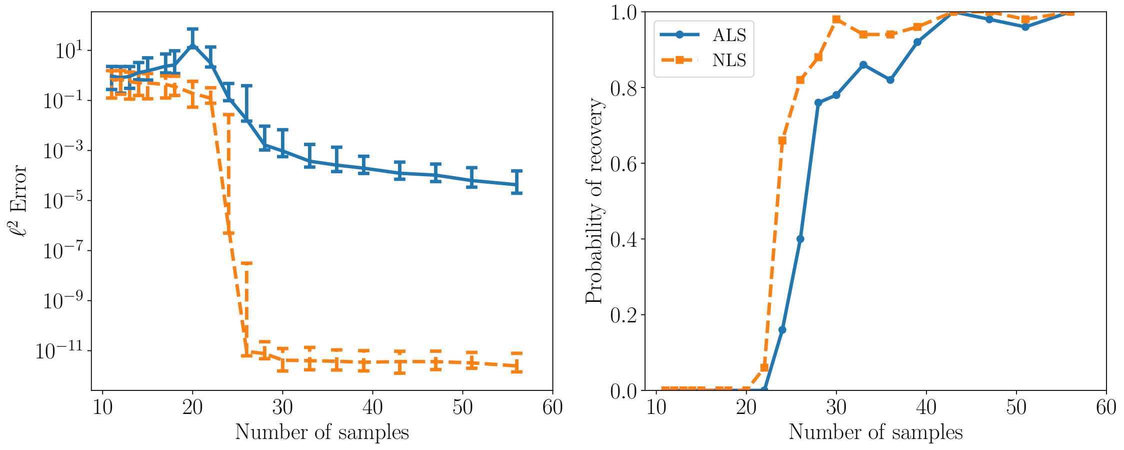


Figure 6: Recovery of two dimensional functions of Section 5.1.1 using $p_1 = p_2 = 5$, and $r = 4$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

we expect to recover them quickly. Indeed we see that both approaches recover the function once nine data points are obtained.

Figure 5 shows a clear example where we achieve more rapid recovery of the tensor using NLS than ALS with high probability. In these results, recovery using NLS occurs starting at 15 samples while ALS is delayed until 20 samples. This setting is more realistic than the previous one because the true coefficient matrix has 25 elements, while the true parameterization has $5(2) + 5(2) + 2(2) = 24$ elements. We see that NLS recovers the function with high probability almost immediately after the problem becomes over determined, while ALS needs an additional buffer of samples.

Figure 6 is another example where the tensor network over parameterizes the underlying tensor. In this setting the underlying coefficient tensor again has 25 elements, while the tensor network is now parameterized by $5(4) + 5(4) + 4(4) = 56$. Once again, we see that NLS immediately fits well once the 25 sample threshold is passed; however, now ALS begins to have trouble identifying the underlying tensor. It clearly undergoes a transition of improved performance in a similar region as NLS, but it appears that the problem is too poorly conditioned for the ALS approach to rapidly converge. Indeed we have empirically noticed that ALS has significant conditioning issues that slow down the rate of convergence.

5.1.2 Tree-tensor network recovery

Next we consider recovery with the tree-network formats. We consider the special cases of balanced and unbalanced binary trees. The latter special case is widely used and given its own name — the tensor-train representation or the

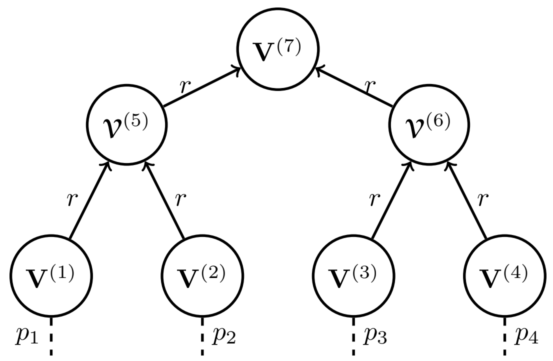


Figure 7: Binary tree-tensor network. Free dimensions are drawn with dashed lines, indicating that this network represents a four dimensional tensor. Note that the leaves are two-dimensional tensors (matrices) while the interior (non-root) nodes are three-way tensors. The root node is a matrix.

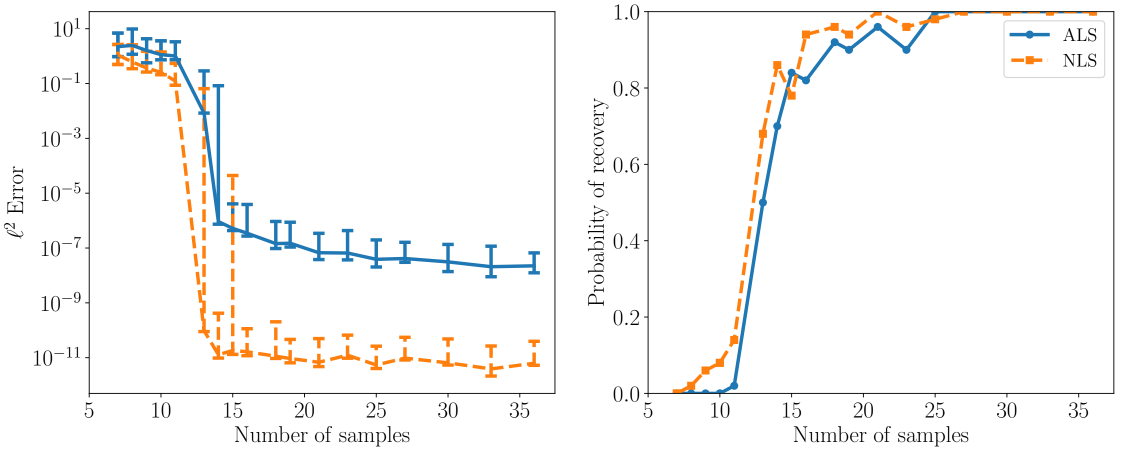


Figure 8: Recovery of four dimensional function built from balanced tree network (Figure 7) using $p_i = 2$, and $r = 2$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

matrix product states representation.

The binary tree format, for the case of four leaves, is shown in Figure 7. In this format only the leaf tensors have free modes — the internal tensors have no free edges. As a result, the tree-Tucker format represents a coefficient tensor with as many dimensions as there are leaves.

Let $p_i = p$, then the four dimensional balanced tree of Figure 7 consists of $4pr + 2r^3 + r^2$ elements. Figures 8 ($p_i = 2$, $r = 2$); 9 ($p_i = 3, r = 3$); and 10 ($p_i = 4, r = 2$) show the recovery behavior under the indicated settings.

Figure 8 corresponds to a four dimensional tensor with $2^4 = 16$ elements and is parameterized by 36 elements. This problem is over parameterized; and both ALS and NLS converge almost immediately after observing 16 data points. Figure 9 shows a similar case for a tensor with $3^4 = 81$ elements. Here the network is parameterized by 87 parameters. Here again we see similar performance between NLS and ALS — however, NLS does seem to converge 10 samples earlier than the ALS.

Practical situations will not allow over-parameterization. Figure 10 shows a case with 52 parameters in a tensor network representing a $4^4 = 256$ element coefficient tensor. Here we again see an earlier transition into convergence by NLS. This transition happens most significantly around approximately 50 samples — matching the number of parameters in the tensor. ALS again requires a slightly higher oversampling ratio to start the transition to convergence.

5.1.3 Non-traditional tensor network recovery

In this section we explore the recovery of non-traditional tensor networks — those that do not obviously fall into one of the most well studied forms (TT, Tree-Tucker, MERA, PEPS, etc..). Our aim is to show that the NLS algorithm

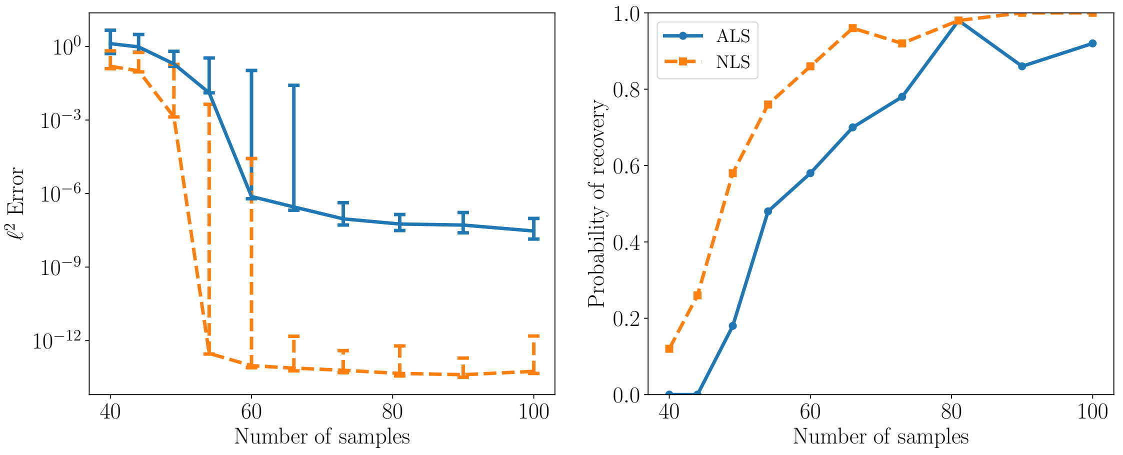


Figure 9: Recovery of four dimensional function built from balanced tree network (Figure 7) using $p_i = 3$, and $r = 3$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

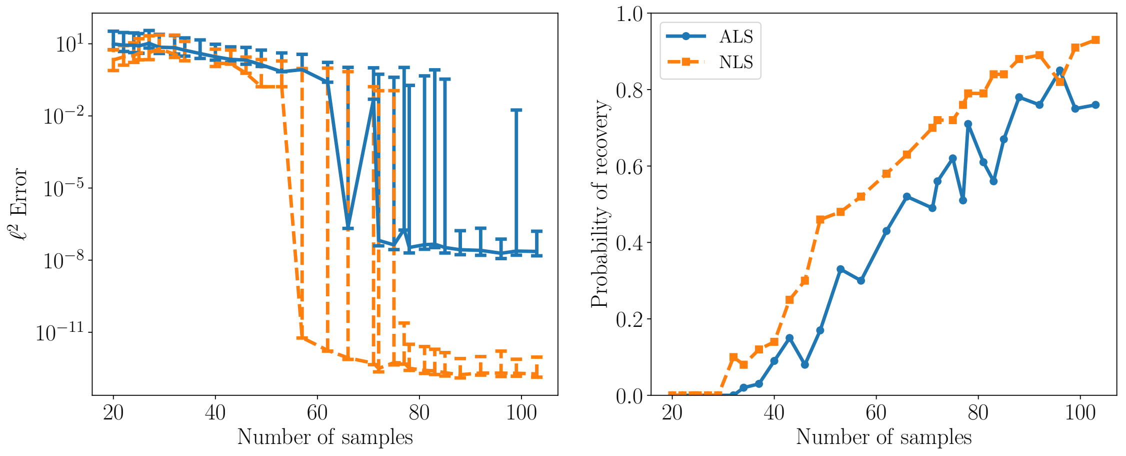


Figure 10: Recovery of four dimensional function built from balanced tree network (Figure 7) using $p_i = 4$, and $r = 2$. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

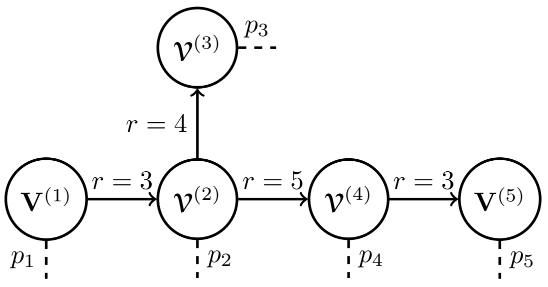


Figure 11: Unbalanced tree network with no interior nodes. Free dimensions are drawn with dashed lines, indicating that this network represents a five dimensional tensor. The left, right, and top most cores are matrices, the rest are three-way tensors.

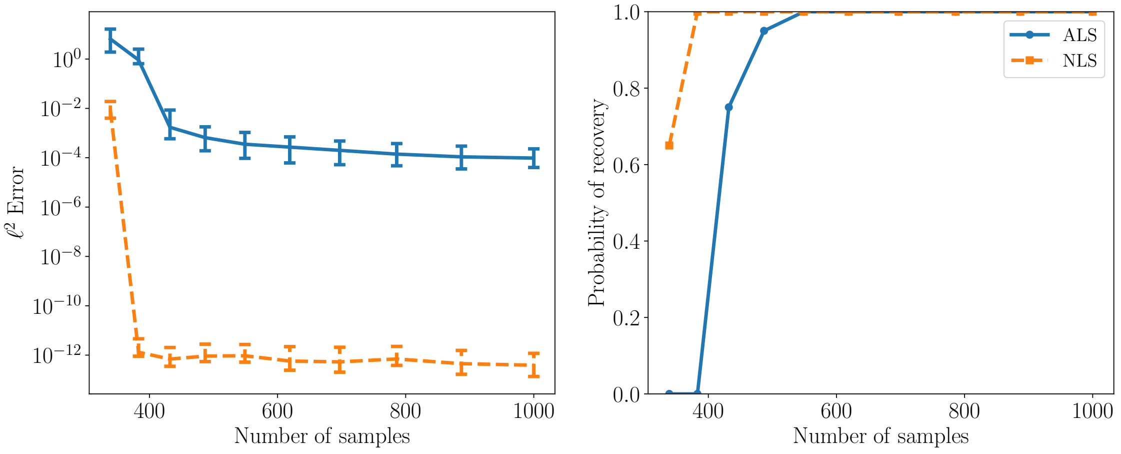


Figure 12: Recovery of a five dimensional function built from the tensor-train network (Figure 11) using $p_i = 5$ and 20 randomized tensors per sample size. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

is widely applicable — even for problem settings requiring novel tensor formats.

Consider the network shown in Figure 11. This network is not quite a Tree-Tucker because there are no internal nodes. Instead, it is similar to a tensor train where we impose a special factorization for the interaction between the second and third dimensions. Indeed, grouping these dimensions together would result in a tensor-train structure. Figure 12 shows results for a five dimensional function with ranks given in Figure 11 and $p_i = 5$. The coefficient tensor has $5^5 = 3125$ elements, while the network parameterization has 410 unknowns. Here we clearly see improved recovery using the NLS solver. Indeed it begins to converge with non-negligent probability almost one hundred samples earlier than the ALS solver. Interestingly, this convergence happens between 400—500 samples — a number commensurate with the number of unknowns.

Figure 13 represents another tree-type network with no internal nodes. However, this network is distinguished from the others because it is not a binary tree — several of the nodes have more than two outgoing edges. In other words, each node has a free index. This network can be viewed as a multi-dimensional sequence of intersecting tensor-trains. Figure 14 shows a significantly better recovery by NLS compared to ALS. This tensor network has 745 unknowns, and recovery proceeds almost immediately following this threshold of samples. The ALS algorithm needs several more times the number of samples to converge.

5.2 Real-world performance

Finally, we compare supervised learning on the same set of examples (identical training data and test data) as [8]. These data sets were originally obtained from [14] for which a set of 22 algorithms[1], 19 nonparametric and 3 parametric, were compared. The three parametric algorithms included a ridge regularized regression algorithm with linear basis functions, and two sparse regularized algorithms, LASSO and LAR. These data sets were obtained from a variety of sources and preprocessed to normalize the inputs and outputs to zero mean and a standard deviation of one. To allow estimation of prediction error, the authors of [14] randomly split each data set in half to generate a training sample set and a separate validation set. We adopt the same precise partitioning here.

In the following we measure performance via the mean squared error

$$MSE = \frac{1}{N} \sum_{i=1}^{n_{\text{validation}}} \left(\hat{f}(x^{(i)}) - y^{(i)} \right)^2,$$

over the validation set. In [8, Table 4] we showed that regression in a functional tensor train format produces results comparable to state-of-the-art, even on this unstructured and not *a-priori* known low rank problems. We repeat these

[1]Kernel Ridge Regression, k-Nearest Neighbors, Nadaraya Watson, Local Linear/Quadratic interpolation, ϵ-Support Vector Regression, ν-Support Vector Regression (nSVR), Gaussian Process Regression, Regression Trees (RT), Gradient Boosted Regression trees (GBRT), RBF interpolation, M5' Model Trees, Shepard Interpolation, Back-fitting with Cubic Splines, Multivariate Adaptive Regression Splines (MARS), Component Selection and Smoothing, Sparse Additive Models, Additive Gaussian Processes, Ridge Regression, Least Absolute Shrinkage and Selection, and Least Angle Regression

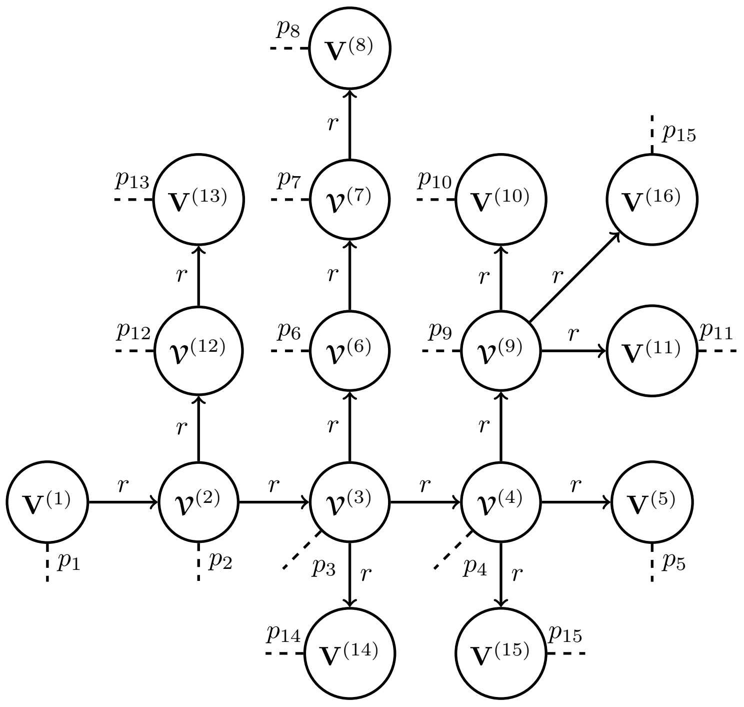


Figure 13: An unbalanced-tree network with no interior nodes. Free dimensions are drawn with dashed lines, indicating that this network represents a sixteen dimensional tensor.

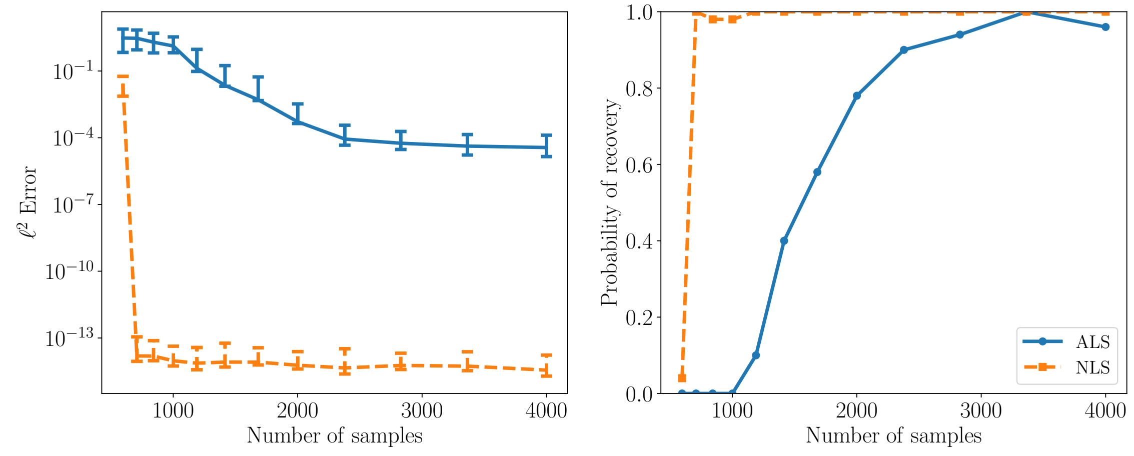


Figure 14: Recovery of a sixteen dimensional function built from the tensor network in Figure 13. This tensor network has 745 parameters, and represents a coefficient tensor with $5^{16} = 152,587,890,625$ elements. Threshold defined as recovery is specified at 10^{-2} to match the ALS convergence region. ALS approach is poorly conditioned and only converges to a level of 10^{-5}. The curves in the left plot represent the median RMSE error and the whiskers are 25% and 75% quantiles.

Table 1: Mean squared error on validation data for a subset of regression algorithms and data sets from [14]. Adapted from [8, Table 4], with the first two columns showing the unregularized and regularized functional tensor train results in that paper. In the last column we provide results of learning in the hierarchical Tucker format using the algorithms proposed here. Best case for each data set is underlined.

Datasets (d, n)	Reg. FT [8]	Unreg. FT [8]	TensNet (this paper)
Housing (12,256)	$\underline{0.32798}$	0.42790	0.3906
Galaxy (20,2000)	0.00056	0.00056	$\underline{0.000017}$
Skillcraft (18,1700)	$\underline{0.54434}$	0.67536	0.70944
CCPP (59,2000)	$\underline{0.06631}$	0.07042	0.06843
Speech (21,520)	0.02684	0.02221	$\underline{0.02327}$
Music (90,1000)	0.72094	0.78521	$\underline{0.6536}$
Telemonit (19,1000)	$\underline{0.04040}$	0.04552	0.079516
Propulsion (15,200)	$\underline{0.00009}$	0.00011	0.0006952
Airfoil (40,750)	0.46920	0.49394	$\underline{0.40708}$
Forestfires (10,211)	0.39465	0.56085	$\underline{0.352}$

experiments with a hierarchical Tucker format and the gradient-based optimization algorithms discussed in the paper. In the outer loop we sweep over univariate polynomials of degrees between 1 and 3 based on Legendre polynomials and consider tensor ranks between 2 and 4 in the inner loop. The objective function in Eq. (20) is augmented with a regularization term that penalizes the squared norm of all the unknown parameters $\lambda||\theta||_2^2$.

Next, we describe our hyperparameter tuning procedure. We start with rank 2 tensor networks that employ univariate polynomials of degree 1 on the outer nodes. The initial tensor entries are derived by perturbing the coefficients of a linear fit to the data. For each experiment, the magnitude of the penalty term λ, the rank, and the polynomial order are chosen via a 10-fold cross-validation over the training data. Once an optimal value for the regularization term magnitude λ was identified, the hyperparameter combination with the lowest cross validation error on the training data is then evaluated on the validation set, and the corresponding errors are provided in Table 1.

We observe an improved performance for 5 datasets (Housing, Speech, Music, Airfoil, Forest fires) and similar results for CCPP compared with the functional tensor-train format. We note that this improved performance is generally obtained for the higher dimensional data sets (dimensions 20, 21, 40, 90). For the second highest dimension set (CCPP), we obtained very similar results. Overall, this suggests that the hierarchical Tucker may be more suitable for higher dimensional problems. This result matches intuition as the tensor-train cannot convey information efficiently across dimensions without large dimensions due to its linear structure.

6 Conclusion

This paper introduced a general approach for reverse-mode differentiation through tensor network contractions. We utilized this approach to improve supervised learning of tensor networks. Our empirical results indicate that our gradient-based minimization of a regularized least squares objective needs a lower oversampling ratio to recover tensors in low-rank tensor network format when compared to alternating least squares minimization. Results also suggest that more complex tensor network formats can yield greater efficiency for higher dimensional problems. Our general procedure for computing derivatives of arbitrary network formats will enable a much wider adoption of tensor network formats for both standalone approximation formats and when embedded as a component of a larger workflow. For instance, we can envision the adoption of general tensor networks within neural network algorithms as reductions of the weight matrices as suggested in [17]. It can also be embedded within other applications of compression and/or optimization. Future work is needed to enable efficient use of gradient based algorithms for compositions of tensor network operations beyond contraction — for instance differentiating through a calculus of operations including addition, multiplication, etc.

Acknowledgments

A. Gorodetsky was supported by the AFOSR Computational Mathematics Program under the Young Investigator Program. C. Safta and J. Jakeman were supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract

DE-NA-0003525. The views expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[2] L. Chi-Chung, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[3] M. Espig, W. Hackbusch, S. Handschuh, and R. Schneider. Optimization problems in contracted tensor networks. *Computing and visualization in science*, 14(6):271–285, 2011.

[4] G. Evenbly and R. N. Pfeifer. Improving the efficiency of variational tensor network algorithms. *Physical Review B*, 89(24):245118, 2014.

[5] G. Evenbly and G. Vidal. Tensor network states and geometry. *Journal of Statistical Physics*, 145(4):891–918, 2011.

[6] G. Evenbly and G. Vidal. Tensor network renormalization. *Physical review letters*, 115(18):180405, 2015.

[7] M. Fannes, B. Nachtergaele, and R. F. Werner. Finitely correlated states on quantum spin chains. *Communications in mathematical physics*, 144(3):443–490, 1992.

[8] A. A. Gorodetsky and J. D. Jakeman. Gradient-based optimization for regression in the functional tensor-train format. *Journal of Computational Physics*, 374:1219–1238, 2018.

[9] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.

[10] L. Grasedyck, M. Kluge, and S. Krämer. Variants of alternating least squares tensor completion in the tensor train format. *SIAM Journal on Scientific Computing*, 37(5):A2424–A2450, 2015.

[11] E. Grelier, A. Nouy, and M. Chevreuil. Learning with tree-based tensor formats. *arXiv preprint arXiv:1811.04455*, 2018.

[12] A. Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[13] W. Hackbusch and S. Kühn. A new scheme for the tensor representation. *Journal of Fourier analysis and applications*, 15(5):706–722, 2009.

[14] K. Kandasamy and Y. Yu. Additive approximations in high dimensional nonparametric regression via the SALSA. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016.

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

[16] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang. Differentiable programming tensor networks. *Physical Review X*, 9(3):031041, 2019.

[17] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450, 2015.

[18] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[20] R. N. Pfeifer, J. Haegeman, and F. Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.

[21] E. Stoudenmire and D. J. Schwab. Supervised learning with tensor networks. In *Advances in Neural Information Processing Systems*, pages 4799–4807, 2016.

[22] E. M. Stoudenmire. Learning relevant features of data with multi-scale tensor networks. *Quantum Science and Technology*, 3(3):034003, 2018.

[23] A. Uschmajew. Local convergence of the alternating least squares algorithm for canonical tensor approximation. *SIAM Journal on Matrix Analysis and Applications*, 33(2):639–652, 2012.

[24] F. Verstraete and J. I. Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[25] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical review letters*, 91(14):147902, 2003.

[26] G. Vidal. Entanglement renormalization. *Physical review letters*, 99(22):220405, 2007.

[27] S. R. White. Density matrix formulation for quantum renormalization groups. *Physical review letters*, 69(19):2863, 1992.

[28] K. Ye and L.-H. Lim. Tensor network ranks. *arXiv preprint arXiv:1801.02662*, 2018.

[29] Y. Zhang and E. Solomonik. On stability of tensor networks and canonical forms. *arXiv preprint arXiv:2001.01191*, 2020.