A Simple and Efficient Tensor Calculus

Sören Laue,[1,2] Matthias Mitterreiter,[1] Joachim Giesen[1]
[1]Friedrich-Schiller-Universität Jena
Faculty of Mathematics and Computer Science
Ernst-Abbe-Platz 2
07743 Jena, Germany
Friedrich-Schiller-University Jena
[2]Data Assessment Solutions GmbH, Hannover

Abstract

Computing derivatives of tensor expressions, also known as tensor calculus, is a fundamental task in machine learning. A key concern is the efficiency of evaluating the expressions and their derivatives that hinges on the representation of these expressions. Recently, an algorithm for computing higher order derivatives of tensor expressions like Jacobians or Hessians has been introduced that is a few orders of magnitude faster than previous state-of-the-art approaches. Unfortunately, the approach is based on Ricci notation and hence cannot be incorporated into automatic differentiation frameworks like TensorFlow, PyTorch, autograd, or JAX that use the simpler Einstein notation. This leaves two options, to either change the underlying tensor representation in these frameworks or to develop a new, provably correct algorithm based on Einstein notation. Obviously, the first option is impractical. Hence, we pursue the second option. Here, we show that using Ricci notation is not necessary for an efficient tensor calculus and develop an equally efficient method for the simpler Einstein notation. It turns out that turning to Einstein notation enables further improvements that lead to even better efficiency.

1 Introduction

Many problems in machine learning are naturally written in terms of tensor expressions. Any algorithmic method for computing derivatives of such expressions is called a tensor calculus. Standard automatic differentiation (deep learning) frameworks like TensorFlow (Abadi and others 2016), PyTorch (Paszke et al. 2017), autograd (Maclaurin, Duvenaud, and Adams 2015), and JAX (Frostig, Johnson, and Leary 2018) are very efficient when computing derivatives of scalar-valued functions. However, evaluating the derivatives of non-scalar-valued functions, for instance, Jacobians or Hessians, in these frameworks is up to three orders of magnitude slower than evaluating the derivatives that are computed by the approach of (Laue, Mitterreiter, and Giesen 2018).

There have been some recent attempts to alleviate this lack of efficiency by accelerating the underlying linear algebra using automatic batching and optimizing the computational graphs of the derivatives (XLA and TensorFlow teams 2017). These improvements have been incorporated into TensorFlow and JAX. However, the improvements are rather small and the efficiency gap of up to three orders of magnitude still persists.

On the other hand, the approach of (Laue, Mitterreiter, and Giesen 2018) relies crucially on Ricci notation and therefore cannot be incorporated into standard deep learning frameworks that use the simpler Einstein notation. Here, we remove this obstacle and provide an efficient tensor calculus in Einstein notation. Already the simple version of our approach is as efficient as the approach by (Laue, Mitterreiter, and Giesen 2018). We provide further improvements that lead to an even better efficiency.

Ricci notation distinguishes between co- and contravariant indices, that is, upper and lower indices. This distinction is necessary in order to compute derivatives in a mathematical correct way. Consider for instance the simple expression $x^\top Ax$. If we want to compute the derivative of this expression with respect to the vector x, then, at some point, we face the problem of computing the derivative of $x^\top$. However, this derivative in Ricci notation is the delta-tensor δ_{ij} that cannot be represented in linear algebra. Note, it is not the identity matrix which is represented in Ricci notation as δ^i_j. Hence, in order to represent the derivative in a mathematical correct way, upper and lower indices are necessary. This problem has its roots in mathematical tensor analysis, where tensors are used for representing multilinear functions by their values on a set of basis vectors. These values are stored in a tensor, that is, a multi-dimensional array. Upper and lower indices are used to distinguish between vector space and dual vector space components that transform differently under basis changes. In the example expression, x is a vector while $x^\top$ is a co-vector from the dual vector space.

In machine learning tensors are typically not used for representing multi-linear functions, but simply as multi-dimensional arrays for storing data and parameters. Hence, there is no need to distinguish different types of components. Indices can just be used for accessing the different tensor components. This is basically Einstein notation that is used in all deep learning frameworks.

The contribution of this paper is an efficient and coherent method for computing tensor derivatives in Einstein notation together with a correctness proof. In reverse mode automatic differentiation, our method is equivalent to the efficient approach in (Laue, Mitterreiter, and Giesen 2018) for computing higher order derivatives. Additionally, we show that reverse mode is not optimal. A combination of reverse and forward mode, known as cross-country mode, can be more efficient. Efficiency can be further improved by compressing higher order derivatives.

For validating our framework we compute Hessians for several machine learning problems. It turns out that our method, because of the additional optimizations, outperforms the approach of (Laue, Mitterreiter, and Giesen 2018) which is already a few orders of magnitude more efficient than TensorFlow, PyTorch, autograd, and JAX.

Related Work. Many details on the fundamentals and more advanced topics of automatic differentiation can be found in the book by (Griewank and Walther 2008). (Baydin et al. 2018) provide an excellent survey on automatic differentiation for machine learning.

Computing derivatives of non-scalar-valued functions is discussed in (Pearlmutter 1994). In this approach, if the function returns an n-dimensional vector, then its derivative is computed by treating each entry as a separate scalar-valued function. The same idea is employed in almost all implementations for computing derivatives of non-scalar-valued functions. (Gebremedhin et al. 2009) introduce some optimizations based on graph coloring algorithms.

(Magnus and Neudecker 2007) can compute derivatives with respect to vectors and matrices. At the core of their approach, matrices are turned into vectors by stacking columns of a matrix into one long vector. Then, the Kronecker matrix product is used to emulate higher order tensors. This approach works well for computing first order derivatives of scalar-valued functions. However, it is not practicable for computing higher order derivatives.

(Giles 2008) collects a number of derivatives for matrix operators, i.e., pushforward and pullback functions for automatic differentiation. Similarly, (Seeger et al. 2017) provide methods and code for computing derivatives of Cholesky factorizations, QR decompositions, and symmetric eigenvalue decompositions. However, they all require that the output function is scalar-valued, and hence, cannot be generalized to higher order derivatives.

(Kakade and Lee 2018) consider non-smooth functions and provide a provably correct algorithm that returns an element from the subdifferential. However, their algorithm is also restricted to scalar-valued functions.

Another line of research focuses on automatic differentiation from a programming language point of view. The goal is to incorporate gradient computations into programming languages with the goal of fully general differentiable programming (LeCun). Work towards this goal includes the Tangent package (van Merriënboer, Moldovan, and Wiltschko 2018), the Myia project (van Merriënboer et al. 2018), and the approach of (Wang et al. 2018). So far this work is again restricted to scalar-valued functions.

Recently, also second order information has been considered for training deep nets. For instance, this information can be exploited for escaping one of the many saddle points, but also for turning the final classifier more robust (Dey et al. 2018). Furthermore, some differences in the convergence behavior can be explained by looking at the spectrum of the Hessian of the objective function (Ghorbani, Krishnan, and Xiao 2019; Sagun et al. 2018; Yao et al. 2018). However, so far it has been prohibitive to compute the full Hessian even for small networks. Here, in our experiments, we also compute the Hessian of a small neural net.

2 Einstein Notation

In tensor calculus one distinguishes three types of multiplication, namely inner, outer, and element-wise multiplication. Indices are important for distinguishing between these types. For tensors A, B, and C any multiplication of A and B can be written as

$$C[s_3] = \sum_{(s_1 \cup s_2) \setminus s_3} A[s_1] \cdot B[s_2],$$

where C is the result tensor and s_1, s_2, and s_3 are the index sets of the left argument, the right argument, and the result tensor, respectively. The summation is only relevant for inner products that in Ricci calculus are denoted by shared upper and lower indices. If one does not want to distinguish between upper and lower indices, then the summation must be made explicit through the result tensor. The standard way to do so is by excluding the index for summation from the index set of the result tensor. Hence, the index set of the result tensor is always a subset of the union of the index sets of the multiplication's arguments, that is, $s_3 \subseteq (s_1 \cup s_2)$. In the following we denote the generic tensor multiplication simply as $C = A *_{(s_1, s_2, s_3)} B$, where s_3 explicitly represents the index set of the result tensor. This notation is basically identical to the tensor multiplication `einsum` in NumPy, TensorFlow, and PyTorch, and to the notation used in the Tensor Comprehension Package (Vasilache et al. 2018).

The $*_{(s_1, s_2, s_3)}$-notation comes close to standard Einstein notation. In Einstein notation the index set s_3 of the output is omitted and the convention is to sum over all shared indices in s_1 and s_2. This, however, restricts the types of multiplications that can be represented. The set of multiplications that can be represented in standard Einstein notation is a proper subset of the multiplications that can be represented by our notation. For instance, standard Einstein notation is not capable of representing element-wise multiplications directly. Still, in the following we refer to the $*_{(s_1, s_2, s_3)}$-notation simply as Einstein notation as it is standard practice in all deep learning frameworks.

Table 1 shows examples of tensor expressions in standard linear algebra notation, Ricci calculus, and Einstein notation. The first group shows an outer product, the second group shows inner products, and the last group shows examples of element-wise multiplications. As can be seen in Table 1, Ricci notation and Einstein notation are syntactically reasonably similar. However, semantically they are quite different. As pointed out above, Ricci notation differentiates between co- and contravariant dimensions/indices and Einstein notation does not. While this might seem like a minor difference, it does have substantial implications when computing derivatives. For instance, when using Ricci notation, forward and reverse mode automatic differentiation can be treated in the same way (Laue, Mitterreiter, and Giesen 2018). This is no longer the case when using Einstein notation.

vectorized	Ricci	Einstein
$yx^\top$	$y^i x_j$	$y *_{(i,j,ij)} x$
Ax	$A^i_j x^j$	$A *_{(ij,j,i)} x$
$y^\top x$	$y_i x^i$	$y *_{(i,i,\emptyset)} x$
AB	$A^i_j B^j_k$	$A *_{(ij,jk,ik)} B$
$y \odot x$	$y^i x^i$	$y *_{(i,i,i)} x$
$A \odot B$	$A^i_j B^i_j$	$A *_{(ij,ij,ij)} B$
$A \cdot \mathrm{diag}(x)$	$A^i_j x^i$	$A *_{(ij,i,ij)} x$

Figure 1: Comparison of different linear algebra notations.

We can show that the generic tensor multiplication operator $*_{(s_1,s_2,s_3)}$ is associative, commutative, and satisfies the distributive property. Our tensor calculus, that we introduce in the next section, makes use of all three properties. By $s_1 s_2$ we denote the concatenation of the index sets s_1 and s_2. Please see the supplemental material for the easy proofs that follow directly from our definition of Einstein notation.

Lemma 1 (Associativity)**.** *Let s_1, s_2, s_3, and s_4 be index sets with $s_3 \subseteq s_1 \cup s_2$ and $s_4 \cap (s_1 \cup s_2) = \emptyset$. Then it holds that*

$$\left(A *_{(s_1,s_2s_4,s_3s_4)} B\right) *_{(s_3s_4,s_4,s_3)} C = A *_{(s_1,s_2,s_3)} \left(B *_{(s_2s_4,s_4,s_2)} C\right).$$

Unlike standard matrix multiplication tensor multiplication is commutative.

Lemma 2 (Commutativity)**.** *It holds that*

$$A *_{(s_1,s_2,s_3)} B = B *_{(s_2,s_1,s_3)} A.$$

Lemma 3 (Distributive property)**.** *Let s_1, s_2, and s_3 be index sets with $s_3 \subseteq s_1 \cup s_2$. It holds that*

$$A *_{(s_1,s_2,s_3)} B + A *_{(s_1,s_2,s_3)} C = A *_{(s_1,s_2,s_3)} (B + C).$$

3 Tensor Calculus

Now we are prepared to develop our tensor calculus. We start by giving the definition of the derivative of a tensor-valued expression with respect to a tensor. For the definition, we use $\|A\| = \sqrt{\sum_s A[s]^2}$ as the norm of a tensor A.

Definition 4 (Fréchet Derivative)**.** *Let $f : \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_k} \to \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l}$ be a function that takes an order-k tensor as input and maps it to an order-l tensor as output. Then, $D \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l \times n_1 \times n_2 \times \ldots \times n_k}$ is called the derivative of f at x if and only if*

$$\lim_{h \to 0} \frac{\|f(x+h) - f(x) - D \circ h\|}{\|h\|} = 0,$$

where $\circ$ is an inner tensor product.

Here, the dot product notation $D \circ h$ is short for the inner product $D *_{(s_1s_2,s_2,s_1)} h$, where s_1s_2 is the index set of D and s_2 is the index set of h. For instance, if $D \in \mathbb{R}^{m_1 \times n_1 \times n_2}$ and $h \in \mathbb{R}^{n_1 \times n_2}$, then $s_1 = \{i, j, k\}$ and $s_2 = \{j, k\}$.

In the following, we first describe forward and reverse mode automatic differentiation for expressions in Einstein notation, before we discuss extensions like cross-country mode and compression of higher order derivatives that are much easier to realize in Einstein than in Ricci notation. As can be seen from our experiments in Section 4, the extensions allow for significant performance gains.

Forward Mode

Any tensor expression has an associated directed acyclic expression graph (expression DAG). Figure 2 shows the expression DAG for the expression

$$X^\top (\exp(X \cdot w) + 1)^{-1} \odot \exp(X \cdot w)), \qquad (1)$$

where $\odot$ denotes the element-wise multiplication and $^{-1}$ the element-wise multiplicative inverse. The nodes of the DAG

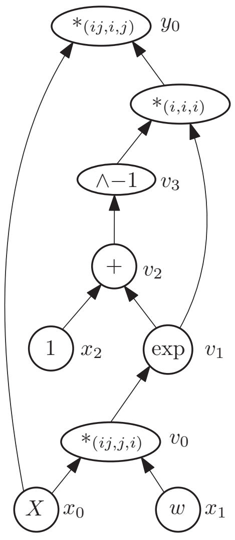

Figure 2: Expression DAG for Expression (1)

that have no incoming edges represent the variables of the expression and are referred to as input nodes. The nodes of the DAG that have no outgoing edges represent the functions that the DAG computes and are referred to as output nodes. Let the DAG have n input nodes (variables) and m output nodes (functions). We label the input nodes as $x_0, ..., x_{n-1}$, the output nodes as $y_0, ..., y_{m-1}$, and the internal nodes as $v_0, \ldots, v_{k-1}$. Every internal and every output node represents either a unary or a binary operator. The arguments of these operators are supplied by the incoming edges.

In forward mode, for computing derivatives with respect to the input variable x_j, each node v_i will eventually store the derivative $\frac{\partial v_i}{\partial x_j}$ which is traditionally denoted as $\dot{v}_i$. It is computed from input to out- put nodes as follows: At the input nodes that represent the variables x_i, the derivatives $\frac{\partial x_i}{\partial x_j}$ are stored. Then, the derivatives that are stored at the remaining nodes, here called f, are iteratively computed by summing over all their incoming edges as

$$\dot{f} = \frac{\partial f}{\partial x_j} = \sum_{z\,:\,(z,f)\in E} \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x_j} = \sum_{z\,:\,(z,f)\in E} \frac{\partial f}{\partial z} \cdot \dot{z},$$

where $\frac{\partial f}{\partial z}$ is the partial derivative of node f with respect to z and the multiplication is tensorial. The so called pushforwards $\dot{z}$ of the predecessor nodes z of f have been computed before and are stored at z. Hence, the derivative of each function is stored at the corresponding output node y of the expression DAG. Obviously, the updates can be done simultaneously for one input variable x_j and all output nodes y_i. Computing the derivatives with respect to all input variables requires n such rounds.

In the following we derive the explicit form of the pushforward for nodes of the expression DAG of a tensor expression. For such a DAG we can distinguish four types of nodes, namely multiplication nodes, general unary function nodes, elementwise unary function nodes, and addition nodes. General unary functions are general tensor-valued functions while elementwise unary functions are applied to each entry of a single tensor. The difference can be best explained by the difference between the matrix exponential function (general unary function) and the ordinary exponential function applied to every entry of the matrix (elementwise unary function). The pushforward for addition nodes is trivially just the sum of the pushforward of the two summands. Thus, it only remains to show how to compute the pushforward for multiplication, general unary functions, and element-wise unary function nodes. Please refer to the supplemental material for the proofs of the following three theorems, or see the similar proof of Theorem 8 for the reverse mode in the next section.

Theorem 5. *Let x be an input variable with index set s_4 and let $C = A *_{(s_1,s_2,s_3)} B$ be a multiplication node of the expression DAG. The pushforward of C is*

$$\dot{C} = B *_{(s_2, s_1 s_4, s_3 s_4)} \dot{A} + A *_{(s_1, s_2 s_4, s_3 s_4)} \dot{B}.$$

Theorem 6. *Let x be an input variable with index set s_3, let f be a general unary function whose domain has index set s_1 and whose range has index set s_2, let A be a node in the expression DAG, and let $C = f(A)$. The pushforward of the node C is $\dot{C} = f'(A) *_{(s_2 s_1, s_1 s_3, s_2 s_3)} \dot{A}$, where f' is the derivative of f.*

In case that the general unary function is simply an elementwise unary function that is applied element-wise to a tensor, Theorem 6 simplifies as follows.

Theorem 7. *Let x be an input variable with index set s_2, let f be an elementwise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f is applied element-wise. The pushforward of the node C is $\dot{C} = f'(A) *_{(s_1, s_1 s_2, s_1 s_2)} \dot{A}$, where f' is the derivative of f.*

Reverse Mode

Reverse mode automatic differentiation proceeds similarly to the forward mode, but from output to input nodes. Each node v_i will eventually store the derivative $\frac{\partial y_j}{\partial v_i}$ which is usually denoted as $\bar{v}_i$, where y_j is the function to be differentiated. These derivatives are computed as follows: First, the derivatives $\frac{\partial y_j}{\partial y_i}$ are stored at the output nodes of the DAG. Then, the derivatives that are stored at the remaining nodes, here called z, are iteratively computed by summing over all their outgoing edges as follows

$$\bar{z} = \frac{\partial y_j}{\partial z} = \sum_{f\,:\,(z,f)\in E} \frac{\partial y_j}{\partial f} \cdot \frac{\partial f}{\partial z} = \sum_{f\,:\,(z,f)\in E} \bar{f} \cdot \frac{\partial f}{\partial z},$$

where the multiplication is again tensorial. The so-called pullbacks $\bar{f}$ have been computed before and are stored at the successor nodes f of z. This means the derivatives of the function y_j with respect to all the variables x_i are stored at the corresponding input nodes of the expression DAG. Computing the derivatives for all the output functions requires m such rounds.

In the following we describe the contribution of unary and binary operater nodes to the pullback of their arguments. Here we have only two types of binary operators, namely tensor addition and tensor multiplication. In the addition case the contribution of C to the pullback of both of its arguments is simply $\bar{C}$. In Theorem 8 we derive the explicit form of the contribution of a multiplication node to the pullback of its arguments, in Theorem 9 the contribution of a general unary function, and in Theorem 10 we derive the contribution of an elementwise unary function node to its argument.

Theorem 8. *Let Y be an output node with index set s_4 and let $C = A *_{(s_1,s_2,s_3)} B$ be a multiplication node of the expression DAG. Then the contribution of C to the pullback $\bar{B}$ of B is $\bar{C} *_{(s_4 s_3, s_1, s_4 s_2)} A$ and its contribution to the pullback $\bar{A}$ of A is $\bar{C} *_{(s_4 s_3, s_2, s_4 s_1)} B$.*

Proof. Here we only derive the contribution of C to the pullback $\bar{B}$. Its contribution to $\bar{A}$ can be computed analogously. The contribution of C to $\bar{B}$ is $\bar{C} \cdot \frac{\partial C}{\partial B}$. By Definition 4 we have for the derivative $\bar{C} = \frac{\partial Y}{\partial C}$ of Y with respect to C that

$$\lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \cdot \left\| Y(C + \tilde{h}) - Y(C) - \bar{C} \circ \tilde{h} \right\| = 0.$$

By specializing $\tilde{h} = A *_{(s_1,s_2,s_3)} h$ we get

$$\begin{aligned}
&Y(C + \tilde{h}) - Y(C) - \bar{C} \circ \tilde{h} \\
&= Y(A *_{(s_1,s_2,s_3)} B + A *_{(s_1,s_2,s_3)} h) \\
&\quad - Y(A *_{(s_1,s_2,s_3)} B) - \bar{C} \circ (A *_{(s_1,s_2,s_3)} h) \\
&= Y(A *_{(s_1,s_2,s_3)} (B + h)) - Y(A *_{(s_1,s_2,s_3)} B) \\
&\quad - \bar{C} \circ (A *_{(s_1,s_2,s_3)} h) \\
&= Y(A *_{(s_1,s_2,s_3)} (B + h)) - Y(A *_{(s_1,s_2,s_3)} B) \\
&\quad - \bar{C} *_{(s_4 s_3, s_3, s_4)} (A *_{(s_1,s_2,s_3)} h) \\
&= Y(A *_{(s_1,s_2,s_3)} (B + h)) - Y(A *_{(s_1,s_2,s_3)} B) \\
&\quad - (\bar{C} *_{(s_4 s_3, s_1, s_4 s_2)} A) *_{(s_4 s_2, s_2, s_4)} h) \\
&= Y(A *_{(s_1,s_2,s_3)} (B + h)) - Y(A *_{(s_1,s_2,s_3)} B) \\
&\quad - (\bar{C} *_{(s_4 s_3, s_1, s_4 s_2)} A) \circ h),
\end{aligned}$$

where the first equality follows from the definitions of C and $\tilde{h}$, the second from Lemma 3, the third from the definition of $\circ$, the fourth from Lemma 1, the fifth from Lemma 2, and the last again from the definition of $\circ$. Hence, we have for $\frac{\partial Y}{\partial C} \cdot \frac{\partial C}{\partial B}$ that

$$\begin{aligned} 0 &= \lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \cdot \left\| Y(C+\tilde{h}) - Y(C) - \bar{C} \circ \tilde{h} \right\| \\ &= \lim_{h\to 0} \frac{1}{\|h\|} \cdot \Big\| Y(A *_{(s_1,s_2,s_3)} (B+h)) \\ &\qquad - Y(A *_{(s_1,s_2,s_3)} B) - (\bar{C} *_{(s_4s_3,s_1,s_4s_2)} A) \circ h) \Big\| \end{aligned}$$

Thus, the contribution of C to the pullback $\bar{B}$ is

$$\frac{\partial Y}{\partial C} \cdot \frac{\partial C}{\partial B} = \bar{C} \cdot \frac{\partial C}{\partial B} = \bar{C} *_{(s_4s_3,s_1,s_4s_2)} A. \qquad \square$$

If the output function Y in Theorem 8 is scalar-valued, then we have $s_4 = \emptyset$ and the pullback function coincides with the function implemented in all modern deep learning frameworks including TensorFlow and PyTorch. Hence, our approach can be seen as a direct generalization of the scalar case.

Theorem 9. *Let Y be an output function with index set s_3, let f be a general unary function whose domain has index set s_1 and whose range has index set s_2, let A be a node in the expression DAG, and let $C = f(A)$. The contribution of the node C to the pullback $\bar{A}$ is*

$$\bar{f} *_{(s_3s_2,s_2s_1,s_3s_1)} f'(A),$$

where f' is the derivative of f.

In case that the general unary function is simply an elementwise unary function that is applied element-wise to a tensor, Theorem 9 simplifies as follows.

Theorem 10. *Let Y be an output function with index set s_2, let f be an elementwise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f where f is applied element-wise. The contribution of the node C to the pullback $\bar{A}$ is*

$$\bar{f} *_{(s_2s_1,s_1,s_2s_1)} f'(A),$$

where f' is the derivative of f.

The proofs of Theorem 9 and 10 are similar to the proofs of Theorems 6 and 8. They can be found in the supplemental material.

Beyond Forward and Reverse Mode

Since the derivative of a function y with respect to an input variable x is the sum over all partial derivatives along all paths from x to y (Griewank and Walther 2008), we can combine forward and reverse mode. Using that $\bar{v} = \frac{\partial y}{\partial v}$ and $\dot{v} = \frac{\partial v}{\partial x}$, we get

$$\frac{\partial y}{\partial x} = \sum_{v\in S} \bar{v} *_{(s_1s_v,s_vs_2,s_1s_2)} \dot{v},$$

where s_v is the index set of node v, s_1 is the index set of the output function y, s_2 is the index set of the input node x, and S is the set of nodes in a cut of the expression DAG. General combinations of forward and reverse mode lead to the so-called cross-country mode. We will show that the differentiation of tensor expressions becomes even more efficient by a special instantiation of the cross-country mode and by compressing higher order derivatives.

Cross-Country Mode. In both forward and reverse mode, derivatives are computed as sums of products of partial derivatives. In general, the time for evaluating the derivatives depends on the order by which the partial derivatives are multiplied. The two modes multiply the partial derivatives in opposite order. Derivatives are multiplied from input to output nodes in forward mode and the other way around in reverse mode.

If the output function is scalar-valued, then reverse mode is efficient for computing the derivative with respect to all input variables. It is guaranteed that evaluating the derivative takes at most six times the time for evaluating the function itself. In practice, usually a factor of two is observed (Griewank and Walther 2008). However, this is no longer true for non-scalar-valued functions. In the latter case, the order of multiplying the partial derivatives has a strong impact on the evaluation time, even for simple functions (Naumann 2004). Reordering the multiplication order of the partial derivatives is known as cross-country mode in the automatic differentiation literature (Bischof, Hovland, and Norris 2002). Finding an optimal ordering is NP-hard (Naumann 2008) in general.

However, it turns out that significant performance gains for derivatives of tensor expressions can be obtained by the re-ordering strategy that multiplies tensors in order of their tensor-order, that is, multiplying vectors first, then matrices, and so on. We illustrate this strategy on the following example

$$f(x) = B \cdot g(h(Ax)), \tag{2}$$

where A and B are two matrices, x is a vector and $g(.)$ and $h(.)$ are vector-valued functions that also take a vector as input. The derivative in this case is $B \operatorname{diag}(u) \operatorname{diag}(v) A$, where $u = g'(h(Ax))$, $v = h'(Ax)$, and $\operatorname{diag}(u)$ is the diagonal matrix with u on its diagonal. Reverse mode multiplies these matrices from left to right while forward mode multiplies them from right to left. However, it is more efficient to first multiply the two vectors u and v element-wise and then to multiply the result with the matrices A and B.

Actually, the structure of Example 2 is not contrived, but fairly common in second order derivatives. For instance, consider the expression $\sum g(h(Ax))$, where g and h are as above and the sum is over the vector components of the vector-valued expression $g(h(Ax))$. Many machine learning problems feature such an expression as subexpression, where A is a data matrix and the optimization variable x is a parameter vector. The gradient of this expression has the form of Example 2 with $B = A^\top$. As can be seen in the experiments in Section 4, reordering the multiplications by our strategy reduces the time for evaluating the Hessian by about 30%.

Compressing Derivatives. Our compression scheme builds on the re-ordering scheme (cross-country mode) from above and on the simple observation that in forward as well as in reverse mode the first partial derivative is always a unit tensor. It is either, in reverse mode, the derivative of the output nodes with respect to themselves or, in forward mode, the derivative of the input nodes with respect to themselves. This unit tensor can always be moved to the end of the multiplications, if the order of multiplication is chosen exactly as in our cross-country mode strategy that orders the tensors in increasing tensor-order. Then, the multiplication with the unit tensor at the end is either trivial, i.e., amounts to a multiplication with a unit matrix that has no effect and thus can be removed, or leads to a compactification of the derivative.

For an example, consider the loss function

$$f(U) = \left\|T - UV^\top\right\|^2$$

of the non-regularized matrix factorization problem. Here, $T \in \mathbb{R}^{n\times n}$, $U, V \in \mathbb{R}^{n\times k}$ and n is usually large while k is small. The Hessian of f is the fourth order tensor

$$H = 2(V *_{(ij,ik,jk)} V) *_{(jl,ik,ijkl)} \mathbb{I} \in \mathbb{R}^{n\times k\times n\times k},$$

where $\mathbb{I}$ is the identity matrix. Newton-type algorithms for this problem solve the Newton system which takes time in $O\left((nk)^3\right)$. However, the Hessian can be compressed to $2(V *_{(ij,ik,jk)} V)$ which is a small matrix of size $k \times k$. This matrix can be inverted in $O(k^3)$ time. The performance gain realized by compression can be significant. For instance, solving the compressed Newton system needs only about 10 μsec whereas solving the original system needs about 1 sec for a problem of size $n = 1000$ and $k = 10$. For more experimental results refer to Section 4.

As another example, consider a simple neural net with a fixed number of fully connected layers, ReLU activation functions, and a softmax cross-entropy output layer. The Hessian of each layer is a fourth order tensor that can be written as $A *_{(ijl,ik,ijkl)} \mathbb{I}$ for a suitable third order tensor A. In this case, the Hessian can be compressed from a fourth order tensor to a third order tensor. We provide expression trees for both derivatives, compressed and uncompressed, in the supplemental material. Computing with the compact representation of the Hessian is of course more efficient which we confirm experimentally in the next section.

4 Experiments

We have implemented both modes of the tensor calculus from the previous section together with the improvements that can be achieved by cross-country mode and the compactification of higher order derivatives. State-of-the-art frameworks like TensorFlow and PyTorch only support reverse mode since it allows to compute derivatives with respect to all input variables at the same time. Similarly to all other frameworks, our implementation performs some expression simplification like constant folding and removal of zero and identity tensors. An anonymized version of our implementation can be found in the supplemental material.

Experimental Setup. We followed the experimental setup of (Laue, Mitterreiter, and Giesen 2018). However, we added one more experiment, a small neural net. We have implemented our algorithms in Python. To evaluate expressions we used NumPy 1.16 and CuPy 5.1. We compared our framework with the state-of-the-art automatic differentiation frameworks TensorFlow 1.14, PyTorch 1.0, autograd 1.2, and JAX 0.1.27 used with Python 3.6 that were all linked against the Intel MKL. All these frameworks support reverse mode automatic differentiation for computing first order derivatives. For scalar-valued functions the reverse mode of each of these frameworks coincides with the reverse mode of our approach. For non-scalar-valued functions all the frameworks compute the derivative for each entry of the output function separately. The expression DAGs that are generated by our reverse mode for general tensor expressions coincide with the derivatives computed by the approach of (Laue, Mitterreiter, and Giesen 2018). The experiments were run in a pure CPU setting (Intel Xeon E5-2643, 8 cores) as well as in a pure GPU setting (NVIDIA Tesla V100), except for autograd that does not provide GPU support.

We computed function values, gradients, and Hessians for each set of experiments. We computed Hessians on the CPU as well as on the GPU. To avoid the issue of sparsity patterns we generated dense, random data for each experiment. In this setting the running time does not depend on whether the data are synthetic or real world.

Logistic Regression. Let $X \in \mathbb{R}^{m\times n}$ be a data matrix and $y \in \{-1, +1\}^m$ the corresponding binary label vector. The logistic regression function is $f(w) = \sum_i \log\left(\exp\left(-y^{(i)}\left(X^{(i)}w\right)\right) + 1\right)$, where $w \in \mathbb{R}^n$ is the weight vector, $X^{(i)}$ is the i-th data point (i-th row of X), and $y^{(i)}$ its label. We set $m = 2n$ in the experiments.

Matrix Factorization. Let $T = \mathbb{R}^{m\times m}$ be some target matrix, $\Omega \in \{0,1\}^{m\times m}$ be an indicator matrix that defines which elements of T are known, and let $U, V \in \mathbb{R}^{m\times k}$ be low-rank factor matrices. Matrix factorization is the problem of solving $\min_{U,V} \|T - UV^\top\|^2_\Omega$. For the experiments, we set $k = 5$ and compute the gradient and Hessian with respect to U.

Neural Net. We have created a small neural net with ten fully connected layers, ReLU activation functions, and a softmax cross-entropy output layer. The weight matrices for the different layers all had the same size, $n \times n$. Here, we report running times for computing the Hessian of the first layer.

Evaluation. In the case of scalar-valued functions all the frameworks basically work in the same way. Thus, it is not surprising that their running times for computing function values and gradients are almost the same, see Figure 1 in the supplemental material.

The situation is different for Hessians. First, it can be seen that the reverse mode in our approach, whose results agree

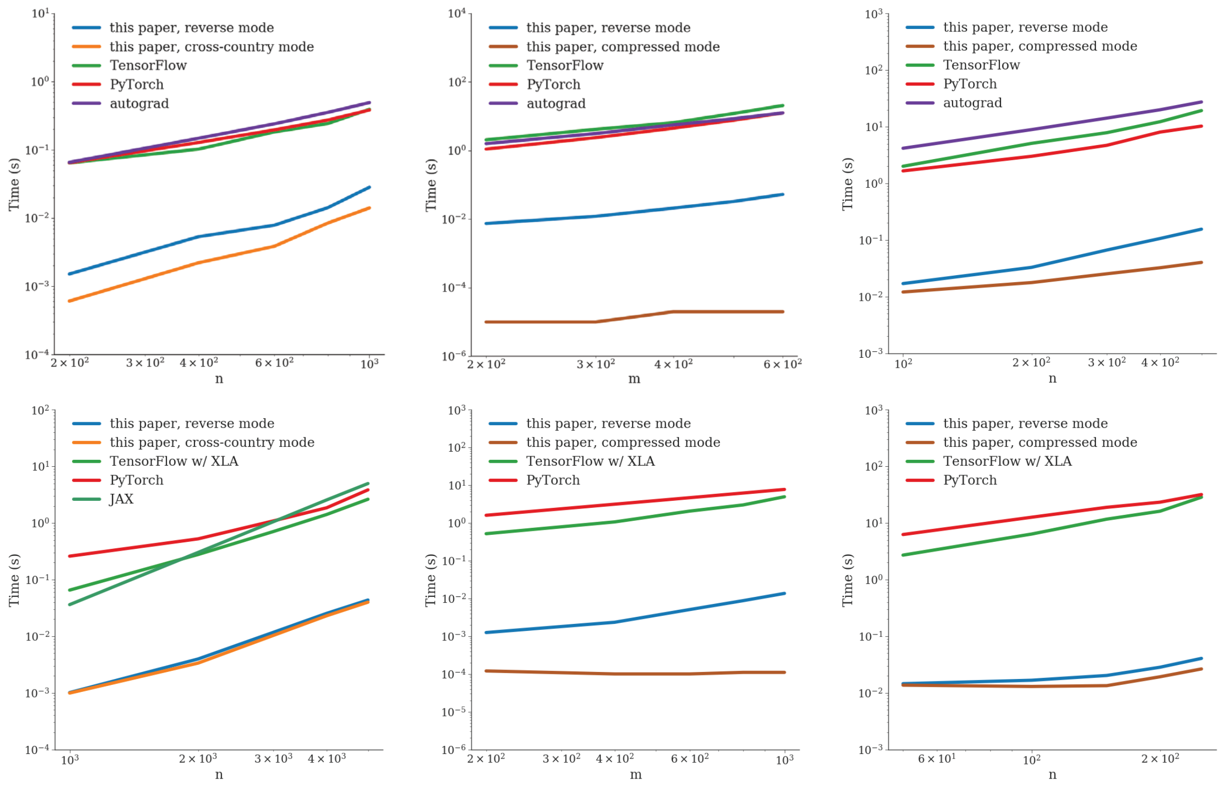

Figure 3: Running times on a CPU (top row) and on a GPU (bottom row) for computing the Hessian of the logistic regression function (left), matrix factorization (middle), and a small neural net (right).

with the results in (Laue, Mitterreiter, and Giesen 2018), is a few orders of magnitude faster than current state-of-the-art frameworks like TensorFlow, PyTorch, autograd, and JAX. This holds true for all experiments on the CPU and on the GPU, see Figure 3. For the logistic regression problem our cross-country mode is about 30% faster on the CPU, while its effect on the GPU is negligible because of the GPU overhead. The performance gain of our compression scheme can be seen on the matrix factorization problem and on the neural net (middle and right column of Figure 3). Computing Hessians for small neural nets has now become feasible.

In our experiments, the effect of recent efforts in speeding up deep learning frameworks turned out to be rather small. Actually, enabling XLA for TensorFlow on the CPU slowed the computation down by a factor of two. Hence, we omit these running times in Figure 3. Enabling XLA on the GPU provided only marginal improvements. JAX which relies on XLA did not finish computations but raised memory errors indicating that it went out of main memory. These errors seem to be caused by the automatic batching function `vmap` that is used by JAX for auto-vectorization when computing Hessians. This was surprising to us since JAX is meant to be more memory efficient than TensorFlow. There is one exception, on the GPU, JAX finished the computation for the logistic regression problem. However, as can be seen in Figure 3, even in this case it is not significantly more efficient than the other deep learning frameworks. This was to be expected since JAX relies on XLA and the authors of XLA report a speed-up of only 15% in the GPU setting (XLA 2019).

5 Conclusion

We have developed a simple, efficient and provably correct framework for computing derivatives of general tensor expressions that is much simpler than previous approaches. Furthermore, it can be easily integrated into state-of-the-art frameworks like TensorFlow and PyTorch that use the same tensor representation, but are a few orders of magnitude slower than our approach when computing higher order derivatives. We have also demonstrated that reverse mode automatic differentiation is not optimal for computing higher order derivatives. Significant speed ups can be achieved by a special instantiation of the cross-country mode and by compressing higher order derivatives. The algorithms presented here also form the basis of an online tool for computing derivatives of matrix and tensor expressions which is accessed by more than 30,000 users per year.

Acknowledgments

Sören Laue has been funded by Deutsche Forschungsgemeinschaft (DFG) under grant LA 2971/1-1.

References

Abadi, M., et al. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 265–283. USENIX Association.

Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.; and Siskind, J. M. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18(153):1–43.

Bischof, C. H.; Hovland, P. D.; and Norris, B. 2002. Implementation of automatic differentiation tools. In *ACM SIGPLAN Notices*, volume 37, 98–107. ACM.

Dey, P.; Nag, K.; Pal, T.; and Pal, N. R. 2018. Regularizing multilayer perceptron for robustness. *IEEE Trans. Systems, Man, and Cybernetics: Systems* 48(8):1255–1266.

Frostig, R.; Johnson, M. J.; and Leary, C. 2018. Compiling machine learning programs via high-level tracing. In *Conference on Systems and Machine Learning (SysML)*.

Gebremedhin, A. H.; Tarafdar, A.; Pothen, A.; and Walther, A. 2009. Efficient computation of sparse hessians using coloring and automatic differentiation. *INFORMS Journal on Computing* 21(2):209–223.

Ghorbani, B.; Krishnan, S.; and Xiao, Y. 2019. An investigation into neural net optimization via hessian eigenvalue density. In *International Conference on Machine Learning (ICML)*.

Giles, M. B. 2008. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation*, 35–44. Springer Berlin Heidelberg.

Griewank, A., and Walther, A. 2008. *Evaluating derivatives - principles and techniques of algorithmic differentiation (2. ed.)*. SIAM.

Kakade, S. M., and Lee, J. D. 2018. Provably correct automatic sub-differentiation for qualified programs. In *Advances in Neural Information Processing Systems (NeurIPS)*. 7125–7135.

Laue, S.; Mitterreiter, M.; and Giesen, J. 2018. Computing higher order derivatives of matrix and tensor expressions. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Laue, S.; Mitterreiter, M.; and Giesen, J. 2019. GENO – GENeric Optimization for Classical Machine Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

LeCun, Y. Deep Learning est mort. Vive Differentiable Programming! https://www.facebook.com/yann.lecun/posts/10155003011462143.

Maclaurin, D.; Duvenaud, D.; and Adams, R. P. 2015. Autograd: Effortless gradients in numpy. In *ICML AutoML workshop*.

Magnus, J. R., and Neudecker, H. 2007. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. John Wiley and Sons, third edition.

Naumann, U. 2004. Optimal accumulation of jacobian matrices by elimination methods on the dual computational graph. *Math. Program.* 99(3):399–421.

Naumann, U. 2008. Optimal jacobian accumulation is np-complete. *Math. Program.* 112(2):427–441.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in pytorch. In *NIPS Autodiff workshop*.

Pearlmutter, B. A. 1994. Fast exact multiplication by the hessian. *Neural Computation* 6(1):147–160.

Sagun, L.; Evci, U.; Güney, V. U.; Dauphin, Y.; and Bottou, L. 2018. Empirical analysis of the hessian of over-parametrized neural networks. In *ICLR (Workshop)*.

Seeger, M. W.; Hetzel, A.; Dai, Z.; and Lawrence, N. D. 2017. Auto-differentiating linear algebra. In *NIPS Autodiff workshop*.

van Merriënboer, B.; Breuleux, O.; Bergeron, A.; and Lamblin, P. 2018. Automatic differentiation in ml: Where we are and where we should be going. In *Advances in Neural Information Processing Systems (NeurIPS)*.

van Merriënboer, B.; Moldovan, D.; and Wiltschko, A. 2018. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Vasilache, N.; Zinenko, O.; Theodoridis, T.; Goyal, P.; DeVito, Z.; Moses, W. S.; Verdoolaege, S.; Adams, A.; and Cohen, A. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*.

Wang, F.; Decker, J.; Wu, X.; Essertel, G.; and Rompf, T. 2018. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *Advances in Neural Information Processing Systems (NeurIPS)*.

XLA and TensorFlow teams. 2017. XLA — TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

2019. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

Yao, Z.; Gholami, A.; Keutzer, K.; and Mahoney, M. W. 2018. Hessian-based analysis of large batch training and robustness to adversaries. In *Advances in Neural Information Processing Systems (NeurIPS)*.