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, for example in the case of convexity checks. 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 input syntax of the calculus is developed in form of a grammar. Then, the parsing and representation of tensor expressions is discussed. ...

4 Input Syntax

In this chapter we develop the language that will be used to input tensor expressions for differentiation. To differentiate a tensor expression, three kinds 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.

The main point of interest is how to represent tensors and their products. We start with an established tensor calculus and it's 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 captures tensors both as n-dimensional arrays and as their dual form, multilinear functions. It differentiates between the two forms by the location of their indices, either in sub- or superscript. While Ricci calculus is elegant in the way it handles the duality of tensors, the number of indices make it less readable.

Since, for our purposes, tensors are simply containers for variables, we have no need for this distinction. 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 may be multiplied in many ways depending on their order, we need a way to describe which tensor axes should be multiplied in what way. 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 multiple 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, two order-1 tensors (vectors) `x` and `y` could have their outer product, `x*(i,j->ij)y`, their inner product, `x*(i,j->)y` or their elementwise product, `x*(i,i->i)y`, taken. The application of a matrix `A` to a vector `v` would be written

as `A*(ij,j->ij)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`.

5 Literature

6 Appendices

7 Declaration of autonomy