Designing and implementing a tensor calculus

Farin Lippmann

August 2022

1 Abstract

2 Table of contents

3 Introduction

With the rise of machine learning applications, interest in optimization problems has been growing. In these problems, one seeks to find values for a number of parameters that maximize or minimize a given scalar objective function. While some optimization problems have a closed-form solution which can be computed directly, many can only be solved approximately. When computing these approximate solutions, derivatives play a crucial role. In particular, the first derivatives of the objective function are used in the ubiquitous gradient descent procedure. In every even slightly advanced optimization problem, multiple parameters need to be optimized simultaneously. This means that derivatives need to be computed with respect to multiple variables, organized either in vectors, matrices or higher-order tensors. Commonly used machine learning frameworks (like Tensorflow, PyTorch, Theano) focus heavily on the first derivatives of the objective function.

Higher-order derivatives can, however, also be of interest. As an example, the matrix of second-order partial derivatives, called the Hessian, is used in the application of Newton's method, which can be more efficient than gradient descent for some problems. Convexity checks constitute another application of higher-order derivatives. The convexity of objective functions and their associated optimization problems play a major role in the field of optimization. If a problem is strictly convex or concave, then it has a unique global optimum and the gradient descent algorithm can approximate it arbitrarily well. To check for convexity, the objective function's Hessian matrix, the matrix of second-order partial derivatives, needs to be computed and checked for semidefiniteness. If the function's inputs are naturally organized in a matrix or higher-order tensor, then the resulting Hessian will also be a higher-order tensor. The prevalent machine learning frameworks have no way to directly compute these derivatives, and classical computer algebra systems struggle with efficiency, as they work on the level of individual tensor entries.

To bridge this gap, we design and implement a tensor calculus that allows for automatic symbolic differentiation of tensor expressions of any order. Additionally, this calculus will use Einstein summation notation to represent tensor products, rather than the more complex notation used in Ricci calculus.

This thesis builds on the work by Laue, Mitterreiter and Giesen, who have built a similar calculus for matrix derivatives [Matrix Calculus] and developed the theoretical foundation for an Einstein notation based tensor calculus [Tensor Paper].

The structure of the thesis is as follows. First, the syntax of the calculus is developed in form of a grammar. Then, the parsing and representation of tensor expressions is discussed. ...

4 A Syntax for Tensor Expressions

4.1 Aims

In this chapter we develop the language that will be used to specify tensor expressions for differentiation. To differentiate a tensor expression, three pieces of information are needed: 1. The tensor expression itself, 2. The variable with respect to which should be differentiated, 3. The tensor rank of all variables in the expression. The neccessity of the first two is obvious, while the third might not immediately be. The ranks of each part of the tensor expression need to be known because they play a crucial role in the differentiation rules that will later be implemented.

Our aims when developing this syntax are the following. The syntax should allow representation of tensor expressions of arbitrary order. It should be as general as possible, not focusing on one single interpretation of tensors. Even so, as the motivation for this thesis comes from the background of optimization and machine learning, the most common specifics of these fields should be covered.

The main point of interest for this syntax is how to represent tensors and their products, as there are many ways to multiply two tensors. We start with an established tensor calculus and it's notation, and break it down to our needs.

4.2 Einstein Summation Convention

There exists a calculus for tensor expressions, called Ricci calculus, that is heavily used in physics, which allows differentiation between tensors as n-dimensional arrays and as multilinear functions. It differentiates between the two forms by the location of their indices, either in sub- or superscript. For example, the inner product of two vectors, $y^T x$, would be written as $y_i x^i$. And the Matrix-vector product Ax would be $A^i_j x^j$. While Ricci calculus can be useful, for example to physicists, the many indices on each tensor make it less readable.

Since, for our purposes, tensors are simply containers for variables, we have no need to expressly define some tensors as multilinear functions. The main part of Ricci calculus that we bring over into our notation is the Einstein summation convention. This is a notational convention that allows elegant representation of tensor products. Since tensors, depending on their order, may be multiplied in many ways, we need a way to describe which tensor axes should be multiplied in what way. For example, two vectors could be multiplied by taking their inner product, their outer product, or even their elementwise product. Einstein notation handles this problem by adding index sets to the inputs and the output of the product, each index representing an axis in the respective tensor. If an index appears in both inputs, the respective axes will be multiplied. If an index appears in inputs but not in the output, this axis will be summed over. This simple set of rules provides an elegant way to express the main tensor operations.

We take a large part of the syntax for tensor multiplication from the established Python software package *Numpy*, specifically it's `einsum` procedure. A product of two tensors `x` and `y` has the form `x*(a,b->c)y`, where a,b and c are strings of indices.

As an example, see the inner, outer and elementwise product of two order-1 tensors (vectors) `x` and `y`: `x*(i,i->)y`, `x*(i,j->ij)y`, `x*(i,i->i)y`. The application of a matrix `A` to a vector `v` would be written as `A*(ij,j->i)v`, while the elementwise product of two matrices `A` and `B` would be `A*(ij,ij->ij)B`. One may even take diagonals of tensors using this notation, the inner product of a square matrix `A`'s main diagonal and a vector `v` would be `A*(ii,i->)v`.

4.3 Grammar

With the representation of tensor products handled, we can now define the full syntax grammar. The grammar, presented here in extended backus-naur form, is based on a standard grammar for mathematical expressions by Julien Klaus. It was adapted to tensor expressions by generalizing multiplication to tensor products using the einstein summation convention.

```
expressionpart = 'expression' expr
expr = term (( '+' | '-' ) term)*
term = factor (( '*(' productindices ')') factor | '/' factor)*
productindices = tensorindices ',' tensorindices '->' tensorindices
tensorindices = (smallalpha)*
factor = {'-'} atom ('^' ('(' expr ')' | atom))*
atom = number | function '(' expr ')' | tensorname | '(' expr ')'
number = ['-'] digit* '.' digit* [( 'e' | 'E' ) ['+' | '-'] (digit)*]
digit = [0-9]
function = 'sin' | 'cos' | 'tan' | 'arcsin' | 'arccos' | 'arctan' |
    'abs' | 'tanh' | 'exp' | 'log' | 'sign' | 'relu' | 'det' | 'inv'
```

Here `{x}` describes that `x` may occur any number of times, while `(x)*` lets `x` appear at least once.

As can be seen in the grammar, we allow the following binary operations on tensors: products (`*(,->)`), sums (`+`), differences (`-`) and quotients (`/`). With the exception of tensor products, which have already been discussed, these operations are executed elementwise. Exponentiation (`^`) is also included, but not as a true binary operator on tensors, as the exponent may only be scalar, with each entry of the basis tensor being combined with the exponent. This is not apparent from the grammar, as tensor variable order is specified separately from the tensor expression itself.

A variety of functions with relevance to optimization and machine learning are included, mostly being applied elementwise to tensors of any order. The exceptions to this are the matrix inverse `inv` and the matrix determinant `det`, both of which are not applied elemtwise and may only operate on order-2 tensors. The inclusion of these functions takes away from the generality of our grammar, as the idea of an inverse and a determinant can only be applied to matrices, not higher-order tensors, but they are crucial to some common optimization problems.

5 Parsing and Representation

With the syntax handled, we may now discuss how a given tensor expression is parsed and represented internally. In our application, tensor expressions are represented using directed acyclic graphs derived from binary trees. This data structure was chosen because graphical representations are well tested for expressions and because the foundational paper by Laue, Mitterreiter and Giesen also assumes such a representation, allowing the differentiation rules to be easily implemented. Furthermore, many of the algorithms to be implemented later are most naturally written recursively. A recursive data structure like a tree lends itself to these algorithms very well.

6 Literature

7 Appendices

8 Declaration of autonomy