

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

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 necessity 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.

The main point of interest 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 its notation, and break it down to our needs.

4.1 Einstein Summation Notation

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 $A_j^i x^j$. 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 notation. 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 its `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) \mathbf{x} and \mathbf{y} : $\mathbf{x}*(i,j\rightarrow)\mathbf{y}$, $\mathbf{x}*(i,j\rightarrow ij)\mathbf{y}$, $\mathbf{x}*(i,i\rightarrow i)\mathbf{y}$. The application of a matrix \mathbf{A} to a vector \mathbf{v} would be written as $\mathbf{A}*(ij,j\rightarrow i)\mathbf{v}$, while the elementwise product of two matrices \mathbf{A} and \mathbf{B} would be $\mathbf{A}*(ij,ij\rightarrow ij)\mathbf{B}$. One may even take diagonals of tensors using this notation, the inner product of a square matrix \mathbf{A} 's main diagonal and a vector \mathbf{v} would be $\mathbf{A}*(ii,i\rightarrow)\mathbf{v}$.

4.2 Grammar

With the representation of tensor products handled, we can now define the full input 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. // TODO: // - Insert Grammar // - Why choose these functions // - Only elementwise operations besides multiplication // - Elaborate on the grammar base by Julien

4.3 Parsing the Expression

5 Literature

6 Appendices

7 Declaration of autonomy