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 kinds of 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 most optimization problems, 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 an Einstein-like 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 are discussed, followed by a chapter on the differentiation process.

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 order of all variables in the expression. The neccessity of the first two is obvious, while the third might not immediately be. The orders 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 the 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 its notation, and generalize it to fit our problem domain.

4.2 Representing Tensor Products

There exists a calculus for tensor expressions, called Ricci calculus, that is heavily used in physics. It which allows differentiation between tensors as n-dimensional arrays and as multilinear functions. The differentiation 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. The main part of Ricci calculus that we bring over into our notation is a derivative of 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. (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*, specifically it's `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.

As an example, see the inner, outer and elementwise product of two order 1 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 $A *_{(ij,ij \to ij)} B$. 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 $A *_{(ii,i \to)} v$.

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.

```
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' | 'adj'
```

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 elementwise 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. This allows the differentiation rules stated in the paper 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. In this chaper we first describe how a tensor expression is parsed into a binary expression tree, then explain the processing steps taken to transform the binary tree into a directed acyclic graph without duplicate subexpressions.

5.1 Parsing

The process of turning a string into a different data structure that allows for further processing is called parsing. It is split into two steps, tokenization and tree building.

5.1.1 Tokenization

To be able to parse an expression string, we first transform it into a sequence of tokens. This is done by the tokenizer, sometimes also called lexical scanner, which recognizes certain defined patterns in the expression string and unites 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 (`'+'`, `'-'`, `'*'`, `'/'`, `'^'`, `','`, `'('`, `')'`, `'>'`), keywords (`'declare'`, `'expression'`, `'derivative'`, `'wrt'`) 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. Examples for special functions are the matrix determinant `det` and the matrix inverse `inv`.

5.1.2 Tree Building

The output of the tokenizer is used to build a binary tree representing the expression.

A binary tree is a type of tree where each node has exactly two child nodes. Like in other trees, each node has exactly one parent node. Exceptions to this are the root, which has no parent, and the leaves, which have no children. Formally, we define a binary tree B as follows: B is either the empty set or a tuple $(\mathrm{left}(B), \mathrm{root}(B), \mathrm{right}(B))$, where $\mathrm{left}(B)$ and $\mathrm{right}(B)$ are also binary trees and have B as their only parent, and $\mathrm{root}(B)$ is the root of the tree, containing some value.

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, an exception is thrown and the program aborted.

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 `'('`.

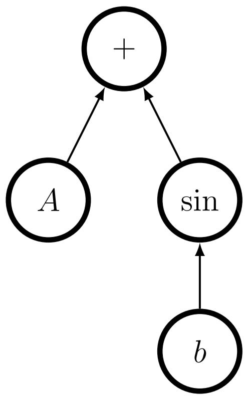

Figure 5.1: Binary expression tree generated from the expression string `A + sin(b)`. Empty nodes are omitted.

- `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 up the call stack and finish.

The binary tree generated from this can be seen in figure 5.1. Note that, by convention, we set the argument of a unary function node to its right child.

5.2 Preprocessing

Having described 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 three 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.

5.2.1 Equivalent Subtree Substitution

Some operations or functions may be viewed as equivalent to, or as a special case of another, more easily differentiable operation or function. 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 5.1.

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

Table 5.1: Table showing all operations that get substituted during preprocessing, along with their substitutes. Note that (.*) is a placeholder for elementwise multiplication and ($\cdot$) is a placeholder for scalar multiplication. Both are replaced with the appropriate tensor products during tensor order determination. elementwise_inverse is an elementwise function for which a differentiation rule is implemented.

5.2.2 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.

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

where $\mathrm{root}(X)$ refers to the root node of X, and $\mathrm{left}(X)$, $\mathrm{right}(X)$ to the left and right child trees of X, respectively. Equality of 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 hash table. When we encounter a subtree that is already present in our hashmap, we let its parent node point to the tree in the hashmap 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.

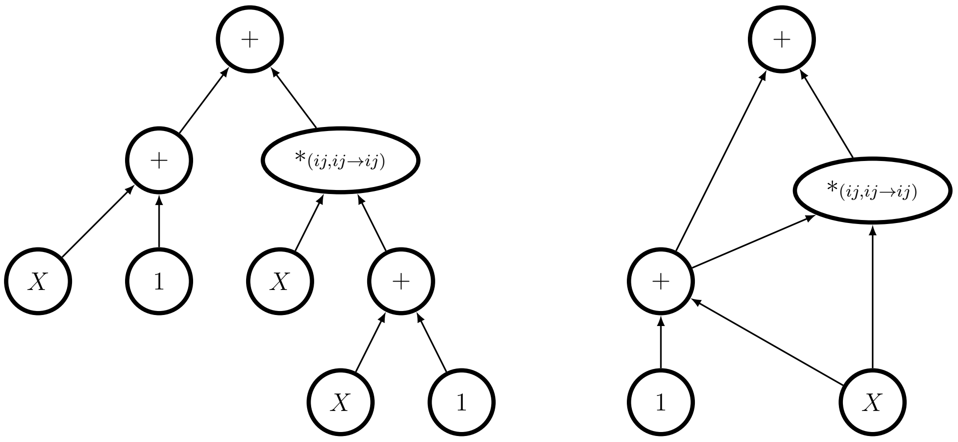

Figure 5.2: Left: Binary tree generated from the input expression `(X+1) + X*(ij,ij->ij) (X+1)`. Right: DAG obtained by eliminating common subtrees in the left tree.

5.2.3 Tensor order Determination

To be able to accurately create new product nodes, which require knowledge of the tensor orders of both inputs and the output, we determine the output tensor order of all nodes in the expression DAG. Starting at the bottom of the DAG, tensor orders are propagated upwards towards 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 of tensor orders. Each parent node calculates its output tensor order based on its child nodes. For example, a sum will have its output order set equal to the output order of its children. A full overview of these order determination rules can be found in table 5.2.

So far we have not addressed how the tensor order of a constant is determined. 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 summand. In a product, the order would be determined by the given input indices. In a special function, the order is the input order the function operates on. Since all our special functions operate on matrices, this is always 2. 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 summands always have the same order, `log(2)`, and by extension `2`, needs to have the same order as `X`.

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

Table 5.2: Rules applied to determine tensor orders of a node X, depending on the type of node it is. Note that the output_indices of a product node are given, that the sum may also take its order from the right child, as it has the same order, and that each special function (`det`, `inv`, `adj`) needs its own rule.

6 Differentiation

The last chapter was concerned with creating a suitable internal representation for tensor expressions, a directed acyclic graph. In this chapter, we use this DAG representation of the expression to differentiate it. We will start by giving some background information on the general differentiation procedure and describe the differentiation rules applied in theory. Following that, we elaborate on the details of our implementation of this procedure and these rules.

6.1 Background

The theory for differentiating tensor expressions using our Einstein-like notation was developed by Laue, Mitterreiter and Giesen. They developed differentiation rules for DAG representations of tensor expressions based on the Fréchet Derivative. In their paper, 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. Since the pullback is known for each parent node u of v, we may apply the chain rule:

$$\frac{dy}{dv} = \sum_{u:(v,u)\in E} \frac{dy}{du} \cdot \frac{du}{dv} \tag{6.1}$$

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. This rule allows us to calculate the pullback of v using the known pullbacks of vs parents u. 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 computation starts with the topmost 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 ys child nodes, which are used to compute their child nodes, continuing recursively until all paths from the top node y to the argument node have been traversed. The pullback for the node containing the argument is then 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 top node y with respect to itself is computed. Since y is a tensor-valued function, its derivative is

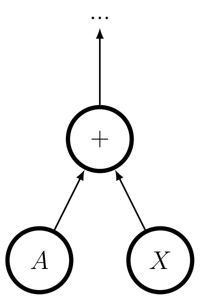

Figure 6.1: A sum node in an expression DAG. Its childrens' pullbacks are the same as its pullback.

also a tensor. Specifically, differentiating y with respect to itself means that each entry of the output of y is differentiated once with respect to all other entries in the output of y. [EXPLAIN DELTA TENSORS HERE]

6.2 Differentiation Rules

Having established the basic derivation 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 [TENSOR PAPER]. Since we are computing $\frac{du}{dv}$, the rule applied depends on the type of node u is.

6.2.1 Sum Rule

If u is a sum node, the derivation 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 childrens only parent, like the one seen in figure 6.1, the pullback of the children is simply the pullback of the sum node.

6.2.2 Product Rule

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

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

and the contribution to the pullback of w is

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

An example of how the product rule is applied to an expression DAG can be seen in figure 6.2. 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 the section on implementation details.

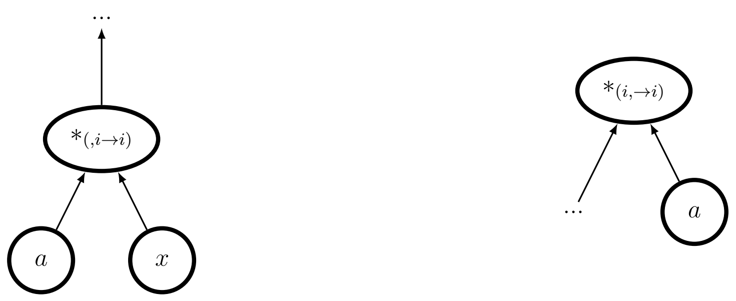

Figure 6.2: **Left**: A product node representing a scalar-vector-product in an expression DAG. "..." represents the rest of the DAG. **Right**: Derivative of the left DAG with respect to x computed using the product rule. "..." represents the pullback of the product node. Note that the topmost output node of the left DAG is assumed to be scalar here. Therefore s_4, the output index string, is empty.

6.2.3 Function Rule

If u is a function node, we differentiate between two cases. u either represents an elementwise function or one of the special functions. 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

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

where s_1 is the index string of v, s_2 is the index string of the output node of the DAG and f' is the derivative of f.

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

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

where s_1 vs 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 output node of the DAG with f' still being the derivative of f. The distinction between s_1 and s_2 is neccessary because domain and range of f are not be the same anymore, as they are with elementwise functions.

The derivatives f' of all accepted functions f need to be known. [Maybe insert a list of them here?]

6.3 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 section.

An overview of the differentiation procedure can be seen in algorithm 1 and 2. The details will be explained in the following paragraphs.

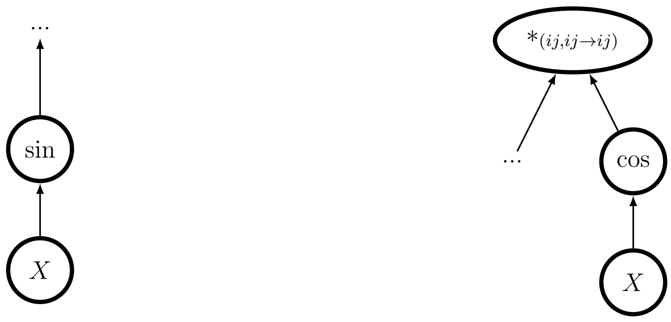

Figure 6.3: **Left**: Part of a DAG that contains the subexpression sin(X). X is an order 2 tensor (matrix) and "..." represents the rest of the DAG. **Right**: Derivative of the left DAG with respect to X computed using the rule for elementwise functions. "..." represents the pullback of the function node. Note that the topmost output node of the left DAG is assumed to be scalar here. Therefore s_2, the output index string, is empty.

Algorithm 1 Differentiate (*root*, *argument*)

```
  diffdag ← Node(delta(order(root)))
  ReverseModeDiff(root, argument, diffdag, EmptyDictionary())
```

Algorithm 2 ReverseModeDiff (*child*, *argument*, *diffdag*, *contributions*)

```
  for each child of node do
      if subtree of child does not contain argument then
          Continue with next child
      end if
      Apply fitting differentiation rule (depending on node) to child
      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 can be seen in algorithm 1, the first node is added specially, before real differentiation algorithm starts. This node is the derivative of the top node y with respect to itself. [Elaborate on delta tensor]

The original DAG is traversed from the top 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.

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 6.1. 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 easily 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. We also need to move all nodes in the differentiation DAG that rely on the pullback of v, which are its 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.

7 Literature

8 Appendices

9 Declaration of autonomy