Automatic Symbolic Differentiation for Tensor Expressions

Farin Lippmann

ai.uni-jena.de

SUMMER SCHOOL 2023

Motivation

AI relies on solving optimization problems.

An optimization problem is solved by finding values for parameters that minimize/maximize some objective function.

Parameters are usually bundled in vectors, matrices, or higher-order tensors.

Thus, computing derivatives of tensor expressions is of interest.

For my bachelor's thesis, I implemented a calculus for automatic symbolic differentiation of tensor expressions.

Language Design

We read tensor expressions made up of sums, tensor products (using einsum notation [1]), power terms, elementwise function applications ($\sin$, $\exp$, abs,...) and special function applications ($\det$, inv, adj), as well as constants (which are treated as broadcast to the appropriate number of axes).

Here are some example expressions:

Linear Algebra Notation	Our Notation
$x^T y$	$x *_{(i,i\to)} y$
Ax	$A *_{(ij,j\to i)} x$
$A \odot B$	$A *_{(ij,ij\to ij)} B$

With the notation for tensor products defined, we can showcase some example inputs into our differentiation program. Each input is made of three parts:

- the declaration of the tensor variables with the number of axes for each
- the expression
- the variable with respect to which the expression is to be differentiated

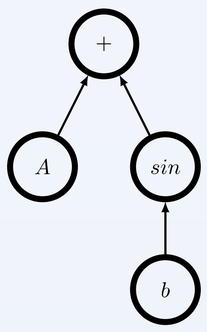
```
declare
 v 1
expression
 v *(i,i→i) v
derivative wrt v
```

```
declare
 A 2
 x 1
expression
 (A + 1) *(ij,j→i) x
derivative wrt A
```

```
declare
 A 3
 x 0
expression
 A *(ijk,→ijk) exp(x)
derivative wrt x
```

Parsing and Preprocessing

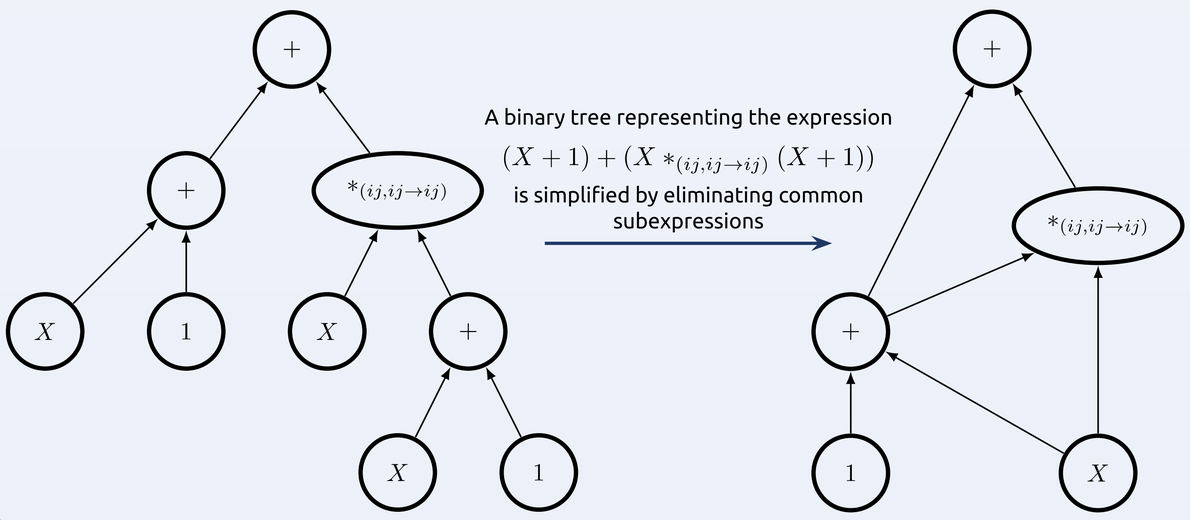
To differentiate expressions, we parse them into an internal representation that is easier to work with, an expression DAG (directed acyclic graph).



DAG for the expression

$$A + \sin(b)$$

We first parse a binary expression tree, then convert it into a DAG by eliminating common subexpressions, such that no subexpression exists twice.



Differentiation

Our algorithm computes derivatives of tensor expressions in reverse mode, by repeatedly applying the chain rule.

We traverse the DAG from top to bottom. The goal is to calculate for each node v the derivative of the whole expression y with respect to it, $\frac{dy}{dv}$. This value is called the pullback of v.

When traversing the DAG, the pullback is known for each parent node u of v. This allows us to apply the chain rule:

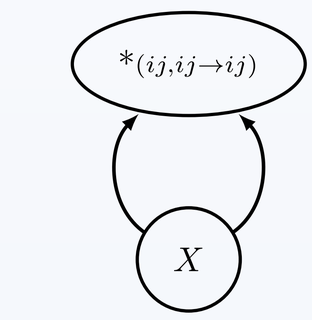
$$\frac{dy}{dv} = \sum_{u\colon u \in \mathrm{parents}(v)} \frac{dy}{du} \cdot \frac{du}{dv}$$

Thus, we need only know the local derivative of each node with respect to its children to compute all pullbacks.

These local derivatives are obtained by applying differentiation rules for each of the allowed types of nodes (sums, tensor products, powers, functions). The rules are taken from Laue, Mitterreiter, and Giesen [2].

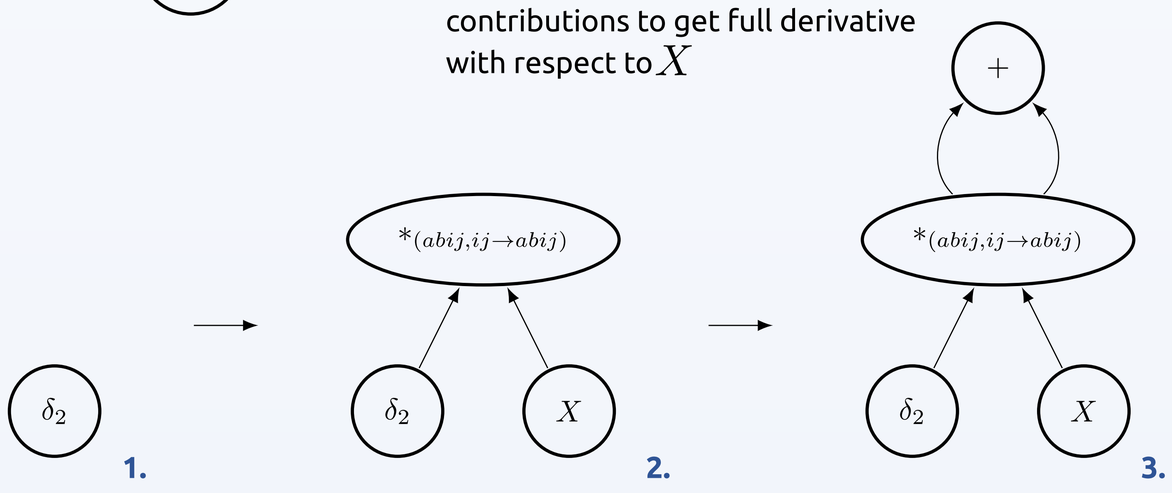
Because we eliminated common subexpressions, a node may have multiple parents. As seen in the above equation for chain rule application, multiple contributions to the same pullback need to be summed.

Here is an example of how our algorithm works:



Given the DAG representing $X *_{(ij,ij\to ij)} X$:

1. Differentiate the top node with respect to itself (returns a delta tensor, analogous to identity matrix)
2. Apply product rule to differentiate top node with respect to X (left child), concatenate with chain rule
3. Apply product rule to differentiate top node with respect to X (right child), sum both contributions to get full derivative with respect to X



Results

Our algorithm allows for automatic symbolic differentiation of tensor expressions of arbitrary order, so long as local differentiation rules have been defined for all of the operations in the expression.

We are also able to automatically produce *Numpy* [3] code for the generated derivative expressions.

This allowed us to verify the correctness of our implementation in dozens of test cases, where our symbolically calculated derivative was compared to a numerical approximation of the derivative of the expression at a random value.

It is also easy to prove that our algorithm produces an expression DAG whose number of nodes and edges scales linearly with:

- the number of nodes and edges ($O(|V| + |E|)$) in the original expression DAG (with common subexpressions eliminated)
- the number of nodes ($O(|V|)$) in the original unsimplified expression tree

Sources

[1] NumPy Developers. "numpy.einsum". In: https://numpy.org/doc/stable/reference/generated/numpy.einsum.html (2022)

[2] Sören Laue, Matthias Mitterreiter, and Joachim Giesen. "A Simple and Efficient Tensor Calculus". In: The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20) (2020), pp. 4527–4534.

[3] Charles R. Harris et al. "Array programming with NumPy". In: Nature 585.7825 (Sept. 2020), pp. 357–362.