FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Designing and Implementing a Tensor Calculus

B A C H E L O R ' S   T H E S I S

In Fulfillment of the Requirements for the Academic Degree of

Bachelor of Science (B.Sc.)

In the Study Program 'Computer Science'

FRIEDRICH SCHILLER UNIVERSITY JENA

Faculty of Mathematics and Computer Science

submitted by Farin Lippmann

born on the 02.02.2000 in Gera

Supervisor: Prof. Dr. Joachim Giesen

Jena, 25.08.2022

Zusammenfassung

Beim Finden von Lösungen für Optimierungsprobleme spielen Ableitungen eine zentrale Rolle. Das automatische symbolische Ableiten von Matrixausdrücken und das numerische Ableiten von Tensorausdrücken sind gut erforscht und werden in der Wirtschaft extensiv genutzt. Das Feld der automatischen symbolischen Ableitung von Tensorausdrücken ist hingegen weniger gut untersucht, obwohl es in Optimierungsalgorithmen und Konvexitätsprüfungen angewandt werden kann. In dieser Arbeit wird zum Schluss dieser Lücke beigetragen, indem ein Kalkül für die automatische symbolische Ableitung von Tensorausdrücken jeglicher Ordnung eingeführt und implementiert wird. Zur Darstellung von Tensorprodukten nutzt das Kalkül ein Derivat der Einsteinnotation. Dieses ist generischer, als die im populären Riccikalkül verwendete Notation. Tensorausdrücke werden im Kalkül durch gerichtete azyklische Graphen repräsentiert, was es ermöglicht, das ableitungsbedingte Wachstum der Ausdrücke als höchstens linear in der Knoten- und Kantenzahl des Graphen zu bestimmen.

Abstract

In optimization problems, derivatives play a crucial role in finding an optimal solution. Automatic symbolic differentiation of matrix expressions and numerical differentiation of tensor expressions are well researched and used extensively in the industry. The field of automatic symbolic differentiation of tensor expressions, however, is not explored as thoroughly, even though it has possible applications in optimization algorithms and convexity checks. In this thesis, we contribute to the closing of this gap by introducing and implementing a calculus for automatic symbolic differentiation of tensor expressions of any order. The calculus uses an Einstein-like product notation that is more general than the notation used in the popular Ricci calculus. Furthermore, its representation of tensor expressions as directed acyclic graphs allows us to bound the differentiation-induced growth of the expression to be at most linear in the number of edges and nodes in the graph.

Table of Contents

1 Introduction

With the rise of machine learning applications, interest in optimization problems has been growing. In these kinds of problems, one seeks to find values for a number of parameters that maximize or minimize a given 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 derivative of the objective function is used in the ubiquitous gradient descent procedure.

In most optimization problems, multiple parameters need to be optimized simultaneously. This requires derivatives 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 or 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 has better convergence time guarantees than gradient descent [5]. 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 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 symbolically, 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 uses an Einstein-like notation to represent tensor products, which is more general than the notation used in Ricci calculus. Using a directed acyclic graph representation for tensor expressions also allows us to bound the growth of the expression induced by differentiation to be at most linear in the number of edges and nodes in the graph.

This thesis builds on the work by Laue, Mitterreiter and Giesen, who have built a similar calculus exclusively for vector and matrix derivatives [4] and developed the theoretical foundation for an Einstein notation based tensor calculus [3].

The structure of the thesis is as follows. First, we develop the syntax of the calculus in form of a grammar. We then discuss the parsing and representation of tensor expressions. Following that, the theoretical background and practical implementation of the differentiation process are described. We also remedy a problem that was present in the original work [3] by Laue, Mitterreiter and Giesen and prove that our algorithm causes tensor expressions to grow at most linearly when differentiating.

A short note on notation: This thesis uses a `monospaced` font when describing strings of characters.

2 A Syntax for Tensor Expressions

2.1 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:

- the tensor expression itself
- the variable with respect to which should be differentiated
- the tensor order of all variables in the expression

The necessity of the third condition is not immediately obvious. During differentiation, the tensor orders of each subexpression need to be known in order to apply the differentiation rules. We compute these tensor orders for all subexpressions from the given tensor orders of variables. Further details are covered in Chapter 3.

Our aims when developing the syntax are the following. It should allow representation of tensor expressions of arbitrary order and should be as general as possible. 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 the notation of an established tensor calculus and generalize it to fit our problem domain.

2.2 Representing Tensor Products

There exists a calculus for tensor expressions, called Ricci calculus, that is heavily used in physics [6]. The notation it uses allows for tensors to represent both n-dimensional arrays and multilinear functions. The distinction between the two forms occurs 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. A part of Ricci calculus that we bring over into our notation is a derived version of the Einstein summation convention [1]. 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 their elementwise product. Einstein notation handles this problem by adding index strings 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 the inputs but not in the output, this axis will be summed over. In true Einstein-notation, output indices are omitted, so indices that appear in both inputs are always summed over. For more generality, we include output indices. 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* [2], specifically its *einsum* procedure. A product of two tensors x and y has the form $x *_{(a,b \to c)} y$, where a,b and c are strings of indices. A formal characterization of the product $C = A *_{(s_1,s_2 \to s_3)} B$ is given in Equation 2.1, where s_1, s_2 and s_3 represent index strings. In the original work [3] by Laue, Mitterreiter and Giesen, tensor products are characterized by index sets instead of index strings. This, however, restricts us in specifying the order of axes. We work with index strings, but may apply the results of their work as strings can be viewed as sets with a specified ordering. Note that for two index strings s_1 and s_2, we write $s_1 s_2$ as shorthand for $s_1 \cup s_2$.

$$C[s_3] = \sum_{(s_1 s_2) \setminus s_3} A[s_1] \cdot B[s_2] \tag{2.1}$$

As an example of tensor products, see the inner, outer and elementwise product of two first order tensors (vectors) x and y: $x *_{(i,i \to)} y$, $x *_{(i,j \to ij)} y$, $x *_{(i,i \to i)} y$. The application of a matrix A to a vector v would be written as $A *_{(ij,j \to i)} v$, while the elementwise product of two matrices A and B would be $A *_{(ij,ij \to ij)} B$. By swapping the order of output indices, the axis order can be manipulated. A special case of this is the matrix transpose, which for a matrix X can be taken using the product $X *_{(ij, \rightarrow ji)} 1$.

2.3 Grammar

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

```
input = declaration expressionpart argument

declaration = 'declare' {tensordeclaration}
tensordeclaration = tensorname {digit}+
tensorname = alpha { alpha | digit }
alpha = smallalpha | largealpha
smallalpha = [a-z]
largealpha = [A-Z]

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 ')' |
       delta
number = ['-'] {digit}+ '.' {digit} [( 'e' | 'E' ) ['+' | '-'] {digit}+]
digit = [0-9]
elementwise_function = 'sin' | 'cos' | 'tan' | 'arcsin' | 'arccos' | 'arctan' |
           'tanh' | 'exp' | 'log' | 'sign' | 'relu' | 'abs'
special_function = 'det' | 'inv' | 'adj'
delta = 'delta(' {digit}+ ')'

argument = 'derivative wrt' tensorname
```

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

Some examples of inputs this grammar accepts are the following; more examples can be found in the appendix.

- `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`

The input is structured around three keywords. After the first keyword, `declare`, come the tensor variables that appear in the expression with their respective orders. The expression to be differentiated follows after `expression`. The last keyword, `derivative wrt`, precedes the name of the argument variable with respect to which is being differentiated.

As can be seen in the grammar, we allow the following binary operations on tensors: products (`*(,->)`), sums (`+`), differences (`-`) and quotients (`/`). Except for 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.

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`), the matrix determinant (`det`) and the adjugate matrix (`adj`), all of which are not applied elementwise and may only operate on order 2 tensors. The inclusion of these functions takes away from the generality of our grammar, as they can only be applied to matrices, not higher-order tensors, but they are crucial to some common optimization problems that work with matrices.

We automatically broadcast constants, like the `1` in the second example above, to the appropriate tensor order for ease of use. How this is accomplished is covered in Chapter 3.

3 Parsing and Representation

Having defined the syntax, we 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 work [3] by Laue, Mitterreiter and Giesen also assumes such a representation. This allows the differentiation rules presented in their work to be easily implemented. Furthermore, many of the algorithms described in this chapter and the next are most naturally written recursively. A recursive data structure like a tree lends itself to these algorithms.

In this chapter we first describe how a tensor expression string is parsed into a binary expression tree and explain the processing steps taken to simplify the binary tree into a directed acyclic graph without duplicate subexpressions. We also describe how we assign a tensor order and an axis tuple to each node in the graph, which is required for differentiation and for numerical evaluation of the derivative expression.

3.1 Parsing

The process of turning a string into a different data structure that allows for further processing is called parsing. We parse an expression string in two separate steps, tokenization and tree building.

3.1.1 Tokenization

To be able to parse an expression string, we first transform it into a sequence of tokens. This is done by a piece of software called a tokenizer, sometimes also called lexical scanner, which recognizes certain defined patterns in the expression string and unifies them into a token. Each of these tokens represents a logical unit that is used in the next step of parsing. Tokenization allows the parser to work only with well-defined tokens, not having to work on the level of individual characters. Each token is made up of two parts, the descriptor and the identifier. The descriptor is one of a few defined categories, describing what kind of logical unit the token forms, while the identifier contains the characters the token was formed from. As an example, the string `A + sin(b)` would be tokenized into the following sequence: `(ALPHANUM, 'A'), (PLUS, '+'), (ELEMENTWISE_FUNCTION, 'sin'), (LRBRACKET, '('), (LOWERCASE_ALPHA, 'b'), (RRBRACKET, ')')`.

We define token descriptors for constants (*e.g.* `'1.05'`), variable names (*e.g.* `'A'`, `'b'`, `'C1'`), special symbols (`'+'`, `'-'`, `'*'`, `'/'`, `'^'`, `','`, `'('`, `')'`, `'>'`) and function names, which are recognized using the list of functions given in the grammar (*e.g.* `'sin'`, `'abs'`, `'det'`). There are two token descriptors for functions, `ELEMENTWISE_FUNCTION` and `SPECIAL_FUNCTION`, because derivative rules are different for these two types.

3.1.2 Tree Building

The output of the tokenizer is used to build a data structure, specifically a binary tree, representing the expression. A binary tree is a type of tree where each node has at most two child nodes. Like in other trees, each node has exactly one parent node. The exception to this is the root, which has no parent. We call nodes that have no children leaf nodes.

Formally, we define a binary tree B as follows: B is either the empty set or a tuple $(\text{left}(B), \text{root}(B), \text{right}(B))$, where $\text{left}(B)$ and $\text{right}(B)$ are also binary trees that have B as their only parent, and $\text{root}(B)$ is the root node, containing some value.

To unify the formal definition with our intuitive characterization, refer to empty nodes as nonexistent and omit them from graphical representations. This allows us to speak of nodes in terms of their number of children.

The process of constructing such a tree from a given expression follows the structure of the grammar defined in the previous chapter. Starting with `expr`, we recursively step through the grammar, using the tokens to decide which options to follow. For each of the words defined in the grammar there is a function in our parser. When a certain word is expected, the function is called. For example, at the start, `expr` is called, which first calls `term`, which calls `factor`, which then looks for any `MINUS`-tokens before calling `atom`, which then checks if there is a constant, a function application, a variable name, or another expression, and so forth. Thus, we traverse the expression from left to right while building binary tree nodes when appropriate.

Leaf nodes are created when we reach constants or variable names, and they are connected using branch nodes containing operators or functions. Each node, with its subtree, represents a subexpression, with the root node of the whole tree representing the entire expression. If there is ever a point at which expectations are not met, for example, if a left round bracket token (`(LRBRACKET, '(')`) is expected but not found, we abort program execution.

As an example, we take the expression from before, `A + sin(b)`, tokenized to `(ALPHANUM, 'A'), (PLUS, '+'), (ELEMENTWISE_FUNCTION, 'sin'), (LRBRACKET, '('), (LOWERCASE_ALPHA, 'b'), (RRBRACKET, ')')`.

The order of function calls generated when parsing this expression would be the following:

- `expr`
- `term`
- `factor`
- `atom` — We identify `'A'` as a variable name, generate a leaf node and jump back up the call stack to `expr`. There, we find the expected `'+'` and add a sum node as the parent of our leaf node.
- `term`
- `factor`
- `atom` — We identify `'sin'` as a function name and generate an elementwise function node as the second child of our sum node. We also find the expected `'('`.
- `expr`
- `term`
- `factor`
- `atom` — We identify `'b'` as a variable name and generate another leaf node as the child of our elementwise function node. We jump back up the call stack, find the expected `')'`, jump all the way to the top of the call stack and finish.

The binary tree generated from this can be seen in Figure 3.1. Note that, by convention, we set the argument of a unary function node to its right child.

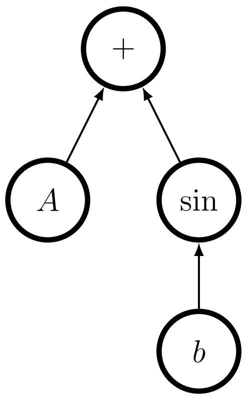

Figure 3.1: Binary expression tree generated from the expression string `A + sin(b)`. We omit the label describing which of a node's children is the left and right child when it is obvious from the figure or makes no difference in the expression the graph represents.

3.2 Preprocessing

Having shown how a binary expression tree is created from a given tensor expression, we now describe how this tree is processed further to prepare it for differentiation. We employ the following preprocessing steps.

- Some subtrees are substituted with equivalent ones which are more easily differentiated.
- Subtrees which occur more than once are fused.
- Each node's output tensor order is determined and stored.
- Each node is assigned a tuple of axis identifiers.

3.2.1 Equivalent Subexpression Substitution

Some operations or functions may be viewed as equivalent to, or as special cases of other, more easily differentiable operations or functions. In these cases, we substitute the more general case to reduce the number of differentiation rules that need to be implemented. For example, when encountering the expression $A - B$, we substitute $A + -(B)$, viewing $(-)$ as an elementwise function. This allows us to use the differentiation rules for sums and elementwise functions, not having to implement a new rule for differences. A full overview of the substituted expressions can be seen in Table 3.1.

Original	Substitute
$A - B$	$A + -(B)$
A^B	$\exp(B *_{(,s \to s)} \log(A))$
A/B	A .* elementwise_inverse(B)
$\mathrm{adj}(X)$	$\det(X) *_{(,ij \to ij)} \mathrm{inv}(X)$

Table 3.1: Table showing all operations that get substituted during preprocessing, along with their substitutes. The transformation of A^B is only applied when both A and B contain the differentiation argument, and is only valid when no entries of A are zero. s is the index string of A. (.*) is a placeholder for elementwise multiplication; it is replaced with the appropriate tensor product during tensor order determination. The transformation for adj(X) is only valid if X is invertible. elementwise_inverse is an elementwise function for which a differentiation rule is implemented.

3.2.2 Common Subtree Elimination

When differentiating, we need to be able to tell when a subexpression is reached multiple times. To help with this, and also to reduce the number of nodes in the expression tree, we fuse subtrees that occur more than once.

First, to identify duplicate subtrees, the notion of equality needs to be defined for trees. We define equality of two trees, A and B, as follows.

$$A = B \iff (\mathrm{root}(A) = \mathrm{root}(B)) \wedge (\mathrm{left}(A) = \mathrm{left}(B)) \wedge (\mathrm{right}(A) = \mathrm{right}(B))$$

Equality of root nodes is handled by comparing all components of the node (node type, name, indices for product nodes) individually.

Now that we may identify when two subtrees are equal, we continue with common subtree elimination. We first obtain all subtrees by traversing the whole tree. Then, we store each unique subtree we find in a hashmap. When we encounter a subtree that is already present in our hashmap, we change its parent node to have the tree in the hashmap as its child instead of the original. In doing so, we remove duplicate subtrees and transform the binary expression tree into a directed acyclic graph (DAG) representing the expression. Nodes in the DAG still only have two child nodes, but may have many parent nodes.

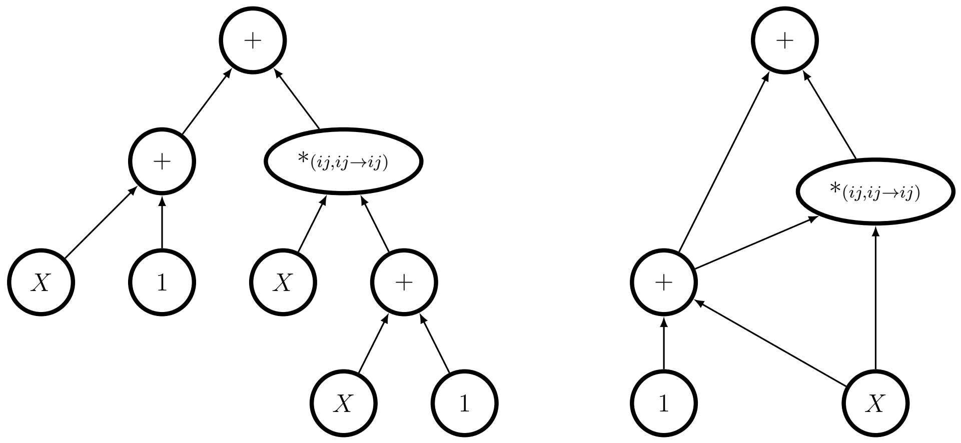

Figure 3.2: *Left*: Binary tree representing the expression $(X+1)+X*_{(ij,ij\to ij)}(X+1)$. *Right*: DAG obtained by eliminating common subtrees in the left tree.

3.2.3 Tensor Order Determination

During differentiation, we often create new product nodes. Because we need to specify product indices when we create a product node, we require knowledge of the tensor orders of both inputs and the output of the new node, which depend on the orders of nodes in the original DAG. Because we may need the tensor order of all nodes in the original DAG, we determine them by propagating orders from the bottom of the DAG toward the root.

The bottom nodes contain either variables or constants. The tensor orders of variables need to be given, as they function as the start for the upwards propagation. Constants are handled specially and are covered in the following paragraph. Starting from the bottom nodes in the graph, each parent node calculates its tensor order based on its child nodes. For example, a sum will have its order set equal to the order of its children. A full overview of these order determination rules can be found in Table 3.2.

Constants may need to be broadcast to fit into the expression. For example, in the expression $1 + \det(X)$ with an order 2 tensor X, 1 would be a scalar (order 0 tensor) since it needs to be summed with another scalar. But in the expression $\det(X + 1)$, 1 would need to be an order 2 tensor filled with ones, because it is summed with another order 2 tensor. To accomplish this broadcasting, constants have their order set to -1 to start with. Then, when an error is encountered during tensor order determination because an order -1 constant does not fit into a part of the expression, the order of the constant is set to the value that makes it fit. In a sum, this would be the order of the other addends. In a product, the order would be determined by the given input indices. With a special function, the order is the input order the function operates on. Since all our special functions operate on matrices, this is always two. The case of an elementwise function is more difficult, since they operate on arbitrary input tensor orders. Nonetheless, we may find a fitting order using the expression in which the output of the elementwise function is used, by using the order that is needed there. This is possible since the input and output order of elementwise functions are always the same. As an example, the order of the tensor constant 2 in the expression $X + \log(2)$ may be determined by finding out what order $\log(2)$ needs to be to fit in the sum. Since addends always have the same order, $\log(2)$, and by extension 2, need to have the same order as X. Using these rules, we determine the tensor orders of all nodes in an expression DAG.

Node Type	Tensor order
`SUM`	order(left(X))
`PRODUCT`	output_indices(X)
`ELEMENTWISE FUNCTION`	order(right(X))
`adj`, `inv`	2
`det`	0

Table 3.2: Rules applied to determine tensor orders of a node X, depending on the node's type. Note that the output_indices of a product node are given after the $\rightarrow$ in the product string, that the sum may also take its order from the right child, and that each special function (`det`, `inv`, `adj`) needs its own rule.

3.2.4 Axis Propagation

In addition to the tensor order, we also require each node to store what its axes are. The general structure of tensor derivatives is invariant to the specific axis lengths of the tensors variables. For example, the derivative of $X *_{(ij,ij\rightarrow)} X$ with respect to the order 2 tensor X is always $X + X$ or $2 *_{(,ij\rightarrow ij)} X$, no matter whether $X \in \mathbb{R}^{3\times 5}$ or $X \in \mathbb{R}^{400\times 2}$. This is why we do not require the length of each axis of each variable in a tensor expression to differentiate it.

However, knowing how the shape of each node depends on the shape of the input variables is still useful. It is necessary for generating program code to numerically compute the derivative we determine and gives us more information about the structure of the derivative. For these reasons, we do not only propagate tensor order through the input and differentiation DAGs, but also the axes.

First, we give each node a tuple of axes. Note that we simply use integers to refer to axes in our DAGs. A node X with tensor order 2 may for example have the axes $(1, 2)$, where 1 and 2 are simply names for axes, not containing any further information. In the same way we propagated tensor order, we start from the leaf nodes of the DAG and move toward the top.

The leaf nodes of the DAG contain variables and constants. Constants start out with no axes while they still have the initial tensor order -1. When a constant gets broadcast, it is assigned fitting axes depending on its surroundings in the expression, in the same way they get fitting tensor orders.

Variables are assigned an n-tuple of axes, n being the tensor order of the variable. Each axis is newly created and not used by any other variable at first. Then, these axes are propagated upwards through the DAG, using similar rules to the propagation of tensor orders, visible in Table 3.3.

Node Type	Axes
`SUM`	axes(left(X))
`PRODUCT`	See Algorithm 1
`ELEMENTWISE FUNCTION`	axes(right(X))
`adj`, `inv`	axes(right(X))
`det`	()

Table 3.3: Rules applied to determine axes of a node X, depending on the type of node it is.

The rule for products is slightly more complex. It chooses the correct axes for the product node from the axes of its children based on the product index strings. See Algorithm 1 for further details.

Algorithm 1 PropagateProductAxesBottomUp (*node*)

```
for each $i$ in output_indices($node$) do
    if $i$ is in left_indices($node$) then
        $p \leftarrow$ position of $i$ in left_indices($node$)
        append axes(left($node$))[$p$] to axes($node$)
    else if $i$ is in right_indices($node$) then
        $p \leftarrow$ position of $i$ in right_indices($node$)
        append axes(right($node$))[$p$] to axes($node$)
    end if
end for
```

After this procedure, each node has a fitting tuple of axes. However, many of these axes with different names are actually the same. For example, in the expression $A + B$ for two order 2 tensors A and B, this procedure may assign A the axis tuple $(1, 2)$ and B the axis tuple $(3, 4)$. A and B are assigned different axes even though the sum requires axis 1 to be equal to axis 3 and axis 2 to be equal to axis 4.

This is why we pass through the DAG a second time, going from the top toward the bottom, unifying axes that are the same. For each node x, we set its children's axes depending on its own, with different procedures for different node types.

Node Type	Left Child Axes	Right Child Axes
`SUM`	axes(X)	axes(X)
`PRODUCT`	See Algorithm 2	See Algorithm 2
`ELEMENTWISE FUNCTION`	-	axes(X)
`adj`, `inv`	-	axes(X)
`det`	-	()

Table 3.4: Rules applied to determine the axes of a node X when propagating downward through a DAG. Note that elementwise and special functions have no left child, as we always set their argument to their right child.

An overview of the applied propagation rules is seen in Table 3.4. For product nodes we employ a more complex procedure, setting each child's axes to the node's axes depending on the product indices, as well as unifying equivalent axes. More details may be seen in Algorithm 2.

Algorithm 2 PropagateProductAxesTopDown (*node*)

```
for each $i$ in output_indices($node$) do
    $p \leftarrow$ position of $i$ in output_indices($node$)
    for each $j$ in left_indices($node$) do
        $q \leftarrow$ position of $j$ in left_indices($node$)
        if $i = j$ then
            axes(left($node$))[$q$] $\leftarrow$ axes($node$)[$p$]
        end if
    end for
    for each $j$ in right_indices($node$) do
        $q \leftarrow$ position of $j$ in right_indices($node$)
        if $i = j$ then
            axes(right($node$))[$q$] $\leftarrow$ axes($node$)[$p$]
        end if
    end for
end for
for each $i$ in left_indices($node$) do    ▷ Unifies equivalent left and right child axes
    $p \leftarrow$ position of first occurence of $i$ in left_indices($node$)
    if $i$ exists in right_indices($node$) then
        $q \leftarrow$ position of first occurence of $i$ in right_indices($node$)
        $l \leftarrow$ axes(left($node$))[p]
        $r \leftarrow$ axes(right($node$))[q]
        rename axis $l$ to $r$ in all nodes of the DAG
    end if
end for
```

After this second pass through the DAG, each node now contains a fitting tuple of axes, unified as much as possible, with each axis originating from a variable in the original DAG.

4 Differentiation

The previous chapter was concerned with developing a suitable internal representation for tensor expressions, which turned out to be a directed acyclic graph (DAG). In this chapter, we use this DAG representation of expressions for differentiation. We start by giving some background information on the general differentiation procedure and describe the differentiation rules in theory. Following that, we elaborate on the details of our implementation of the procedure and present a problem we discovered in the differentiation rule for tensor products, along with a solution for it. We finish this chapter with a description of how the correctness of our implementation is verified and a discussion of how tensor expressions grow when we differentiate them.

4.1 Background

The theory for differentiating tensor expressions using our Einstein-like notation was developed by Laue, Mitterreiter and Giesen [3]. They developed differentiation rules for DAG representations of tensor expressions based on the Fréchet Derivative. In their work, they present two modes of differentiation, forward and reverse mode. This thesis utilizes reverse mode.

In reverse mode, the computation traverses 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 v. This value, $\frac{dy}{dv}$, is also called the pullback of v. When traversing the DAG, the pullback is known for each parent node u of v, so we may apply the chain rule:

$$\frac{dy}{dv} = \sum_{u:(v,u)\in E} \frac{dy}{du} \cdot \frac{du}{dv}, \tag{4.1}$$

where $(v, u) \in E$ means that there is an edge going from v to u in the DAG, which makes u a parent of v. Note also that $\cdot$ is a placeholder for an appropriate tensor product, which is specified in the differentiation rules. Equation 4.1 allows us to

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad\qquad \begin{pmatrix} \frac{\partial x_1}{\partial x_1} & \frac{\partial x_2}{\partial x_1} & \frac{\partial x_3}{\partial x_1} \\ \frac{\partial x_1}{\partial x_2} & \frac{\partial x_2}{\partial x_2} & \frac{\partial x_3}{\partial x_2} \\ \frac{\partial x_1}{\partial x_3} & \frac{\partial x_2}{\partial x_3} & \frac{\partial x_3}{\partial x_3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 4.1: *Left*: An order 1 tensor. *Right*: The derivative of the left tensor with respect to itself, resulting in an order 2 delta tensor.

calculate the pullback of v using the known pullbacks of v's parents. The only new calculation to be done for each node is the derivation of the parent nodes u with respect to v, $\frac{du}{dv}$. This is accomplished by a number of differentiation rules covered in the next section of this chapter.

The procedure starts with the root node y, representing the entire expression, and computes its pullback, which is the derivative with respect to itself. This pullback is then used to compute the pullback of y's child nodes, which are used to compute their child nodes, continuing recursively until all paths from the root node y to the argument node have been traversed. The pullback for the node containing the argument is the result of the differentiation, the derivative of the whole expression y with respect to the argument.

In the very first step of the procedure, the derivative of the root node y with respect to itself is computed. Since y is a tensor-valued function, its derivative is also a tensor. Specifically, differentiating y with respect to itself means that each entry of the tensor represented by y is differentiated once with respect to all other entries. This differentiation creates a new tensor of the same size as y for each entry in y, where for each of these tensors, only one entry is 1, while the rest are 0, visible in Figure 4.1. Concretely, if y is an order n tensor, then its derivative is a tensor δ_n of order $2n$, where

$$\delta_n[i_1, .., i_{2n}] = \begin{cases} 1 & \text{if } (i_1, .., i_n) = (i_{n+1} ... i_{2n}) \\ 0 & \text{else} \end{cases} \tag{4.2}$$

Intuitively, the first n indices point to the entry of y that is being differentiated, while the last n indices point to the entry with respect to which we are differentiating. Only when we differentiate an entry with respect to itself is the result a 1, otherwise it is 0. Tensors that satisfy Equation 4.2 are called delta tensors. Note that we call a delta tensor of order $2n$ by the name δ_n and not δ_{2n}, because delta tensors may only have even order. The first node created in the differentiation procedure is always a

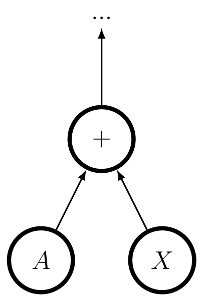

Figure 4.2: A sum node in an expression DAG. The pullbacks of its children A and X are the same as its pullback.

delta tensor representing the root node y differentiated with respect to itself.

4.2 Differentiation Rules

Having established the basic differentiation procedure, we will now describe the rules used to compute the derivative $\frac{du}{dv}$ for a node v and its parent u. Proofs for the correctness of these rules can be found in the work [3] of Laue, Mitterreiter and Giesen. Since we are computing $\frac{du}{dv}$, the applied rule depends on the type of node u is.

4.2.1 Sum Rule

If u is a sum node, the differentiation is very simple. In this case, u adds nothing of its own to the pullback of v, so the contribution of u to the pullback of v is only $\frac{dy}{du}$. Thus, when differentiating a sum node that is their children's only parent, like the one seen in Figure 4.2, the pullback of the children is simply the pullback of the sum node.

4.2.2 Product Rule

If u is a product node representing the subexpression $v *_{(s_1,s_2 \to s_3)} w$ with index strings s_1, s_2 and s_3 in a tensor expression where the root node has indices s_4, then we may apply the product rule as follows. The contribution of u to the pullback of v is

$$\frac{dy}{du} *_{(s_4 s_3, s_2 \to s_4 s_1)} w,$$

and the contribution to the pullback of w is

$$\frac{dy}{du} *_{(s_4 s_3, s_1 \to s_4 s_2)} v.$$

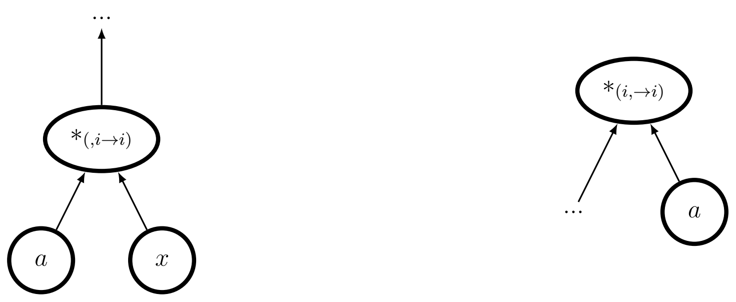

Figure 4.3: *Left*: A product node representing a scalar-vector-product in an expression DAG. "..." represents the rest of the DAG. *Right*: Derivative of the left DAG with respect to x computed using the product rule. "..." represents the pullback of the product node. Note that the product indices in the right DAG depend on the indices s_4 of the root node in the left DAG (not visible here), which is assumed to be scalar in this case.

An example of how the product rule is applied to an expression DAG can be seen in Figure 4.3. There is a caveat to the product rule, because by itself it is not applicable to all tensor products. This detail will be discussed further in section 4.6.

4.2.3 Function Rules

If u is a function node, we distinguish between two cases. The node u either represents an elementwise function or a special function. The elementwise case allows for a slight simplification over the special function case.

If the node u represents an elementwise function f being applied to the child node v, then the contribution of u to the pullback of v is

$$\frac{dy}{du} *_{(s_2 s_1, s_1 \to s_2 s_1)} f'(v),$$

where s_1 is the index string of v, s_2 is the index string of the root node of the DAG and f' is the derivative of f. An example can be seen in Figure 4.4.

If the node u represents a special function f being applied to the child node v, then the contribution of u to the pullback v is

$$\frac{dy}{du} *_{(s_3 s_2, s_2 s_1 \to s_3 s_1)} f'(v),$$

where s_1 is v's input, representing the domain of f, s_2 is the index string of v, representing the range of f, and s_3 is the index string of the root node of the DAG with f' still being the derivative of f. The distinction between s_1 and s_2 is necessary because domain and range of f may not be the same with these functions. The derivatives f' of all accepted functions f need to be known. They are listed in Table 4.1.

Function	Derivative wrt. x
$-(x)$	$-(1)$
$\sin(x)$	$\cos(x)$
$\cos(x)$	$-(\sin(x))$
$\tan(x)$	$1./(\cos(x).^*\cos(x))$
$\arcsin(x)$	$1./(1+-(x.^*x))^{0.5}$
$\arccos(x)$	$-(1./(1+-(x.^*x))^{0.5})$
$\arctan(x)$	$1./(1+(x.^*x))$
$\exp(x)$	$\exp(x)$
$\log(x)$	$1./x$
$\tanh(x)$	$1+-(\tanh(x).^*\tanh(x))$
$\text{abs}(x)$	$\text{sign}(x)$
$\text{sign}(x)$	0
$\text{relu}(x)$	$\text{relu}(\text{sign}(x))$
$1./x$	$-(1./(x.^*x))$
$\text{inv}(x)$	$-(\text{inv}(x)) *_{(ij,kl\to kjli)} \text{inv}(x)$
$\det(x)$	$\text{adj}(x) *_{(ij,\to ji)} 1$

Table 4.1: Derivatives of all used functions. $.^*$ is a placeholder for elementwise multiplication with the fitting indices. The elementwise inverse $1./x$ is internally handled as an elementwise function, which is why its derivative is listed here.

4.2.4 Exponentiation Rules

If u is a power node representing the subexpression A^B, where A has indices s_1, in an expression where the root node has indices s_2, there are two possible differentiation rules to apply. If only A contains the differentiation argument, the pullback of A^B with respect to A is

$$\frac{dy}{du} *_{(s_2 s_1, s_1 \to s_2 s_1)} (B *_{(,s_1\to s_1)} A^{B-1}).$$

If only B contains the differentiation argument, the pullback of A^B with respect to B is

$$\frac{dy}{du} *_{(s_2 s_1, s_1 \to s_2)} (A^B *_{(s_1,s_1\to s_1)} \log(A)).$$

The case of both A and B containing the differentiation argument is covered by a previous transformation we discuss in section 3.2.1.

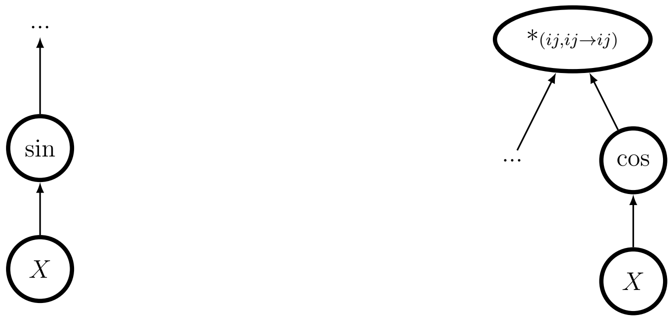

Figure 4.4: *Left*: Part of a DAG that contains the subexpression sin(X). X is an order 2 tensor (matrix) and "..." represents the rest of the DAG. *Right*: Derivative of the left DAG with respect to X computed using the rule for elementwise functions. "..." represents the pullback of the function node. Note that the root node of the left DAG is assumed to be scalar here, which has an impact on the indices in the product node in the right DAG.

4.3 Example of Differentiation Rule Application

In this section, we present a detailed example of the differentiation procedure to demonstrate how the differentiation rules of the last chapter are used in practice. Consider the simple DAG in Figure 4.5, representing the expression $X *_{(ij,ij\to ij)} X$, the elementwise product of a matrix X with itself. We will be differentiating this DAG with respect to X. Each step of the process is visualized in Figure 4.6.

Following the differentiation procedure, we start by differentiating the root node with respect to itself. Since the expression is an order 2 tensor, this creates a delta tensor δ_2 of order 4 (Step 1 in Figure 4.6).

Continuing from the root node, we compute its contribution to the pullback of the left child first. In this case, the left child and the right child are the same, but are still handled separately. Using the product rule, we can see that the contribution of the root node y to the left child is $\frac{dy}{dy} *_{(abij,ij\to abij)} X$, with $\frac{dy}{dy}$ being δ_2 (Step 2 in Figure 4.6).

Since X has no children, we continue with the right child of y. Here, we apply the product rule and obtain the same $\delta_2 *_{(abij,ij\to abij)} X$ as the contribution to the pullback of X. As contributions to the pullbacks of the same node need to be added (see Equation 4.1), we add a sum node as the root of our differentiation DAG (Step 3 in Figure 4.6). Because all ways from the root node to the argument have been processed, we are finished.

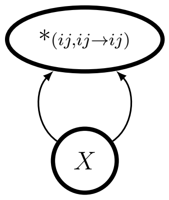

Figure 4.5: Expression DAG representing the elementwise product of a matrix X with itself.

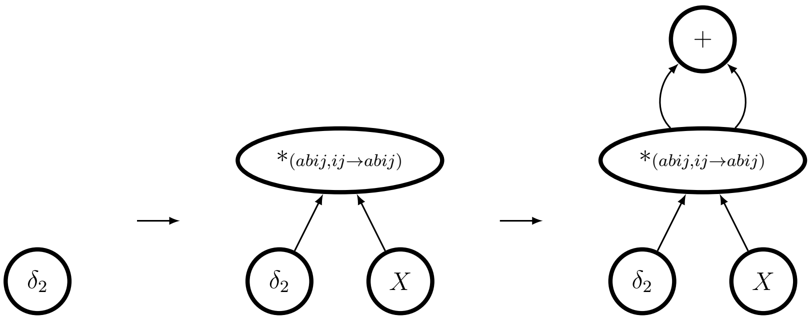

Figure 4.6: Progression of the differentiation procedure on the DAG in Figure 4.5. *Left*: Pullback of the root node. *Middle*: One contribution to the pullback of X. *Right*: Pullback of X, both contributions summed.

4.4 Implementation

In this chapter we describe how the reverse mode differentiation procedure is implemented using the DAG representation of tensor expressions and the differentiation rules introduced in the previous sections.

An overview of the differentiation procedure can be seen in Algorithm 3 and Algorithm 4. The details will be explained in the following paragraphs.

Algorithm 3 Differentiate (*root*, *argument*)

```
*diffdag* ← delta(order(*root*))        ▷ Creates a node representing $\delta_{\text{order}(root)}$
**return** ReverseModeDiff(*root*, *argument*, *diffdag*, EmptyDictionary())
```

As can be seen in Algorithm 3, the first node is added specially, before the real differentiation algorithm starts. This node, a delta tensor, is the derivative of the root node y with respect to itself and has double the order of the original root node.

The original DAG is traversed from the root node down to the argument node, building up the differentiation DAG alongside it. With each node u of the original DAG that is processed, its contributions to the pullbacks of its child nodes are added to the differentiation DAG. These pullbacks are computed using the differentiation rules discussed in the previous chapter.

Algorithm 4 ReverseModeDiff (*node*, *argument*, *diffdag*, *contributions*)

```
for each child of node do
    if subtree of child does not contain argument then
        Continue with next child
    end if
    Add node's contribution to child's pullback to diffdag using fitting diff. rule
    if child is in keys(contributions) then
        Add new contribution to contributions[child] using a sum node
        Move all parents of the contributions[child] to the sum node
        Update contributions[child] as the sum node
    else
        contributions[child] ← diffdag
        diffdag ← ReverseModeDiff(child, argument, diffdag, contributions)
    end if
end for
return diffdag
```

As we are operating on a DAG and not a tree, a single node may be reached multiple times during differentiation. When differentiating a node u, one of its children v may have already been reached through a different parent. In this case, a part of the pullback of v has already been computed and is present in the differentiation DAG. The new pullback contribution we computed from u needs to be added to the old part, as can be seen from Equation 4.1. To be able to do this, we keep a dictionary mapping from nodes in the original DAG which we have already reached to their pullback in the differentiation DAG. This lets us add the old and new pullback contributions with a sum node. Afterwards, we update the entry in the dictionary to the sum node that was just added so that further contributions are added there. We also need to move all nodes in the differentiation DAG that rely on the pullback of v, specifically the parents. Thus, we set all parents of the old pullback to have the updated pullback as their child. This way, we build up the correct pullback over time, adding contributions as we find new ways to reach each node.

4.5 Example of Multiple Pullback Contributions

To better illustrate the process of building a pullback piece by piece, we showcase an example. Consider the DAG in Figure 4.7 representing the expression $\sin(v) + \cos(\sin(v))$ for an order 1 tensor v. We differentiate this expression with respect to v.

Starting out, we compute the pullback of the root node. Since the output of the root node is an order 1 tensor, its pullback, the derivative with respect to itself, is

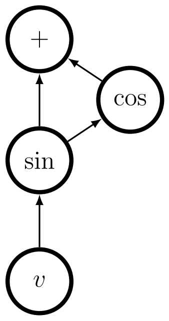

Figure 4.7: DAG representing the expression $\sin(v) + \cos(\sin(v))$.

an order 2 delta tensor δ_1 (Step 1 in Figure 4.8).

Next, we compute the contribution of the root node to its children's pullbacks. However, since the root node is a sum node, it adds nothing of its own, which makes its children's pullbacks the same as its own.

Continuing with the node's children, we start (arbitrarily) with the left child. The left child is an elementwise function node representing $\sin(v)$. Using the differentiation rule for elementwise functions along with the derivative of the sin function, we see that its contribution to its child's pullback is $\delta_1 *_{(ba,a \to ba)} \cos(v)$. This is the first contribution to the pullback of v (Step 2 in Figure 4.8). Since the only child of the sin node is the argument node v, we need not continue further and instead jump back up to explore the second child of the root node.

The second child is an elementwise function node representing $\cos(\sin(v))$. Using the differentiation rule for elementwise functions along with the derivative of the cos function, we obtain $\delta_1 *_{(ba,a \to ba)} -(\sin(v))$ as the contribution of the node to its child's pullback.

The cos node's child is the sin node we have been to before. This means we need to add the new contribution to this node's pullback to the old one. The old contribution to the pullback of $\sin(v)$ was δ_1, to which we now add $\delta_1 *_{(ba,a \to ba)} -(\sin(v))$ using a sum node. Since the pullback of v depends on the pullback of $\sin(v)$, we need to make sure to update it. We manage this by changing all parents of the old pullback to be parents of the new pullback instead. The left child of the product $\delta_1 *_{(ba,a \to ba)} \cos(v)$ gets updated to be the sum node in the new, complete pullback (Step 3 in Figure 4.8).

Since there are no more unvisited children, we are done. The finished differentiation DAG represents the expression $(\delta_1 + (\delta_1 *_{(ba,a \to ba)} -(\sin(v)))) *_{(ba,a \to ba)} \cos(v)$. The complete, simplified DAG is visible in Figure 4.9.

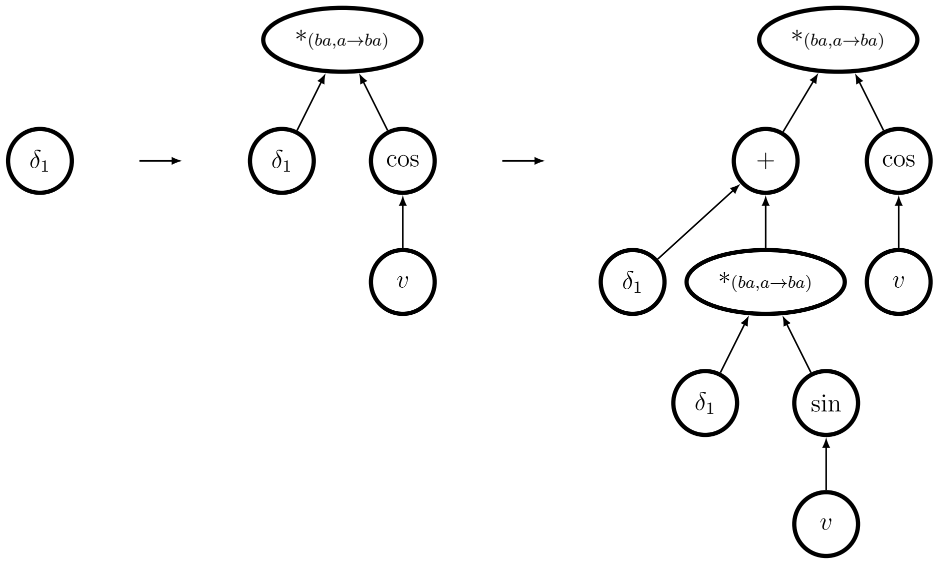

Figure 4.8: Progression of the differentiation procedure on the DAG in Figure 4.7. *Left*: Pullback of the +, sin and cos nodes. *Middle*: First contribution to the pullback of v. *Right*: Full pullback of v.

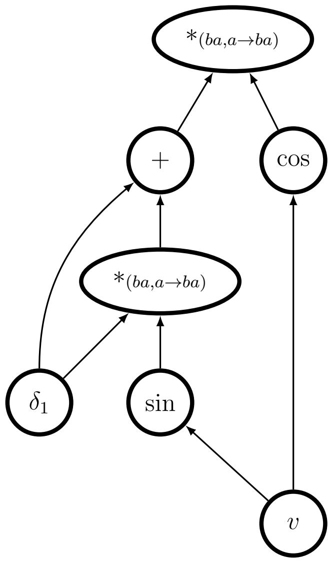

Figure 4.9: Complete, simplified differentiation DAG of the original DAG in Figure 4.7.

4.6 Invalid Nodes Created by the Product Rule

In this section we present a problem with the differentiation of tensor products that was discovered during development, along with a solution.

4.6.1 Problem

We discovered that the differentiation rule for products sometimes creates invalid product nodes. Specifically in the case where an index appears only in the index string of exactly one input of a product node, but not in the output index string. An example of this would be the left DAG in Figure 4.10, containing a node representing the product $A *_{(ij,j\rightarrow)} v$, where the index i only appears in the left index string. When differentiating this DAG with respect to A, we compute the contribution of the product node to the pullback of A and receive $\delta_0 *_{(,j\rightarrow ij)} v$ using the product rule. The resulting product node is not valid, because the index i appears in the result index string, but not in either input index string.

The root of this problem can be found in the proof of the reverse mode differentiation product rule (Theorem 8) in the work [3] by Laue, Mitterreiter and Giesen. In this proof, Laue, Mitterreiter and Giesen find the derivative of a product node $A *_{(s_1,s_2\rightarrow s_3)} B$ with respect to B, although all statements we make are analogously applicable to the differentiation with respect to A. They utilize the Fréchet Derivative, which is defined as follows.

For a function f that maps from an order k tensor to an order l tensor, we call a tensor D of order $k + l$ the derivative of f at x if and only if

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

Here, $\circ$ describes an inner product. In our notation, an inner product has the form $D *_{(s_1s_2,s_2\rightarrow s_1)} h$ for two index strings s_1 and s_2.

Laue, Mitterreiter and Giesen start from this definition applied to the product node $A *_{(s_1,s_2\rightarrow s_3)} B$ and transform it to describe the pullback of B. Important to us is the fourth equality of their proof, where the associative property for tensor products is used to transform $\bar{C} *_{(s_4s_3,s_3\rightarrow s_4)} (A *_{(s_1,s_2\rightarrow s_3)} h)$ into $(\bar{C} *_{(s_4s_3,s_1\rightarrow s_4s_2)} A) *_{(s_4s_2,s_2\rightarrow s_4)} h$. Their goal is to create an expression that ends with a form of $D \circ h$, because that makes D the Fréchet Derivative they are looking for.

However, consider the definition of the associative property for tensor products given by Laue, Mitterreiter and Giesen.

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

where $s_3 \subseteq s_1 \cup s_2$ and $s_4 \cap (s_1 \cup s_2) = \emptyset$. Applying the associative property to our expression would require that $s_1 = s_3s_2$, which is not the case for all product nodes. Thus, we cannot always apply the associative property here.

4.6.2 Solution

To solve this problem, we forego the stated version of the associative property and find another way to transform $\bar{C} *_{(s_4s_3,s_3\to s_4)} (A *_{(s_1,s_2\to s_3)} h)$ into an inner product.

We start by explicitly writing the Einstein-like tensor products as sums. The indices we sum over are the indices of the inputs with the result indices removed.

$$\sum_{s_3} \bar{C}[s_4s_3] \left(\sum_{s_1s_2\setminus s_3} A[s_1]\ h[s_2] \right)$$

As both sums go over separate, non-overlapping index sets, we may collapse them into a single sum.

$$\sum_{s_1s_2s_3} \bar{C}[s_4s_3]\ A[s_1]\ h[s_2]$$

As the first step towards writing our expression as an inner product, we split off part of the large sum to create two smaller sums over different indices.

$$\sum_{s_2} \left(\sum_{s_1s_3\setminus s_2} \bar{C}[s_4s_3]\ A[s_1] \right)\ h[s_2]$$

Next, we write the inner sum in our Einstein-like tensor product notation. This requires that we determine the output indices of the tensor product by removing the indices that are summed over, $s_1s_3 \setminus s_2$, from the input indices, s_4s_3 and s_1.

The result, $s_1s_3s_4 \setminus (s_1s_3 \setminus s_2)$, can be simplified in the following way:

$$\begin{aligned}
& s_1s_3s_4 \setminus (s_1s_3 \setminus s_2) \\
&= s_1s_3s_4 \setminus (s_1s_3 \cap \overline{s_2}) \\
&= s_1s_3s_4 \cap \overline{(s_1s_3 \cap \overline{s_2})} \\
&= s_1s_3s_4 \cap (\overline{s_1s_3} \cup s_2)
\end{aligned}$$

$$
\begin{aligned}
&= (s_1s_3s_4 \cap \overline{s_1s_3}) \cup (s_1s_3s_4 \cap s_2)\\
&= (s_1s_3s_4 \setminus s_1s_3) \cup (s_1s_3s_4 \cap s_2)\\
&= (s_4 \setminus s_1s_3) \cup (s_1s_3s_4 \cap s_2)\\
&= s_4 \cup (s_1s_3s_4 \cap s_2)\\
&= s_4 \cup (s_1s_3 \cap s_2)
\end{aligned}
$$

where $\bar{x}$ is the set complement of x. The last two simplification steps are possible because the output indices of the root node, s_4, are always be chosen to be disjoint to s_1, s_2 and s_3.

With these output indices, we may rewrite our expression.

$$\sum_{s_2} (C *_{(s_4s_3,s_1 \to s_4 \cup (s_1s_3 \cap s_2))} A)\ h[s_2]$$

We know that the output indices of this expression need to be s_4 from the version we started with. Using this, we can rewrite the outer sum in our tensor product notation as well.

$$(C *_{(s_4s_3,s_1 \to s_4 \cup (s_1s_3 \cap s_2))} A) *_{((s_4 \cup (s_1s_3 \cap s_2)),s_2 \to s_4)} h$$

At this point, if we had s_4s_2 instead of $s_4 \cup (s_1s_3 \cap s_2)$ in the left and right products, the right product would have the form $*_{(s_4s_2,s_2 \to s_4)}$ and would be an inner product. We can see that $s_4s_2 \neq s_4 \cup (s_1s_3 \cap s_2)$ if and only if there are indices in s_2 that are not present in either s_1 or s_3. If we could remove this case, we would be successful in creating an inner product and thus in computing the derivative.

To implement this idea and prevent invalid product nodes from appearing, we modify the original DAG before it is differentiated. For any product node of the form $A *_{(s_1,s_{21}s_{22} \to s_3)} B$ where $s_{21} \cap (s_1s_3) = s_{21}$ and $s_{22} \cap (s_1s_3) = \emptyset$, we call s_{22} the missing indices of this node and transform the node by making the sum over the missing indices explicit.

We append the missing indices to the output index string of the product node, leading to $A *_{(s_1,s_{21}s_{22} \to s_3s_{22})} B$. This prevents an invalid product node from being created, because the indices in s_{22} are now also present in the output indices. However, this also changes the node, since it now has more output axes than before. To remedy this, we add another product node where we explicitly sum over these axes while multiplying with a tensor full of ones, leading to $(A *_{(s_1,s_{21}s_{22} \to s_3s_{22})} B) *_{(s_3s_{22},s_{22} \to s_3)} 1$. Since s_3 appears in both the left input and the output, and s_{22} appears in both

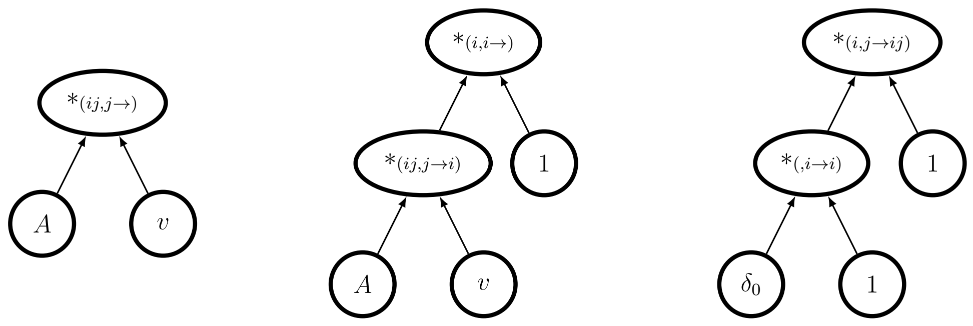

Figure 4.10: *Left*: DAG with a disappearing index i. *Middle*: Modified DAG where the sum over index i is made explicit. *Right*: Derivative of the middle DAG with respect to A.

inputs, this new product never has any missing indices. This transformed DAG achieves the same result as the original DAG, but by making the sum over the axes explicit in another product node, we prevent the problematic case.

In the example in Figure 4.10, we would transform the original expression from $A *_{(ij,j\to)} v$ to $(A *_{(ij,j\to i)} v) *_{(i,i\to)} 1$. Both the modified DAG and the derivative are visible in Figure 4.10.

4.7 Numerical Checks

To be certain of the correctness of our algorithm even in edge cases, we implement test cases. Each test case consists of a tensor expression describing a function f, returning an order q tensor. Our algorithm computes the derivative f' of this function with respect to an argument x_k of order p. This computed derivative function is evaluated with random values for all tensor variables $x_1, x_2, ..., x_n$ in the function, including the differentiation argument x_k. We call these specific values $v_1, ..., v_n$. Note that since each expression only specifies the number of axes a tensor variable has, but not the length of each axis, we need to arbitrarily assign the lengths. We assign the length 3 to every axis. We then compare $f'(v_1, ..., v_n)$ to a numerical approximation of the derivative we call $\bar{f}'$. If the tensor of differences, $|\bar{f}' - f'(v_1, ..., v_n)|$, is small enough in every entry (we use 10^{-6} as the maximum error allowed) then the check is passed.

This approximation, $\bar{f}'$, is computed using a variant of the definition of the derivative. For any function g with n variables $x_1, ..., x_n$, including a scalar argument x_k, we may approximate the derivative with respect to x_k at a point $(v_1, ..., v_n)$ using the following formula with a very small number h (We use 10^{-8}).

$$\bar{g'} = \frac{g(v_1, ..., v_k - h, ..., v_n) - g(v_1, ..., v_k + h, ..., v_n)}{2h}$$

Since the argument for our differentiation of f is an order p tensor, it is not always scalar. Thus, we have to compute this approximation for each entry of the argument x_k. In each computation, we only add and subtract h to one element in x_k, leaving the rest constant. Each computation returns a tensor of order q, approximating the partial derivative of f with respect to one element of x_k. We combine all the computed order q tensors in a tensor $\bar{f'}$ of order $p+q$, resulting in an approximation of the full derivative.

These checks are performed on a number of expressions to cover all basic features of our developed language. A full list of them can be found in the appendix.

4.8 Expression Growth

For performance purposes, it is of interest how the sizes of expressions change when we differentiate them. The size of an expression DAG may be measured in the number of nodes $|V|$ or the number of edges $|E|$ in the DAG. We include discussions on both size measures. In the differentiation process, there are two points where nodes and edges are added to the differentiation DAG: when applying the differentiation rules and when summing contributions to the same pullback.

We apply one differentiation rule per node, with each differentiation rule adding a constant number of nodes and edges to the differentiation DAG. Thus, the number of nodes and the number of edges both scale linearly ($O(|V|)$) during this step.

When we sum contributions to the pullback of a node X, we add one sum node and two edges for each contribution after the first one. Because each contribution stems from one incoming edge (from a parent node) of X, we add one sum node and a constant 2 edges for each incoming edge of X in total. At most, we add one node and two edges for every edge in the DAG, leading to a growth of $O(|E|)$ both in the number of nodes and edges.

This leads to a total growth in the number of nodes and edges of $O(|V| + |E|)$ from the original expression DAG to the differentiation DAG. However, since each incoming edge of a node in the DAG represents one occurrence of that node in the original unsimplified binary tree, we may also say that the total growth in both the number of nodes and the number of edges is $O(|V|)$ from the original unsimplified binary expression tree to the differentiation DAG.

5 Summary

In this thesis, we covered the design and implementation of a calculus for automatic symbolic differentiation of tensor expressions. We created a language for general tensor expressions using an Einstein-like notation for tensor products and discussed how expressions formulated in this language may be parsed into a suitable representation, a directed acyclic graph (DAG). We also demonstrated multiple preprocessing steps, substituting some subexpressions for more easily differentiable ones, eliminating multiple occurrences of the same subexpression and storing both tensor order and an axis tuple for each node in the DAG. We gave a review of the theory behind the differentiation of tensor expressions using our notation and demonstrated our implementation, which is able to differentiate the preprocessed expression DAGs. A problem with the differentiation of some tensor products in the original work [3] by Laue, Mitterreiter and Giesen, on which this thesis builds, was identified and remedied. The correctness of our implementation was verified by numerical checks, where the computed derivatives of a number of example inputs were compared with numerical approximations for random inputs. We were also able to bound the growth of tensor expressions induced by differentiation to be linear in the number of nodes and edges in the directed acyclic graph representation of the expression.

Possible further development could address the flexibility and readability of the input language, for example by adding a generalized transpose operation that allows axis permutations, which have to be embedded in tensor products in the current language. The tensor products could also be generalized even further, possibly by adding a unary version that allows one to sum over a single tensor's axes. Furthermore, our algorithms could be analyzed and optimized in regard to efficiency and performance for different inputs. Lastly, one possible use of this thesis' results are convexity checks for arbitrary functions. How our representation of tensor function derivatives can be used to detect convexity and concavity is worth investigating.

Bibliography

[1] Albert Einstein. “Die Grundlage der allgemeinen Relativitätstheorie”. In: *Annalen der Physik* 49.7 (1916), pp. 284–339.

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

[3] 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.

[4] Sören Laue, Matthias Mitterreiter, and Joachim Giesen. “Computing Higher Order Derivatives of Matrix and Tensor Expressions”. In: *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)* (2018).

[5] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* 2nd ed. Springer Science and Business Media LLC, 2006.

[6] Gregorio Ricci and Tullio Levi-Civita. “Méthodes de calcul différentiel absolu et leurs applications”. In: *Mathematische Annalen* 54 (1-2) (Mar. 1900), pp. 125–201.

Listings

List of Figures

List of Tables

List of Algorithms

Appendix

List of Examples used in Numerical Checks

This list of example inputs with their respective derivatives is structured in the following way. Each example consists of two parts separated by an empty line, with examples being separated by a horizontal rule. The first part is the input string, containing the tensor variable names with their orders, the expression itself and the name of the variable with respect to which is being differentiated. The second part is the output of our algorithm, the resulting derivative.

```
declare a 0 expression a derivative wrt a

delta(0)
```

```
declare a 1 b 1 c 0 expression (a*(i,i->)b)*(,->)c derivative
wrt c

(a *(i,i->) b)
```

```
declare a 0 expression (a*(,->)a) derivative wrt a

(a + a)
```

```
declare a 0 b 0 expression (b*(,->)a) *(,->) a derivative wrt a

((a *(,->) b) + (b *(,->) a))
```

```
declare a 0 b 0 expression a + b derivative wrt a

delta(0)
```

```
declare a 0 b 0 expression a + a + b derivative wrt a

(delta(0) + delta(0))
```

```
declare a 0 b 0 expression a + b + a derivative wrt a

(delta(0) + delta(0))
```

```
declare a 0 b 0 expression a - b derivative wrt b

-1
```

```
declare a 0 b 0 expression a - a - b derivative wrt a

(delta(0) + -1)
```

```
declare a 0 b 0 expression a - b - a derivative wrt a

(delta(0) + -1)
```

```
declare a 1 b 1 c 1 expression a*(i,i->i)a + b + c derivative
wrt a

((delta(1) *(ai,i->ai) a) + (delta(1) *(ai,i->ai) a))
```

```
declare a 1 b 1 c 1 expression a*(i,i->i)b + a*(i,i->i)c
derivative wrt a

((delta(1) *(ai,i->ai) b) + (delta(1) *(ai,i->ai) c))
```

```
declare a 1 b 1 c 0 expression a*(i,i->)b + c derivative wrt a

b
```

```
declare a 2 b 1 expression a*(ij,j->i)b derivative wrt a

(delta(1) *(ai,j->aij) b)
```

```
declare a 2 b 1 expression a*(ij,j->i)b derivative wrt a

(delta(1) *(ai,j->aij) b)
```

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

((delta(1) *(ai,ij->aj) A) + (delta(1) *(ai,ij->aj) B))
```

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

(delta(1) *(ai,ij->aj) (A + B))
```

```
declare A 2 B 2 x 1 expression (A*(ij,ij->ij)B) * (ij,j->i)x
derivative wrt x

(delta(1) *(ai,ij->aj) (A *(ij,ij->ij) B))
```

```
declare x 1 expression x*(i,i->)x derivative wrt x

(x + x)
```

```
declare a 1 b 1 c 1 d 1 expression a*(i,i->)b + a*(i,i->)c
+ a*(i,i->)d derivative wrt a

((b + c) + d)
```

```
declare a 1 b 1 c 1 expression a*(i,i->i)a - b - c derivative
wrt a

((delta(1) *(ai,i->ai) a) + (delta(1) *(ai,i->ai) a))
```

```
declare a 1 b 1 c 1 expression a*(i,i->i)b - a*(i,i->i)c
derivative wrt a

((delta(1) *(ai,i->ai) b) + ((delta(1) *(ba,a->ba) -1)
*(ai,i->ai) c))
```

```
declare A 2 B 2 x 1 expression (A-B)*(ij,j->i)x derivative wrt x

(delta(1) *(ai,ij->aj) (A - B))
```

```
declare x 1 expression sin(x) derivative wrt x

(delta(1) *(ba,a->ba) (cos(x)))
```

```
declare A 2 x 1 expression A *(ij,j->i) (sin(x)) derivative wrt x

((delta(1) *(ai,ij->aj) A) *(ba,a->ba) (cos(x)))
```

```
declare A 2 x 1 expression sin( A*(ij,j->i)x ) derivative wrt x

((delta(1) *(ba,a->ba) (cos((A *(ij,j->i) x)))) *(ai,ij->aj) A)
```

```
declare x 1 expression cos(x) + sin(x) derivative wrt x

((delta(1) *(ba,a->ba) (-((sin(x))))) + (delta(1)
*(ba,a->ba) (cos(x))))
```

```
declare x 1 expression exp(x) derivative wrt x

(delta(1) *(ba,a->ba) (exp(x)))
```

```
declare A 2 x 1 expression A *(ij,j->i) exp(x) derivative wrt x

((delta(1) *(ai,ij->aj) A) *(ba,a->ba) (exp(x)))
```

```
declare y 1 x 1 expression y / x derivative wrt x

((delta(1) *(ba,a->ba) y) *(ba,a->ba) (-((1 / ((x
*(a,a->a) x))))))
```

```
declare x 1 expression tan(x) derivative wrt x

(delta(1) *(ba,a->ba) (1 / (((cos(x)) *(a,a->a) (cos(x))))))
```

```
declare x 1 expression arcsin(x) derivative wrt x

(delta(1) *(ba,a->ba) (1 / (((1 - (x ^ 2)) ^ 0.5))))
```

```
declare x 1 expression arccos(x) derivative wrt x

(delta(1) *(ba,a->ba) (-((1 / (((1 - (x ^ 2)) ^ 0.5))))))
```

```
declare x 1 expression arctan(x) derivative wrt x

(delta(1) *(ba,a->ba) (1 / (((x *(a,a->a) x) + 1))))
```

```
declare x 1 expression log(x) derivative wrt x

(delta(1) *(ba,a->ba) (1 / (x)))
```

```
declare x 1 expression tanh(x) derivative wrt x

(delta(1) *(ba,a->ba) (1 - ((tanh(x)) *(a,a->a) (tanh(x)))))
```

```
declare x 1 expression abs(x) derivative wrt x

(delta(1) *(ba,a->ba) (sign(x)))
```

```
declare x 1 expression sign(x) derivative wrt x

(delta(1) *(ba,a->ba) 0)
```

```
declare x 1 expression relu(x) derivative wrt x

(delta(1) *(ba,a->ba) (relu((sign(x)))))
```

```
declare x 1 a 0 expression x^a derivative wrt x

(delta(1) *(ba,a->ba) (a *(,a->a) (x ^ (a - 1))))
```

```
declare x 1 a 0 expression (x^a) *(i,i->) x derivative wrt x

((x *(a,a->a) (a *(,a->a) (x ^ (a - 1)))) + (x ^ a))
```

```
declare x 0 a 1 expression a^x derivative wrt x

((a ^ x) *(b,b->b) (log(a)))
```

```
declare x 0 a 1 expression x^x derivative wrt x

(((x ^ x) *(,->) (log(x))) + (((x ^ x) *(,->) x) *(,->)
(1 / (x))))
```

```
declare A 2 expression inv(A) derivative wrt A

(delta(2) *(efcd,cdab->efab) ((-((inv(A)))) *(ij,kl->kjli)
(inv(A))))
```

```
declare A 2 expression det(A) derivative wrt A

((adj(A)) *(ij,->ji) 1)
```

```
declare X 2 expression adj(X) derivative wrt X

(((inv(X)) *(cd,ab->cdab) ((adj(X)) *(ij,->ji) 1)) + ((delta(2)
*(abij,->abij) (det(X))) *(efcd,cdab->efab) ((-((inv(X))))
*(ij,kl->kjli) (inv(X)))))
```

```
declare X 2 expression 1*(,ij->)X derivative wrt X

(1 *(ij,->ij) 1)
```

```
declare X 2 expression X + X derivative wrt X

(delta(2) + delta(2))
```

```
declare X 2 expression 1*(ij, ij -> )(X + X) derivative wrt X

(1 + 1)
```

```
declare X 2 expression 1*(ij, ij -> )X + 1*(ij, ij -> )X
derivative wrt X

(1 + 1)
```

```
declare X 2 expression 1*(ij, ij -> )X + 1*(ij, ij -> ) (X+1)
derivative wrt X

(1 + 1)
```

```
declare v 1 expression v*(i, i -> )v + 1*(i, i -> )(v)
derivative wrt v

((v + v) + 1)
```

```
declare v 1 expression (v *(i,i->i) v) + (v *(i,i->i) v)
derivative wrt v

(((delta(1) + delta(1)) *(ai,i->ai) v) + ((delta(1) + delta(1))
*(ai,i->ai) v))
```

```
declare v 1 expression (delta(0) + 1) *(,i->i) v derivative wrt v

(delta(1) *(ai,->ai) (delta(0) + 1))
```

```
declare X 2 expression sin(delta(1)) *(ij,ij->) X derivative
wrt X

(sin(delta(1)))
```

```
declare a 2 X 4 expression a*(ij,ijkl->ijkl)X + cos(X) derivative
wrt X

((delta(4) *(abcdijkl,ij->abcdijkl) a) + (delta(4)
*(efghabcd,abcd->efghabcd) (-((sin(X))))))
```

```
declare X 2 expression sin(delta(1)) *(ij,ij->) X derivative wrt a

0
```

Selbstständigkeitserklärung

Jena, 25.08.2022